From Hoare Logic to Matching Logic

Grigore Roşu and Andrei Ştefănescu
Department of Computer Science, University of Illinois at Urbana-Champaign
{grosu, stefanel}@illinois.edu

Abstract. Matching logic 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 provides a language-independent and sound proof system that directly uses the trusted operational semantics of the language as axioms. Matching logic 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 is as powerful as Hoare logic. This paper introduces a technique to mechanically translate Hoare logic proof derivations into equivalent matching logic proof derivations. The presented technique has two consequences: first, it suggests that matching logic 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 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:

\[
\begin{align*}
\text{while}(e)s & \Rightarrow \text{if}(e)s;\text{while}(e)s \text{ else skip} \\
\text{proc}() & \Rightarrow \text{body} \quad \text{where “proc()body” is a procedure}
\end{align*}
\]

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.

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 under the form of Hoare
logic proof systems for deriving Hoare triples \( [\text{precondition}] \) code \( [\text{postcondition}] \). For example, the proof rules below correspond to the operational semantics rules above:

\[
\begin{align*}
\mathcal{H} &\vdash [\psi \land e \neq 0] s \{\psi\} \\
\mathcal{H} &\vdash [\psi] \text{while}(e) s [\psi \land e = 0] \\
\mathcal{H} \cup [\psi] \text{proc}() [\psi'] &\vdash [\psi] \text{body} [\psi'] \\
\mathcal{H} &\vdash [\psi] \text{proc}() [\psi']
\end{align*}
\]

where “proc() body” is a procedure

The second rule takes into account the fact that the procedure proc might be recursive; several instances of the rule are needed for mutually recursive procedures. Both these rules 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 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 semantics; the state-of-the-art approaches in mechanical verification do precisely that [1, 8–10, 12, 17].

Matching logic [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 provides a high-level proof system for deriving program properties, like Hoare logic. In matching logic, program properties are specified as reduction rules \( \varphi \Rightarrow \varphi' \) between patterns, abstractly capturing the idea of reachability in the corresponding transition system: program configuration \( \gamma \) that matches \( \varphi \) reduces in zero, one or more steps to a configuration \( \gamma' \) that matches \( \varphi' \). Patterns are 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. Unlike in Hoare logic, the proof rules of matching logic are all language-independent, taking the given operational semantics as a set of axiom reduction rules. The key proof rule of matching logic is Circularity, which is meant to language-independently capture the various circular behaviors that appear in languages, due to loops, recursion, etc.

\[
\begin{align*}
\mathcal{A} &\vdash \varphi \Rightarrow^+ \varphi'' \\
\mathcal{A} \cup [\varphi \Rightarrow \varphi'] &\vdash \varphi' \\
\mathcal{A} &\vdash \varphi \Rightarrow \varphi'
\end{align*}
\]

\( \mathcal{A} \) initially contains the operational semantics rules. Circularity adds new reductions 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 has a clear pragmatic advantage over Hoare logic: it eliminates the need for an additional semantics of the same language only to reason about programs, and implicitly eliminates the need for non-trivial and tedious correctness proofs. The soundness of matching logic has already been shown in [16]. Its practicality and usability have been demonstrated through the MatchC automatic program verifier for a C fragment [15], which is a faithful implementation of the matching logic proof system. What is missing is a formal treatment of the completeness of matching logic. 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. A technique to mechanically translate Hoare logic proof derivations into equivalent matching logic proof derivations is presented and proved correct. The generated matching logic 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, where any Hoare logic proof derivation can be mimicked using the matching logic 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 capabilities, without going through its Hoare 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 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 (proved generically, for all languages).

Section 2 recalls operational semantics and Hoare logic, and Section 3 matching logic. Section 4 illustrates the differences between Hoare logic and matching logic. 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. Fig. 1 shows its syntax, an operational semantics based on evaluation contexts, and a Hoare logic for it. IMP has only integer expressions, which can also be used as conditions of if and while (zero means false and any non-zero integer 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 \( \langle \text{code}, \sigma \rangle \), where \text{code} is a program fragment and \( \sigma \) is a state term mapping program variables into integers. As usual, appropriate definitions of the domains of integers (including arithmetic operations \( i_1 \text{op} \_ \text{Int} \_ i_2 \), etc.) and of maps (including lookup \( \sigma(x) \) and update \( \sigma[x \leftarrow i] \) operations) are assumed. IMP’s operational semantics has seven reduction rule schemas between program configurations, which make use of first-order variables: \( \sigma \) is a variable of sort \text{State}; \( x \) is a variable of sort \text{PVar}; \( i, i_1, i_2 \) are variables of sort \text{Int}; \( e \) is a variable of sort \text{Exp}; \( s, s_1, s_2 \) are variables of sort \text{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 \text{op} is
We also recall some mathematical notions and notations, although we generally assume with such rules in general (taking them as axioms), and is agnostic to the particular evaluation contexts [7], we also have the chemical abstract machine [2] and K

\[
\Sigma \text{ semantics have been defined as initial }
\]

extend to algebraic specifications. Many mathematical structures needed for language

Σ-algebras: boolean algebras, natural/integer/rational numbers, lists, sets, bags (or multisets), maps (e.g., IMP’s states), trees, queues, stacks, etc. We refer the reader to the CASL [11] and Maude [4] manuals for examples.

