



HAL
open science

A Language-Independent Proof System for Mutual Program Equivalence

Stefan Ciobaca, Dorel Lucanu, Vlad Rusu, Grigore Rosu

► **To cite this version:**

Stefan Ciobaca, Dorel Lucanu, Vlad Rusu, Grigore Rosu. A Language-Independent Proof System for Mutual Program Equivalence. Formal Aspects of Computing, 2016. hal-01245528v1

HAL Id: hal-01245528

<https://inria.hal.science/hal-01245528v1>

Submitted on 17 Dec 2015 (v1), last revised 25 Jan 2016 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Language-Independent Proof System for Mutual Program Equivalence

Ștefan Ciobâcă¹, Dorel Lucanu¹, Vlad Rusu² and Grigore Roșu³

¹Faculty Of Computer Science, “Alexandru Ioan Cuza” University, Iași, Romania,

²Inria Lille, France

³University of Illinois at Urbana-Champaign, USA

Abstract.

Two programs are mutually equivalent if, for the same input, either they both diverge or they both terminate with the same result. 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. The Collatz sequence is an interesting case study since it is not known whether the sequence terminates or not; nevertheless, our proof system shows that the two programs are mutually equivalent (even if we cannot establish termination or divergence of either one).

1. Introduction

Two terminating programs are equivalent if the final states that they reach are similar (they contain the same result). Nontermination can be incorporated in equivalence in several ways. If termination is ignored, we obtain **partial equivalence**: two programs are partially equivalent if, for all inputs on which they both terminate, they return the same result. A program that never terminates is therefore partially equivalent to any other program. If both programs are required to terminate, we obtain **total equivalence**: two programs are totally equivalent if, for the same input, they both terminate and they return the same result. Finally, we have **mutual equivalence**, which we explore in this article and which we consider to be the most appropriate notion of equivalence for general purposes. Mutual equivalence is an equivalence relation that is also known in the literature as full equivalence [9]. Two programs are said to be **mutually equivalent** iff, when given the same input, 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.

As a motivating example for our paper, we consider two programs that both compute the Collatz sequence (also known as the $3x + 1$ sequence) starting from an arbitrary integer n . In the Collatz sequence, after a number x , we have $x/2$ if x is even or $3x + 1$ if x is odd. For example, the Collatz sequence that starts with 6 is: 6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \dots . It is conjectured that, if we start with any positive integer, the sequence reaches 1 in a finite number of steps. Consider the following programs, which both compute the number of steps in the Collatz sequence from n up to the point where 1 is reached:

| | |
|---|--|
| <pre> c := n; n := 1; while (c != 1) n := n + 1; if (c % 2 != 0) then c := 3 * c + 1 else c := c / 2 </pre> | <pre> μf. λn. λa. if n != 1 then if n % 2 != 0 then f (3 * n + 1) (a + 1) else f (n / 2) (a + 1) else a </pre> |
|---|--|

The program on the left -hand side is written in an imperative language with while-loops and assignment statements (we will call this language IMP). The initial configuration of the program should map the program variable n (program variables are written using `teletype` font) to the actual number n (a mathematical variable) that starts the Collatz sequence. The program on the right-hand side is written in a functional language (lambda-calculus extended with a fix-point operator μ). The functional program is actually a recursive function that takes two¹ parameters: n – the current value in the sequence – and a – an accumulator that remembers how many steps we took so far. The initial configuration of the second program consists of the recursive function itself called on the two arguments n (the number which starts the sequence) and 1 (the length of the sequence so far).

Obviously the two programs compute the same thing. The imperative program will end up with the length of the Collatz sequence in the program variable n and the function program will reduce to a value that is exactly the length of the Collatz sequence. However, it is not known if the programs terminate on all inputs (it is only conjectured) and such a proof does not seem to be in reach of present-day mathematics. Paul Erdős even offered a 500\$ reward [11] for anyone who would prove or disprove the conjecture. Therefore, a proof of mutual equivalence of the two programs must deal with the fact that once the input n has been fixed (arbitrarily), it is not known if the programs terminate or not and therefore the proof cannot proceed by separating the proof into a proof of termination and another proof of partial equivalence. We deal with this difficulty by relying on a proof rule inspired from co-induction that we call CIRCULARITY:

$$\text{CIRCULARITY} \frac{\models \varphi_1 \rightarrow_1^+ \varphi'_1 \quad \models \varphi_2 \rightarrow_2^+ \varphi'_2 \quad \vdash \langle \varphi'_1, \varphi'_2 \rangle \Downarrow^\infty E \cup \{ \langle \varphi_1, \varphi_2 \rangle \}}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E}$$

CIRCULARITY is inspired from previous work [7], where we use a form of circularity to prove partial correctness of programs. In previous work, the circularity rule (for partial correctness) was used to prove properties of programs that contain repetitive behaviours such as loops, recursive functions, jumps, etc. Even if partial equivalence can be reduced to partial correctness [3], mutual equivalence presents additional difficulties that we have to deal with. In particular, the circularity rule for partial correctness is un-sound in case the program diverges.

Our CIRCULARITY rule (introduced in Figure 9 on page 20) allows us to implicitly postulate synchronisation points in the two programs. To prove that two programs φ_1 and φ_2 either both diverge or reach a set of good states E (known to be equivalent), we make progress from each of the programs ($\varphi_1 \rightarrow_1^+ \varphi'_1$ and $\varphi_2 \rightarrow_2^+ \varphi'_2$) and we show that in the new pair of programs ($\langle \varphi'_1, \varphi'_2 \rangle$) either both programs diverge or they reach the set of good states E or they end up in the same state as they started ($\langle \varphi_1, \varphi_2 \rangle$). The CIRCULARITY

¹ Of course, formally, the recursive function bound to the name f by the μ operator only takes one parameter (n) and returns another function which takes a as parameter; however, for our purposes, we can think of the curried functions as one function with two parameters.

rule therefore allows us to prove divergence or convergence to the same result of the two programs simultaneously. Its soundness is critically based on the fact that the two configurations φ_1 and φ_2 are required to make progress before reaching $\langle \varphi_1, \varphi_2 \rangle$ again. The soundness of the rule is based on a Circularity Principle (Lemma 4) that we state and prove in Section 7.

There are other ways to show mutual equivalence (e.g. [9]). However, we adhere to the belief in [18] that formal analyses and proofs of programs (proofs of partial/total correctness, proofs of equivalence, etc.) should be based on the operational semantics of the programming language. In particular, each programming language should have an operational semantics that everyone agrees on and trusts and which completely defines the behaviour of programs from that language. Any approach to prove programs should be shown to be sound with respect to that particular operational semantics, or, even better, should be parametrized by it. This way, two different formal verification tools will always return the same answer for the same verification task, and, in case a bug is present in the tools, it can be traced back to the semantics of the language. The proof system that we present here is parametric in the operational semantics of the programming languages used. We assume that the operational semantics of the two languages is given as a set of reachability rules (which generalize rewrite rules) that can faithfully capture all major styles of operational semantics [27]. Our proof system is shown to be sound once and it can be used with any particular language that has an operational semantics without redoing the soundness proof.

A difficulty in expressing equivalence properties when the two programs are written in different languages is to relate values from one program to the other. In particular, assume that the two languages both have an integer datatype, but it is called *Int* in one language and *Integer* in the other. We overcome this difficulty by identifying a common signature and model for the parts of the languages that are shared (e.g. Strings, Integers, Booleans, etc.). This common part is also a parameter of our construction. Once the common part has been identified, we can **aggregate** the two languages into a single language where programs consist of pairs of programs from the initial languages. In the aggregated language, the common sorts have the same name and are interpreted in the same way. The aggregation relies on the existence of push-outs for the signatures and on model amalgamation for the model. Section 4 is devoted to this construction.

To summarise, 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 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.

In Section 2 we introduce preliminaries needed in the rest of the paper. This includes matching logic, a logic introduced by Roşu [22], that we use in order to reason about program configurations. A critical advantage of matching logic is that it is language independent, being capable of reasoning with arbitrary program configurations. We also show here how reachability rules (based on matching logic) can be used to give operational semantics to programming languages. In Section 3 we introduce the semantics of two languages, IMP and FUN, that we use throughout the paper. We explain how the syntax of each language can be encoded as a many-sorted matching logic signature. The semantics of the two languages are given as sets of axioms (reachability logic formulae or simply reachability rules). The reachability rules describes how a program configuration advances in one step into another configuration.

In Section 4 we show how to aggregate two programming languages. The aggregated language will have as configurations pairs of configurations from the initial languages. We show that the signature of the aggregated language can be constructed as a push-out of a diagram and that the configuration model can be obtained by amalgamation from the configuration models of the two languages. In Section 5 we investigate several ways in which we can combine the axioms (the operational semantics) of the two languages. We have several choices of operational semantics for the aggregated language: the two programs can take turns to advance one after the other or they can advance in parallel, or both. We show how each of these possibilities can be specified using reachability rules.

Section 6 shows how our formalism can be used to specify equivalent programs. Here we define the notion of mutual equivalence and we illustrate it on a few examples. In Section 7, we introduce our 5-rule proof system for mutual equivalence, we show that it is sound and we apply it in order to prove the equivalence of

two programs that both compute the Collatz sequence. Section 8 discusses related work and concludes the paper.

Acknowledgments This paper is supported by the Sectorial Operational Programme Human Resource Development (SOP HRD), financed from the European Social Fund and by the Romanian Government under the contract number POSDRU/159/1.5/S/137750 as well as by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2014-2015, the NSF grants CCF-1218605, CCF-1318191 and CCF-1421575, and the DARPA grant under agreement number FA8750-12-C-0284.

Note to editors/reviewers This article is the journal version of the paper with the same title that was presented at ICFEM 2014 [4]. The main differences between the ICFEM version and this paper are:

1. all the proofs are included;
2. the present approach now uses of the latest version of matching logic [22], which allows us to be more general while describing certain patters more cleanly; for this we had to redo the presentation and proofs;
3. aggregation of the language definitions is now formalised completely; to accommodate the most general version of matching logic, we had to create a new proof of model amalgamation since existing proofs do not longer apply;
4. the examples are now fully included (we had to skip the formal definition of FUN in the conference version due to space constraints);
5. we also show how to aggregate the semantics of the two languages in three different ways and we prove them to be correct;
6. significant overall clean-up and simplification of the paper, including the explanatory text and proofs.

