A Language-Independent Proof System for Mutual Program Equivalence

Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, Grigore Rosu

A Language-Independent Proof System for Mutual Program Equivalence

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

Abstract. Two programs are mutually equivalent if they both diverge or they end up in similar states. Mutual equivalence is an adequate notion of equivalence for programs written in deterministic languages. It is useful in many contexts, such as capturing the correctness of program transformations within the same language, or capturing the correctness of compilers between two different languages. In this paper we introduce a language-independent proof system for mutual equivalence, which is parametric in the operational semantics of two languages and in a state-similarity relation. The proof system is sound: if it terminates then it establishes the mutual equivalence of the programs given to it as input. We illustrate it on two programs in two different languages (an imperative one and a functional one), that both compute the Collatz sequence.

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. In this article, we explore *mutual equivalence*, an equivalence relation that is also known in the literature as full equivalence [6]. 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 are written in two possibly different 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 that is closed under the transition relations given by the corresponding operational semantics. This involves constructing a single language that is capable of executing pairs of programs written in the two languages. 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 aggregated language must be capable of independently "executing" pairs of programs in the original languages. Once the aggregated language constructed, the most import rule in our proof system for mutual equivalence is the Circularity rule, which incrementally postulates synchronization points in the two

programs. We illustrate the proof system on two programs (Fig. 6 on page 14) that both compute the Collatz sequence, but in different 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, of course, knowing whether they terminate.

In Section 2 we introduce the preliminaries needed in the rest of the paper. This includes many-sorted first-order signatures, their models, and the amalgamation theorem that relates the models of pushouts of two first-order signatures with the models of the individual signatures. Section 3 presents matching logic, a specialization of many-sorted first-order logic, and shows how it can be used to give operational semantics to programming languages. Section 4 then shows how to aggregate matching logic semantics, a core operation for the mutual equivalence of programs from possibly distinct languages. Section 5 shows how our formalism can be used to specify equivalent programs and Section 6 presents our proof system for mutual equivalence and applies it to two programs computing the Collatz sequence. Section 7 discusses related work and concludes. Proofs not included due to space constraints can be found in the technical report [3].

2 Preliminaries

In this section, we recall notions and notations that used throughout the paper. We present the syntax and semantics of **many-sorted first-order logic**, which is used to define matching logic later in the paper. We then state the **a**malgamation theorem for many-sorted first-order logic, a known result that allows us to construct a model for the pushout construction of two first-order signatures, from the the models of the two signatures, even when the two signature share function symbols. We use the amalgamation result later in the article in order to construct the aggregated semantics of two languages from their individual semantics.

2.1 Many-sorted First Order Logic

Let S be a set of **sorts**, Σ an S-sorted algebraic signature (i.e., an indexed set $\Sigma = \cup_{w \in S^*, s \in S} \Sigma_{w,s}$, where $\Sigma_{w,s}$ is the set of function symbols of arity w with a result of sort s) and Π an indexed set $\Pi = \cup_{w \in S^*} \Pi_w$ of predicate symbols. Then $\Phi = (S, \Sigma, \Pi)$ is called a **many-sorted FOL signature**. We write $x \in \Phi$ instead of $x \in S \cup \Sigma \cup \Pi$. By $T_{\Sigma,s}(Var)$ we denote the set of terms of sort s built over the variables Var with function symbols in Σ. We sometimes omit Σ if it is clear from the context and we write $T_s(Var)$ instead of $T_{\Sigma,s}(Var)$.

Example 1. The signatures $\Phi_I = (S_I, \Sigma_I, \Pi_I)$ and $\Phi_F = (S_F, \Sigma_F, \Pi_F)$ in Figure 1 model the syntax of an imperative and a functional programming language, with sorts $S_I = \{\texttt{Int}, \texttt{Var}, \texttt{ExpI}, \texttt{Stmt}, \texttt{Code}, \texttt{CfgI}\}$ in `IMP` and sorts $S_F = \{\texttt{Var}, \texttt{Int}, \texttt{ExpF}, \texttt{Val}, \texttt{CfgF}\}$ in `FUN`, function symbols

$\Sigma_I = \{_\texttt{+}_, _\texttt{*}_, _\texttt{-}_, _ + __{Int}, _ - __{Int}, _ * __{Int}, _\texttt{:=}_, \texttt{skip}, _\texttt{;}_, \texttt{if_then_else_}, \texttt{while_do_}, \langle _\texttt{,}_ \rangle\}$in `IMP` and
$\Sigma_F = \{\_\texttt{+}\_, \_\texttt{*}\_, \_\texttt{-}\_, \_ + \__{Int}, \_ - \__{Int}, \_ * \__{Int}, \_\_, \texttt{letrec}\_\_\texttt{=}\_\texttt{in}\_, \texttt{if\_then\_else\_}, \mu \_ . \_, \lambda \_ . \_, \langle \_ \rangle\}$ in `FUN`.

```
ExpI ::= Var | Int | ExpI + ExpI
Stmt ::= Var := ExpI
       | skip | Stmt ; Stmt
       | if ExpI then Stmt else Stmt
       | while ExpI do Stmt
Code ::= ExpI | Stmt
CfgI ::= ⟨Code, Map{Var,Int}⟩
```

```
ExpF ::= Var | Val | ExpF + ExpF
       | ExpF ExpF
       | letrec Var Var = ExpF in ExpF
       | if ExpF then ExpF else ExpF
       | μ Var . ExpF
Val  ::= Int | λ Var . ExpF
CfgF ::= ⟨ExpF⟩
```

