University "Alexandru Ioan Cuza" of Iaşi
Faculty of Computer Science

TECHNICAL REPORT

A Language-Independent Proof System for Mutual Program Equivalence

Ştefan Ciobâcă, Dorel Lucanu,
Vlad Rusu and Grigore Roşu

TR 13-03, December 2013

ISSN 1224-9327

Universitatea "Alexandru Ioan Cuza" Iaşi
Facultatea de Informatică
Str. Berthelot 16, 6600-Iaşi, Romania
Tel. +40-32-201090, email: bibl@infoiasi.ro

A Language-Independent Proof System for Mutual Program Equivalence

Ştefan Ciobâcă[1], Dorel Lucanu[1], Vlad Rusu[2], and Grigore Roşu[1,3]
[1] Universitatea Alexandru Ioan Cuza, Romania
[2] Inria Lille, France
[3] University of Illinois at Urbana-Champaign, USA

Abstract. Two programs or fragments of program are mutually equivalent iff either they both diverge or they end up in similar states. Mutual equivalence is desirable in many contexts, ranging from capturing program equivalence or correctness of program transformations within the same language, to capturing correctness of compilers from one language to another. This paper introduces a language-independent proof system for mutual equivalence. The proof system is parametric in the operational semantics of the two languages and in a state similarity relation.

1 Introduction

Two terminating programs are equivalent if the final states that they reach are similar. Nontermination can be incorporated in equivalence in several ways. Two programs are said to be *mutually equivalent* iff they either both diverge or they both terminate and then the final states that they reach are similar. Mutual equivalence is thus an adequate notion of equivalence for programs written in deterministic sequential languages and is useful, e.g., in compiler verification.

In this paper we formalize the notion of mutual equivalence and propose a logic with a deductive system for stating and proving mutual equivalence of two programs that can belong to two different languages. The deductive system is language-independent, in the sense that it is parametric in the semantics of the two-languages. We prove that the proposed system is sound: when it succeeds it proves the mutual equivalence of the programs given to it as input. The key idea is to use the proof system to build a relation on configurations which is closed under the transition relations given by the corresponding operational semantics. The challenge is how to achieve that generically, where the two languages are given by their formal semantics, without relying on the specifics of any particular language. The representative rule of our proof system is Circularity, which allows us to incrementally postulate synchronization points in the two programs.

We illustrate the proof system on two programs (Fig. 5 on page 10) that both compute the Collatz sequence in two ways: one is written in an imperative language and the other one in a functional language. We prove with our system that they are mutually equivalent without knowing whether they terminate.

Section 2 introduces matching logic and shows how it can be used to give operational semantics to programming languages, on which the rest of the developments build. Section 3 then shows how to aggregate matching logic semantics, a core operation for stating and proving mutual equivalence. Section 4 presents our proof system for mutual equivalence and applies it to the two programs computing the Collatz sequence. Section 5 discusses related work and concludes.

```
Exp_L ::= Var | Int | Exp_L op Exp_L
Stmt ::= Var := Exp_L
       | skip | Stmt ; Stmt
       | if Exp_L then Stmt else Stmt
       | while Exp_L do Stmt
Code ::= Exp_L | Stmt
Cfg_L ::= ⟨Code, Map{Var, Int}⟩
```

```
Exp_R ::= Var | Val | Exp_R op Exp_R
        | Exp_R Exp_R
        | letrec Var Var = Exp_R in Exp_R
        | if Exp_R then Exp_R else Exp_R
        | μ Var . Exp_R
Val  ::= Int | λ Var . Exp_R
Cfg_R ::= ⟨Exp_R⟩
```

Fig. 1. Syntax of the IMP language on the left and of the FUN language on the right. The symbol op ranges over the binary arithmetic operations +, %, /, ==, !=, etc.

2 Matching Logic and Programming Language Semantics

Here we introduce notation used throughout the paper and discuss the recently introduced *matching logic* [16, 15], a language-parametric logic for reasoning about program configurations, and its use in programming language semantics. Matching logic extends FOL with *basic patterns*, which are open terms (i.e., terms with variables) that can be used as basic formulae in the logic. The semantics of a basic pattern as a FOL formula is given by matching.

For example, if program configurations are described as pairs $\langle \mathit{code}\,\texttt{,}\,\mathit{env}\rangle$ of fragments of code *code* and state *env*, the matching logic formula $\langle \texttt{skip,}\,(\texttt{x} \mapsto n, \_)\rangle \wedge n > 0$ matches configurations where there is no more code to execute (the `skip` instruction does nothing) and where the program variable `x` is mapped to a positive integer. Here the underscore ($\_$) denotes an anonymous variable matching the rest of the program state map. We only consider first-order matching logic in this paper, whose signatures are many-sorted first-order signatures with a distinguished sort for configurations. We let Σ and Π range over multi-sorted algebraic signatures and, respectively, over multi-sorted first-order predicate sets.

Definition 1. *A **matching logic (ML) signature** $(\mathit{Cfg}, \Sigma, \Pi)$ is a first-order many-sorted signature (Σ, Π) with a distinguished sort Cfg for **configurations**.*

Example 1. It is well-known that the abstract (context-free) syntax of programming languages can be defined as first-order signatures. Fig. 1 shows the syntax of a simple imperative language IMP and of a simple functional language FUN, that we use as case studies in this paper. We call the respective first-order signatures Σ_L (for IMP—the language on the **l**eft-hand side of the figure) and Σ_R (for FUN—the language on the **r**ight-hand side of the figure). Each non-terminal in the syntax context-free grammars corresponds to a sort and each production to an operation (or function) symbol. When we refer to the corresponding operation symbol of a production, we use the *mixfix* notation. For example, the operation corresponding to IMP's assignment is $\_\texttt{:=}\_ : \texttt{Var} \times \texttt{Exp}_L \to \texttt{Stmt}$, and that corresponding to FUN's λ-abstraction is $\lambda\_.\_ : \texttt{Var} \times \texttt{Exp}_R \to \texttt{Val}$. We assume the integer numbers (sort Int) and variable symbols (sort Var), together with their usual operations and predicates, shared by both IMP and FUN. Sorts $\texttt{Cfg}_L$ and respectively $\texttt{Cfg}_R$ represent the configurations of each programming language, which, for uniformity, we assume constructed with a grouping operation $\langle ... \rangle$.

$\langle x\ ,\ env\rangle \Rightarrow \langle env(x)\ ,\ env\rangle \in \mathcal{A}_{\texttt{IMP}}$
$\langle i_1\ \texttt{op}\ i_2\ ,\ env\rangle \Rightarrow \langle i_1\ \texttt{op}_{Int}\ i_2\ ,\ env\rangle \in \mathcal{A}_{\texttt{IMP}}$
$\langle x\ \texttt{:=}\ i\ ,\ env\rangle \Rightarrow \langle \texttt{skip}\ ,\ env[x \mapsto i]\rangle \in \mathcal{A}_{\texttt{IMP}}$
$\langle \texttt{skip;}s\ ,\ env\rangle \Rightarrow \langle s\ ,\ env\rangle \in \mathcal{A}_{\texttt{IMP}}$
$\langle \texttt{if i then}\ s_1\ \texttt{else}\ s_2\ ,\ env\rangle \wedge i \neq 0 \Rightarrow \langle s_1\ ,\ env\rangle \in \mathcal{A}_{\texttt{IMP}}$
$\langle \texttt{if}\ 0\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ ,\ env\rangle \Rightarrow \langle s_2\ ,\ env\rangle \in \mathcal{A}_{\texttt{IMP}}$
$\langle \texttt{while}\ e\ \texttt{do}\ s\ ,\ env\rangle \Rightarrow \langle \texttt{if}\ e\ \texttt{then}\ s\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{else skip}\ ,\ env\rangle \in \mathcal{A}_{\texttt{IMP}}$
$\langle C[code]\ ,\ env\rangle \Rightarrow \langle C[code']\ ,\ env'\rangle \in \mathcal{A}_{\texttt{IMP}}$ if $\langle code\ ,\ env\rangle \Rightarrow \langle code'\ ,\ env'\rangle \in \mathcal{A}_{\texttt{IMP}}$

where $C\ ::=\ \_ \mid C\ \texttt{op}\ e \mid i\ \texttt{op}\ C \mid \texttt{if}\ C\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid v\ \texttt{:=}\ C \mid C\ s$

Fig. 2. Matching logic semantics of IMP as a set $\mathcal{A}_{\texttt{IMP}}$ of reachability rules (schemata). op ranges over the binary function symbols and $\texttt{op}_{Int}$ is their denotation in $\mathcal{T}$.