2. Preliminaries

Here we introduce preliminary notions and our notations for many-sorted sets and algebraic signatures, for matching logic and its use in programming language semantics.

2.1. Many-sorted Sets and Algebraic Signatures

We assume that the reader is familiar with the main notions and concepts used in many sorted algebra. Our purpose in this subsection is to introduce our notation for these notions and concepts.

If S is a set, whose elements are called **sorts**, an S -sorted set is an S -indexed collection of disjoint sets $A = \bigcup_{s \in S} A_s$, where $s \neq s'$ implies $A_s \cap A_{s'} = \emptyset$. We often write $a \in A$ or $a:s$ for $a \in A_s$. If A and B are two S -sorted sets, then:

- $A \subseteq B$ iff $A_s \subseteq B_s$ for all $s \in S$;
- $A \cup B = \bigcup_{s \in S} (A \cup B)_s$, where $(A \cup B)_s = A_s \cup B_s$;
- a relation $R \subseteq A \times B$ is a collection of relations $R = \bigcup_{s \in S} R_s$ with $R_s \subseteq A_s \times B_s$. We often write $(a, b) \in R$ for $(a, b) \in R_s$, where $a \in A_s$, $b \in B_s$;
- in particular, a function $f : A \rightarrow B$ is a collection of functions $f = \bigcup_{s \in S} f_s$ with $f_s : A_s \rightarrow B_s$. We often write $f(a)$ for $f_s(a)$, where $a \in A_s$ and $f_s(a) \in B_s$.

If S is a set of sorts, an **algebraic S -sorted signature** is an $S^* \times S$ -indexed set $\Sigma = \bigcup_{w \in S^*, s \in S} \Sigma_{w,s}$ of **functional symbols**. If $\sigma \in \Sigma_{w,s}$ then w is called the **arity** of σ and s the **sort** of σ . We often write $\sigma : s_1 \times \dots \times s_n \rightarrow s$ for $\sigma \in \Sigma_{s_1 \dots s_n, s}$. If $n = 0$, i.e. $\sigma : \rightarrow s$, then σ is called **constant**.

Given two algebraic signatures (S, Σ) and (S', Σ') , an **algebraic signature morphism** $h : (S, \Sigma) \rightarrow (S', \Sigma')$ consists of two functions $h_{srt} : S \rightarrow S'$ and $h_{sgn} : \Sigma \rightarrow \Sigma'$ such that, for each $\sigma \in \Sigma$, if $\sigma :$

$s_1 \times s_2 \times \dots \times s_n \rightarrow s$ then $h_{sgn}(\sigma) : h_{srt}(s_1) \times h_{srt}(s_2) \times \dots \times h_{srt}(s_n) \rightarrow h_{srt}(s) \in \Sigma'$. We often write $h(s)$ for $h_{srt}(s)$ and $h(\sigma)$ for $h_{sgn}(\sigma)$. It is easy to see that the signatures together with their morphisms form a category *Sig*.

Example 1. Let (S, Σ) be the signature

$$S = \{Bool\},$$

$$\Sigma = \{false, true : \rightarrow Bool, _and_ : Bool \times Bool \rightarrow Bool\},$$

and (S', Σ') be the signature

$$S' = \{Prop, \dots\},$$

$$\Sigma' = \{\perp, \top : \rightarrow Prop, _ \wedge _ : Prop \times Prop \rightarrow Prop, \dots\}$$

where \dots means that (S', Σ') may include more objects (sorts and functional symbols). An example of a signature morphism $h : (S, \Sigma) \rightarrow (S', \Sigma')$ is given by $h(Bool) = Prop$, $h(false) = \perp$, $h(true) = \top$, and $h(_and_) = _ \wedge _$.

2.2. Matching Logic

Matching logic [22, 26] was proposed by Roşu and others for reasoning about program configurations in a language-parametric way. The characteristic feature of matching logic in contrast to FOL is that it makes no distinction between function vs. predicate symbols, its symbols being uniformly used to build *patterns*, which can be regarded both as terms and as predicates. They are regarded as terms when building structure, such as (symbolic) program configurations, and as predicates when stating properties about structure, such as a configuration matching a certain pattern. In this section, we recall the basic concepts of matching logic, as defined in [22].

Definition 1 (Signature). A **matching logic signature** is an algebraic S -sorted signature (S, Σ) .

Definition 2 (Matching Logic Models). Let (S, Σ) be a matching logic signature. An (S, Σ) -**model** consists of:

1. an S -sorted **carrier set** $\bigcup_{s \in S} \llbracket s \rrbracket_M$ such that, for any sort $s \in S$, the carrier set $\llbracket s \rrbracket_M$ of sort s is not empty; and
2. a function $\llbracket \sigma \rrbracket_M : M_{s_1} \times \dots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$ for each symbol $\sigma \in \Sigma_{s_1, \dots, s_n, s}$; the function $\llbracket \sigma \rrbracket_M$ is called the **interpretation** of σ .

In the above definition, $\mathcal{P}(M_s)$ is the power-set of M_s . We extend the domain of $\llbracket \sigma \rrbracket_M$ such that

$$\llbracket \sigma \rrbracket_M(A_1, \dots, A_n) = \bigcup_{a_1 \in A_1, \dots, a_n \in A_n} \llbracket \sigma \rrbracket_M(a_1, \dots, a_n). \quad (1)$$

The power-set $\mathcal{P}(M)$ of the S -sorted set M is the S -sorted set $\bigcup_{s \in S} \mathcal{P}(M_s)$ of power-sets.

Therefore, unlike the FOL models, the matching logic models interpret symbols as relations, that is, as functions returning *sets* of values. This is to account for the fact that patterns can be matched by many different values. For example, if $list : Nat \rightarrow Map$ is a symbol for describing linked list patterns starting with a given location (regarded as a natural number) in the program heap (regarded as a map), then $list(7)$ consists of a set of infinitely many maps, each representing a linked list starting at location 7.

Example 2. Let (S', Σ') be the signature defined in Example 1. An example of (S', Σ') -model M' is defined as follows: $\llbracket Prop \rrbracket_{M'} = \{0, 1\}$, $\llbracket \perp \rrbracket_{M'} = \{0\}$, $\llbracket \top \rrbracket_{M'} = \{1\}$ and $\llbracket _ \wedge _ \rrbracket_{M'}$ is addition modulo 2, i.e. $\llbracket _ \wedge _ \rrbracket_{M'}(a, b) = \{(a + b) \% 2\}$. This model example is very simple and close to many-sorted algebra. See [22] for more complex models that go beyond the many-sorted algebra.

Definition 3 (Reduct of a Model). Let $h : (S, \Sigma) \rightarrow (S', \Sigma')$ be a matching logic signature morphism and M' a (S', Σ') -model. The **reduct** of M' , written as $M' \upharpoonright_h$, is defined as follows:

$$\llbracket s \rrbracket_{M' \upharpoonright_h} = \llbracket h(s) \rrbracket_{M'}$$

$$\llbracket \sigma \rrbracket_{M' \upharpoonright_h}(a_1, \dots, a_n) = \llbracket h(\sigma) \rrbracket_{M'}(a_1, \dots, a_n)$$

Example 3. Let h be the signature morphism defined in Example 1 and M' the (S', Σ') -model defined in Example 2. Then $M' \upharpoonright_h$ is the model M defined by: $\llbracket Bool \rrbracket_M = \{0, 1\}$, $\llbracket false \rrbracket_M = \{0\}$, $\llbracket true \rrbracket_M = \{1\}$, $\llbracket _and _ \rrbracket_M(a, b) = \llbracket _ \wedge _ \rrbracket_{M'}(a, b) (= \{(a + b) \% 2\})$.

The next definition shows how matching logic's patterns make no distinction between functional and predicate symbols, allowing them to build terms but also to combine them with logical connectives and quantifiers:

Definition 4 (Syntax). Let (S, Σ) be a matching logic signature. For any sort $s \in S$, the set of (S, Σ) -patterns (matching logic (S, Σ) -formulae) of sort s are defined inductively as follows:

$$\begin{array}{l} \varphi_s ::= x \quad \text{if } x \in Var_s \\ \quad \quad \quad \sigma(\varphi_{s_1}, \dots, \varphi_{s_n}) \quad \text{if } \sigma \in \Sigma_{s_1, \dots, s_n, s} \\ \quad \quad \quad \neg \varphi_s \\ \quad \quad \quad \varphi_s \wedge \varphi_s \\ \quad \quad \quad \exists x. \varphi_s \quad \text{for any } x \in Var \end{array}$$

Note that, as opposed to many-sorted FOL, formulae have sorts and function symbols are applied not only to terms, but also to formulae of appropriate sorts. As variables are also formulae, matching logic formulae generalize FOL terms. Also, there is no need for predicate symbols, as we will see further on.

We also consider the following derived constructs:

$$\begin{aligned} \perp_s &\equiv x:s \wedge \neg x:s \\ \top_s &\equiv \neg \perp_s \\ \varphi_1 \vee \varphi_2 &\equiv \neg(\neg \varphi_1 \wedge \neg \varphi_2) \\ \varphi_1 \rightarrow \varphi_2 &\equiv \neg \varphi_1 \vee \varphi_2 \\ \forall x. \varphi &\equiv \neg \exists x. \neg \varphi \end{aligned}$$

Example 4. Let (S, Σ) be the signature

$$\begin{aligned} S &= \{Int, Pred\}, \\ \Sigma &= \{-, +, -, * _ : Int \times Int \rightarrow Int, - \leq _, - < _ : Int \times Int \rightarrow Pred\}. \end{aligned}$$

Examples of formulae are $x + 2$, $x \leq 5$, $\exists x. x + y * 2$, $\forall x. \exists y. x \leq y$, and so on.

The set of patterns of sort s is denoted by $PATTERNS(S, \Sigma)_s$ and the set of patterns (of all sorts) is denoted by $PATTERNS(S, \Sigma)$. For simplicity, we write them $PATTERNS_s$, respectively $PATTERNS$ whenever (S, Σ) and/or s are understood from the context.