**IMP language syntax**

\[
PVar ::= \text{ program variables}
\]
\[
Exp ::= PVar \mid Int \mid Exp \text{ op } Exp
\]
\[
Stmt ::= \text{ skip} \mid PVar \text{ := } Exp \mid Stmt;Stmt \mid \text{ if}(Exp)Stmt \text{ else } Stmt \mid \text{ while}(Exp)Stmt
\]

**IMP evaluation contexts syntax**

\[
Context ::= \square
\mid \langle Context, State \rangle
\mid Context \text{ op } Exp \mid Exp \text{ op } Context
\mid PVar := Context \mid Context,Stmt
\mid \text{ if}(Context)Stmt \text{ else } Stmt
\]

**IMP operational semantics**

\[
\text{lookup } (C, \sigma)[x] \Rightarrow (C, \sigma)[r(x)]
\]
\[
\text{op } i_1 \text{ op } i_2 \Rightarrow i_1 \text{ op } i_2
\]
\[
\text{asgn } (C, \sigma)[x := i] \Rightarrow (C, \sigma[x \leftarrow i])[\text{skip}]
\]
\[
\text{seq } \text{skip}; s_2 \Rightarrow s_2
\]
\[
\text{cond}_1 \text{ if}(i)_1\text{ else } s_2 \Rightarrow s_1 \text{ if } i \neq 0
\]
\[
\text{cond}_2 \text{ if}(0)_1\text{ else } s_2 \Rightarrow s_2
\]
\[
\text{while } (e)s \Rightarrow \text{if}(e)s;\text{while}(e)s \text{ else } \text{skip}
\]

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

In fact \((C, \sigma)[i_1 \text{ op } i_2] \Rightarrow (C, \sigma)[i_1 \text{ op } i_2]\). The code context meta-variable \(C\) allows us to instantiate a schema into reduction rules, one for each redex of each code fragment.

We can therefore regard the operational semantics of IMP above as a (recursively enumerable) set of reduction rules of the form “\(l \Rightarrow r \text{ if } b\)”, where \(l \text{ and } r\) 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 [7], we also have the chemical abstract machine [2] and \(\Sigma\) [14].

Large languages have been given semantics with only rules of the form “\(l \Rightarrow r \text{ if } b\)”, including \(C\) [6] (defined in \(\Sigma\) with more than 1200 such rules). Matching logic works with such rules in general (taking them as axioms), and is agnostic to the particular operational 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 \(\Sigma\), we let \(T_\Sigma\) denote the initial \(\Sigma\)-algebra of ground terms (i.e., terms without variables) and let \(T_\Sigma(\text{Var})\) denote the free \(\Sigma\)-algebra of terms with variables in \(\text{Var}\). \(T_{\Sigma,\text{st}}(\text{Var})\) is the set of \(\Sigma\)-terms of sort \(s\). Maps \(\rho : \text{Var} \rightarrow T\) with \(T\) a \(\Sigma\)-algebra extend uniquely to morphisms of \(\Sigma\)-algebras \(\rho : T_\Sigma(\text{Var}) \rightarrow T\). These notions extend to algebraic specifications. Many mathematical structures needed for language semantics have been defined as initial \(\Sigma\)-algebras: boolean algebras, natural/integer/rational numbers, lists, sets, bags (or multisets), maps (e.g., 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 $\Sigma$, associated to some desired configuration syntax, with distinguished sorts $\text{Cgf}$ and $\text{Bool}$, (2) a sort-wise infinite set of variables $\text{Var}$, and (3) a $\Sigma$-algebra $T$, the configuration model, which may but needs not necessarily be the initial or free $\Sigma$-algebra. As usual, $T_{\Sigma,\text{Cgf}}$ denotes the elements of $T$ of sort $\text{Cgf}$, which we call configurations. Let $S$ (from “semantics”) be a set of reduction rules “$l \Rightarrow r$ if $b$” like above, where $l, r \in T_{\Sigma,\text{Cgf}}(\text{Var})$ and $b \in T_{\Sigma,\text{Bool}}(\text{Var})$.

**Definition 1.** $S$ yields a transition system $(T, \Rightarrow^S)$, where $\gamma \Rightarrow^S \gamma'$ for $\gamma, \gamma' \in T_{\text{Cgf}}$ iff there is a “$l \Rightarrow r$ if $b$” in $S$ and a $\rho : T \rightarrow T$ with $\rho(l) = \gamma, \rho(r) = \gamma'$ and $\rho(b)$ holds. $(T, \Rightarrow^S)$ is a conventional transition system, i.e. a set with a binary relation on it (in fact, $\Rightarrow^S \subseteq T_{\text{Cgf}} \times T_{\text{Cgf}}$), and captures the operational behaviors of the language defined by $S$.

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 $\{ \psi \} \text{code} \{ \psi' \}$ is semantically valid, written $\models \{ \psi \} \text{code} \{ \psi' \}$, if and only if $\sigma \models \psi$ and $\langle \text{code}, \sigma \rangle \Rightarrow^T \langle \text{skip}, \sigma' \rangle$. The Hoare logic proof system in Fig. 1 is sound if and only if $\Gamma \models \{ \psi \} \text{code} \{ \psi' \}$ implies $\models \{ \psi \} \text{code} \{ \psi' \}$.

In Definition 2, we tacitly identified the ground configurations $\langle \text{code}, \sigma \rangle$ and $\langle \text{skip}, \sigma' \rangle$ with their (unique) interpretation in the configuration model $T$. First-order logic (FOL) validity, both in Definition 2 and in the $\text{HL-csq}$ in Fig. 1, is relative to $T$. 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

This section recalls matching logic [13, 16]. In matching logic, patterns specify configurations and reduction rules specify operational transitions or program properties. A language-independent proof system takes a set of reduction rules (operational semantics) as axioms and derives new reduction 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 $\langle \text{code}, \sigma \rangle$ with code a fragment of program and $\sigma$ a State.

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

**Definition 3.** [13] A matching logic formula, or a pattern, is a first-order logic (FOL) formula which allows terms in $T_{\Sigma,\text{Cgf}}(\text{Var})$, called basic patterns, as predicates. We define the satisfaction $(\gamma, \rho) \models \varphi$ over configurations $\gamma \in T_{\text{Cgf}}$, valuations $\rho : \text{Var} \rightarrow T$ and patterns $\varphi$ as follows (among the FOL constructs, we only show 3):

\[(\gamma, \rho) \models \exists X \varphi \text{ iff } (\gamma, \rho') \models \varphi \text{ for some } \rho' : \text{Var} \rightarrow T \text{ with } \rho'(y) = \rho(y) \text{ for all } y \in \text{Var}\setminus X\]

\[(\gamma, \rho) \models \pi \text{ iff } \gamma = \rho(\pi) \text{, where } \pi \in T_{\Sigma,\text{Cgf}}(\text{Var})\]

We write $\models \varphi$ when $(\gamma, \rho) \models \varphi$ for all $\gamma \in T_{\text{Cgf}}$ and all $\rho : \text{Var} \rightarrow T$. 