We now define matching logic formulae, or patterns, and their satisfaction. For the rest of this section we fix an ML signature (Cfg, Σ, Π) and use the standard notation $T_{\Sigma,s}(\mathcal{X})$ for the set of Σ-terms of sort s with variables in $\mathcal{X}$.

Definition 2. *Matching logic formulae, or* ***patterns****, extend the first-order formulae with terms $\pi \in T_{\Sigma,Cfg}(\mathcal{X})$, called* ***basic patterns****:*

$$\varphi \quad ::= \quad \forall x.\varphi \ \mid\ \varphi \wedge \varphi \ \mid\ \neg\varphi \ \mid\ P(t_1, \ldots, t_n) \ \mid\ \pi$$

Example 2. For example, $\varphi = \langle \texttt{skip}\ ,\ \texttt{x} \mapsto i\rangle \wedge i \geq 0$ is such a pattern. Intuitively, it denotes terminal configurations (the skip) with a state holding exactly one variable $\texttt{x} \in \texttt{Var}$ bound to an integer $i \in \mathcal{X}$ greater than or equal to 0.

Definition 3. *Let $\mathcal{T}$ be a fixed (Σ, Π) first-order model, whose elements of sort $\mathcal{T}_{Cfg}$ are called* ***concrete configurations****. We define the* ***satisfaction relation*** *between pairs $(\gamma \in \mathcal{T}_{Cfg}, \rho : \mathcal{X} \to \mathcal{T})$ and patterns as follows:*

$(\gamma, \rho) \models \forall x.\varphi$	*iff* $(\gamma, \rho') \models \varphi$ *for any* ρ' *with* $\rho'(y) = \rho(y)$ *for all* $y \in \mathcal{X} \setminus \{x\}$
$(\gamma, \rho) \models \varphi_1 \wedge \varphi_2$	*iff* $(\gamma, \rho) \models \varphi_1$ *and* $(\gamma, \rho) \models \varphi_2$
$(\gamma, \rho) \models \neg\varphi$	*iff it is not the case that* $(\gamma, \rho) \models \varphi$
$(\gamma, \rho) \models P(t_1, ..., t_n)$	*iff the predicate* P *holds on* $\rho(t_1), \ldots, \rho(t_n)$
$(\gamma, \rho) \models \pi$	*iff* $\gamma = \rho(\pi)$

When $(\gamma, \rho) \models \varphi$ we say that "γ matches pattern φ with witness ρ".

Example 3. Consider again pattern $\langle \texttt{skip}, \texttt{x} \mapsto i\rangle \wedge i \geq 0$. Then $(\langle \texttt{skip}, \texttt{x} \mapsto 42\rangle, i \mapsto 42)$ and $(\langle \texttt{skip}, \texttt{x} \mapsto 0\rangle, i \mapsto 0)$ satisfy it. However, $(\langle \texttt{skip}, \texttt{x} \mapsto 3\rangle, i \mapsto -3)$ and $(\langle \texttt{skip}, \texttt{x} \mapsto 3\rangle, i \mapsto 4)$ do not satisfy it, the former because i is negative and the latter because the *matching* fails: $\langle \texttt{skip}, \texttt{x} \mapsto 3\rangle \neq (i \mapsto 4)(\langle \texttt{skip}, \texttt{x} \mapsto i\rangle)$.

Since matching logic patterns generalize configuration terms, the transition relation between configurations, which is at the core of programming language semantics, can be extended to a relation between matching logic patterns [16]:

Definition 4. *A* ***reachability rule*** *is a pair $\varphi \Rightarrow \varphi'$, with φ and φ' patterns.*

Example 4. The reachability rule $\langle x\texttt{:=}\ i\texttt{;}s\texttt{,}\ env\rangle \Rightarrow \langle s\texttt{,}\ env[x \mapsto i]\rangle$ gives the semantics of IMP's assignment, and $\langle \texttt{if}\ i\ \texttt{then}\ s\ \texttt{else}\ t\texttt{,}\ env\rangle \wedge i \neq 0 \Rightarrow \langle s\texttt{,}\ env\rangle$ gives the semantics of the conditional statement for the positive case. In these rules, $x \in \mathcal{X}_{\texttt{Var}}$, $i \in \mathcal{X}_{\texttt{Int}}$, and $s, t \in \mathcal{X}_{\texttt{Stmt}}$ are variables of appropriate sorts.

$\langle i_1 \,\texttt{op}\, i_2 \rangle \Rightarrow \langle i_1 \,\texttt{op}_{Int}\, i_2 \rangle \in \mathcal{A}_{\texttt{FUN}}$
$\langle \texttt{if i then } e_1 \texttt{ else } e_2 \rangle \wedge i \neq 0 \Rightarrow \langle e_1 \rangle \in \mathcal{A}_{\texttt{FUN}}$
$\langle \texttt{if } 0 \texttt{ then } e_1 \texttt{ else } e_2 \rangle \Rightarrow \langle e_2 \rangle \in \mathcal{A}_{\texttt{FUN}}$
$\langle \texttt{letrec } f\ x = e \texttt{ in } e' \rangle \Rightarrow \langle e'[f \mapsto (\mu f.\lambda x.e)] \rangle \in \mathcal{A}_{\texttt{FUN}}$
$\langle (\lambda x.e)\ v \rangle \Rightarrow \langle e[x \mapsto v] \rangle \in \mathcal{A}_{\texttt{FUN}}$
$\langle \mu x.e \rangle \Rightarrow \langle e[x \mapsto (\mu x.e)] \rangle \in \mathcal{A}_{\texttt{FUN}}$
$\langle C[c] \rangle \Rightarrow \langle C[c'] \rangle \in \mathcal{A}_{\texttt{FUN}}$ if $\langle c \rangle \Rightarrow \langle c' \rangle \in \mathcal{A}_{\texttt{FUN}}$

where C ::= _ | C op e | if C then e_1 else e_2 | C e | v C

Fig. 3. Matching logic semantics of FUN as a set $\mathcal{A}_{\texttt{FUN}}$ of reachability rules.

Definition 5. *A **matching logic**(-based) **semantics** for a language is a tuple $(Cfg, \Sigma, \Pi, \mathcal{A}, \mathcal{T}, \rightarrow_{\mathcal{T}})$, where (Cfg, Σ, Π) is a matching logic signature and $\mathcal{A}$ a set of reachability rules, $\mathcal{T}$ is a configuration model like in Definition 3 and $\rightarrow_{\mathcal{T}}$ is the transition system generated by $\mathcal{A}$ on $\mathcal{T}$, that is, $\rightarrow_{\mathcal{T}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ with $\gamma \rightarrow_{\mathcal{T}} \gamma'$ iff there exist $\varphi \Rightarrow \varphi' \in \mathcal{A}$ and ρ such that $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$.*

As discussed in [16], conventional operational semantics of programming languages can be regarded as matching logic semantics: Σ includes the abstract syntax of the language as well as the syntax of the various operations in the needed mathematical domains; $\mathcal{A}$ is the (possibly infinite) set of operational semantics rules of the language; $\mathcal{T}$ is the model of configurations of the language together merged with the needed mathematical domains, and the relation $\rightarrow_{\mathcal{T}}$ is precisely the transition relation defined by the operational semantics. Fig. 2 and Fig. 3 show matching logic semantics of the IMP and FUN languages, respectively, obtained by mechanically representing conventional operational semantics of these languages based on reduction semantics with evaluation contexts into matching logic. The only observable difference between the original semantics of these languages and their matching logic semantics is that the side conditions have been conjuncted with the left-hand-side patterns in the positive case of the conditionals. Note that $\mathcal{A}_{\texttt{IMP}}$ and $\mathcal{A}_{\texttt{FUN}}$ are infinite, as the rules in Figs. 2 and 3 are schemata in meta-variable C (the evaluation context).