Definition 5 (Patterns Translation via a Signature Morphism). A matching logic signature morphism $h : (S, \Sigma) \rightarrow (S', \Sigma')$ is extended to $h : PATTERNS(S, \Sigma) \rightarrow PATTERNS(S', \Sigma')$ as follows:

$$\begin{aligned} h(x:s) &= x:h(s), \\ h(\sigma(\varphi_1, \dots, \varphi_n)) &= h(\sigma)(h(\varphi_1), \dots, h(\varphi_n)) \quad \text{if } \sigma \in \Sigma_{s_1, \dots, s_n, s}, \\ h(\neg \varphi) &= \neg h(\varphi), \\ h(\varphi_1 \wedge \varphi_2) &= h(\varphi_1) \wedge h(\varphi_2), \\ h(\exists x:s. \varphi) &= \exists x:h(s). h(\varphi). \end{aligned}$$

Example 5. In the context of Example 1, $h(x:Bool \text{ and } (true))$ is $x:Prop \wedge \top$.

Note that, in the above example, the same variable x has different sorts in the two formulae. The following definition formalises this mechanism of renaming.

Definition 6. Let $h : (S, \Sigma) \rightarrow (S', \Sigma')$ be a matching logic signature morphism and $Var = \bigcup_{s' \in S'} Var_{s'}$ an S' -sorted set of variables. Then $Var \upharpoonright_h = \bigcup_{s \in S} Var \upharpoonright_{h,s}$, where $Var \upharpoonright_{h,s} = \{x \mid x \in Var_{h(s)}\}$, is an S -sorted set of variables.

Whenever we work in the context of a signature morphism $h : (S, \Sigma) \rightarrow (S', \Sigma')$, we consider Var being an S' -sorted set of variables and the formulae over (S, Σ) are given over $Var \upharpoonright_h$. This convention is generalised

to the case of diagrams in Sig having colimits. Moreover, by abuse of notation, we often use Var instead of $Var \upharpoonright_h$.

Definition 7 (Valuation). Let M be an (S, Σ) -model. An M -**valuation** is any function $\rho : Var \rightarrow M$ of S -sorted sets.

Definition 8 (Valuation Extension). Given an (S, Σ) -model M and an M -valuation ρ , we construct its extension $\bar{\rho} : \text{PATTERNS} \rightarrow \mathcal{P}(M)$ as follows:

1. $\bar{\rho}(x) = \{\rho(x)\}$
2. $\bar{\rho}(\sigma(\varphi_1, \dots, \varphi_n)) = \llbracket \sigma \rrbracket_M(\bar{\rho}(\varphi_1), \dots, \bar{\rho}(\varphi_n))$
3. $\bar{\rho}(\neg\varphi) = M_s \setminus \bar{\rho}(\varphi)$
4. $\bar{\rho}(\varphi_1 \wedge \varphi_2) = \bar{\rho}(\varphi_1) \cap \bar{\rho}(\varphi_2)$
5. $\bar{\rho}(\exists x.\varphi) = \bigcup \{\bar{\rho}'(\varphi) \mid \rho' : Var \rightarrow \mathcal{P}(M), \rho'|_{Var \setminus \{x\}} = \rho|_{Var \setminus \{x\}}\}$

Unlike FOL models, in which a valuation ρ takes terms into a single element of the domain, in matching logic the extension of the valuation, $\bar{\rho}$, maps each formula into a set of elements (a subset of the domain). In particular, for variables, $\bar{\rho}$ returns the singleton set. Note that in the second case in the definition above, we use the notation in Equation (1) and therefore we might obtain several elements in the right-hand side.

Example 6. Let (S, Σ) be the signature defined in Example 4 and M be an (S, Σ) -model, where $\llbracket Int \rrbracket_M = \mathbb{Z}$, $\llbracket Pred \rrbracket_M = \{True\}$, and the operators have the usual interpretation. If $\rho(x) = 3$ and $\rho(y) = 4$, then $\bar{\rho}(x+2) = \{5\}$, $\bar{\rho}(x \leq 5) = \{True\}$, $\bar{\rho}(y < x) = \emptyset$, and $\bar{\rho}(\exists x.x + y * 2) = \mathbb{Z}$.

Definition 9 (Reduct of a Valuation). Let $h : (S, \Sigma) \rightarrow (S', \Sigma')$ be a matching logic signature morphism, M' an (S', Σ') -model, and $\rho : Var \rightarrow M'$ a valuation. The **reduct of ρ** , written $\rho \upharpoonright_h : Var \upharpoonright_h \rightarrow M' \upharpoonright_h$, is given by $\rho \upharpoonright_h(y:s) = \rho(y:h(s))$.

Lemma 1. Let $h : (S, \Sigma) \rightarrow (S', \Sigma')$ be a matching logic signature morphism, M' be a (S', Σ') -model, and $\rho : Var \rightarrow M'$ a valuation. If φ is a (S, Σ) -pattern, then $\bar{\rho}(h(\varphi)) = \bar{\rho} \upharpoonright_h(\varphi)$.

Proof. Recall that, according to our convention, φ is defined over the variables $Var \upharpoonright_h$. We proceed by structural induction on φ :

$$\begin{aligned}
\bar{\rho}(h(x:s)) &= \{\rho(x:h(s))\} = \{\rho \upharpoonright_h(x:s)\} \\
\bar{\rho}(h(\sigma(\varphi_1, \dots, \varphi_n))) &= \llbracket h(\sigma) \rrbracket_{M'}(\bar{\rho}(h(\varphi_1)), \dots, \bar{\rho}(h(\varphi_n))) && \text{(def.}\bar{\rho}\text{)} \\
&= \llbracket h(\sigma) \rrbracket_{M'}(\bar{\rho} \upharpoonright_h(\varphi_1), \dots, \bar{\rho} \upharpoonright_h(\varphi_n)) && \text{(ind.hyp)} \\
&= \llbracket \sigma \rrbracket_{M' \upharpoonright_h}(\bar{\rho} \upharpoonright_h(\varphi_1), \dots, \bar{\rho} \upharpoonright_h(\varphi_n)) && \text{(def.}M' \upharpoonright_h\text{)} \\
&= \bar{\rho} \upharpoonright_h(\sigma(\varphi_1, \dots, \varphi_n)) && \text{(def.}\bar{\rho} \upharpoonright_h\text{)} \\
\bar{\rho}(h(\neg\varphi)) &= M'_{h(s)} \setminus \bar{\rho}(h(\varphi)) && \text{(def.}\bar{\rho}\text{)} \\
&= M' \upharpoonright_{h,s} \setminus \bar{\rho} \upharpoonright_h(\varphi) && \text{(ind.hyp)} \\
&= \bar{\rho} \upharpoonright_h(\neg\varphi) && \text{(def.}\bar{\rho} \upharpoonright_h\text{)} \\
\bar{\rho}(h(\varphi_1 \wedge \varphi_2)) &= \bar{\rho}(h(\varphi_1) \wedge h(\varphi_2)) && \text{(def.}h\text{)} \\
&= \bar{\rho}(h(\varphi_1)) \cap \bar{\rho}(h(\varphi_2)) && \text{(def.}\bar{\rho}\text{)} \\
&= \bar{\rho} \upharpoonright_h(\varphi_1) \cap \bar{\rho} \upharpoonright_h(\varphi_2) && \text{(ind.hyp)} \\
&= \bar{\rho} \upharpoonright_h(\varphi_1 \wedge \varphi_2) && \text{(def.}\bar{\rho} \upharpoonright_h\text{)} \\
\bar{\rho}(h(\exists x:s.\varphi)) &= \bar{\rho}(\exists x:h(s).h(\varphi)) && \text{(def.}h\text{)} \\
&= \bigcup \{\bar{\rho}'(h(\varphi)) \mid \rho' : Var \rightarrow \mathcal{P}(M'), \rho'|_{Var \setminus \{x\}} = \rho|_{Var \setminus \{x\}}\} && \text{(def.}\bar{\rho}\text{)} \\
&= \bigcup \{\bar{\rho}' \upharpoonright_h(\varphi) \mid \rho' : Var \rightarrow \mathcal{P}(M'), \rho'|_{Var \setminus \{x\}} = \rho|_{Var \setminus \{x\}}\} && \text{(ind.hyp)} \\
&= \bigcup \{\bar{\rho}' \upharpoonright_h(\varphi) \mid \rho' \upharpoonright_h : Var \upharpoonright_h \rightarrow \mathcal{P}(M' \upharpoonright_h), \rho' \upharpoonright_h|_{Var \upharpoonright_h \setminus \{x\}} = \rho' \upharpoonright_h|_{Var \upharpoonright_h \setminus \{x\}}\} && \text{(conv.*}\upharpoonright_h\text{)} \\
&= \bar{\rho} \upharpoonright_h(\exists x.\varphi) && \text{(def.}\bar{\rho} \upharpoonright_h\text{)}
\end{aligned}$$

where "conv.* \upharpoonright_h " refers our convention on the variable names among morphisms h . \square

Results similar to Lemma 1, showing how the semantics is preserved along signature morphisms, are given in [2], where the matching logic is organised as a stratified institution. Lemma 1 is used later in proving the correctness of the aggregation.

Definition 10 (Semantics). A model M **satisfies** a pattern $\varphi \in \text{PATTERNS}_s$ of sort s (written $M \models \varphi$) if $\bar{\rho}(\varphi) = M_s$ for all M -valuations $\rho : \text{Var} \rightarrow M$.

Matching logic becomes more expressive in the presence of the **definedness** symbol $[-]_{s_1}^{s_2} \in \Sigma_{s_1, s_2}$ together with a pattern axiom

$$[x:s_1]_{s_1}^{s_2}$$

for each pair of sorts $s_1, s_2 \in S$. The axiom enforces $\llbracket [-]_{s_1}^{s_2} \rrbracket_M(m_1) = \llbracket s_2 \rrbracket_M$ in all models M and for all $m_1 \in M_{s_1}$, which means that for any $\rho : \text{Var} \rightarrow M$, $\bar{\rho}([x]_{s_1}^{s_2})$ is either \emptyset when $\bar{\rho}(\varphi) = \emptyset$ (i.e., φ undefined in ρ), or is $\llbracket s_2 \rrbracket_M$ when $\bar{\rho}(\varphi) \neq \emptyset$ (i.e., φ defined). By convention, we take the freedom to not write the undefinedness symbol whenever s_1 is a boolean or a formula sort, and s_2 can be non-ambiguously inferred from the context.