---

5
A basic pattern \( \pi \) is satisfied by all the configurations \( \gamma \) that match it; the \( \rho \) in \( (\gamma, \rho) \models \pi \) can be thought of as the “witness” of the matching, and can be further constrained in a pattern. If \( \text{SUM} \) is the IMP code “\( s:=0; \text{while}(n>0)(s:=s+n; n:=n-1) \)” e.g., then \( \exists s ((\text{SUM}, (s \mapsto s, n \mapsto n)) \land n \geq_\text{Int} 0) \) is a pattern that matches the configurations with code \( \text{SUM} \) and state binding program variables \( s,n \) with \( n \geq_\text{Int} 0 \). Note that we use typewriter for program variables in \( P\text{Var} \) and italic for mathematical variables in \( \text{Var} \). Pattern reasoning reduces to FOL reasoning in the configuration model \( \mathcal{T} \) \cite{16}.

**Definition 4.** A (matching logic) reduction rule is a pair \( \varphi \Rightarrow \varphi' \), where \( \varphi \), called the left-hand side (LHS), and \( \varphi' \), called the right-hand side (RHS), are matching logic patterns (which can have free variables). A reduction system \( S \) induces a transition system \((\mathcal{T}, \Rightarrow_S^\mathcal{T})\) on the configuration model: \( \gamma \Rightarrow_S^\mathcal{T} \gamma' \) for \( \gamma, \gamma' \in \mathcal{T}_{\mathcal{C}_S} \) iff there is a \( \varphi \Rightarrow \varphi' \) in \( S \) and a \( \rho : \text{Var} \rightarrow \mathcal{T} \) with \( (\gamma, \rho) \models \varphi \) and \( (\gamma', \rho) \models \varphi' \). Configuration \( \gamma \in \mathcal{T}_{\mathcal{C}_S} \) terminates in \((\mathcal{T}, \Rightarrow_S^\mathcal{T})\) iff there is no infinite \( \Rightarrow_S^\mathcal{T} \)-sequence starting with \( \gamma \). A rule \( \varphi \Rightarrow \varphi' \) is well-defined iff for any \( \gamma \in \mathcal{T}_{\mathcal{C}_S} \) and \( \rho : \text{Var} \rightarrow \mathcal{T} \) with \( (\gamma, \rho) \models \varphi \), there is a \( \gamma' \in \mathcal{T}_{\mathcal{C}_S} \) with \( (\gamma', \rho) \models \varphi' \). Reduction system \( S \) is well-defined iff each rule is well-defined, and is deterministic iff so is \((\mathcal{T}, \Rightarrow_S^\mathcal{T})\).

Operational semantics defined with rules “\( l \Rightarrow r \) if \( b \)”, like those in Section 2, are particular well-defined reduction systems with rules of the form \( l \land b \Rightarrow r \) (see \cite{16}).

Matching logic reduction rules can also specify program properties. For our \( \text{SUM} \) above, \( \exists s ((\text{SUM}, (s \mapsto s, n \mapsto n)) \land n \geq_\text{Int} 0) \Rightarrow (\text{skip}, (s \mapsto n +_\text{Int} (n +_\text{Int} 1)/_\text{Int} 2, n \mapsto 0)) \) specifies the property of \( \text{SUM} \). Unlike Hoare triples, which only specify properties about the final states of programs, reduction rules can also specify properties about intermediate states. Hoare triples correspond to reduction rules whose basic pattern in the RHS holds the code \( \text{skip} \), like the one above. Semantic validity in matching logic captures the same intuition of partial correctness as Hoare logic, but in more general terms of reachability.

**Definition 5.** Let \( S \) be a reduction system and \( \varphi \Rightarrow \varphi' \) a reduction rule. We define \( S \models \varphi \Rightarrow \varphi' \) iff for all \( \gamma \in \mathcal{T}_{\mathcal{C}_S} \) such that \( \gamma \) terminates in \((\mathcal{T}, \Rightarrow_S^\mathcal{T})\) and for all \( \rho : \text{Var} \rightarrow \mathcal{T} \) such that \( (\gamma, \rho) \models \varphi \), there exists some \( \gamma' \in \mathcal{T}_{\mathcal{C}_S} \) such that \( \gamma \Rightarrow_S^\mathcal{T} \gamma' \) and \( (\gamma', \rho) \models \varphi' \).

If \( \varphi' \) holds the empty code \( \text{skip} \), then so does \( \gamma' \) in the definition above, and, in the case of IMP, \( \gamma' \) is unique and thus we recover the Hoare validity as a special case.

The reduction rule property of \( \text{SUM} \) above is valid, although the proof is tedious, involving low-level IMP transition system details and induction. Instead, matching logic gives us an abstract proof system for deriving such reduction rules, which avoids the transition system. Fig. 3 shows the language-independent matching logic proof system. Initially, \( \mathcal{A} \) contains the operational semantics of the target language. Reflexivity, Axiom, Substitution, and Transitivity have an operational nature and are needed to (symbolically) execute reduction systems. 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 \( \mathcal{A} \Rightarrow \varphi \Rightarrow \varphi' \) whenever we can derive \( \varphi \Rightarrow \varphi' \) by starting with one or more reduction 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, \( \varphi \Rightarrow \varphi' \). The first step can for example be an operational loop unrolling step in the case of loops, or a function invocation step in the case of recursive functions, etc.
Rules of operational nature

Reflexivity:

\[ A \vdash \phi \Rightarrow \phi \]

Axiom:

\[ \varphi \Rightarrow \varphi' \in A \]
\[ A \vdash \phi \Rightarrow \phi' \]

Substitution:

\[ A \vdash \phi \Rightarrow \phi' \]
\[ \theta : \text{Var} \rightarrow T_F(\text{Var}) \]
\[ A \vdash \theta(\varphi) \Rightarrow \theta(\varphi') \]