The following definitions allow us to state properties and to reason about the dynamic behavior of configurations denoted by matching logic patterns:

Definition 6. *Let $\mathcal{S}$ be a matching logic semantics $(Cfg, \Sigma, \Pi, \mathcal{A}, \mathcal{T}, \rightarrow_{\mathcal{T}})$. We define the following relations and predicates on patterns of $\mathcal{S}$:*

- *$\varphi \Rightarrow_{\mathcal{S}} \varphi'$ iff for all $(\gamma, \rho) \models \varphi$ there is a γ' such that $\gamma \rightarrow_{\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$;*
- *$\varphi \Rightarrow^{*}_{\mathcal{S}} \varphi'$ iff for all $(\gamma, \rho) \models \varphi$ there is a γ' such that $\gamma \rightarrow^{*}_{\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$;*
- *$\varphi \Rightarrow^{+}_{\mathcal{S}} \varphi'$ iff for all $(\gamma, \rho) \models \varphi$ there is a γ' such that $\gamma \rightarrow^{+}_{\mathcal{T}} \gamma'$ and $(\gamma', \rho) \models \varphi'$;*
- *$\varphi \Uparrow^{\infty}_{\mathcal{S}}$, read φ diverges, iff for all $(\gamma, \rho) \models \varphi$ we have that γ does not terminate in $\rightarrow_{\mathcal{T}}$, i.e., there is an infinite sequence $\gamma \rightarrow_{\mathcal{T}} \gamma_1 \rightarrow_{\mathcal{T}} \gamma_2 \rightarrow_{\mathcal{T}} \ldots$.*

Example 5. Let $\mathcal{S}_{\texttt{IMP}}$ denote the matching logic semantics of IMP in Fig. 2. Then $\langle \texttt{x:=}x\texttt{;}\ \texttt{y:=}y\texttt{,}\ \emptyset \rangle \Rightarrow^{*}_{\mathcal{S}_{\texttt{IMP}}} \langle \texttt{skip,}\ (\texttt{x} \mapsto x, \texttt{y} \mapsto y) \rangle$ (if we perform the two assignments) and $\langle \texttt{x:=}x\texttt{;}\ \texttt{y:=}y\texttt{,}\ \emptyset \rangle \Rightarrow^{*}_{\mathcal{S}_{\texttt{IMP}}} \langle \texttt{x:=}x\texttt{;}\ \texttt{y:=}y\texttt{,}\ \emptyset \rangle$ (by not taking any step). The former also holds when we replace $\Rightarrow^{*}_{\mathcal{S}_{\texttt{IMP}}}$ with $\Rightarrow^{+}_{\mathcal{S}_{\texttt{IMP}}}$, but the latter not. Also, $\langle \texttt{while 1 do skip,}\ \emptyset \rangle \Uparrow^{\infty}_{\mathcal{S}_{\texttt{IMP}}}$, but it is not the case that $\langle \texttt{x:=}x\texttt{;}\ \texttt{y:=}y\texttt{,}\ \emptyset \rangle \Uparrow^{\infty}_{\mathcal{S}_{\texttt{IMP}}}$.

3 Aggregating Matching Logic Semantics

In order to prove mutual equivalence of patterns in two (possibly) different matching logic semantics defining two (possbily) different programming languages, it is technically convenient to first put the two matching logic semantics together into one (larger) matching logic semantics. In this section we show how to aggregate two matching logic semantics, noticing that the construction works also for more than two semantics. The idea is to add infrastructure for pairing a configuration of one matching logic semantics with a configuration of the other, into a configuration of the aggregated semantics. Care needs to be exercised to factor out the common parts of the two semantics, such as the mathematical domains that happen to be used in both semantics (integers, maps, variables, etc.). The key result that allows for matching logic semantics to be aggregated is the amalgamation theorem of first-order (multi-sorted) logic [5, 4].

Let $\mathcal{S}_L = (\mathit{Cfg}_L, \Sigma_L, \Pi_L, \mathcal{A}_L, \mathcal{T}_L, \rightarrow_{\mathcal{T}_L})$ and $\mathcal{S}_R = (\mathit{Cfg}_R, \Sigma_R, \Pi_R, \mathcal{A}_R, \mathcal{T}_R, \rightarrow_{\mathcal{T}_R})$ be two fixed matching logic semantics, for example two programming language definitions, like those of `IMP` and `FUN` defined in Section 2. Also, suppose that their signatures and predicates share a common part, (Σ_0, Π_0), for example the domains of integers, maps, etc., which can be possibly renamed in the two semantics. Formally, we assume that there exist first-order logic morphisms

$$(\Sigma_L, \Pi_L) \xleftarrow{h_L} (\Sigma_0, \Pi_0) \xrightarrow{h_R} (\Sigma_R, \Pi_R)$$

and that the two configuration models $\mathcal{T}_L$ and $\mathcal{T}_R$ agree on their common part, that is, $\mathcal{T}_L{\restriction_{h_L}} = \mathcal{T}_R{\restriction_{h_R}}$, where $\restriction$ is the first-order model reduct operation.

We next show define the aggregation of $\mathcal{S}_L$ and $\mathcal{S}_R$. First, compute the first-order logic pushout of the morphisms h_L and h_R above, known to exist [5, 4]:

$$(\Sigma_L, \Pi_L) \xrightarrow{h'_L} (\Sigma', \Pi') \xleftarrow{h'_R} (\Sigma_R, \Pi_R)$$

Recall that a pushout puts together the two first-order signatures, possibly renaming sorts and operations happening to have the same name, but making sure that the shared sorts and symbols are correctly preserved. In particular, $h_L; h'_L = h_R; h'_R$ (we use the diagramatic notation for morphism composition).

Second, we add infrastructure for pairing configurations in the two object matching logic semantics. Specifically, add a new sort, say Cfg, and a new pairing operation, say $\langle \_, \_ \rangle : \mathit{Cfg}_L \times \mathit{Cfg}_R \rightarrow \mathit{Cfg}$, to Σ', and let us name the resulting first-order signature (Σ, Π)—no predicates were added to Π', so $\Pi = \Pi'$.

The constructions above are summarized in the following diagram

$$\begin{array}{ccccc}
(\Sigma_0, \Pi_0) & \xrightarrow{h_R} & (\mathit{Cfg}_R, \Sigma_R, \Pi_R) & & \\
\downarrow{\scriptstyle h_L} & & \downarrow{\scriptstyle h'_R} & & \\
(\mathit{Cfg}_L, \Sigma_L, \Pi_L) & \xrightarrow{h'_L} & (\Sigma', \Pi') & \xrightarrow{\iota} & (\mathit{Cfg}, \Sigma, \Pi)
\end{array}$$

with ι an inclusion (matching logic signatures also list their configuration sorts).

Third, we can aggregate $\mathcal{A}_L$ and $\mathcal{A}_R$ into a set of rules $\mathcal{A}$, by replacing each basic pattern π in $\mathcal{A}_L$ by a basic pattern $\langle \pi, \mathit{cfg}_R \rangle$ (here π is now regarded as a *term* of sort Cfg_L, not as a formula) for a fresh arbitrary but fixed variable cfg_R of sort Cfg_R, and similarly for $\mathcal{A}_R$. Intuitively, $\mathcal{A}$ consists of all the semantic steps of $\mathcal{A}_L$ and all of $\mathcal{A}_R$, on their respective position in the aggregated configuration.