Fig. 1. $\Phi_I = (S_I, \Sigma_I, \Pi_I)$ and $\Phi_F = (S_F, \Sigma_F, \Pi_F)$, the signatures of IMP and FUN, written using BNF notation and detailed in Example 1. Only the function symbols are detailed in the figure and the predicates for the two languages consist of the arithmetic comparison operators: $\Pi_I = \Pi_F = \{=_{Int}, <_{Int}, \leq_{Int}\}$. The difference between the operators $\_\texttt{+}\_$, $\_\texttt{*}\_$, etc. and their correspondants $\_ + \_{Int}$, $\_ * \_{Int}$, etc. is that the former are the syntactic language constructs for addition, multiplication, etc. (i.e. they take language expressions as arguments), while the later are the actual function symbols for denoting integer addition, multiplication, etc. (they take integers as arguments).

The function symbols above are written using Maude-like notation, where the underscore ($\_$) denotes the position of an argument. Althought not written explicitly above, the signatures also include the one-argument injections needed to inject sorts like `Int` and `Var` into `ExpI`.

Definition 1. *We say that* $\mathcal{T} = (\llbracket\cdot\rrbracket^{\boldsymbol{S}}_{\mathcal{T}}, \llbracket\cdot\rrbracket^{\boldsymbol{F}}_{\mathcal{T}}, \llbracket\cdot\rrbracket^{\boldsymbol{P}}_{\mathcal{T}})$ *is a* ***model*** *of a many-sorted signature* $\Phi = (S, \Sigma, \Pi)$ *if:*

1. $\llbracket s\rrbracket^{\boldsymbol{S}}_{\mathcal{T}}$*, the interpretation of the sort* s *in the model* $\mathcal{T}$*, is a set for each* $s \in S$
2. $\llbracket f\rrbracket^{\boldsymbol{F}}_{\mathcal{T}}$*, read as the interpretation of the function symbol* f *in the model* $\mathcal{T}$*, is a function defined on* $\llbracket s_1\rrbracket_{\mathcal{T}} \times \ldots \times \llbracket s_n\rrbracket_{\mathcal{T}}$ *with values in* $\llbracket s\rrbracket_{\mathcal{T}}$*, for every function symbol* $f \in \Sigma_{s_1,\ldots,s_n,s}$*.*
3. $\llbracket p\rrbracket^{\boldsymbol{P}}_{\mathcal{T}}$*, read as the interpretation of the predicate symbol* p *in the model* $\mathcal{T}$*, is a subset of* $\llbracket s_1\rrbracket_{\mathcal{T}} \times \ldots \times \llbracket s_n\rrbracket_{\mathcal{T}}$ *for every predicate symbol* $p \in \Pi_{s_1,\ldots,s_n}$*.*

From now on, we write $\llbracket\cdot\rrbracket_{\mathcal{T}}$ instead of $\llbracket\cdot\rrbracket^{\mathbf{S}}_{\mathcal{T}}$, $\llbracket\cdot\rrbracket^{\mathbf{F}}_{\mathcal{T}}$ and $\llbracket\cdot\rrbracket^{\mathbf{P}}_{\mathcal{T}}$ when the type of the argument (sort, function symbol or predicate symbol), is clear from context.

Example 2. We consider $\mathcal{T}_I$ to be a model of $\Phi_I = (S_I, \Sigma_I, \Pi_I)$ where the interpretation $\llbracket\texttt{Var}\rrbracket_{\mathcal{T}_I}$ of the sort `Var` is the set of strings, the interpretation $\llbracket\texttt{Int}\rrbracket_{\mathcal{T}_I}$ of the sort `Int` is the set of integers and the function symbols are interpreted syntactically (as terms). The predicates $=_{Int}, <_{Int}, \leq_{Int}$ are interpreted as the respective comparison relations between integers.

Definition 2. *Let* $\Phi = (S, \Sigma, \Pi)$ *and* $\Phi' = (S', \Sigma', \Pi')$ *be two many-sorted FOL signatures and let* h *be a function from* $S \cup \Sigma \cup \Pi$ *to* $S' \cup \Sigma' \cup \Pi'$*. The function* h *is a* ***morphism*** *between* Φ *and* Φ' *if it preserves sort compatibility:*

1. $h(S) \subseteq S'$*,*
2. *if* $f \in \Sigma_{s_1\ldots s_n,s}$ *then* $h(f) \in \Sigma'_{h(s_1)\ldots h(s_n),h(s)}$ *and*
3. *if* $p \in \Pi_{s_1\ldots s_n}$ *then* $h(p) \in \Pi'_{h(s_1)\ldots h(s_n)}$*.*

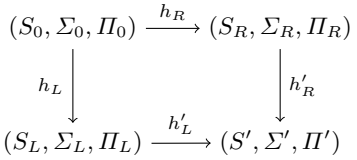

Fig. 2. Push-out diagram assumed throughout the paper.