Transitivity:

\[ A \vdash \phi_1 \Rightarrow \phi_2 \]
\[ A \vdash \phi_2 \Rightarrow \phi_3 \]
\[ A \vdash \phi_1 \Rightarrow \phi_3 \]

Rules of deductive nature

Case Analysis:

\[ A \vdash \phi_1 \Rightarrow \phi \]
\[ A \vdash \phi_2 \Rightarrow \phi \]
\[ A \vdash \phi_1 \lor \phi_2 \Rightarrow \phi \]

Logic Framing:

\[ \varphi' \in A \]
\[ A \vdash \varphi \Rightarrow \varphi' \]
\[ \psi \text{ is a (patternless) FOL formula} \]
\[ A \vdash \varphi \land \psi \Rightarrow \varphi' \land \psi \]

Consequence:

\[ \models \varphi_1 \Rightarrow \varphi_1' \]
\[ A \vdash \varphi_1 \Rightarrow \varphi_2 \]
\[ \models \varphi_2 \Rightarrow \varphi_2' \]
\[ A \vdash \varphi_1 \Rightarrow \varphi_2' \]

Abstraction:

\[ A \vdash \varphi \Rightarrow \varphi' \]
\[ \psi \text{ is a (patternless) FOL formula} \]
\[ X \cap \text{FreeVars}(\varphi') = \emptyset \]
\[ A \vdash \exists X \varphi \Rightarrow \varphi' \]

Rule for circular behavior

Circularity:

\[ A \vdash \varphi \Rightarrow \varphi' \]
\[ A \vdash \varphi \Rightarrow \varphi' \]
\[ A \cup \{ \varphi \Rightarrow \varphi' \} \vdash \varphi' \Rightarrow \varphi' \]

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

Theorem 1. (partial correctness) [10] Let S be a well-defined and deterministic matching logic reduction system (typically corresponding to an operational semantics), and let \( S \vdash \varphi \Rightarrow \varphi' \) be a sequent derived with the proof system in Fig. 2. Then S \( \models \varphi \Rightarrow \varphi' \).

4 Hoare Logic versus Matching Logic

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

Consider again the SUM program "\( s:=0; \text{while}(n>0)(s:=s+n; \ n:=n-1) \)" in IMP. The main property of SUM can be specified as the following Hoare triple:

\[ \{ n = \text{oldn} \land n \geq 0 \} \text{SUM}\{ s = \text{oldn}\times(\text{oldn}+1)/2 \land n = 0 \} \]

The oldn variable is needed to remember the initial value of n. Let us derive this Hoare triple using the Hoare logic proof system in Fig. 1. Let LOOP be the actual loop of SUM, namely "\( \text{while}(n>0)(s:=s+n; \ n:=n-1) \)". Let \( \psi_{\text{inv}} \) be the formula

\[ s = (\text{oldn}-n)\times(\text{oldn}+n+1)/2 \land n \geq 0 \]

We can derive our original Hoare triple by first deriving the triples

\[ \{ n = \text{oldn} \land n \geq 0 \} \text{SUM}\{ s = \text{oldn}\times(\text{oldn}+1)/2 \land n = 0 \} \]

and then using the proof rule HL-seq in Fig. 1. To keep the proof small, we skip the FOL reasoning steps (within the state model) and thus the applications of HL-csq. The first triple follows by HL-asgn. The second follows by HL-while, after first deriving

\[ \{ \psi_{\text{inv}} \land n > 0 \} \text{LOOP}\{ s = \text{oldn}\times(\text{oldn}+1)/2 \land n = 0 \} \]

by using two instances of the HL-asgn rule and one instance of HL-seq.

Before we discuss the matching logic proof derivation, let us recall some important facts about Hoare logic. 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 $\text{oldn}$ 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 $+_{\text{int}}$ operation that the integer domain provides. These simplifying assumptions make proofs like above 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.

Let us now show how to use the proof system in Fig. 2 to derive the matching logic reduction rule specifying the property of $\text{SUM}$, already discussed in Section 11 namely

$$\exists s ((\text{SUM}, (s \mapsto s, n \mapsto n)) \land n \geq_{\text{Int}} 0) \Rightarrow (\text{skip}, (s \mapsto n *_{\text{Int}} (n +_{\text{Int}} 1)/_{\text{Int}} 2, n \mapsto 0))$$

The “$\exists$” quantifier is optional. Let us drop it and let us name the resulting rule $\mu_{\text{SUM}}^{1} = (\varphi_{\text{LHS}} \Rightarrow \varphi_{\text{RHS}})$. The original rule follows from $\mu_{\text{SUM}}^{1}$ by Abstraction. Let $S_{\text{IMP}}$ be the operational semantics of IMP in Fig. 1 and let $\varphi_{\text{inv}}$ be the pattern