Fourth, since $\mathcal{T}_L{\restriction_{h_L}} = \mathcal{T}_R{\restriction_{h_R}}$, the amalgamation theorem of (multi-sorted) first-order logic [5, 4] tells us that there exists a *unique* (Σ', Π')-model $\mathcal{T}'$ such that $\mathcal{T}'{\restriction_{h'_L}} = \mathcal{T}'{\restriction_{h'_R}}$. We can further (freely) extend $\mathcal{T}'$ into a (Σ, Π)-model $\mathcal{T}$ by adding a new carrier $\mathcal{T}_{\mathit{Cfg}} = \{\langle \gamma_L, \gamma_R \rangle \mid \gamma_L \in \mathcal{T}'_{\mathit{Cfg}_L}, \gamma_R \in \mathcal{T}'_{\mathit{Cfg}_R}\}$ and interpreting the pairing operations as expected: $\mathcal{T}_{\langle \_,\_ \rangle}(\gamma_L, \gamma_R) = \langle \gamma_L, \gamma_R \rangle$.

Finally, we construct the transition relation $\to_{\mathcal{T}}$ like in Definition 5, and this way obtain our aggregated matching logic semantics $\mathcal{S} = (\mathit{Cfg}, \Sigma, \Pi, \mathcal{A}, \mathcal{T}, \to_{\mathcal{T}})$.

Example 6. The following aggregated matching logic pattern matches all the pairs of `IMP` and `FUN` configurations which cannot take any further steps and where the `FUN` configuration holds a non-negative integer that can be retrieved in the `x` variable in the `IMP` configuration:

$$\exists i.(\langle\langle \texttt{skip}, \ (\texttt{x} \mapsto i, \_)\rangle, \langle i \rangle\rangle \wedge i > 0).$$

The underscore ($\_$) denotes an anonymous variable that matches the rest of the environment. The variable i is of sort integer and appears in both configurations (the sort `Int` is shared). The program variable `x` is a constant of sort `Var`.

By looking only at the left-hand side (or only at the right-hand side) component of each pair of configurations in an aggregated matching logic pattern and keeping all the other logical connectives unchanged, we obtain a pattern that specifies configurations in the corresponding matching logic semantics. We call these operations *projections*:

Definition 7. *The* ***left projection*** $pr_L(\varphi)$ *(resp. the* ***right projection*** $pr_R(\varphi)$*) of an aggregated matching logic formula φ is obtained by replacing each pattern of the form $\langle t_L, t_R \rangle$ by its left component t_L (resp. its right component t_R).*

Example 7. For $\varphi \equiv \exists i.(\langle\langle \texttt{skip}, (\texttt{x} \mapsto i, \_), \langle i \rangle\rangle \wedge i > 0)$, the aggregated matching logic pattern from the previous example, the left projection is $pr_L(\varphi) \equiv \exists i.(\langle \texttt{skip}, (\texttt{x} \mapsto i, \_)\rangle \wedge i > 0)$ and the right projection is $pr_R(\varphi) \equiv \exists i.(\langle i \rangle \wedge i > 0)$.

Note that by taking the left or the right projection of an aggregated pattern, we may not end up with a matching logic pattern over the syntax of the corresponding matching logic semantics. Indeed, for a pattern φ of the aggregated matching logic semantics $\mathcal{S} = (\mathit{Cfg}, \Sigma, \Pi, \mathcal{A}, \mathcal{T}, \to_{\mathcal{T}})$ constructed above, the left projection $pr_L(\varphi)$ is a pattern in a matching logic semantics that extends $\mathcal{S}_L$ to first-order syntax (Σ', Π'), and similarly for the right projection. Nevertheless, the following property holds: for any $\gamma_L \in \mathcal{T}_{\mathit{Cfg}_L}$, $\gamma_R \in \mathcal{T}_{\mathit{Cfg}_R}$, and $\rho : \mathcal{X} \to \mathcal{T}$, $(\langle \gamma_L, \gamma_R \rangle, \rho) \models \varphi$ iff $(\gamma_L, \rho) \models pr_L(\varphi)$ and $(\gamma_R, \rho) \models pr_R(\varphi)$, where for simplicity we used the same γ_L, γ_R, and ρ for their images through the model reducts.

4 Proving Mutual Program Equivalence

Here we provide a language-parametric foundation for showing equivalence of programs written in possibly different languages. Like in the previous section, we generically assume that the two languages are given as matching logic semantics $\mathcal{S}_L = (Cfg_L, \Sigma_L, \Pi_L, \mathcal{A}_L, \mathcal{T}_L, \rightarrow_{\mathcal{T}_L})$ and $\mathcal{S}_R = (Cfg_R, \Sigma_R, \Pi_R, \mathcal{A}_R, \mathcal{T}_R, \rightarrow_{\mathcal{T}_R})$ with aggregation $\mathcal{S} = (Cfg, \Sigma, \Pi, \mathcal{A}, \mathcal{T}, \rightarrow_{\mathcal{T}})$, but when we discuss examples we assume them to be the semantics $\mathcal{S}_{\texttt{IMP}}$ and $\mathcal{S}_{\texttt{FUN}}$ of, respectively, `IMP` and `FUN`.

4.1 Specifying Equivalent Programs

Aggregate matching logic patterns can be used to specify pairs of configurations of the two involved languages:

Definition 8. *The denotation of an aggregated matching logic pattern φ, written $[\![\varphi]\!]$, is the set of all pairs of configurations that satisfy it:*

$$[\![\varphi]\!] = \{(\langle \gamma_L, \gamma_R \rangle \mid \textit{there exists a valuation } \rho \textit{ such that } (\langle \gamma_L, \gamma_R \rangle, \rho) \models \varphi\}.$$

This extends sets of patterns E, written $[\![E]\!]$, as expected:

$$[\![E]\!] = \{(\gamma_L, \gamma_R) \mid \textit{there exist } \varphi \in E \textit{ such that } (\gamma_L, \gamma_R) \in [\![\varphi]\!]\}.$$

Example 8. The following set

$$E = \{\exists i.(\langle\langle \texttt{skip}, (\texttt{x} \mapsto i, \_)\rangle, \langle i \rangle\rangle)\}$$

captures all pairs of `IMP` and respectively `FUN` configurations that have terminated (since there is no more code to execute) and where the `IMP` variable `x` holds the same integer as the result of the `FUN` program.

Suppose we have an `IMP` program that computes its result in a variable `x` and suppose we want to show it computes the same integer result as a `FUN` program. Then the denotation $[\![E]\!]$ of set E above holds exactly the set of pairs of terminal configurations in which the two programs should end in order for them to compute the same result.

When trying to prove that two programs compute the same result, it is tempting to say that the two programs should reach the same configuration at the end. However, this is not feasible since the configuration might contain additional information (such as temporary variables) that was used in the computation but is not part of the result. When testing if the final configurations are the same in the two programs, it is important to ignore such additional information. In the example above, only the variable `x` is inspected (the values of all other variables are ignored) when comparing final configurations. Another aspect is that, when working in a general setting where we are comparing programs from two arbitrary programming languages, the configurations of the two languages might be significantly different. This is the case above, with the configuration for `IMP` holding code and an environment and the configuration for `FUN` holding only (extended) lambda expressions. Therefore, in general, to show that two

programs end up with the same result there is a need to design such a set $[\![E]\!]$ of "base" pairs which are known to be equivalent.