Definition 3. *Let h be a morphism from $\Phi = (S, \Sigma, \Pi)$ to $\Phi' = (S', \Sigma', \Pi')$ and $\mathcal{T}'$ be a model of $\Phi' = (S', \Sigma', \Pi')$. We define $\mathcal{T}'\restriction_h$ (the **reduct** of $\mathcal{T}'$ through h) to be the model of Φ such that:*

1. $[\![s]\!]_{\mathcal{T}'\restriction_h} = [\![h(s)]\!]_{\mathcal{T}'}$ *for all* $s \in S$.
2. $[\![f]\!]_{\mathcal{T}'\restriction_h}(e_1, \ldots, e_n) = [\![h(f)]\!]_{\mathcal{T}'}(e_1, \ldots, e_n)$ *for all* $f \in \Sigma_{s_1,\ldots,s_n,s}$ *and for all* $e_1 \in [\![s_1]\!]_{\mathcal{T}'\restriction_h}$, …, $e_n \in [\![s_n]\!]_{\mathcal{T}'\restriction_h}$.
3. $(e_1, \ldots, e_n) \in [\![p]\!]_{\mathcal{T}'\restriction_h}$ *iff* $(e_1, \ldots, e_n) \in [\![h(p)]\!]_{\mathcal{T}'}$ *for all* $p \in \Pi_{s_1,\ldots,s_n}$ *and for all* $e_1 \in [\![h(s_1)]\!]_{\mathcal{T}'}, \ldots, e_n \in [\![h(s_n)]\!]_{\mathcal{T}'}$.

Example 3. Let $\Phi = (\{\texttt{Int}\}, \{op_{Int} \mid op \in \{\_\texttt{+}\_, \_\texttt{-}\_, \_\texttt{*}\_\}\}, \{op_{Int} \mid op \in \{=, <, \leq\}\})$ be a signature and let h (with $h(\texttt{Int}) = \texttt{Int}$ and $h(op_{Int}) = op_{Int}$ for $op \in \{\_\texttt{+}\_, \_\texttt{-}\_, \_\texttt{*}\_\}$) be a morphism from Φ to Φ_I (defined above in Example 1). Let $\mathcal{T}_I$ be the model of Φ_I considered above in Example 2. We have that $[\![\texttt{Int}]\!]_{\mathcal{T}'\restriction_h}$ is the set of integers, $[\![\_\texttt{+}\_{Int}]\!]_{\mathcal{T}'\restriction_h}$ is the addition of integers, etc.

2.2 The Amalgamation Theorem

Theorem 1 (Pushout). *Let Φ_R, Φ_L and Φ_0 be three FOL signatures, h_R a morphism from Φ_0 to Φ_R and h_L a morphism from Φ_0 to Φ_L. There exists a tuple (h'_L, Φ', h'_R), called the **pushout** of $\Phi_L \xleftarrow{h_L} \Phi_0 \xrightarrow{h_R} \Phi_R$, where h'_L is a morphism from Φ_L to Φ' and h'_R a morphism from Φ_R to Φ' such that the following conditions hold:*

1. *(commutativity) $h'_L(h_L(x)) = h'_R(h_R(x))$ for all $x \in \Phi_0$ and*
2. *(minimality) if there exist Φ'' and morphisms h''_L (from Φ_L to Φ'') and h''_R (from Φ_R to Φ'') such that $h''_L(h_L(x)) = h''_R(h_R(x))$ for all $x \in \Phi_0$ then there exists a morphism h from Φ' to Φ''.*

Furthermore, the pushout is unique (up to renaming).

See, e.g., [7], for a proof. The push-out is summarised in Figure 2, which is used throughout the paper.

Proposition 1. *In the push-out in Figure 2, if $x' \in \Phi' = (S', \Sigma', \Pi')$ such that there exist $x_L \in \Phi_L$ and $x_R \in \Phi_R$ with $h'_R(x_R) = x' = h'_L(x_L)$, then there exists $x \in \Phi$ such that $h_L(x) = x_L$ and $h_R(x) = x_R$.*

See, e.g. [7], for the proof.

Theorem 2. *If $\mathcal{T}_R$, $\mathcal{T}_L$ and $\mathcal{T}_0$ are models of $\Phi_R = (S_R, \Sigma_R, \Pi_R)$, $\Phi_L = (S_L, \Sigma_L, \Pi_L)$ and respectively $\Phi_0 = (S_0, \Sigma_0, \Pi_0)$ such that $\mathcal{T}_R{\restriction_{h_R}} = \mathcal{T}_L{\restriction_{h_L}} = \mathcal{T}_0$, there exists a unique model $\mathcal{T}'$ of Φ' such that $\mathcal{T}'{\restriction_{h'_L}} = \mathcal{T}_L$ and $\mathcal{T}'{\restriction_{h'_R}} = \mathcal{T}_R$.*

The proof can be found in our accompanying technical report ([3]).