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

We derive $S_{\text{IMP}} \vdash \mu_{\text{SUM}}^{1}$ 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 $\text{asgn}$ (Fig. 1) within the $\text{SUM}$ context followed by Substitution with $\theta(\sigma) = (s \mapsto s, n \mapsto n), \theta(s) = s$ and $\theta(i) = 0$ followed by Logic Framing with $n \geq_{\text{Int}} 0$, we derive $\varphi_{\text{LHS}} \Rightarrow (\text{skip}; \text{LOOP}, (s \mapsto 0, n \mapsto n)) \land n \geq_{\text{Int}} 0$. This “operational” sequence of Axiom, Substitution and Logic Framing is quite common; we abbreviate it ASLF. Further, by ASLF with $\text{seq}$ and Transitivity, we derive $\varphi_{\text{LHS}} \Rightarrow (\langle \text{LOOP}, (s \mapsto s, n \mapsto n) \rangle \land n \geq_{\text{Int}} 0). S_{\text{IMP}} \vdash \mu_{1}$ now follows by Consequence. We derive $S_{\text{IMP}} \vdash \mu_{2}$ by Circularity with $S_{\text{IMP}} \vdash \exists n' \varphi_{\text{inv}} \Rightarrow \varphi_{\text{LHS}}$ and $S_{\text{IMP}} \cup \{\mu_{2}\} \vdash \varphi_{\text{LHS}} \Rightarrow \varphi_{\text{RHS}}$, where $\varphi_{\text{LHS}}$ is the formula obtained from $\varphi_{\text{inv}}$ replacing its code with “if $(n>0) \ (s := s+n; \ n := n-1; \ \text{LOOP}) \ \text{else} \ \text{skip}$. ASLF (while) followed by Abstraction derive $S_{\text{IMP}} \vdash \exists n' \varphi_{\text{inv}} \Rightarrow \varphi_{\text{LHS}}$. For the other, we use Case Analysis with $\varphi_{\text{LHS}} \land n \leq_{\text{Int}} 0$ and $\varphi_{\text{LHS}} \land n >_{\text{Int}} 0$. ASLF ($\text{lookup}_{n}, \text{op}_{-}, \text{cond}_{2}$) together with some Transitivity and Consequence steps derive $S_{\text{IMP}} \cup \{\mu_{2}\} \vdash \varphi_{\text{LHS}} \land n' \leq_{\text{Int}} 0 \Rightarrow \varphi_{\text{RHS}}$ ($\mu_{2}$ is not needed in this derivation). Similarly, ASLF ($\text{lookup}_{n}, \text{op}_{-}, \text{cond}_{1}, \text{lookup}_{n}, \text{op}_{-}, \text{asgn}, \text{seq}, \text{lookup}_{n}, \text{op}_{-}, \text{asgn}, \text{seq}$, and $\mu_{2}$) together with Transitivity and Consequence steps derive $S_{\text{IMP}} \cup \{\mu_{2}\} \vdash \varphi_{\text{LHS}} \land n' >_{\text{Int}} 0 \Rightarrow \varphi_{\text{RHS}}$. This time $\mu_{2}$ 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

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

The circularity $\mu_{2}$ 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 $n' -_{\text{Int}} 1$).

The matching logic proof above may seem low-level when compared to the Hoare logic proof. However, note that it is quite mechanical, the only interesting part being to provide the invariant ($\varphi_{\text{inv}}$), same like in the Hoare logic proof. 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 \textit{if}, a Case Analysis is applied. Our current MatchC implementation can prove it automatically, as well as much more complex programs \cite{15,16}. 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 matching logic’s reduction rules are more expressive than the Hoare triples, since they can specify reachable configurations which are not necessarily final. For example, the rule
\begin{align*}
&\langle \text{SUM}, (s \mapsto s, n \mapsto n) \rangle \land n > \text{Int}_0 \Rightarrow \langle \text{LOOP}, (s \mapsto n, n \mapsto n - \text{Int}_1) \rangle
\end{align*}
is also derivable and states that if the value $n$ of $n$ is strictly positive, then the loop is taken once and, when the loop is reached again, $s$ is $n$ and $n$ is $n - \text{Int}_1$.

5 Translating Hoare Logic Proofs into Matching Logic Proofs

Here we show how proof derivations using the IMP-specific Hoare logic proof system in Fig. 1 are mechanically translated into proof derivations using the language-independent matching logic proof system in Fig. 2 with IMP’s operational semantics in Fig. 1 as axioms. Moreover, the sizes of the two proof derivations are within a linear factor.

5.1 The Translation

Without restricting the generality, we make the following simplifying assumptions about the Hoare triples $\{\psi\} \text{code} \{\psi'\}$ that appear in the Hoare logic proof derivation that we translate into a matching logic proof: (1) the variables appearing in \text{code} belong to an arbitrary but fixed finite set $X \subset PVar$; and (2) the additional variables appearing in $\psi$ and $\psi'$ but not in \text{code} belong to an arbitrary but fixed finite set $Y \subset PVar$ such that $X \cap Y = \emptyset$. In other words, we fix the finite disjoint sets $X, Y \subset PVar$, 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 $X$ to be the union of all the variables appearing in the program about which we want to reason and $Y$ 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.

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

\begin{align*}
H2M(\{\psi\} & \text{code} \{\psi'\} ) & \overset{\text{def}}{=} \exists X (\langle \text{code}, \sigma_X \rangle \land \psi_{X,Y}) \Rightarrow \exists X (\langle \text{skip}, \sigma_X \rangle \land \psi'_{X,Y})
\end{align*}

where:

1. $X, Y \subset \text{Var}$ (written using italic font) are finite sets of variables corresponding to the sets $X, Y \subset PVar$ fixed above, one variable $x$ or $y$ in $\text{Var}$ (written using italic font) for each variable $x$ or $y$ in $PVar$ (written using typewriter font);
2. $\sigma_X$ is the state binding each $x \in X$ to its corresponding $x \in X$; and
3. \(\psi_{X,Y} \) and \(\psi'_{X,Y} \) are \(\psi \) and respectively \(\psi' \) with \(x \in X \) or \(y \in Y \) replaced by its corresponding \(x \in X \) or \(y \in Y \), respectively, and each expression construct \(\text{op} \) replaced by its mathematical correspondent \(\text{op}_{\text{Int}} \).

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](http://fsl.cs.uiuc.edu/index.php/Special:MatchCOnline).

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

\[
[\{n = \text{old}n \land n \geq 0\} \text{SUM} \{s = \text{old}n \ast (\text{old}n + 1) / 2 \land n = 0\}]
\]

from Section 4 is translated into the following matching logic reduction rule:

\[
\exists s, n ((\text{SUM}, (s \mapsto s, n \mapsto n)) \land n = \text{old}n \land n \geq_{\text{Int}} 0) \\
\Rightarrow \exists s, n ((\text{skip}, (s \mapsto s, n \mapsto n)) \land s = \text{old}n \ast_{\text{Int}} (\text{old}n +_{\text{Int}} 1) /_{\text{Int}} 2 \land n = 0)
\]

Not surprisingly, we can use the matching logic proof system in Fig. 2 to prove this reduction rule equivalent to the one that we gave for \(\text{SUM} \) in Section 4. Indeed, using FOL reasoning and Consequence we can show the above equivalent to

\[
\exists s ((\text{SUM}, (s \mapsto s, n \mapsto \text{old}n)) \land \text{old}n \geq_{\text{Int}} 0) \Rightarrow (\text{skip}, (s \mapsto \text{old}n \ast_{\text{Int}} (\text{old}n +_{\text{Int}} 1) /_{\text{Int}} 2, n \mapsto 0))
\]

which, by Substitution \((n \mapsto \text{old}n)\), is equivalent to the reduction rule in Section 4.