Using the definedness symbol and the convention above, we may have patterns of the form $(x+y*2) \wedge (x < y)$, which is equivalent to $(x+y*2) \wedge [x < y]_{\text{Pred}}^{\text{Int}}$. To understand the semantics of this pattern we consider two valuations. Let ρ_1 be a valuation such that $\rho_1(x) = 2$ and $\rho_1(y) = 5$. Then

$$\begin{aligned} \bar{\rho}_1(x+y*2 \wedge x < y) &= \bar{\rho}_1(x+y*2 \wedge [x < y]_{\text{Pred}}^{\text{Int}}) \\ &= \bar{\rho}_1(x+y*2) \cap \bar{\rho}_1([x < y]_{\text{Pred}}^{\text{Int}}) \\ &= \{12\} \cap \mathbb{Z} && (\bar{\rho}_1(x < y) = \{\text{True}\}) \\ &= \{12\} \end{aligned}$$

If ρ_2 is a valuation such that $\rho_2(x) = 5$ and $\rho_2(y) = 2$. Then

$$\begin{aligned} \bar{\rho}_2(x+y*2 \wedge x < y) &= \bar{\rho}_2(x+y*2 \wedge [x < y]_{\text{Pred}}^{\text{Int}}) \\ &= \bar{\rho}_2(x+y*2) \cap \bar{\rho}_2([x < y]_{\text{Pred}}^{\text{Int}}) \\ &= \{12\} \cap \emptyset && (\bar{\rho}_2(x < y) = \emptyset) \\ &= \emptyset \end{aligned}$$

The name matching logic comes from the fact that a matching logic formula is **matched** by all elements of the model that make it true. For example $x \wedge (x > 0)$ is matched by all positive integers and the pattern $(x+y*2) / (x < y)$ is matched by precisely all the numbers in the set $\{a+b*2 \mid a, b \in \mathbb{Z}, a < b\}$. Matching logic is crucially important to us because it allows us to reason about program configurations in a language independent way. A complete reference for matching logic can be found in [22].

2.3. Specifying Languages using Matching Logic

Matching logic can express and reason about *static properties* of program configurations. That is, about properties that program configurations should satisfy at a certain place during the execution of a program, without taking the programming language dynamic semantics into account. To reason about *dynamic properties*, we have introduced Reachability Logic in a series of papers [23, 24, 25, 7] as means for specifying the operational semantics of programming languages and for stating reachability properties between states of program executions. Here we recall main definitions that are used in the paper.

Definition 11 (Reachability Logic Syntax). A **reachability rule** of sort s is a pair $\varphi \Rightarrow \varphi'$ of matching logic formulae φ and φ' of sort s .

Example 7. A simple example of reachability rule is the one used to bubble-sort a list:

$$\langle L_1:\text{List}, x:\text{Int}, y:\text{Int}, L_2:\text{List} \rangle \wedge x > y \Rightarrow \langle L_1, y, x, L_2 \rangle$$

Another example is the one specifying the sorting problem:

$$\langle L:\text{List} \rangle \Rightarrow \langle L':\text{List} \rangle \wedge \text{isSorted}(L') \wedge \text{equalAsMultisets}(L, L')$$

A reachability rule $\varphi \Rightarrow \varphi'$ intuitively states that a configuration matching φ advances, in zero or more steps, into a configuration matching φ' . Hence the models of reachability logic are transition systems over elements (typically program configurations or states) of appropriate matching logic models.

Definition 12 (Reachability Logic Models). An (S, Σ) -reachability model N consists of an (S, Σ) -matching logic model M and an S -sorted relation $\rightarrow_N \subseteq M \times M$, i.e., $\rightarrow_N = \bigcup_{s \in S} \rightarrow_N^s$ with $\rightarrow_N^s \subseteq M_s \times M_s$.

Example 8. Consider a model M such that $\llbracket List \rrbracket_M$ is the set of sequences of integers x_1, \dots, x_n and the transition relation \rightarrow_N consisting of pairs

$$x_1, \dots, x_{i-1}, x_i, x_{i+1}, x_{i+2}, \dots, x_n \rightarrow_N x_1, \dots, x_{i-1}, x_{i+1}, x_i, x_{i+2}, \dots, x_n.$$

Note that in this example model, x_i is not constrained to be smaller than x_{i+1} and therefore all permutations of the initial list are reachable.

Definition 13 (Semantics of Reachability Logic). We say that N is a model of $\varphi \Rightarrow \varphi'$ (and we write $N \models \varphi \Rightarrow \varphi'$) if, for all valuations ρ , we have that $\bar{\rho}(\varphi) \times \bar{\rho}(\varphi') \subseteq \rightarrow_N^s$.

Example 9. Let $N = (M, \rightarrow_N)$ be the transition system defined in Example 8 and $\varphi_1 \Rightarrow \varphi'_1$ be the reachability formula expressing a list sorting step given in Example 7. Since the transitions $\bar{\rho}(\varphi_1) \times \bar{\rho}(\varphi'_1)$ are of the form

$$x_1, \dots, x_{i-1}, x_i, x_{i+1}, x_{i+2}, \dots, x_n \rightarrow_N x_1, \dots, x_{i-1}, x_{i+1}, x_i, x_{i+2}, \dots, x_n \text{ iff } x_i > x_{i+1},$$

they are included in \rightarrow_N , and therefore $N \models \varphi \Rightarrow \varphi'$ (even if N allows for more transitions). N is not a model of the second reachability formula (the one specifying the list sorting problem) given in Example 7 because \rightarrow_N does not include always transitions between lists and their sorted versions. However, $N' = (M, \rightarrow_N^*)$ is a model for this second formula, since the model contains the transitions that link a list to its sorted version.

In the following, we will assume that a programming language is defined by a set of reachability rules of the same sort. When coupled with a matching logic signature and model, such a set of reachability rules forms the **matching logic semantics** of a programming language:

Definition 14 (Programming Language Semantics Specification). A **(matching logic based-) semantics specification of a programming language (PLSS)** is a tuple $L = (S, \Sigma, M, \mathcal{A})$ where (S, Σ) is a matching logic signature, M is an (S, Σ) -model and \mathcal{A} is a set of reachability rules of the same sort s .

The tuple (S, Σ, M) is called the **matching logic semantic domain** of L .

Definition 15 (PLSS Model). Given a programming language semantics specification $L = (S, \Sigma, M, \mathcal{A})$, an **L -model** is any model $N = (M, \rightarrow_N)$ such that:

$$N \models \varphi \Rightarrow \varphi' \text{ for any } \varphi \Rightarrow \varphi' \in \mathcal{A}.$$

The **standard L -model** $N(S, \Sigma, M, \mathcal{A})$ is the model (M, \rightarrow_N) , where:

$$\rightarrow_N^s = \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho: \text{Var} \rightarrow M}} \bar{\rho}(\varphi) \times \bar{\rho}(\varphi')$$

$$\rightarrow_N^{s'} = \emptyset \text{ for } s' \neq s.$$

where s is the sort formulae in \mathcal{A} . So the standard model contains precisely the set of transitions that *match* the semantic rules in the semantic domain.

If the L -models are organized as a category where the morphisms are given by inclusion of transition relations, the standard model is the initial model in that category. The next section presents examples of PLSSs and examples of models for them.

3. Running Example

We use as running example two toy languages: IMP and FUN. IMP is a simple imperative language and FUN is a simple functional language. In Figure 1 we introduce the signature (S_1, Σ_1) for the language IMP.

```

Int   ::= 0 | 1 | -1 | 2 | -2 | ...
Var   ::= x | y | z | ...
Op    ::= + | - | * | / | < | <= | = | ...
ExpI  ::= Var | Int | ExpI Op 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}⟩

```

Fig. 1. The signature (S_1, Σ_1) of the IMP language written in BNF notation.

```

Int   ::= 0 | 1 | -1 | 2 | -2 | ...
Var   ::= x | y | z | ...
Val   ::= Int
        | λVar.ExpF
        | μVar.ExpF
ExpF  ::= Var | Val | ExpF Op ExpF
        | ExpF ExpF
        | if ExpF then ExpF else ExpF
CfgF  ::= ⟨ExpF⟩

```

Fig. 2. The signature (S_2, Σ_2) of FUN.

In Figure 2 we introduce the signature (S_2, Σ_2) for the language FUN. The syntax of both languages uses the following conventions.

The sorts S_i are the corresponding sets of non-terminals and each function symbol $\sigma \in \Sigma_i$ is represented by a production. We adopt a Maude-like notation, where the function symbols are written using either mixfix notation (e.g. $1 + 2$) or in prefix notation, with arguments replaced by $_$ (e.g. $_{+}(1, 2)$). Some productions (e.g. the two productions for `Code`) do not use any syntax (we simply write either an `ExpI` or a `Stmt` to understand `Code`) – but note that we remain within a many-sorted algebra. The program variables x, y , etc. are constants of sort `Var` in the term algebra. The construction $\text{Map}\{\text{Var}, \text{Int}\}$ denotes maps from `Var` to `Int`. A singleton map that maps x to 10 is written using the following syntax: $x \mapsto 10$. Maps can be concatenated using $;$: $x \mapsto 10, y \mapsto 11, z \mapsto 12$. Incorrect maps are allowed syntactically: $x \mapsto 10, x \mapsto 9$, but they are rejected semantically because the comma operator is partial (just like it is allowed to write $10/0$ as an expression but because $/$ is partial, the expression does not evaluate to anything).

The model M_1 for (S_1, Σ_1) (the signature of IMP) interprets all constructs in Figure 1 except maps syntactically. Maps are interpreted as "real" maps, so that $x \mapsto 10, y \mapsto 9$ is interpreted by the same object as $y \mapsto 9, x \mapsto 10$ (i.e. the comma operator is commutative and associative). For each operator $\text{op} \in \text{Op}$ in the syntax in Figure 1, Σ_i also contains a function symbol op_{Int} that is interpreted by M_i as the corresponding integer operation. For example, while $\llbracket \text{<=} \rrbracket_{M_i}(4, 11)$ is $\text{<=}(4, 11)$ (<= is interpreted syntactically), we have that $\llbracket \text{<=} \rrbracket_{M_1}(4, 11) = \{1\}$ (4 is less than 11, and therefore the entire term is interpreted as the singleton set $\{1\}$). Note that we use a C-like convention for booleans in our languages: 0 means false, anything different from 0 is true and the operators returning truth values (<= , = , < , etc.) return 1 for true and 0 for false. The model M_2 for (S_2, Σ_2) (the signature of FUN) interprets all constructs in Figure 2 syntactically.