3 Matching Logic Syntax and Semantics

We introduce notation used throughout the paper and discuss the recently introduced *matching logic* [17, 16], a language-parametric logic for reasoning about program configurations, and its use in 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.

We first introduce **matching logic signatures** (ML signatures), which extend FOL signatures by fixing a sort of program **configurations**.

Definition 4. *A matching logic signature is a tuple (Cfg, S, Σ, Π), where (S, Σ, Π) is a FOL signature and $Cfg \in S$.*

Example 4. Recall the first-order signature $\Phi_I = (S_I, \Sigma_I, \Pi_I)$ in Example 1. We have that $(\texttt{CfgI}, S_I, \Sigma_I, \Pi_I)$ is a matching logic signature. Note that ground instances of sort `CfgI` represent actual configurations of `IMP` programs.

Matching logic formulae extend FOL formulae with terms of sort *Cfg* as atomic formulae called **basic patterns**:

Definition 5. *Given a matching logic signature (Cfg, S, Σ, Π), the following are **matching logic formulae** (ML formulae) over (Cfg, S, Σ, Π) and the set of sorted variables Var: $\varphi ::= \neg\varphi, \varphi \wedge \varphi, \exists x.\varphi, \pi$ where $\pi \in T_{Cfg}(Var), x \in Var$.*

Example 5. Continuing Example 4, $\varphi = \langle \texttt{skip}, \texttt{x} \mapsto x, \texttt{y} \mapsto y \rangle \wedge x >_{Int} 10$ is a matching logic formula over the matching logic signature $(\texttt{CfgI}, S_I, \Sigma_I, \Pi_I)$. Note that `x` and `y` (written in `teletype` font) are program variables (therefore constant symbols in Σ_I, while x and y (written in *italics*) are variables. Intuitively, and as we will see later on, the formula above denotes `IMP` configurations that have terminated (only the instruction `skip` is left in the code to execute) and in which the program variable `x` is mapped to an integer strictly greater than 10 and the program variable `y` is mapped to an integer y that is unconstrained.

Models of ML signatures are simply FOL models:

Definition 6. *We say that $\mathcal{T}$ is a **matching logic model** of (Cfg, S, Σ, Π) if $\mathcal{T}$ is a first order model of (S, Σ, Π).*

Example 6. The model $\mathcal{T}$ defined in Example 2 is also a model of the matching logic signature $(\texttt{CfgI}, S_I, \Sigma_I, \Pi_I)$.

In what follows, we fix a model $\mathcal{T}$ of (Cfg, S, Σ, Π). Elements of $[\![Cfg]\!]_{\mathcal{T}}$ are called concrete **configurations**. We represent concrete configurations by $\gamma, \gamma', \gamma_1$ and variations thereof. Valuations $\rho : Var \to \mathcal{T}$ of matching logic are simply valuations of the corresponding first order logic. The satisfaction relation of matching logic is defined between pairs (γ, ρ) of configurations and valuations and ML formulae φ as follows:

Definition 7. *The **matching logic satisfaction** relation $\models$ (written as $(\gamma, \rho) \models \varphi$ and read as (γ, ρ) is a **model** of φ) is defined inductively as follows:*

1. $(\gamma, \rho) \models \neg\varphi'$ *if it is not true that* $(\gamma, \rho) \models \varphi'$
2. $(\gamma, \rho) \models \varphi_1 \wedge \varphi_2$ *if* $(\gamma, \rho) \models \varphi_1$ *and* $(\gamma, \rho) \models \varphi_2$
3. $(\gamma, \rho) \models \exists x.\varphi'$*, where x is of sort s, if there exists $e \in [\![s]\!]_{\mathcal{T}}$ such that $(\gamma, \rho[e/x]) \models \varphi'$ (where $\rho[e/x]$ is the valuation obtained from ρ by updating the value of x to be e).*
4. $(\gamma, \rho) \models \pi$*, where π is a basic pattern if $\rho(\pi) = e$.*

Example 7. We continue Example 5, where we defined $\varphi = \langle \texttt{skip}, \texttt{x} \mapsto x, \texttt{y} \mapsto y \rangle \wedge x >_{Int} 10$. Let ρ be a valuation where $\rho(x) = 12$ and $\rho(y) = 3$. Let $\gamma = \langle \texttt{skip}, \texttt{x} \mapsto 12, \texttt{y} \mapsto 3 \rangle$. We have that $(\gamma, \rho) \models \varphi$. Considering $\gamma' = \langle \texttt{skip}, \texttt{x} \mapsto 3, \texttt{y} \mapsto 13 \rangle$ and a valuation ρ' with $\rho'(x) = 3$ and $\rho'(y) = 13$, we have that $(\gamma', \rho') \not\models \varphi$ because the condition $x >_{Int} 10$ is not satisfied. Furthermore, if $\gamma'' = \langle \texttt{skip}, \texttt{x} \mapsto 3, \texttt{y} \mapsto 13 \rangle$ and ρ'' is a valuation with $\rho''(x) = 7$ and $\rho''(y) = 13$, we have that $(\gamma'', \rho'') \not\models \varphi$ because γ'' will not match against the basic pattern $\langle \texttt{skip}, \texttt{x} \mapsto x, \texttt{y} \mapsto y \rangle$ with the valuation ρ'' (the valuation ρ'' assigns 7 to the variable x, while x should be 3 due to matching).