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

\[
H2M([\text{true}] n := 4 \ast n + 3 \{\exists z (n = 2 \ast z + 1)\})
\]

is the reduction rule

\[
\exists s, n ((n := 4 \ast n + 3, (s \mapsto s, n \mapsto n)) \land \text{true}) \\
\Rightarrow \exists s, n ((\text{skip}, (s \mapsto s, n \mapsto n)) \land \exists z (n = 2 \ast z +_{\text{Int}} 1))
\]

Using FOL reasoning and Consequence, this rule can be shown equivalent to

\[
\exists s, n (n := 4 \ast n + 3, (s \mapsto s, n \mapsto n)) \Rightarrow \exists s, z (\text{skip}, (s \mapsto s, n \mapsto 2 \ast z +_{\text{Int}} 1))
\]

5.2 Helping Lemmas

The following holds for matching logic in general:

**Lemma 1.** If \(S \vdash \varphi \Rightarrow \varphi' \) is derivable then \(S \vdash \exists X \varphi \Rightarrow \exists X \varphi' \) is also derivable.

**Proof.** We have \(\varphi \Rightarrow \exists X \varphi' \). By Consequence, we derive \(S \vdash \varphi \Rightarrow \exists X \varphi' \). Since \(X \cap \text{FreeVars}(\exists X \varphi') = 0\), by Abstraction we get that \(S \vdash \exists X \varphi \Rightarrow \exists X \varphi' \) is also derivable.

Symbolical evaluation of IMP expressions is actually derivable in matching logic:

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

\[
S_{\text{IMP}} \vdash (C, \sigma)[e] \Rightarrow (C, \sigma)[\sigma(e)]
\]

where \(\sigma(e) \) replaces each \(x \in \text{PVar} \) in \(e \) by \(\sigma(x) \) (i.e., a term of sort \(\text{Int} \)) and each operation symbol \(\text{op} \) by its mathematical correspondent in the \(\text{Int} \) domain, \(\text{op}_{\text{Int}} \).
Proof. By induction on the structure of $e$. If $e$ is a variable $x \in PVar$, then the result follows by Axiom with $\text{lookup}$ in Fig. 1. If $e$ is of the form $e_1 \text{ op } e_2$, then let $C_1, C_2$ be the contexts obtained from $C$ by replacing $\Box$ with "$\Box \text{ op } e_2$" and respectively "$\sigma(e_1) \text{ op } \Box$".

Then, by the induction hypothesis, the following are derivable

$S_{\text{IMP}} \vdash \langle C_1, \sigma \rangle[e_1] \Rightarrow \langle C_1, \sigma \rangle[\sigma(e_1)]$

$S_{\text{IMP}} \vdash \langle C_2, \sigma \rangle[e_2] \Rightarrow \langle C_2, \sigma \rangle[\sigma(e_2)]$

We also have the following pattern identities

$\langle C, \sigma \rangle[e] = \langle C_1, \sigma \rangle[e_1]$

$\langle C_1, \sigma \rangle[\sigma(e_1)] = \langle C_2, \sigma \rangle[e_2]$

$\langle C_2, \sigma \rangle[\sigma(e_2)] = \langle C, \sigma \rangle[\sigma(e_1) \text{ op } \sigma(e_2)]$

Thus, by Transitivity, we derive $S_{\text{IMP}} \vdash \langle C, \sigma \rangle[e] \Rightarrow \langle C, \sigma \rangle[\sigma(e_1) \text{ op } \sigma(e_2)]$, and then the result follows by Axiom with $\text{op}$ and by noticing that $\sigma(e) = \sigma(e_1) \text{ op } \sigma(e_2)$.

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

Proof. (sketch) Let $\text{append}(\mathcal{A}, s)$ be the set of rules obtained from $\mathcal{A}$ by replacing each rule $\varphi_i \Rightarrow \varphi_i', \in \mathcal{A} \setminus S_{\text{IMP}}$ by the rule $\text{append}(\varphi_i, s) \Rightarrow \text{append}(\varphi_i, s)$, that is

$\text{append}(\mathcal{A}, s) = (\mathcal{A} \setminus S_{\text{IMP}}) \cup \{ \text{append}(\varphi_i, s) \Rightarrow \text{append}(\varphi_i, s) \mid \varphi_i \Rightarrow \varphi_i', \in \mathcal{A} \setminus S_{\text{IMP}} \}$

Let $\mathcal{P}$ be a proof tree deriving $S_{\text{IMP}} \vdash \varphi \Rightarrow \varphi'$. We prove the more general result that for each sequent $\mathcal{A} \vdash \varphi_i \Rightarrow \varphi_i', \in \mathcal{P}$, we can also derive the sequent $\text{append}(\mathcal{A}, s) \vdash \text{append}(\varphi_i, s) \Rightarrow \text{append}(\varphi_i, s)$. The lemma follows as particular case. The proof goes by induction on the structure of $\mathcal{P}$. If the last step is $\text{Reflexivity}$, the result trivially holds. If the last step is one of $\text{Substitution}$, $\text{Transitivity}$, $\text{Case Analysis}$, $\text{Logic Framing}$, $\text{Consequence}$, $\text{Abstraction}$ or $\text{Circularity}$, then the result holds by applying the induction hypothesis, and by noticing that since $s$ does not have any logical variables, then $\text{append}(\theta(\varphi), s) = \theta(\text{append}(\varphi, s))$ ($\text{Substitution}$), $\models \varphi_i \Rightarrow \varphi_i'$ iff $\models \text{append}(\varphi_i, s) \Rightarrow \text{append}(\varphi_i, s)$ ($\text{Consequence}$) and $\text{FreeVars}(\text{append}(\varphi, s)) = \text{FreeVars}(\varphi)$ ($\text{Abstraction}$). If the last step is Axiom with a rule in $\mathcal{A} \setminus S_{\text{IMP}}$, again the result trivially holds. If the last step is Axiom with a rule in $S_{\text{IMP}}$, then the redex always goes to the left of ";". Since none of the reduction rule schemas of IMP mention ";" in the LHS or in the side condition, we can conclude that $\varphi \Rightarrow \varphi' \in S_{\text{IMP}}$ iff $\text{append}(\varphi, s) \Rightarrow \text{append}(\varphi', s) \in S_{\text{IMP}}$.