The PLSS of IMP is the tuple $(S_1, \Sigma_1, M_1, \mathcal{A}_1)$, where \mathcal{A}_1 is defined in Figure 3. The variable x is of sort `Var`, i, i_1, i_2 are variables of sort `Int`, h (for heap) is a variable of sort $\text{Map}\{\text{Var}, \text{Int}\}$, s, s_1, s_2 are of sort `Stmt`, e is of sort `ExpI` and c, c' are of sort `Code`. C denotes the evaluation context. By $e[x \mapsto i]$ we denote the environment obtained from e by setting x to i .

The PLSS of FUN is $(S_2, \Sigma_2, M_2, \mathcal{A}_2)$, where \mathcal{A}_2 defined in Figure 4. Note that \mathcal{A}_2 includes reachability rules schemata. The variables i, i_1, i_2 are of sort `Int`, op ranges over all binary operators (<= , + , etc), e, e_1, e_2 are variable of sort `ExpF`, x is of sort `Var` and v is of sort `Val`. C denotes the evaluation contexts. The notation $e[x \mapsto e']$ defines the expression obtained from e by replacing all free occurrences of x in e by e' .

$$\begin{aligned}
\langle x \mid x \mapsto i, h \rangle &\Rightarrow \langle i \mid x \mapsto i, h \rangle \\
\langle i_1 \text{ op } i_2 \mid h \rangle &\Rightarrow \langle i_1 \text{ op}_{Int} i_2 \mid h \rangle \\
\langle x := i \mid h \rangle &\Rightarrow \langle \text{skip} \mid e[x \mapsto i] \rangle \\
\langle \text{skip}; s \mid h \rangle &\Rightarrow \langle s \mid h \rangle \\
\langle \text{if } i \text{ then } s_1 \text{ else } s_2 \mid h \rangle \wedge i \neq_{Int} 0 &\Rightarrow \langle s_1 \mid h \rangle \\
\langle \text{if } 0 \text{ then } s_1 \text{ else } s_2 \mid h \rangle &\Rightarrow \langle s_2 \mid h \rangle \\
\langle \text{while } e \text{ do } s \mid h \rangle &\Rightarrow \langle \text{if } e \text{ then } (s; \text{while } e \text{ do } s) \text{ else skip} \mid h \rangle \\
\langle C[c] \mid h \rangle &\Rightarrow \langle C[c'] \mid h' \rangle \quad \text{if } \langle c \mid h \rangle \Rightarrow \langle c' \mid h' \rangle
\end{aligned}$$

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

Fig. 3. Reachability Rules for IMP.

$$\begin{aligned}
\langle i_1 \text{ op } i_2 \rangle &\Rightarrow \langle i_1 \text{ op}_{Int} i_2 \rangle \\
\langle \text{if } i \text{ then } e_1 \text{ else } e_2 \rangle \wedge i \neq_{Int} 0 &\Rightarrow \langle e_1 \rangle \\
\langle \text{if } 0 \text{ then } e_1 \text{ else } e_2 \rangle &\Rightarrow \langle e_2 \rangle \\
\langle (\lambda x. e) v \rangle &\Rightarrow \langle e[x \mapsto v] \rangle \\
\langle \mu x. e \rangle &\Rightarrow \langle e[x \mapsto (\mu x. e)] \rangle \\
\langle C[e] \rangle &\Rightarrow \langle C[e'] \rangle \quad \text{if } \langle e \rangle \Rightarrow \langle e' \rangle
\end{aligned}$$

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

Fig. 4. Reachability Rules for FUN.

4. Aggregation of Matching Logic Semantic Models

In the following, we assume that (S_1, Σ_1, M_1) and (S_2, Σ_2, M_2) are two matching logic semantic domains. We also assume that the two semantic domains share a common signature (S_0, Σ_0) , i.e. there exist two morphisms $h_1 : (S_0, \Sigma_0) \rightarrow (S_1, \Sigma_1)$ and $h_2 : (S_0, \Sigma_0) \rightarrow (S_2, \Sigma_2)$. Furthermore, we assume that the models M_1 and M_2 agree on the common signature: $M_1 \upharpoonright_{h_1} = M_2 \upharpoonright_{h_2}$. We call this common model M_0 ($M_0 = M_1 \upharpoonright_{h_1} = M_2 \upharpoonright_{h_2}$).

We show how to construct an aggregated semantic domain (S', Σ', M') , in which the patterns consist of pairs of patterns from L_1 and L_2 .

4.1. Signature Aggregation

First, we show how to aggregate the matching logic signatures.

Theorem 1 (Pushout Theorem). Let (S_1, Σ_1) , (S_2, Σ_2) and (S_0, Σ_0) be three matching logic signatures, h_1 a morphism from (S_0, Σ_0) to (S_1, Σ_1) and (S_2, Σ_2) a morphism from (S_0, Σ_0) to (S_2, Σ_2) . There exists a tuple $(h'_1, (S', \Sigma'), h'_2)$, called the **pushout** of $(S_1, \Sigma_1) \xleftarrow{h_1} (S_0, \Sigma_0) \xrightarrow{h_2} (S_2, \Sigma_2)$ where h'_1 is a morphism from (S_1, Σ_1) to (S', Σ') and h'_2 a morphism from (S_2, Σ_2) to (S', Σ') such that:

1. (commutativity) $h'_1(h_1(x)) = h'_2(h_2(x))$ for all $x \in S_0 \cup \Sigma_0$ and
2. (minimality) if there exist (S'', Σ'') and morphisms h''_1 (from (S_1, Σ_1) to (S'', Σ'')) and h''_2 (from (S_2, Σ_2) to (S'', Σ'')) such that $h''_1(h_1(x)) = h''_2(h_2(x))$ for all $x \in S_0 \cup \Sigma_0$ then there exists a morphism h from (S', Σ') to (S'', Σ'') .

Furthermore, the pushout is unique (up to renaming). See, e.g., [12], for a proof. The push-out is summarised in Figure 5, which is used throughout the paper.

$$\begin{array}{ccc}
(S_0, \Sigma_0) & \xrightarrow{h_2} & (S_2, \Sigma_2) \\
h_1 \downarrow & & \downarrow h'_2 \\
(S_1, \Sigma_1) & \xrightarrow{h'_1} & (S', \Sigma')
\end{array}$$

Fig. 5. Push-out diagram assumed throughout the paper.

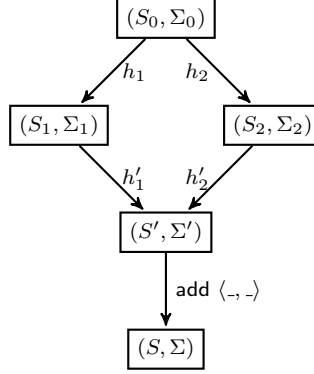


Fig. 6. The aggregation of the two signatures.

One of the most important properties of the push-out construction is captured by the following proposition:

Proposition 1. In the push-out in Figure 5, if $o' \in S' \cup \Sigma'$ such that there exist $o_1 \in S_1 \cup \Sigma_1$ and $o_2 \in S_2 \cup \Sigma_2$ with $h'_2(o_2) = o' = h'_1(o_1)$, then there exists $o_0 \in S_0 \cup \Sigma_0$ such that $h_1(o) = o_1$ and $h_2(o) = o_2$.

It says that if an object $x \in S' \cup \Sigma'$ comes from both sides ((S_1, Σ_1) and (S_2, Σ_2)), then it must come from the common part of the signatures.

The matching logic signature (S', Σ') obtained through the pushout theorem contains symbols from both languages, but it lacks a way to put two such programs together. Given two sorts $s_1 \in S_1$ and $s_2 \in S_2$ that denote the sort of configurations, we construct (S, Σ) from (S', Σ') by adding a new sort s and pairing symbol $\langle -, - \rangle : h'_1(s_1) \times h'_2(s_2) \rightarrow s$. Formally:

1. $S = S' \uplus \{s\}$ (\uplus denotes disjoint union),
2. $\Sigma = \Sigma' \uplus \{\langle -, - \rangle : h'_1(s_1) \times h'_2(s_2) \rightarrow s\}$.

The aggregation of the two signatures is summarized in Figure 6.

4.2. Model Aggregation

In this subsection, we show how to construct the aggregated model of the aggregated signature from the models of the two initial signatures through amalgamation.

Theorem 2 (Amalgamation). If M_1, M_2 and M_0 are models of $(S_1, \Sigma_1), (S_2, \Sigma_2)$ and respectively (S_0, Σ_0) such that $M_1 \upharpoonright_{h_1} = M_2 \upharpoonright_{h_2} = M_0$, there exists a unique model M' of (S', Σ') such that $M' \upharpoonright_{h'_1} = M_1$ and $M' \upharpoonright_{h'_2} = M_2$.

Proof. We have that $M_0 = M_1 \upharpoonright_{h_1} = M_2 \upharpoonright_{h_2}$. We choose M' as follows:

1. Let $s' \in S'$ be any sort. We let

$$\llbracket s' \rrbracket_{M'} = \begin{cases} \llbracket s_1 \rrbracket_{M_1} & \text{if there exists } s_1 \in S_1 \text{ such that } h'_1(s_1) = s' \\ \llbracket s_2 \rrbracket_{M_2} & \text{if there exists } s_2 \in S_2 \text{ such that } h'_2(s_2) = s' \end{cases}$$

Note that $\llbracket s' \rrbracket_{M'}$ is well defined. In case there exist both s_1 and s_2 such that $h'_1(s_1) = s' = h'_2(s_2)$, then, by Proposition 1, we have that there exists $s_0 \in S_0$ such that $s_1 = h_1(s_0)$ and $s_2 = h_2(s_0)$. But then, $\llbracket s_1 \rrbracket_{M_1} = \llbracket s_0 \rrbracket_{M_0} = \llbracket s_2 \rrbracket_{M_2}$.