Definition 8. *A **matching logic semantic domain** for a language is a tuple $(Cfg, S, \Sigma, \Pi, \mathcal{T})$, where (Cfg, S, Σ, Π) is a matching logic signature and $\mathcal{T}$ a matching logic model of (Cfg, S, Σ, Π).*

Example 8. Assuming $\mathcal{T}_I$ is the model in Example 6, we have that $(\texttt{CfgI}, S_I, \Sigma_I, \Pi_I, \mathcal{T}_I)$ is a matching logic semantic domain for the `IMP` language.

Note that the matching logic semantic domain fixes the abstract syntax of the language (programs are first-order terms of sort Cfg) and the configuration space (given by the model $\mathcal{T}$). However, the matching logic semantic domain does not say anything about the dynamic behavior of configurations. This is the role of the matching logic semantics. A matching logic semantics for a programming language extends the matching logic semantic domain by the addition of several reachability rules:

Definition 9. *A **reachability rule** is a pair $\varphi \Rightarrow \varphi'$ of matching logic formulae.*

Example 9. Let us consider the rule $\langle \texttt{skip;}s, m \rangle \Rightarrow \langle s, m \rangle$. In the rule above, s is a variable of sort `Stmt` and m is a variable of sort `Map{Var,Int}`. It describes what happens in the `IMP` language when the code to execute is a sequence composed of the `skip` instruction and another statement s. The `skip` instruction is simply dissolved and the sequence is simply replaced by s. The environment (captured in the variable m) is not changed during this step.

Definition 10. *A **matching logic semantics** for a language is a tuple $(Cfg, \Sigma, \Pi, \mathcal{T}, \mathcal{A}, \rightarrow_{\mathcal{T}})$, where $(Cfg, \Sigma, \Pi, \mathcal{T})$ is matching logic semantic domain, $\mathcal{A}$ a set of reachability rules 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 $(\gamma', \rho) \models \varphi'$.*

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

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

Fig. 3. Matching logic semantics of IMP as a set $\mathcal{A}_I$ of reachability rules (schemata). *op* ranges over the binary function symbols and op_{Int} is their denotation in $\mathcal{T}_I$.

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

where $C\ \texttt{::=}\ \_ \mid C\ \texttt{op}\ e \mid \texttt{if}\ C\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid C\ \ e \mid v\ \ C$

Fig. 4. Matching logic semantics of FUN as a set $\mathcal{A}_F$ of reachability rules schemata. *op* ranges over the binary function symbols and op_{Int} is their denotation in $\mathcal{T}_F$

Example 10. Figures 3 and 4 presents the set of reachability rules $\mathcal{A}_{\texttt{I}}$ and $\mathcal{A}_{\texttt{F}}$ required to define the IMP and, respectively, the FUN languages.

As discussed in [17], 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 merged together with the needed mathematical domains, and the relation $\rightarrow_{\mathcal{T}}$ is precisely the transition relation defined by the operational semantics. Fig. 3 and Fig. 4 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. 3 and 4 are schemata in meta-variable C (the evaluation context).

Given a matching logic language semantics given as a set of reachability rules, it is possible to derive other reachability rules that "hold" as consequences of the initial set of rules. This is captured by the following definition.

Definition 11. *Given a matching logic semantics* $(Cfg, \Sigma, \Pi, \mathcal{T}, \mathcal{A}, \rightarrow_{\mathcal{T}})$*, we say that* $\varphi \rightarrow^* \varphi'$ *(resp.* $\varphi \rightarrow^+ \varphi'$*) is a semantic consequence of* $\mathcal{A}$*, and we write* $A \models \varphi \rightarrow^* \varphi'$ *(resp.* $A \models \varphi \rightarrow^+ \varphi'$*), if for any* $\gamma, \gamma' \in [\![Cfg]\!]_{\mathcal{T}}$*, for any valuation* ρ *such that* $(\gamma, \rho) \models \varphi$ *and* $(\gamma, \rho) \models \varphi'$*, we have that* $\gamma \rightarrow^*_{\mathcal{T}} \gamma'$ *(resp.* $\gamma \rightarrow^+_{\mathcal{T}} \gamma'$*).*

Example 11. In the set of rateability rules A_I for the IMP languages (given in Figure 3), if we let $\texttt{SUM} \equiv \texttt{while i <= n do (s := s + i; i := i + 1)}$ be the program that computes the sum of all numbers between `i` and `n`, then we have

$$A_I \models \langle \texttt{SUM, n} \mapsto n, \texttt{i} \mapsto 0; \texttt{s} \mapsto 0\rangle \wedge n \geq_{Int} 0 \rightarrow^+ \langle \texttt{skip, } [\texttt{n} \mapsto n; \texttt{i} \mapsto n+1; \texttt{s} \mapsto n(n+_{Int}1)/_{Int}2]\rangle.$$

Intuitively the above reachability rule that is a semantic consequence of the IMP set of leachability rules claims that the program `SUM` indeed computes the sum of the numbers 1 upto n.