5.3 The Main Result

Theorem 2. Let $S_{\text{IMP}}$ be the operational semantics of IMP in Fig. 7 regarded as a matching logic reduction system, and let $\langle \psi \rangle \text{ code } \langle \psi' \rangle$ be derivable with the IMP-specific Hoare logic proof system in Fig. 7 Then $S_{\text{IMP}} \vdash H2M(\langle \psi \rangle \text{ code } \langle \psi' \rangle)$ is derivable with the language-independent matching logic proof system in Fig. 2.

Proof. We prove that for any Hoare logic proof of $\langle \psi \rangle \text{ code } \langle \psi' \rangle$ one can construct a matching logic proof of $S_{\text{IMP}} \vdash H2M(\langle \psi \rangle \text{ code } \langle \psi' \rangle)$. The proof goes by structural induction on the formal proof derived using the Hoare logic proof system in Fig. 7. We consider each proof rule in Fig. 1 and show how corresponding matching logic proofs for the hypotheses can be composed into a matching logic proof for the conclusion.

\[
\begin{align*}
\text{HL-skip} & \quad \frac{}{\langle \psi \rangle \text{ skip } \langle \psi \rangle}
\end{align*}
\]
Reflexivity (Fig. 2), derives \( S_{\text{imp}} \vdash \exists X ((\text{skip}, \sigma_X) \land \psi_{X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{X,Y}). \)

**HL-asgn**

\[
[\psi[e/x]] x := e \psi
\]

We have to derive \( S_{\text{imp}} \vdash \exists X ((x := e, \sigma_X) \land \psi[e/x]_{X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{X,Y}). \)

By using Lemma 2, Logical Framing and Lemma 1, we derive

\[
S_{\text{imp}} \vdash \exists X ((x := e, \sigma_X) \land \psi[e/x]_{X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{X,Y})
\]

Further, by using Axiom with \( \text{asgn} \) in Fig. 1, Substitution and Logical Framing, followed by Lemma 1, we derive

\[
S_{\text{imp}} \vdash \exists X ((x := \sigma_X(e), \sigma_X) \land \psi[e/x]_{X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X(e)) \land \psi[e/x]_{X,Y})
\]

Then, the result follows by Transitivity with the rules above and by Consequence with

\[
\models \exists X ((\text{skip}, \sigma_X(x \leftarrow \sigma_X(e))) \land \psi[e/x]_{X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{X,Y}),
\]

which holds because \( \sigma_X(x \leftarrow \sigma_X(e)) \) and \( \psi[e/x]_{X,Y} \) are nothing but \( \sigma_X \) and respectively \( \psi_{X,Y} \) with \( x \in X \) replaced by \( \sigma_X(e) \).

**HL-seq**

\[
\frac{\{\psi_1\} s_1 \{\psi_2\} \quad \{\psi_2\} s_2 \{\psi_3\}}{\{\psi_1\} s_1; s_2 \{\psi_3\}}
\]

We have to derive \( S_{\text{imp}} \vdash \exists X ((s_1; s_2, \sigma_X) \land \psi_{1,X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{3,X,Y}). \) By the induction hypothesis, the following sequents are derivable

\[
S_{\text{imp}} \vdash \exists X ((s_1, \sigma_X) \land \psi_{1,X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{2,X,Y})
\]

\[
S_{\text{imp}} \vdash \exists X ((s_2, \sigma_X) \land \psi_{2,X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{3,X,Y})
\]

By applying Lemma 3 with the former rule, we derive

\[
S_{\text{imp}} \vdash \exists X ((s_1; s_2, \sigma_X) \land \psi_{1,X,Y}) \Rightarrow \exists X ((\text{skip}; s_2, \sigma_X) \land \psi_{2,X,Y})
\]

Further, Axiom with \( \text{seq} \) (Fig. 1), Substitution and Logical Framing, followed by Lemma 1, imply \( S_{\text{imp}} \vdash \exists X ((s_1; s_2, \sigma_X) \land \psi_{1,X,Y}) \Rightarrow \exists X ((s_2, \sigma_X) \land \psi_{2,X,Y}). \) Then, the result follows by Transitivity with the rules above and the second induction hypothesis.

**HL-cond**

\[
\frac{\{\psi_1 \land e \neq 0\} s_1 \{\psi_2\} \quad \{\psi_1 \land e = 0\} s_2 \{\psi_2\}}{\{\psi_1\} \text{if}(e) s_1 \text{else} s_2 \{\psi_2\}}
\]

We have to derive

\[
S_{\text{imp}} \vdash \exists X ((\text{if}(e) s_1 \text{else} s_2, \sigma_X) \land \psi_{1,X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{2,X,Y})
\]

By the induction hypothesis, the following sequents are derivable

\[
S_{\text{imp}} \vdash \exists X ((s_1, \sigma_X) \land (\psi_1 \land e \neq 0)_{X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{2,X,Y})
\]

\[
S_{\text{imp}} \vdash \exists X ((s_2, \sigma_X) \land (\psi_1 \land e = 0)_{X,Y}) \Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_{2,X,Y})
\]

By using Lemma 2, Logical Framing, and Lemma 1, we derive

\[
S_{\text{imp}} \vdash \exists X ((\text{if}(e) s_1 \text{else} s_2, \sigma_X) \land \psi_{1,X,Y})
\]

\[
\Rightarrow \exists X ((\text{if}(\sigma_X(e)) s_1 \text{else} s_2, \sigma_X) \land \psi_{1,X,Y})
\]