Two programs are then considered *mutually equivalent* when, for all inputs, they both diverge or they both reach a pair in the base equivalence $[\![E]\!]$. This intuition is captured by the following definition:

Definition 9. *We write* $\models \varphi \Downarrow^{\infty} E$*, and say that* φ ***reaches*** E*, iff for all configurations* γ_L, γ_R *and for all valuations* ρ *such that* $(\langle \gamma_L, \gamma_R \rangle, \rho) \models \varphi$ *we have that at least one of the following conditions holds:*

1. *both* γ_L *and* γ_R *diverge;*
2. *there are configurations* γ'_L, γ'_R *with* $\gamma_L \rightarrow_{\mathcal{T}}^{*} \gamma'_L$, $\gamma_R \rightarrow_{\mathcal{T}}^{*} \gamma'_R$ *and* $(\gamma'_L, \gamma'_R) \in [\![E]\!]$.

Example 9. Let $E = \{\exists i.(\langle\langle \texttt{skip, } (\texttt{x} \mapsto i, \_)\rangle, \langle i \rangle\rangle)$ and let

$$\varphi_1 = \exists n.\langle\langle \texttt{code}_1\texttt{, n} \mapsto n\rangle, \langle \texttt{exp}_1(n)\rangle\rangle$$
$$\varphi_2 = \langle\langle \texttt{while 1 do skip, } \emptyset\rangle, \langle \texttt{letrec f x = f(x + 1) in f(1)}\rangle\rangle$$
$$\varphi_3 = \exists n.\langle\langle \texttt{code}_3\texttt{, n} \mapsto n\rangle, \langle \texttt{exp}_3(n)\rangle\rangle.$$

where $\texttt{code}_1 \equiv$ `i:=1; x:=0; while i<=n do (x:=x+i; i:=i+1)` is the `IMP` program that computes the sum of the numbers from 1 to `n`, where $\texttt{exp}_1(\texttt{n}) \equiv$ `letrec f x = if x=1 then 1 else x+f(x-1) in f(n)` is the `FUN` program computing the same sum and where $\texttt{code}_3 \equiv \texttt{PGM}_L$ and, resp., $\texttt{exp}_3(\texttt{n}) \equiv \texttt{PGM}_R(\texttt{n})$ are the `IMP` and `FUN` programs in Fig. 5 that compute the Collatz function.

We have that $\models \varphi_1 \Downarrow^{\infty} E$ since both programs end up in a pair from $[\![E]\!]$: $\langle \texttt{code}_1\texttt{, n} \mapsto n\rangle \rightarrow_{\mathcal{T}}^{*} \langle \texttt{skip, x} \mapsto 1{+}2{+}\ldots{+}n\rangle$ and $\langle \texttt{exp}_1(n)\rangle \rightarrow_{\mathcal{T}}^{*} \langle 1{+}2{+}\ldots{+}n\rangle$. We also have that $\models \varphi_2 \Downarrow^{\infty} E$, since both configurations in φ_2 clearly diverge. We also have that $\models \varphi_3 \Downarrow^{\infty} E$, but this is more difficult to establish. In fact, it is currently only conjectured (not proven) that the programs terminate no matter what the input value n is. But it can be proven that if one does not terminate, the other does not terminate either and therefore $\models \varphi_3 \Downarrow^{\infty} E$ holds independently of the Collatz conjecture. However, $\models \varphi_3 \Downarrow^{\infty} E$ is more difficult to show than the previous examples since it is not clear if both programs terminate or diverge.

We next propose a proof system that allows us to derive such properties.

4.2 Proof System

In this section, we introduce a proof system that is able to derive sequents of the form $\vdash \varphi \Downarrow^{\infty} E$ denoting mutual equivalences that are sound in the sense that $\vdash \varphi \Downarrow^{\infty} E$ implies $\models \varphi \Downarrow^{\infty} E$. Fig. 4 contains the 5-rule proof system for proving mutual equivalence of programs.

The first rule is Axiom. There is nothing suprizing about this rule; it simply states that if an equivalence is known to be true, then it can be derived.

The second rule is Step. This rule allows to take an arbitrary finite number of steps (zero, one or more steps) in each of the two programs. If by taking such steps from φ to φ', we reach an equivalence φ' that is derivable, then

$$\textsc{Axiom}\ \frac{\varphi \in E}{\vdash \varphi \Downarrow^{\infty} E} \qquad \textsc{Step}\ \frac{\varphi \Rightarrow^{*} \varphi' \quad \vdash \varphi' \Downarrow^{\infty} E}{\vdash \varphi \Downarrow^{\infty} E} \qquad \textsc{Conseq}\ \frac{\models \varphi \to \varphi' \quad \vdash \varphi' \Downarrow^{\infty} E}{\vdash \varphi \Downarrow^{\infty} E}$$

$$\textsc{Case Analysis}\ \frac{\vdash \varphi \Downarrow^{\infty} E \quad \vdash \varphi' \Downarrow^{\infty} E}{\vdash \varphi \vee \varphi' \Downarrow^{\infty} E} \qquad \textsc{Circularity}\ \frac{\vdash \varphi' \Downarrow^{\infty} E \cup \{\varphi\} \quad \varphi \Rightarrow^{+} \varphi'}{\vdash \varphi \Downarrow^{\infty} E}$$

Fig. 4. Mutual Equivalence Proof System. We use $\varphi \Rightarrow^{*} \varphi'$ as syntactic sugar for $pr_L(\varphi) \Rightarrow^{*}_{\mathcal{S}_L} pr_L(\varphi')$ and $pr_R(\varphi) \Rightarrow^{*}_{\mathcal{S}_R} pr_R(\varphi')$ and $\varphi \Rightarrow^{+} \varphi'$ as syntactic sugar for $pr_L(\varphi) \Rightarrow^{+}_{\mathcal{S}_L} pr_L(\varphi')$ and $pr_R(\varphi) \Rightarrow^{+}_{\mathcal{S}_R} pr_R(\varphi')$.

we conclude that φ must also be derivable. The Step rule requires an oracle to reason about reachability in operational semantics. This oracle can be, for example, the reachability proof system in [16], but any other valid reasoning will also work.

The third rule is Conseq(uence). This rule states that if an equivalence formula φ implies another equivalence formula φ' (which means that φ' is more general than φ) and the formula φ' is derivable, then φ must also be derivable. The required implication might seem surprizing at first (we might expect it in reverse), but the intuition is that φ' is more general than φ. Therefore if we are able to prove the equivalence φ', then φ must also hold. This rule is used in the example proof tree below (in Fig. 5) to rearrange a formula of the form $(n > 0 \vee n = 0) \wedge \ldots$ into $n \geq 0 \wedge \ldots$. Another possible use of Conseq would be, for example, to transform a more particular case, like "$n = 20$", into a more general case "n is even" in order to be able to apply other rules.

The fourth case is Case Analysis. This allows to branch the proof depending on the different cases to considner. Typically, Case Analysis is used to branch the proof when the two programs also branch. In the proof tree below (in Fig. 5), this rule is used to perform a case analysis between the case where both programs end (because of reaching the termination condition $n = 0$) and where the programs continue ($n > 0$).

The fifth rule is Circularity. This rule is used to handle repetitive program structures such as loops or recursive functions. Circularity allows to *postulate* that the equivalence being proven (φ) holds, make progress ($\varphi \Rightarrow^{+} \varphi'$) in both programs that we want to show equivalent, and then derive φ' possibly using φ as an axiom, i.e., $\vdash \varphi' \Downarrow^{\infty} E \cup \{\varphi\}$. We use this rule in the proof tree below to assume that at the start of the repetitive behavior (the loop for the program on the left and the recursive call for the program on the right) the two programs are equivalent; we make progress by executing the body of the loop on the left and the body of the recursive on the right and end up with the equivalence that we assumed to hold. The rule is sound because we require both programs to make progress. Therefore, intuitively, when $\vdash \varphi' \Downarrow^{\infty} E \cup \{\varphi\}$ is derivable, either both programs diverge because φ is applied as an axiom in the proof tree or the programs end up in E. As for the first rule, an oracle to reason about reachability in operational semantics is also needed here.