We have previously shown (see [17] and subsequent papers) that there exists a sound and (relatively) complete proof system for establishing semantic consequences such as the above. In this article, we assume that such a system is available as an oracle to our proof system for program equivalence.

4 Aggregation of Matching Logic Semantic Domains

In this section we show how, given the matching logic semantic domains for two languages, we can construct a matching logic semantic domain for the **aggregation** of the two languages. The aggregation of two languages is a new language in which programs consists of pairs of programs in the two languages. The challenge is how to construct the domain such that sorts that are common in the two languages (i.e. the sort of integers) has a common interpretation in the aggregated domain. We rely on pushout construction in Section 2 (Theorem 1) and the amalgamation theorem (Theorem 2) in order to perform the aggregation as expected. This construction involves significant technical and conceptual difficulties and, to our knowledge, it has not been described before.

Let $\mathcal{S}_i = (\mathit{Cfg}_i, S_i, \Sigma_i, \Pi_i, \mathcal{T}_i)$, $i \in \{L, R\}$ be the matching logic semantic domains of two languages, (S_0, Σ_0, Π_0) a matching logic signature, h_L and h_R morphisms from (S_0, Σ_0, Π_0) to (S_L, Σ_L, Π_L) and from (S_0, Σ_0, Π_0) to (S_R, Σ_R, Π_R). Let $\mathcal{T}_L, \mathcal{T}_R, \mathcal{T}_0$ be models of $(S_L, \Sigma_L, \Pi_L), (S_R, \Sigma_R, \Pi_R)$ and respectively (S_0, Σ_0, Π_0) such that $T_L{\restriction_{h_L}} = \mathcal{T}_0 = T_R{\restriction_{h_R}}$. Let $(h'_L, (S', \Sigma', \Pi'), h'_R)$ be the pushout of $(S_L, \Sigma_L, \Pi_L) \xleftarrow{h_L} (S_0, \Sigma_0, \Pi_0) \xrightarrow{h_R} (S_R, \Sigma_R, \Pi_R)$.

By Theorem 2, there exists a unique (S', Σ', Π')-model $\mathcal{T}'$ such that $\mathcal{T}'{\restriction_{h'_L}} = \mathcal{T}_L$ and $\mathcal{T}'{\restriction_{h'_R}} = \mathcal{T}_R$. We define now the **aggregation** of the two matching logic semantic domains. We let $\mathcal{S} = (\mathit{Cfg}, S, \Sigma, \Pi, \mathcal{T})$, where

- Cfg is a new distinguished sort;
- $S = S' \cup \{\mathit{Cfg}\}$
- $\Sigma = \Sigma' \cup \{\langle \_, \_\rangle : h_L(\mathit{Cfg}_L) \times h_R(\mathit{Cfg}_R) \rightarrow \mathit{Cfg}, pr_i : \mathit{Cfg} \rightarrow \mathit{Cfg}_i,\ i \in \{L, R\}\}$;
- $\Pi = \Pi'$;
- $\mathcal{T}_{\mathit{Cfg}} = {\mathcal{T}'}_{h'_L(\mathit{Cfg}_L)} \times {\mathcal{T}'}_{h'_R(\mathit{Cfg}_R)}$
- $\mathcal{T}_{\langle \_,\_\rangle}(\gamma_L, \gamma_R) = (\gamma_L, \gamma_R)$, $\mathcal{T}_{pr_L}((\gamma_L, \gamma_R)) = \gamma_L$, $\mathcal{T}_{pr_R}((\gamma_L, \gamma_R)) = \gamma_R$.
- $\mathcal{T}_o = {\mathcal{T}'}_o$ for any other object $o \in S \cup \Sigma \cup \Pi$.

We define a new matching logic semantic domain $\mathcal{S}'_i = (h'_i(\mathit{Cfg}_i), S', \Sigma', \Pi', \mathcal{T}')$ for each $i \in \{L, R\}$. The matching logic semantic domain $\mathcal{S}'_i$ is the embedding of

$\mathcal{S}_i$ into S. The main difference between $\mathcal{S}_i$ and $\mathcal{S}'_i$ is that $\mathcal{S}'_i$ works in a slightly larger algebra that contains symbols from the other language. However, since the matching logic semantics rules do no mention these additional symbols, executions of programs in $\mathcal{S}_i$ coincide with executions of programs in $\mathcal{S}'_i$. In the rest of this section we formally show that this is indeed the case and we establish relations between executions of the aggregate language and the individual languages.

Remark 1. Let $i \in \{L, R\}$. For every valuation $\rho : Var \to \mathcal{T}_i$, we define $h(\rho)$ to be the valuation $h(\rho) : h(Var) \to \mathcal{T}_i$, with $h(\rho)(x) = \rho(x)$ for all $x \in Var$.

We first show that applying the morphism h'_i on both the matching logic formula and valuation does not change the matching logic satisfaction relation.