By using Axiom with \( \text{cond}_1 \) and \( \text{cond}_2 \) in Fig. 1, each followed by Substitution, Logical Framing and by Lemma 1, we also derive

\[
S_{\text{imp}} \vdash \exists X ((\text{if}(\sigma_X(e)) s_1 \text{else} s_2, \sigma_X) \land (\psi_1 \land e \neq 0)_{X,Y})
\]

\[
\Rightarrow \exists X ((s_1, \sigma_X) \land (\psi_1 \land e \neq 0)_{X,Y})
\]

\[
S_{\text{imp}} \vdash \exists X ((\text{if}(\sigma_X(e)) s_1 \text{else} s_2, \sigma_X) \land (\psi_1 \land e = 0)_{X,Y})
\]

\[
\Rightarrow \exists X ((s_2, \sigma_X) \land (\psi_1 \land e = 0)_{X,Y})
\]

Further, by Transitivity with the rules above and the induction hypotheses, we derive
We apply Case Analysis with \( \sigma \). We derive Axiom with \( \text{skip} \) and Axiom with \( \text{cond} \). Then the result follows by Case Analysis, Consequence and Transitivity.

Then the result follows by Case Analysis, Consequence and Transitivity.

\[
\text{HL-while}\quad \vdash \{\psi \land e \neq 0\} s \{\psi\}
\]

Let \( \mu \) be the matching logic rule that we have to derive, namely

\[
\mathcal{S}_{\text{IMP}} \vdash \exists X ((\text{if}(\sigma_X(e)) s \text{ else } s_2, \sigma_X) \land (\psi_1 \land e \neq 0)_{X,Y})
\]

\[
\Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_2)_{X,Y}
\]

\[
\mathcal{S}_{\text{IMP}} \vdash \exists X ((\text{if}(\sigma_X(e)) s \text{ else } s_2, \sigma_X) \land (\psi_1 \land e \neq 0)_{X,Y})
\]

\[
\Rightarrow \exists X ((\text{skip}, \sigma_X) \land \psi_2)_{X,Y}
\]

Then the result follows by Case Analysis with \( \text{skip} \). We consider the following syntax for procedures:

\[
\text{Procedure} ::= \text{ProcedureName}() \text{ Stmt} \]

\[
\text{Stmt} ::= \text{ProcedureName}()
\]

5.4 Adding Recursion

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

\[
\text{ProcedureName} ::= \text{proc} | \ldots
\]

\[
\text{Procedure} ::= \text{ProcedureName}() \text{ Stmt}
\]

\[
\text{Stmt} ::= \text{ProcedureName}()
\]
Our procedures therefore have the syntax “proc() body”, where proc is the name of the procedure and body the body statement. Procedure invocations are statements of the form “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:

\[
\text{call proc()} \Rightarrow \text{body}
\]

where “proc() body” is a procedure

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

\[
\begin{align*}
H \cup \{\psi\} \text{proc()} \{\psi'\} & \vdash \{\psi\} \text{body} \{\psi'\} \\
H & \vdash \{\psi\} \text{proc()} \{\psi'\}
end{align*}
\]

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 \(H\), 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 \(\{\psi\} \text{code} \{\psi'\}\) in Fig. 1 by a sequent “\(H \vdash \{\psi\} \text{code} \{\psi'\}\)”.

**Theorem 3.** Let \(\text{SIMP}\) be the operational semantics of IMP in Fig. 1 extended with the rule call for procedure calls above, and let \(H \vdash \{\psi\} \text{code} \{\psi'\}\) be a sequent derivable with the extended Hoare logic proof system. Then \(\text{SIMP} \cup H2M(H) \vdash H2M(\{\psi\} \text{code} \{\psi'\})\) is derivable with the language-independent matching logic proof system in Fig. 2.

**Proof.** Like in Theorem 2 we prove by structural induction that for any Hoare logic proof of \(H \vdash \{\psi\} \text{code} \{\psi'\}\) one can construct a matching logic proof of \(\text{SIMP} \cup H2M(H) \vdash H2M(\{\psi\} \text{code} \{\psi'\})\), by showing for each Hoare logic proof rule how corresponding matching logic proofs for the hypotheses can be composed into a matching logic proof for the conclusion. The proofs for the (extended) Hoare rules in Fig. 1 are similar to those in Theorem 2 so we only discuss the new Hoare rule for procedure calls:

\[
\begin{align*}
H \cup \{\psi\} \text{proc()} \{\psi'\} & \vdash \{\psi\} \text{body} \{\psi'\} \\
H & \vdash \{\psi\} \text{proc()} \{\psi'\}
end{align*}
\]

Let \(\mu\) be the matching logic reduction rule \(H2M(\{\psi\} \text{proc() } \{\psi'\})\), that is,

\[
\exists X (\{\text{proc()}, \sigma_X \land \psi_{XY}\} \Rightarrow \exists X (\{\text{skip}, \sigma_X \land \psi_{XY}'\}).
\]

The induction hypothesis gives us that the matching logic sequent

\[
\text{SIMP} \cup H2M(H) \cup \{\mu\} \vdash \exists X (\{\text{body}, \sigma_X \land \psi_{XY}\} \Rightarrow \exists X (\{\text{skip}, \sigma_X \land \psi_{XY}'\})
\]

is derivable with the generic proof system in Fig. 2. Using Axiom with call, Logic Framing with \(\psi_{XY}\), and then Lemma 4 we derive (note the \(\Rightarrow^+\), as this derivation does not use Reflexivity):

\[
\text{SIMP} \cup H2M(H) \vdash \exists X (\{\text{proc()}, \sigma_X \land \psi_{XY}\} \Rightarrow^+ \exists X (\{\text{body}, \sigma_X \land \psi_{XY}\})
\]

Circularity with the two rules above now derives \(\text{SIMP} \cup H2M(H) \vdash \mu\).
6 Conclusion

Matching logic provides a sound and language-independent method of reasoning about programs, based solely on the operational semantics of the target programming language [16]. This paper addressed the other important aspect of matching logic deduction, namely its (relative) completeness. A mechanical translation of Hoare logic proof trees into equivalent matching logic proof trees was presented. The size of the generated proofs is linear in the size of the original proofs. 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 for a standard language, by reduction to the relative completeness of Hoare logic, and thus show that matching logic 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.

References