$\texttt{PGM}_L := \texttt{c := 1; LOOP}_L$
$\texttt{LOOP}_L := \texttt{while (n != 1)}$
$\texttt{c := c + 1;}$
$\texttt{if (n \% 2 != 0)}$
$\texttt{then n := n + n + n + 1}$
$\texttt{else n := n / 2}$

$\texttt{PGM}_R(\texttt{n}) := \texttt{letrec f n = LOOP}_R \texttt{ in f(n)}$
$\texttt{LOOP}_R := \texttt{1 + if (n != 1)}$
$\texttt{then if (n \% 2 != 0)}$
$\texttt{then f(n + n + n + 1)}$
$\texttt{else f(n / 2)}$
$\texttt{else 0}$

$\varphi := \exists i, n.(n > 0 \wedge \langle \texttt{LOOP}_L\texttt{, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle)$

1. $\vdash \exists i, n.(\langle \texttt{skip, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \rangle) \Downarrow^{\infty} E$ — Axiom
2. $\vdash \exists i, n.(\langle \texttt{skip, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \rangle) \Downarrow^{\infty} E \cup \{\varphi\}$ — Axiom
3. $\vdash \exists i, n.(n = 0 \wedge \langle \texttt{LOOP}_L\texttt{, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle) \Downarrow^{\infty} E$ — Step(1)
4. $\vdash \exists i, n.(n = 0 \wedge \langle \texttt{LOOP}_L\texttt{, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle) \Downarrow^{\infty} E \cup \{\varphi\}$ — Step(2)
5. $\vdash \exists i, n.(n > 0 \wedge \langle \texttt{LOOP}_L\texttt{, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle) \Downarrow^{\infty} E \cup \{\varphi\}$ — Axiom
6. $\vdash \exists i, n.(n \geq 0 \wedge \langle \texttt{LOOP}_L\texttt{, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle) \Downarrow^{\infty} E \cup \{\varphi\}$ — Conseq(CA(4, 5))
7. $\vdash \exists i, n.(n > 0 \wedge \langle \texttt{LOOP}_L\texttt{, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle) \Downarrow^{\infty} E$ — Circularity (6)
8. $\vdash \exists i, n.(n \geq 0 \wedge \langle \texttt{LOOP}_L\texttt{, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle) \Downarrow^{\infty} E$ — Conseq(CA(3, 7))
9. $\vdash \exists i, n.(n \geq 0 \wedge \langle \texttt{PGM}_L\texttt{, n} \mapsto n \rangle, \langle \texttt{PGM}_R(n) \rangle) \Downarrow^{\infty} E$ — Step (8)

Fig. 5. Formal proof showing that the two Collatz programs are mutually equivalent. `CA` stands for Case Analysis. Conseq is used in the proof tree above to show that $n > 0 \vee n = 0$ implies $n \geq 0$. For simplicity, `letrec` is not desugared into μ.

Theorem 1 (Soundness). *For any set of aggregated matching logic patterns E and for any aggregated matching logic pattern φ, if the sequent $\vdash \varphi \Downarrow^{\infty} E$ is derivable using the proof system given in Fig. 4 then $\models \varphi \Downarrow^{\infty} E$.*

Proof. (complete proof in the appendix)

4.3 Example

Here we show the proof tree for the equivalence of the two Collatz programs in Fig. 5. As we have already discussed, in order to talk about mutual equivalence, we have to establish a "base" equivalence that contains programs that are clearly equivalent. For this case study, for the "base" equivalence, we choose to equate `FUN` programs that terminate by returning an integer i with `IMP` programs that terminate with the same integer i in variable c. The following set $E = \{\exists i.\langle\langle \texttt{skip, } (\texttt{c} \mapsto i, \_)\rangle, \langle i \rangle\rangle\}$ captures exactly the intuition above. It says that an `IMP` configuration $\langle \texttt{skip, } (\texttt{c} \mapsto i, \_)\rangle$ (describing programs that stopped (because the code cell contains `skip`) and that have the integer i in the c memory cell) is equivalent to a `FUN` configuration that contains exactly the integer i. The proof tree in Fig. 5 shows that the two programs are equivalent.

5 Discussion, Related Work and Conclusion

We have introduced mutual matching logic, a 5-rule proof system for proving mutual equivalence of programs. Mutual equivalence is a natural equivalence between programs: two programs are mutually equivalent if either they both diverge or if they eventually reach the same state. Mutual equivalence can be used, for example, to prove that compiler transformations preserve behavior.

Our approach is language independent. The proof system takes as input two language semantics (in the form of reachability rules) that share certain domains such as integers and produces sequents of the form $\vdash \varphi \Downarrow^{\infty} E$ whose semantics is that for any pair of programs that matches φ, both programs diverge or they reach a state in E. Note that in our running example (the two Collatz programs), both programs have a parameter `n` that is left unspecified. This shows that our approach allows parameterized programs. Although because of space limitations we do not explicitly state this, our approach can handle symbolic programs as well. For example, we can show using our proof system that in `IMP` extended with a `for` loop, the two programs:

```
i := 1;
while (i <= n) do
  s;
  i := i + 1;
```

```
for i := 1 to n do
  s;
```

are equivalent for a symbolic statement `s`. We specify the symbolic statement `s` as an additional statement in the signature of `IMP` and we render explicit the constraints on `s` in its semantics using reachability rules.

Related Work. It was first remarked by Hoare in [8] that program equivalence might be easier than program correcntess. Among the recent works on equivalence we mention [7, 6, 2]. The first one targets programs that include recursive procedures, the second one exploits similarities between single-threaded programs in order to prove their equivalence, and the third one extends the equivalence-verification to multi-threaded programs. They use operational semantics (of a specific language they designed, called LPL) and proof systems, and formally prove their proof system's soundness. In [7] a classification of equivalence relations used in program-equivalence research is given, one of which is mutual equivalence (called *full equivalence* there). The main difference with our approach is that our proof system is language-independent, i.e., it is parametric in the semantics of the two languages in which candidate equivalent programs are written; whereas the deductive system of [7] proves equivalence for LPL programs. On the other hand, [7] propose deductive systems for several kinds of equivalences, whereas we focus on mutual (a.k.a. full) equivalence only. In [9], an implementation of a parametrized equivalence prover is presented.

A lot of work on program equivalence arise from the verification of compilation in a broad sense. One approach is full compiler verification [11], and another one is the individual verification of each compilation [13] (we only cite two of the most relevant recent works). Other work target specific classes of languages: functional [14], microcode [1], CLP [3]. In order to be less language-specific some approaches advocate the use of intermediate languages, such as [10], which works on the Boogie intermediate language. Finally, our own related work [12] gives a proof system for another equivalence relation between programs that is based on bisimulation and an observation relation and that uses other technical mechanisms. We believe that the equivalence relation that we consider in this article in more natural for certain classes of applications such as proving compilers.

Further Work. Our definition (Definition 9) of mutual equivalence is *existential* in the sense that two programs are equivalent when there exists execution

paths in each of the programs such that the paths diverge or end in configurations that are known to be equivalent. Although for deterministic languages this cannot constitute a problem (there exists exactly one execution path for each program), for non-deterministic languages stronger equivalences might be desirable. We leave such stronger equivalences as object of further study. Another issue is completeness. Although relative completeness results have been shown for matching logic based proof systems for showing partial correctness [16], it is less clear how a relevant relative-completeness result can be obtained for equivalence, since the problem is known to be Π_2^0-complete. Another issue that we leave for further study is compositionality. Our goal here was just to obtain a sound and useful language independent proof system for reasoning about equivalence.