2. Let $\sigma' \in \Sigma'$ be any function symbol. We let

$$\llbracket \sigma' \rrbracket_{M'} = \begin{cases} \llbracket \sigma_1 \rrbracket_{M_1} & \text{if there exists } \sigma_1 \in \Sigma_1 \text{ such that } h'_1(\sigma_1) = \sigma' \\ \llbracket \sigma_2 \rrbracket_{M_2} & \text{if there exists } \sigma_2 \in \Sigma_2 \text{ such that } h'_2(\sigma_2) = \sigma' \end{cases}$$

By the same reasoning as above, we have that $\llbracket \sigma' \rrbracket_{M'}$ is well defined.

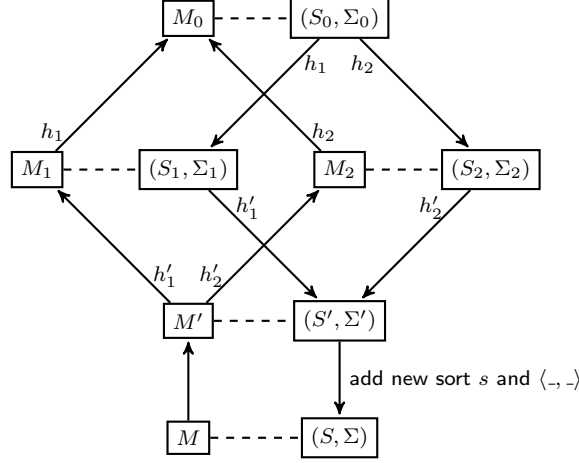


Fig. 7. Construction of the aggregated model M of (S, Σ)

Note that M' is indeed a model of (S', Σ') , since $\llbracket s' \rrbracket_{M'}$ is indeed a set for every sort $s' \in S'$ and $\llbracket \sigma' \rrbracket_{M'} : \llbracket s'_1 \rrbracket_{M'} \times \dots \times \llbracket s'_n \rrbracket_{M'} \rightarrow \mathcal{P}(\llbracket s' \rrbracket_{M'})$ for any $\sigma' \in \Sigma'_{w', s'}$, where $w' = s'_1, \dots, s'_n$.

Next, we show that M' is unique. Suppose that M'' is a model for (S', Σ') such that $M'' \upharpoonright_{h'_1} = M_1$ and $M'' \upharpoonright_{h'_2} = M_2$. We show that $M'' = M'$:

1. Let $s' \in S'$ be an arbitrary sort. We will show that $\llbracket s' \rrbracket_{M'} = \llbracket s' \rrbracket_{M''}$. Since $s' \in S'$, it follows from the pushout diagram that (1) there exists $s_1 \in S_1$ such that $h'_1(s_1) = s'$ or that (2) there exists $s_2 \in S_2$ such that $h'_2(s_2) = s'$.
 In the first case, because $M'' \upharpoonright_{h'_1} = M_1$, it follows that $\llbracket s' \rrbracket_{M''} = \llbracket s_1 \rrbracket_{M_1}$; by the definition of M' , we also have that $\llbracket s' \rrbracket_{M'} = \llbracket s_1 \rrbracket_{M_1}$ and therefore $\llbracket s' \rrbracket_{M''} = \llbracket s' \rrbracket_{M'}$.
 In the second case, because $M'' \upharpoonright_{h'_2} = M_2$, it follows that $\llbracket s' \rrbracket_{M''} = \llbracket s_2 \rrbracket_{M_2}$; by the definition of M' , we also have that $\llbracket s' \rrbracket_{M'} = \llbracket s_2 \rrbracket_{M_2}$ and therefore $\llbracket s' \rrbracket_{M''} = \llbracket s' \rrbracket_{M'}$.
2. Let $\sigma' \in \Sigma'$ be an arbitrary function symbol. Since $\sigma' \in \Sigma'$, it follows from the pushout diagram that (1) there exists $\sigma_1 \in \Sigma_1$ such that $h'_1(\sigma_1) = \sigma'$ or that (2) there exists $\sigma_2 \in \Sigma_2$ such that $h'_2(\sigma_2) = \sigma'$.
 In the first case, because $M'' \upharpoonright_{h'_1} = M_1$, it follows that $\llbracket \sigma' \rrbracket_{M''} = \llbracket \sigma_1 \rrbracket_{M_1}$; by the definition of M' , we also have that $\llbracket \sigma' \rrbracket_{M'} = \llbracket \sigma_1 \rrbracket_{M_1}$ and therefore $\llbracket \sigma' \rrbracket_{M''} = \llbracket \sigma' \rrbracket_{M'}$.
 In the second case, because $M'' \upharpoonright_{h'_2} = M_2$, it follows that $\llbracket \sigma' \rrbracket_{M''} = \llbracket \sigma_2 \rrbracket_{M_2}$; by the definition of M' , we also have that $\llbracket \sigma' \rrbracket_{M'} = \llbracket \sigma_2 \rrbracket_{M_2}$ and therefore $\llbracket \sigma' \rrbracket_{M''} = \llbracket \sigma' \rrbracket_{M'}$.

We have shown that for any model M'' of (S', Σ') such that $M'' \upharpoonright_{h'_1} = M_1$ and $M'' \upharpoonright_{h'_2} = M_2$ is equal to M' , thereby establishing the uniqueness of M' . \square

Note that we need to prove our amalgamation theorem because, even if the amalgamation result is known for many-sorted FOL, our models are different since valuations map terms into subsets of the domains (and not just elements of the domain like in FOL models).

So far, we have obtained a model M' for (S', Σ') by amalgamation. We say that M' is the amalgamated model of M_1 and M_2 . But the signature (S', Σ') does not contain the pairing symbol required to construct pairs of programs. Recall that (S, Σ) is obtained from (S', Σ') by adding the pairing symbol $\langle -, - \rangle : s_1 \times s_2 \rightarrow s$, where $s \in S$ is the new sort of configurations.

We now construct a model M for (S, Σ) from the model M' for (S', Σ') by interpreting the new objects (the new sort and the new function symbol) as follows:

1. $\llbracket s \rrbracket_M = \llbracket s_1 \rrbracket_{M'} \times \llbracket s_2 \rrbracket_{M'}$ (the set of configurations is the cartesian product of the sets of configurations of the individual programs) and
2. $\llbracket \langle -, - \rangle \rrbracket_M(a, b) = \{ \langle \llbracket a \rrbracket_{M'}, \llbracket b \rrbracket_{M'} \rangle \}$ (the interpretation of the pairing symbol applied to two arguments a

and b is the singleton set that contains only the pair of the interpretation of a and the interpretation of b).

We say that M is the aggregated model from M_1 and M_2 . The entire construction of the aggregated language specification is summarised in Figure 7.

Next, we show the link, in terms of matching logic formulae, between the aggregated model and the initial models.

Lemma 2. Let φ_1 and φ_2 be matching logic formulae over (S_1, Σ_1) and respectively (S_2, Σ_2) . For any (well-sorted) valuation $\rho : \text{Var} \rightarrow M$, if x, y are fresh variables, we have that

$$\begin{aligned}\bar{\rho}(\langle h'_1(\varphi_1), y \rangle) &= \bar{\rho} \upharpoonright_{h'_1}(\varphi_1) \times \bar{\rho}(y) \\ \bar{\rho}(\langle x, h'_2(\varphi_2) \rangle) &= \bar{\rho}(x) \times \bar{\rho} \upharpoonright_{h'_2}(\varphi_2)\end{aligned}$$

Proof. For simplicity, we prove the conclusion for $i = 1$ (since the case with $i = 2$ is analogous).

$$\begin{aligned}\bar{\rho}(\langle h'_1(\varphi_1), y \rangle) &= \llbracket \langle -, - \rangle \rrbracket_M(\bar{\rho}(h'_1(\varphi_1)), \bar{\rho}(y)) && \text{(def. } \bar{\rho} \text{)} \\ &= \{ \llbracket \langle -, - \rangle \rrbracket_M(\bar{\rho} \upharpoonright_{h'_1}(\varphi_1), \bar{\rho}(y)) \mid \gamma_1 \in \bar{\rho}(h'_1(\varphi_1)) \} && \text{(ind.hyp.)} \\ &= \bar{\rho} \upharpoonright_{h'_1}(\varphi_1) \times \bar{\rho}(y) && \text{(def. } \llbracket \cdot \rrbracket_M \text{)}\end{aligned}$$

□

5. Language Aggregation

In the following, we assume that $L_1 = (S_1, \Sigma_1, M_1, \mathcal{A}_1)$ and $L_2 = (S_2, \Sigma_2, M_2, \mathcal{A}_2)$ are two PLSSs that share a signature (S_0, Σ_0) , i.e. there exist two morphisms $h_1 : (S_0, \Sigma_0) \rightarrow (S_1, \Sigma_1)$ and $h_2 : (S_0, \Sigma_0) \rightarrow (S_2, \Sigma_2)$, and the models M_1 and M_2 agree on the common signature: $M_1 \upharpoonright_{h_1} = M_2 \upharpoonright_{h_2}$. We further consider the standard model $N_i = (M_i, \rightarrow_i)$ of the language L_i and s_i the sort of the formalae in \mathcal{A}_i , $i = 1, 2$.

Having shown how to construct the aggregated matching logic semantic domain (S, Σ, M) , we now show how the transition relation \rightarrow and the axioms A of the aggregated language are defined from the relations $\rightarrow_1, \rightarrow_2$, and respectively the axioms A_1, A_2 of the initial languages. Let s denote the new added sort in S and $\rightarrow_i^{h'_i(s_i)} \subseteq M_{h'_i(s_i)} \times M_{h'_i(s_i)}$ denote the relation

$$\begin{aligned}\rightarrow_i^{h'_i(s_i)} &= \{ \gamma \rightarrow \gamma' \in M_{i, h'_i(s_i)} \times M_{i, h'_i(s_i)} \mid \gamma \rightarrow_i \gamma' \in M_{i, s_i} \times M_{i, s_i} \} \\ &= \{ \gamma \rightarrow \gamma' \in M_{i, h'_i(s_i)} \times M_{i, h'_i(s_i)} \mid \gamma \rightarrow_i \gamma' \in M \upharpoonright_{h'_i, s_i} \times M \upharpoonright_{h'_i, s_i} \}\end{aligned}$$

for $i = 1, 2$.