Proposition 2. *For any pattern π and any valuation ρ, $\rho(\pi) = h(\rho)(h(\pi))$.*

Let $\gamma_i \in [\![h'_i(Cfg_i)]\!]_{\mathcal{T}'}$ be a configuration and φ_i a matching logic formula over (S_i, Σ_i, Π_i) and the set of variables Var, for each $i \in \{L, R\}$. Note that the same set of variables Var is used for both semantic domains S_L and S_R.

Lemma 1. *For all valuations $\rho : Var \to \mathcal{T}_i$, $(\gamma_i, h'_i(\rho)) \models h'_i(\varphi_i)$ iff $(\gamma_i, \rho) \models \varphi_i$ (where $i \in \{L, R\}$).*

The above lemma allow us to conclude that executions in $\mathcal{S}_i$ and $\mathcal{S}'_i$ coincide:

Proposition 3. *If $\gamma_i, \gamma'_i \in [\![h'_i(Cfg_i)]\!]_{\mathcal{T}'}$, then $\gamma_i \to_{\mathcal{S}'_i} \gamma'_i$ iff $\gamma_i \to_{\mathcal{S}_i} \gamma'_i$.*

We now establish the connection between matching logic formulae over the aggregate language and the two individual languages. We first define the **left-** and **right-projection** of matching logic formulae.

Definition 12. *Let φ be a (S, Σ, Π)-matching logic formula. For $i \in \{L, R\}$, we define the $(S'^i, \Sigma'^i, \Pi'^i)$-matching logic formula $pr_i(\varphi)$ (for $i = L$, the **left-projection** and for $i = R$, the **right-projection**) to be φ where every term $\langle t_L, t_R \rangle$ of sort Cfg is replaced by t_i.*

We now distinguish a class of matching logic formulae which behave well with respect to the aggregate semantics.

Definition 13. *A (S, Σ, Π)-matching logic formula is **pure** if no term of sort Cfg in the formula appears under a negation.*

For such pure formulae, we establish the following proposition, which connects satisfaction of matching logic formulae over the aggregate language with satisfaction of matching logic formulae over the individuals languages:

Proposition 4. *Let $(\gamma_L, \gamma_R) \in [\![Cfg]\!]_{\mathcal{T}}$ be a configuration. Let φ be a pure matching logic formula with no variables of sort Cfg. For any valuation $\rho : Var \to \mathcal{T}$, we have that $((\gamma_L, \gamma_R), \rho) \models_{\mathcal{S}} \varphi$ iff $(\gamma_L, \rho) \models_{\mathcal{S}'_L} pr_L(\varphi)$ and $(\gamma_R, \rho) \models_{\mathcal{S}'_R} pr_R(\varphi)$.*

Note that all of the matching logic formulae that we have used so far are pure. In fact, in order to define programming language semantics, which have been shown to be written as sets of rewrite rules of the form $a \Rightarrow b$ *if* c [17], only pure matching logic formulae are needed. In the rest of this article, we will assume that we only deal with such formulae.

5 Specifying Equivalent Programs

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

Definition 14. *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 notation extends to sets E of patterns, written $[\![E]\!]$, as expected:

$$[\![E]\!] = \cup_{\varphi \in E} [\![\varphi]\!].$$

Example 12. The following set

$$E = \{\exists i. \langle \langle \texttt{skip}, (\texttt{x} \mapsto i, \_) \rangle, \langle j \rangle \rangle \wedge i =_{Int} j\} \tag{1}$$

containing one matching logic formula, captures in its denotation 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. Note that in the above pattern, _ is an anonymous variable meant to capture all of the variable bindings other than `x`.

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.

6 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`.

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 15. *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 (i.e. $\gamma_C \rightarrow_{\mathcal{T}} \gamma_C^1 \rightarrow_{\mathcal{T}} \ldots \rightarrow_{\mathcal{T}} \gamma_C^i \rightarrow_{\mathcal{T}}$ for any natural number i and any $C \in \{L, R\}$);*
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 13. 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. 6 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.

6.1 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. 5 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. It 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 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 [17], but any other valid reasoning will also work.

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

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

Fig. 5. Mutual Equivalence Proof System. We use $\varphi \Rightarrow^{*} \varphi'$ as syntactic sugar for $A_L \models pr_L(\varphi) \to^{*} pr_L(\varphi')$ and $\mathcal{A}_R \models pr_R(\varphi) \to^{*} pr_R(\varphi')$ and $\varphi \Rightarrow^{+} \varphi'$ as syntactic sugar for $A_L \models pr_L(\varphi) \to^{+} pr_L(\varphi')$ and $A_L \models pr_R(\varphi) \to^{+} pr_R(\varphi')$.

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. 6) 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 consinder. Typically, Case Analysis is used to branch the proof when the two programs also branch. In the proof tree below (in Fig. 6), 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 call 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.

Theorem 3 (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. 5 then $\models \varphi \Downarrow^{\infty} E$.*

In order to prove the above theorem, we need several intermediate steps that follow. 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 16. *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 17. *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 18. *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 19. *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 2. *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 3 (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 $\models F \Downarrow^{\infty} E$.