References

1. T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and L. D. Zuck. Formal verification of backward compatibility of microcode. In *CAV, LNCS 3576*, pages 185–198, 2005.
2. S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *VMCAI, LNCS 7148*, pages 119–135, 2012.
3. S. Craciunescu. Proving the equivalence of CLP programs. In *ICLP, LNCS 2401*, pages 287–301, 2002.
4. R. Diaconescu. *Institution-independent Model Theory*. Birkhauser Basel, 2008.
5. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*. Springer, 1985.
6. B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*. To appear.
7. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
9. S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, pages 327–337. ACM, 2009.
10. S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV, LNCS 7358*, pages 712–717, 2012.
11. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
12. D. Lucanu and V. Rusu. Program equivalence by circular reasoning. Technical Report RR-8116, INRIA, 2012.
13. G. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94. ACM, 2000.
14. A. Pitts. Operational semantics and program equivalence. In *Applied Semantics Summer School, LNCS 2395*, pages 378–412, 2002.
15. G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST, LNCS 6486*, pages 142–162, 2010.
16. G. Roşu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.

A Proofs

This appendix contains the proof of the soundness theorem that was left out for space reasons.

In the following, we let $C \in \{L, R\}$ denote either left or right. By $\bar{C}$ we denote the single element of the set $\{L, R\} \setminus \{C\}$.

Let E be a set of mutual matching logic formulae. Let $\mathcal{A}_L$ and $\mathcal{A}_R$ be a set of reachability formulae which describe the semantics of two languages: $\mathcal{A}_L$ the "left" language and $\mathcal{A}_R$ the "right" language. We extend the definition of $\models \varphi \Downarrow^\infty E$ to sets of mutual matching logic formulae as expected:

Definition 10. *If F is a set of mutual matching logic formulae, then we write*

$$\models F \Downarrow^\infty E \text{ if } \models \varphi \Downarrow^\infty E \text{ for all } \varphi \in F.$$

The following definitions will be useful in the proof of soundness. Let G denote a set of pairs of configurations.

Definition 11. *We say that a pair (γ_L, γ_R) reaches G, written $(\gamma_L, \gamma_R) \to^* G$, if there exist configurations γ'_L and γ'_R such that $\gamma_L \to^*_{\mathcal{A}_L} \gamma'_L$, $\gamma_R \to^*_{\mathcal{A}_R} \gamma'_R$ and $(\gamma'_L, \gamma'_R) \in G$.*

Definition 12. *We say that a pair (γ_L, γ_R) diverges, written $(\gamma_L, \gamma_R)\uparrow^\infty$, if both γ_L and γ_R diverge (in $\mathcal{A}_L$ and respectively $\mathcal{A}_R$).*

Definition 13. *We say that a pair (γ_L, γ_R) co-reaches G, written $(\gamma_L, \gamma_R) \to^{*,\infty} G$, if at least one of the following conditions holds:*

1. *(γ_L, γ_R) diverges (i.e. $(\gamma_L, \gamma_R)\uparrow^\infty$),*
2. *(γ_L, γ_R) reaches G (i.e. $(\gamma_L, \gamma_R) \to^* G$).*

The following utility lemma establishes the link between models of mutual matching logic formulae and the notion of co-reachability introduced above. Its proof following trivially by unrolling the above definitions.

Lemma 1. *For all sets of mutual matching logic formulae E and for any mutual matching logic formula φ, we have that:*

$$\models \varphi \Downarrow^\infty E \text{ iff for all } \gamma_L, \gamma_R \text{ such that } (\gamma_L, \gamma_R) \in [\![\varphi]\!], (\gamma_L, \gamma_R) \to^{*,\infty} [\![E]\!].$$

The next lemma is the core of our soundness proof.

Lemma 2 (Circularity Principle).
Let F be a set of mutual matching formulae. If for each $(\gamma_L, \gamma_R) \in [\![F]\!]$ there exist γ'_L, γ'_R such that $\gamma_L \to^+_{\mathcal{A}_L} \gamma'_L$, $\gamma_R \to^+_{\mathcal{A}_R} \gamma'_R$, and $(\gamma'_L, \gamma'_R) \to^{*,\infty} [\![E \cup F]\!]$, then $\models F \Downarrow^\infty E$.

Proof. Let $(\gamma_L, \gamma_R) \in [\![F]\!]$ be arbitrarily chosen. We show that $\models F \Downarrow^\infty E$. By Lemma 1 it is enough to show that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$.

For any natural number $i > 0$, let $P(i)$ denote the following predicate:

$$\begin{aligned} P(i) = \text{either there exist configurations } & \gamma_L^1, \ldots, \gamma_L^i, \gamma_R^1, \ldots, \gamma_R^i \text{ such that} \\ & \gamma_L \rightarrow^+_{\mathcal{A}_L} \gamma_L^1 \rightarrow^+_{\mathcal{A}_L} \gamma_L^2 \rightarrow^+_{\mathcal{A}_L} \cdots \rightarrow^+_{\mathcal{A}_L} \gamma_L^i, \\ & \gamma_R \rightarrow^+_{\mathcal{A}_R} \gamma_R^1 \rightarrow^+_{\mathcal{A}_R} \gamma_R^2 \rightarrow^+_{\mathcal{A}_R} \cdots \rightarrow^+_{\mathcal{A}_R} \gamma_R^i \text{ and} \\ & (\gamma_L^1, \gamma_R^1) \rightarrow^{*,\infty} E \cup F, \ldots, (\gamma_L^i, \gamma_R^i) \rightarrow^{*,\infty} E \cup F \\ \text{or } & (\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!] \end{aligned}$$

By induction on i, we show that for all $i > 0$, $P(i)$ holds:

1. for $i = 1$, we have that by the hypothesis, there exist γ_L^1, γ_R^1 such that $\gamma_C \rightarrow^+_{\mathcal{A}_C} \gamma_C^1$ (for all $C \in \{L, R\}$), and $(\gamma_L^1, \gamma_R^1) \rightarrow^{*,\infty} [\![E \cup F]\!]$.
2. for $i > 1$, we assume that $P(i-1)$ holds and we prove that $P(i)$ holds. If $P(i-1)$ holds because $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$, then by definition $P(i)$ holds as well.
   Otherwise, we have that there exist configurations $\gamma_L^1, \ldots, \gamma_L^{i-1}, \gamma_R^1, \ldots, \gamma_R^{i-1}$ such that:
   (a) $\gamma_L \rightarrow^+_{\mathcal{A}_L} \gamma_L^1 \rightarrow^+_{\mathcal{A}_L} \gamma_L^2 \rightarrow^+_{\mathcal{A}_L} \cdots \rightarrow^+_{\mathcal{A}_L} \gamma_L^{i-1}$,
   (b) $\gamma_R \rightarrow^+_{\mathcal{A}_R} \gamma_R^1 \rightarrow^+_{\mathcal{A}_R} \gamma_R^2 \rightarrow^+_{\mathcal{A}_R} \cdots \rightarrow^+_{\mathcal{A}_R} \gamma_R^{i-1}$ and
   (c) $(\gamma_L^1, \gamma_R^1) \rightarrow^{*,\infty} E \cup F, \ldots, (\gamma_L^{i-1}, \gamma_R^{i-1}) \rightarrow^{*,\infty} E \cup F$
   As $(\gamma_L^{i-1}, \gamma_R^{i-1}) \rightarrow^{*,\infty} [\![E \cup F]\!]$, we distinguish between the following two cases:
   (a) either $(\gamma_L^{i-1}, \gamma_R^{i-1}) \rightarrow^{*,\infty} [\![E]\!]$ and, because $\gamma_L \rightarrow^+_{\mathcal{A}_L} \gamma_L^{i-1}$, we have that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$. This implies that $P(i)$ holds (by the second condition in definition of the predicate).
   (b) or $(\gamma_L^{i-1}, \gamma_R^{i-1}) \rightarrow^{*,\infty} [\![F]\!]$. By the hypothesis, there exist γ_L^i and γ_R^i such that $\gamma_L^{i-1} \rightarrow^+_{\mathcal{A}_L} \gamma_L^i$, $\gamma_R^{i-1} \rightarrow^+_{\mathcal{A}_R} \gamma_R^i$ and $(\gamma_L^i, \gamma_R^i) \rightarrow^{*,\infty} [\![E \cup F]\!]$. But this implies that $P(i)$ holds (by the first condition in the definition of the predicate).
   In either case, we have shown that $P(i)$ whenever $P(i-1)$ holds.