We identify three types of language aggregations, depending on how the \rightarrow transition relation is defined from \rightarrow_1 and \rightarrow_2 . Each of the three constructions could be useful in various contexts:

1. $\rightarrow_1 \otimes_p \rightarrow_2 = \bigcup_{s' \in S} (\rightarrow_1 \otimes_p \rightarrow_2)_{s'}$ is the **parallel** product of the two transition relations, i.e.

$$\begin{aligned}(\rightarrow_1 \otimes_p \rightarrow_2)_s &= \{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma'_2 \rangle \mid \gamma_1 \rightarrow_1^{h'_1(s_1)} \gamma'_1 \text{ and } \gamma_2 \rightarrow_2^{h'_2(s_2)} \gamma'_2 \} \\ (\rightarrow_1 \otimes_p \rightarrow_2)_{s'} &= \emptyset \text{ for } s' \neq s\end{aligned}$$

2. $\rightarrow_1 \otimes_a \rightarrow_2 = \bigcup_{s' \in S} (\rightarrow_1 \otimes_a \rightarrow_2)_{s'}$ is the **asynchronous interleaving** product of the two transition relations, i.e.

$$\begin{aligned}(\rightarrow_1 \otimes_a \rightarrow_2)_s &= (id^{h'_1(s_1)} \otimes_p \rightarrow_2^{h'_2(s_2)}) \cup (\rightarrow_1^{h'_1(s_1)} \otimes_p id^{h'_2(s_2)}) \\ (\rightarrow_1 \otimes_a \rightarrow_2)_{s'} &= \emptyset \text{ for } s' \neq s\end{aligned}$$

where $id = \bigcup_{s' \in S} id^{s'}$ is the identity relation.

3. finally, $\rightarrow_1 \otimes \rightarrow_2 = (\rightarrow_1 \otimes_a \rightarrow_2) \cup (\rightarrow_1 \otimes_p \rightarrow_2)$ is the (general) product of \rightarrow_1 and \rightarrow_2 .

The asynchronous product with interleaving semantics means that in one step of the aggregated language, either the left-hand side takes a step (in the first language) or the right-hand side takes a step (in the second language). The parallel product forces both sides to take steps simultaneously. The (general) product requires at least one side to take a step and it allows (but not requires) the other side to do the same.

We next show how to construct a set of axioms A for the aggregated language from the set of axioms A_1 and A_2 of the initial languages, depending on which of the three constructions is chosen for the aggregated transition relation. The main result is that, for each of the three constructions, the transition relation is a model of the aggregated axioms. This means that we can construct the formal semantics of the aggregated language directly from the formal semantics of the initial languages.

5.1. The Axioms for the Parallel Product

We let

$$\mathcal{A}_1 \otimes_p \mathcal{A}_2 = \bigcup_{\substack{\varphi_1 \Rightarrow \varphi'_1 \in \mathcal{A}_1 \\ \varphi_2 \Rightarrow \varphi'_2 \in \mathcal{A}_2}} \{ \langle h'_1(\varphi_1), h'_2(\varphi_2) \rangle \Rightarrow \langle h'_1(\varphi'_1), h'_2(\varphi'_2) \rangle \}$$

We show formally that $A_1 \otimes_p A_2$ is indeed a formal specification of the language $(M, \rightarrow_1 \otimes_p \rightarrow_2)$:

Theorem 3 (correctness for parallel product). Let $\mathcal{A} = \mathcal{A}_1 \otimes_p \mathcal{A}_2$ and $\rightarrow = \rightarrow_1 \otimes_p \rightarrow_2$. Then (M, \rightarrow) is the standard model for $L_1 \otimes_p L_2 = (S, \Sigma, M, \mathcal{A})$.

Proof. We have $(\rightarrow_1 \otimes_p \rightarrow_2)_{s'} = \emptyset$ for $s' \neq s$ by the definition of $\rightarrow_1 \otimes_p \rightarrow_2$, so we have only to show that

$$(\rightarrow_1 \otimes_p \rightarrow_2)_s = \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho: Var \rightarrow M}} \bar{\rho}(\varphi) \times \bar{\rho}(\varphi').$$

We may assume w.l.o.g. that the formulae in \mathcal{A}_1 and those in \mathcal{A}_2 have disjoint free variables. Consequently, for each $\varphi_1 \Rightarrow \varphi'_1 \in \mathcal{A}_1$, $\varphi_2 \Rightarrow \varphi'_2 \in \mathcal{A}_2$, $\rho_1, \rho_2: Var \rightarrow M$, there exists $\rho: Var \rightarrow M$ such that

$$\bar{\rho}(\varphi_i) = \bar{\rho}_i(\varphi_i), \quad \bar{\rho}(\varphi'_i) = \bar{\rho}_i(\varphi'_i), \quad i = 1, 2. \quad (2)$$

We have:

$$\begin{aligned} (\rightarrow_1 \otimes_p \rightarrow_2)_s &= \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma'_2 \rangle \mid \gamma_1 \xrightarrow{h'_1(s_1)} \gamma'_1, \gamma_2 \xrightarrow{h'_2(s_2)} \gamma'_2 \right\} && \text{(def. } \otimes_p) \\ &= \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma'_2 \rangle \mid \gamma_1 \xrightarrow{s_1} \gamma'_1, \gamma_2 \xrightarrow{s_2} \gamma'_2 \right\} && \text{(def. } \rightarrow_i^{h'_i(s_i)}) \\ &= \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma'_2 \rangle \mid \gamma_1 \rightarrow \gamma'_1 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho_1: Var \upharpoonright_{h'_1} \rightarrow M_1}} \bar{\rho}_1(h'_1(\varphi)) \times \bar{\rho}_1(h'_1(\varphi')), \right. \\ &\quad \left. \gamma_2 \rightarrow \gamma'_2 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho_2: Var \upharpoonright_{h'_2} \rightarrow M_2}} \bar{\rho}_2(h'_2(\varphi)) \times \bar{\rho}_2(h'_2(\varphi')) \right\} && ((M_i, \rightarrow_i) \text{ is std.}) \\ &= \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma'_2 \rangle \mid \gamma_1 \rightarrow \gamma'_1 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho: Var \rightarrow M}} \bar{\rho} \upharpoonright_{h'_1}(\varphi) \times \bar{\rho} \upharpoonright_{h'_1}(\varphi'), \right. \\ &\quad \left. \gamma_2 \rightarrow \gamma'_2 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho: Var \rightarrow M}} \bar{\rho} \upharpoonright_{h'_2}(\varphi) \times \bar{\rho} \upharpoonright_{h'_2}(\varphi') \right\} && \text{(Lemma 1)} \end{aligned}$$

$$\begin{aligned}
&= \bigcup_{\substack{\varphi_1 \Rightarrow \varphi'_1 \in \mathcal{A}_1 \\ \varphi_2 \Rightarrow \varphi'_2 \in \mathcal{A}_2 \\ \rho: \text{Var} \rightarrow M}} \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma'_2 \rangle \mid \begin{array}{l} \gamma_1 \rightarrow \gamma'_1 \in \bar{\rho} \upharpoonright_{h'_1(\varphi_1)} \times \bar{\rho} \upharpoonright_{h'_1(\varphi'_1)}, \\ \gamma_2 \rightarrow \gamma'_2 \in \bar{\rho} \upharpoonright_{h'_2(\varphi_2)} \times \bar{\rho} \upharpoonright_{h'_2(\varphi'_2)} \end{array} \right\} & \text{(cf. (2))} \\
&= \bigcup_{\substack{\varphi_1 \Rightarrow \varphi'_1 \in \mathcal{A}_1 \\ \varphi_2 \Rightarrow \varphi'_2 \in \mathcal{A}_2 \\ \rho: \text{Var} \rightarrow M}} \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma'_2 \rangle \mid \begin{array}{l} \gamma_1 \rightarrow \gamma'_1 \in \bar{\rho}(h'_1(\varphi_1)) \times \bar{\rho}(h'_1(\varphi'_1)), \\ \gamma_2 \rightarrow \gamma'_2 \in \bar{\rho}(h'_2(\varphi_2)) \times \bar{\rho}(h'_2(\varphi'_2)) \end{array} \right\} & \text{(Lemma 1)} \\
&= \bigcup_{\substack{\varphi_1 \Rightarrow \varphi'_1 \in \mathcal{A}_1 \\ \varphi_2 \Rightarrow \varphi'_2 \in \mathcal{A}_2 \\ \rho: \text{Var} \rightarrow M}} \bar{\rho}(\langle h'_1(\varphi_1), h'_2(\varphi_2) \rangle) \times \bar{\rho}(\langle h'_1(\varphi'_1), h'_2(\varphi'_2) \rangle) & \text{(reorg.)} \\
&= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho: \text{Var} \rightarrow M}} \bar{\rho}(\varphi) \times \bar{\rho}(\varphi') & \text{(def.}\mathcal{A}\text{)}
\end{aligned}$$

□

5.2. The Axioms for the Asynchronous Product with Interleaving Semantics

We let

$$\begin{aligned}
\mathcal{A}_1 \otimes_a \mathcal{A}_2 = & \bigcup_{\varphi \Rightarrow \varphi' \in \mathcal{A}_1} \{ \langle h'_1(\varphi), y \rangle \Rightarrow \langle h'_1(\varphi'), y \rangle \} \cup \\
& \bigcup_{\varphi \Rightarrow \varphi' \in \mathcal{A}_2} \{ \langle x, h'_2(\varphi) \rangle \Rightarrow \langle x, h'_2(\varphi') \rangle \}
\end{aligned}$$

where x (of sort $h'_1(s_1)$) and y (of sort $h'_2(s_2)$) are fresh variables. The intuition is that x captures any left-hand side and allow the right-hand to take a step while y captures any right-hand side and allow the left-hand side to take a step. We show formally that $\mathcal{A}_1 \otimes_a \mathcal{A}_2$ is indeed a formal specification of the language $(M, \rightarrow_1 \otimes_a \rightarrow_2)$:

Theorem 4 (correctness for asynchronous product). Let $\mathcal{A} = \mathcal{A}_1 \otimes_a \mathcal{A}_2$ and $\rightarrow = \rightarrow_1 \otimes_a \rightarrow_2$. We have that (M, \rightarrow) is the standard model for $L_1 \otimes_a L_2 = (S, \Sigma, M, \mathcal{A})$.