It lies at the core of the proof for Theorem 3, which can be found in our accompanying technical report [3].

6.2 Example

We next show the proof tree for the equivalence of the two Collatz programs in Fig. 6. 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 the variable `c`. The set $E = \{\exists i. \langle \texttt{skip}, (\texttt{c} \mapsto i, \_)\rangle, \langle i \rangle\}$ defined in Equation 1 captures 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. 6 shows that the two programs are equivalent.

$\texttt{PGM}_L := \texttt{c := 1; LOOP}_L$

$\texttt{LOOP}_L := \texttt{while (n != 1) c := c + 1; if (n \% 2 != 0) then n := 3 * n + 1 else n := n / 2}$

$\texttt{PGM}_R(\texttt{n}) := \texttt{letrec f n = LOOP}_R \texttt{ in f(n)}$

$\texttt{LOOP}_R := \texttt{1 + if (n != 1) then if (n \% 2 != 0) then f(3 * n + 1) else f(n / 2) else 0}$

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

1.	$\vdash (\langle \texttt{skip, c} \mapsto i, \_ \rangle, \langle i \rangle)$	$\Downarrow^\infty E$	Axiom
2.	$\vdash (\langle \texttt{skip, c} \mapsto i, \_ \rangle, \langle i \rangle)$	$\Downarrow^\infty E \cup \{\varphi\}$	Axiom
3.	$\vdash (\langle \texttt{skip, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \rangle)$	$\Downarrow^\infty E$	Conseq(1)
4.	$\vdash (\langle \texttt{skip, n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \rangle)$	$\Downarrow^\infty E \cup \{\varphi\}$	Conseq(2)
5.	$\vdash (n = 0 \wedge \langle \texttt{LOOP}_L\texttt{ , n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle)$	$\Downarrow^\infty E$	Step(3)
6.	$\vdash \exists i, n.(n = 0 \wedge \langle \texttt{LOOP}_L\texttt{ , n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle)$	$\Downarrow^\infty E \cup \{\varphi\}$	Step(4)
7.	$\vdash \exists i, n.(n > 0 \wedge \langle \texttt{LOOP}_L\texttt{ , n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle)$	$\Downarrow^\infty E \cup \{\varphi\}$	Axiom
8.	$\vdash \exists i, n.(n \geq 0 \wedge \langle \texttt{LOOP}_L\texttt{ , n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle)$	$\Downarrow^\infty E \cup \{\varphi\}$	Conseq(CA(6, 7))
9.	$\vdash \exists i, n.(n > 0 \wedge \langle \texttt{LOOP}_L\texttt{ , n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle)$	$\Downarrow^\infty E$	Circularity (8)
10.	$\vdash \exists i, n.(n \geq 0 \wedge \langle \texttt{LOOP}_L\texttt{ , n} \mapsto n, \texttt{c} \mapsto i \rangle, \langle i \texttt{ + LOOP}_R \rangle)$	$\Downarrow^\infty E$	Conseq(CA(5, 9))
11.	$\vdash \exists i, n.(n \geq 0 \wedge \langle \texttt{PGM}_L\texttt{ , n} \mapsto n \rangle, \langle \texttt{PGM}_R(n) \rangle)$	$\Downarrow^\infty E$	Step (10)

Fig. 6. 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. To make reading easier, the existential quantifiers ($\exists i, n.$) in each step (1 to 11) are skipped for brevity.

7 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.This kind of symbolic

programs are also considered in [12] but for a different notion of bisimulation-based program equivalence.

Related Work. It was first remarked by Hoare in [8] that program equivalence might be easier than program correctness. Among the recent works on equivalence we mention [6, 5, 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 [6] 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 [6] proves equivalence for LPL programs. On the other hand, [6] 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 (e.g. CompCert [11]), which is incomparable to our work since it produces computer-checked proofs of equivalence for a particular language, while our own work produces proofs (not computer-checked) of equivalence for any language. Another approach is the individual verification of each compilation [14] (we only cite two of the most relevant recent works). Other work targets specific classes of languages: functional [15], microcode [1], CLP [4]. 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. However, our approach is better, since our proof system works directly with the language semantics; therefore there is no need to trust the compiler from the original language to Boogie. Finally, our own related work [13] 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 15) 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 [17], 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. Ciobaca, D. Lucanu, V. Rusu, and G. Rosu. A language independent proof system for mutual program equivalence. Technical Report TR14-01, Faculty of Computer Science, May 2014.
4. S. Craciunescu. Proving the equivalence of CLP programs. In *ICLP, LNCS 2401*, pages 287–301, 2002.
5. B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*. To appear.
6. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.
7. A. E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In *AMAST*, pages 132–147, 1996.
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. D. Lucanu and V. Rusu. Program equivalence by circular reasoning. In *IFM*, Lecture Notes in Computer Science, pages 362–377. Springer, 2013.
14. G. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94. ACM, 2000.
15. A. Pitts. Operational semantics and program equivalence. In *Applied Semantics Summer School, LNCS 2395*, pages 378–412, 2002.
16. G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST, LNCS 6486*, pages 142–162, 2010.
17. G. Roşu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.