Therefore, we conclude by induction that, for all naturals $i > 0$, $P(i)$ holds.

We will now show by contradiction that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$. We will assume by contradiction that $(\gamma_L, \gamma_R) \not\rightarrow^{*,\infty} [\![E]\!]$. As $P(i)$ holds for all naturals $i > 0$, it follows that for any $i > 0$, there exist $\gamma_L^1, \ldots, \gamma_L^i, \gamma_R^1, \ldots, \gamma_R^i$ such that $\gamma_L \rightarrow^+_{\mathcal{A}_L} \gamma_L^1 \rightarrow^+_{\mathcal{A}_L} \cdots \rightarrow^+_{\mathcal{A}_L} \gamma_L^i$ and $\gamma_R \rightarrow^+_{\mathcal{A}_R} \gamma_R^1 \rightarrow^+_{\mathcal{A}_R} \cdots \rightarrow^+_{\mathcal{A}_R} \gamma_R^i$. This means that both γ_L and γ_R diverge (in $\mathcal{A}_L$ and respectively $\mathcal{A}_R$) and therefore $(\gamma_L, \gamma_R)\uparrow^\infty$. But this implies $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$, which yields a contradiction. As our assumption $(\gamma_L, \gamma_R) \not\rightarrow^{*,\infty} [\![E]\!]$ led to a contradiction, it follows that it must hold and therefore $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$, which is what we had to show.

We are now ready to show the main result of this section, that the proof system is sound.

Theorem 2 (Soundness).
For any set of mutual matching logic formulae E, for any mututal matching logic formula φ, if the sequent $\vdash \varphi \Downarrow^{\infty} E$ is derivable using the proof system given in Figure 4, then $\models \varphi \Downarrow^{\infty} E$.

Proof. By induction on the proof tree and case analysis on the last rule applied:

1. if the last rule to be applied is Axiom, we have that $\varphi \in E$ and we show that $\models \varphi \Downarrow^{\infty} E$. This is immediate by reflexivity.
2. if the last rule to be applied is Step, we have by the induction hypothesis that there exists φ' such that $\mathcal{A}_L \models pr_L(\varphi) \rightarrow^* pr_L(\varphi')$, that $\mathcal{A}_R \models pr_R(\varphi) \rightarrow^* pr_R(\varphi')$ and that $\models \varphi' \Downarrow^{\infty} E$. We show that $\models \varphi \Downarrow^{\infty} E$.
   Let γ_L, γ_R be arbitrary configurations such that $(\gamma_L, \gamma_R) \in [\![\varphi]\!]$. By Lemma 1, it is sufficient to show that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$.
   As $(\gamma_L, \gamma_R) \in [\![\varphi]\!]$, $\mathcal{A}_L \models pr_L(\varphi) \rightarrow^* pr_L(\varphi')$ and $\mathcal{A}_R \models pr_R(\varphi) \rightarrow^* pr_R(\varphi')$, it follows that there exist γ'_L, γ'_R such that $\gamma_R \rightarrow^*_{\mathcal{A}_R} \gamma'_R$, $\gamma_L \rightarrow^*_{\mathcal{A}_L} \gamma'_L$ and $(\gamma'_L, \gamma'_R) \models \varphi'$.
   But $E \models \varphi'$ and therefore, by Lemma 1, $(\gamma'_L, \gamma'_R) \rightarrow^{*,\infty} [\![E]\!]$. But $\gamma_L \rightarrow^*_{\mathcal{A}_L} \gamma'_L$ and $\gamma_R \rightarrow^*_{\mathcal{A}_R} \gamma'_R$ and therefore $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$, which is what we had to prove.
3. if the last rule to be applied is Consequence, we have by the induction hypothesis that there exists φ' such that $\models \varphi \rightarrow \varphi'$ and $\models \varphi' \Downarrow^{\infty}$. We show that $\models \varphi \Downarrow^{\infty} E$.
   Let γ_L, γ_R be arbitrary configurations such that $(\gamma_L, \gamma_R) \in [\![\varphi]\!]$. By Lemma 1, it is sufficient to show that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$.
   As $(\gamma_L, \gamma_R) \in [\![\varphi]\!]$ and $\models \varphi \rightarrow \varphi'$, we have that $(\gamma_L, \gamma_R) \in [\![\varphi']\!]$. But we have that $\models \varphi' \Downarrow^{\infty} E$, and therefore, by Lemma 1, $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$, which is what we had to show.
4. if the last rule to be applied is Case Analysis, we have by the induction hypothesis that $\models \varphi \Downarrow^{\infty} E$ and that $\models \varphi' \Downarrow^{\infty} E$. We show that $\models \varphi \vee \varphi' \Downarrow^{\infty} E$.
   Let γ_L, γ_R be arbitrary configurations such that $(\gamma_L, \gamma_R) \in [\![\varphi \vee \varphi']\!]$. By Lemma 1, it is sufficient to show that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$.
   As $(\gamma_L, \gamma_R) \in [\![\varphi \vee \varphi']\!]$, we have that $(\gamma_L, \gamma_R) \in [\![\varphi]\!]$ or that $(\gamma_L, \gamma_R) \in [\![\varphi']\!]$:
   (a) in the first case, we have that $(\gamma_L, \gamma_R) \in [\![\varphi]\!]$. We also have that $\models \varphi \Downarrow^{\infty} E$ (by the induction hypothesis). By Lemma 1, we obtain that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$.
   (b) in the second case, we have that $(\gamma_L, \gamma_R) \in [\![\varphi']\!]$. We also have that $\models \varphi' \Downarrow^{\infty} E$ (by the induction hypothesis). By Lemma 1, we obtain that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$.
   In either case we have obtained that $(\gamma_L, \gamma_R) \rightarrow^{*,\infty} [\![E]\!]$, which is what we had to prove.
5. if the last rule to be applied is Circularity then, by the induction hypothesis, there exists φ' such that $\mathcal{A}_L \models pr_L(\varphi) \rightarrow^+ pr_L(\varphi')$, $\mathcal{A}_R \models pr_R(\varphi) \rightarrow^+ pr_R(\varphi')$ and $\models \varphi' \Downarrow^{\infty} E \cup \{\varphi\}$. We show that $\models \varphi \Downarrow^{\infty} E$.
   We let $F = \{\varphi\}$ and we apply the Circularity Principle (Lemma 2). Let γ_L, γ_R be two arbitrary configurations and let ρ be an arbitrary valuation

such that $(\langle \gamma_L, \gamma_R \rangle, \rho) \models \varphi$. As $\mathcal{A}_C \models \pi_C(\varphi) \to^+ \pi_C(\varphi')$ (for $C \in \{L, R\}$), there exists γ'_C such that $\gamma_C \to^+_{\mathcal{A}_C} \gamma'_C$ and $(\gamma'_C, \rho) \models \pi_C(\varphi')$ (for $C \in \{L, R\}$). It follows that $(\langle \gamma'_L, \gamma'_R \rangle, \rho) \models \varphi'$. Since $\models \varphi' \Downarrow^\infty E \cup \{\varphi\}$, we get

$$(\gamma'_L, \gamma'_R) \to^{*,\infty} [\![E \cup \{\varphi\}]\!],$$

by Lemma 1. Now the hypotheses of the Circularity Principle are satisfied and hence obtain $\models \varphi \Downarrow^\infty E$.