Proof. The proof is similar to that of Theorem 3. We have $(\rightarrow_1 \otimes_a \rightarrow_2)_{s'} = \emptyset$ for $s' \neq s$ by the definition of $\rightarrow_1 \otimes_a \rightarrow_2$, so we have only to show that

$$(\rightarrow_1 \otimes_a \rightarrow_2)_s = \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho: \text{Var} \rightarrow M}} \bar{\rho}(\varphi) \times \bar{\rho}(\varphi').$$

We have:

$$\begin{aligned}
(\rightarrow_1 \otimes_a \rightarrow_2)_s &= (id^{h'_1(s_1)} \otimes_p \rightarrow_2^{h'_2(s_2)}) \cup (\rightarrow_1^{h'_1(s_1)} \otimes_p id^{h'_2(s_2)}) & \text{(def.}\otimes_a\text{)} \\
&= \{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma_1, \gamma'_2 \rangle \mid \gamma_2 \xrightarrow{s_2} \gamma'_2 \} \cup \\
&\quad \{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma_2 \rangle \mid \gamma_1 \xrightarrow{s_1} \gamma'_1 \} & \text{(def.}\otimes_p\text{)}
\end{aligned}$$

$$\begin{aligned}
&= \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma_1, \gamma'_2 \rangle \mid \gamma_2 \rightarrow \gamma'_2 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho_2: \text{Var} \upharpoonright_{h'_2} \rightarrow M_2}} \bar{\rho}_2(h'_2(\varphi)) \times \bar{\rho}_2(h'_2(\varphi')) \right\} \cup \\
&\quad \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma_2 \rangle \mid \gamma_1 \rightarrow \gamma'_1 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho_1: \text{Var} \upharpoonright_{h'_1} \rightarrow M_1}} \bar{\rho}_1(h'_1(\varphi)) \times \bar{\rho}_1(h'_1(\varphi')) \right\} \quad (\rightarrow_i \text{ is std.}) \\
&= \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma_1, \gamma'_2 \rangle \mid \gamma_2 \rightarrow \gamma'_2 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho: \text{Var} \rightarrow M}} \bar{\rho} \upharpoonright_{h'_2}(\varphi) \times \bar{\rho} \upharpoonright_{h'_2}(\varphi') \right\} \cup \\
&\quad \left\{ \langle \gamma_1, \gamma_2 \rangle \rightarrow \langle \gamma'_1, \gamma_2 \rangle \mid \gamma_1 \rightarrow \gamma'_1 \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho: \text{Var} \rightarrow M}} \bar{\rho} \upharpoonright_{h'_1}(\varphi) \times \bar{\rho} \upharpoonright_{h'_1}(\varphi') \right\} \quad (\text{Lemma 1}) \\
&= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho: \text{Var} \rightarrow M}} \left\{ \langle \rho(x), \gamma_2 \rangle \rightarrow \langle \rho(x), \gamma'_2 \rangle \mid \gamma_2 \rightarrow \gamma'_2 \in \bar{\rho}(h'_2(\varphi)) \times \bar{\rho}(h'_2(\varphi')) \right\} \cup \\
&\quad \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho: \text{Var} \rightarrow M}} \left\{ \langle \gamma_1, \rho(y) \rangle \rightarrow \langle \gamma'_1, \rho(y) \rangle \mid \gamma_1 \rightarrow \gamma'_1 \in \bar{\rho}(h'_1(\varphi)) \times \bar{\rho}(h'_1(\varphi')) \right\} \quad (\text{reorg.}) \\
&= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho: \text{Var} \rightarrow M}} \bar{\rho}(\langle x, h'_2(\varphi) \rangle) \times \bar{\rho}(\langle x, h'_2(\varphi') \rangle) \cup \\
&\quad \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho: \text{Var} \rightarrow M}} \bar{\rho}(\langle h'_1(\varphi), y \rangle) \times \bar{\rho}(\langle h'_1(\varphi'), y \rangle) \quad (\text{reorg.}) \\
&= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho: \text{Var} \rightarrow M}} \bar{\rho}(\varphi) \times \bar{\rho}(\varphi') \quad (\text{def. } \mathcal{A}, \text{ cf. (2)})
\end{aligned}$$

where $\bar{\rho} \upharpoonright_{h'_2}(x) = \gamma_1$, $\bar{\rho} \upharpoonright_{h'_1}(y) = \gamma_2$. \square

5.3. The Axioms for the General Product

For the general product, which allows for both interleaving and parallel steps, we define

$$A_1 \otimes A_2 = A_1 \otimes_a A_2 \cup A_1 \otimes_p A_2.$$

The correctness result for the general product follows quickly from Theorem 4 and Theorem 3.

Theorem 5 (correctness for the general product). Let $A = A_1 \otimes A_2$ and $\rightarrow = \rightarrow_1 \otimes \rightarrow_2$. We have that (M, \rightarrow) is the standard model for $L_1 \otimes L_2 = (S, \Sigma, M, \mathcal{A})$.

In the following sections we will assume that we have constructed an aggregated language $L = (S, \Sigma, M, \mathcal{A})$ out of two languages $L_1 = (S_1, \Sigma_1, M_1, \mathcal{A}_1)$ and $L_2 = (S_2, \Sigma_2, M_2, \mathcal{A}_0)$, where the common part is given by the matching logic semantics domain (S_0, Σ_0, M_0) that has the properties discussed in Section 4. The construction is summarised in Figure 7. We assume that $s \in S$ is the sort of aggregated configurations, while $s_1 \in S_1$ and $s_2 \in S_2$ are the sorts of configurations of the two initial languages. In our examples L_1 will be the IMP language and L_2 will be the FUN language.

We will denote by $\gamma \rightarrow_i \gamma'$ the one-step transition relation of language L_i (for $i \in \{1, 2\}$). We denote by \rightarrow_i^* and \rightarrow_i^+ the reflexive-and-transitive and respectively the transitive closures of the one-step transition relation.

If φ and φ' are matching logic formulae of sort $h'_i(s_i)$ (for $i \in \{1, 2\}$), we make the following notations:

1. $\varphi \rightarrow_i \varphi'$ means that for any ρ and for any γ such that $\gamma \in \bar{\rho}(\varphi)$, there exists γ' such that $\gamma \rightarrow_i \gamma'$ and $\gamma' \in \bar{\rho}(\varphi')$ (any configuration γ matching φ moves in one step into a configuration γ' matching φ')
2. $\varphi \rightarrow_i^+ \varphi'$ means that for any ρ and for any γ such that $\gamma \in \bar{\rho}(\varphi)$, there exists γ' such that $\gamma \rightarrow_i^+ \gamma'$ and $\gamma' \in \bar{\rho}(\varphi')$ (any configuration γ matching φ moves in one or more steps into a configuration γ' matching φ')
3. $\varphi \rightarrow_i^* \varphi'$ means that for any ρ and for any γ such that $\gamma \in \bar{\rho}(\varphi)$, there exists γ' such that $\gamma \rightarrow_i^* \gamma'$ and $\gamma' \in \bar{\rho}(\varphi')$ (any configuration γ matching φ moves in zero or more steps into a configuration γ' matching φ')

6. Specifying Equivalent Programs

Matching logic patterns of sort s (where $\{s\} = S \setminus S'$) over the signature (S, Σ) can be used to specify pairs of configurations of two languages:

Definition 16. Let $\varphi \in \text{PATTERNS}_s$ be a matching logic pattern over (S, Σ) . The denotation of φ (written $\llbracket \varphi \rrbracket$), is the set of all concrete configurations that satisfy it:

$$\llbracket \varphi \rrbracket = \bigcup_{\rho: \text{Var} \rightarrow M} \bar{\rho}(\varphi).$$

This notation extends to sets E of patterns, written $\llbracket E \rrbracket$, as expected:

$$\llbracket E \rrbracket = \bigcup_{\varphi \in E} \llbracket \varphi \rrbracket.$$

Example 10. The following set

$$E = \{ \langle \langle \text{skip} \mid (x \mapsto i, -) \rangle, \langle j \rangle \rangle \wedge i =_{\text{Int}} j \} \quad (3)$$

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 the program variable x and suppose we want to show it computes the same integer result as a FUN program. Then the denotation $\llbracket E \rrbracket$ 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 $\llbracket E \rrbracket$ of "base" pairs which are known to be equivalent.

All definitions for program equivalence contain a part of the form "executions of P_1 and P_2 on equal inputs". This constraint can be captured by an matching logic formula φ in the aggregated language.

Partial equivalence is defined as: "For any terminating executions of P_1 and P_2 on equal inputs the results of the executions are equal". This is captured by the following definition:

Definition 17 (Partial Equivalence). We write $\models \varphi \Downarrow E$, and say that φ **partially reaches** E , if for all configurations $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$ such that γ_1 terminates in \rightarrow_1 with γ'_1 and γ_2 terminates in \rightarrow_2 with γ'_2 , we have that $\langle \gamma'_1, \gamma'_2 \rangle \in \llbracket E \rrbracket$.

Mutual termination is defined as "Given equal inputs, P_1 terminates iff P_2 terminates":

Definition 18 (Mutual Termination). We say that φ **mutually terminates**, if for all configurations $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$, γ_1 terminates in \rightarrow_1 if and only if γ_2 terminates in \rightarrow_2 .

Mutual equivalence (or *full equivalence*) is defined as "partially equivalent and mutually terminate" and is formally expressed as follows:

Definition 19 (Mutual Equivalence). We write $\models \varphi \Downarrow^\infty E$, and say that φ **reaches** E , iff for all configurations $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$ we have that:

1. both γ_1 and γ_2 diverge (with respect to \rightarrow_1 and respectively \rightarrow_2), or
2. there exist γ'_1 and γ'_2 such that $\gamma_1 \rightarrow_1^* \gamma'_1$, $\gamma_2 \rightarrow_2^* \gamma'_2$ and $\langle \gamma'_1, \gamma'_2 \rangle \in \llbracket E \rrbracket$.

Total equivalence is defined as " P_1 and P_2 are partially equivalent and both terminate" and is captured by the following formal definition:

Definition 20 (Total Equivalence). We write $\models \varphi \Downarrow_1 E$, and say that φ **totally reaches** E , iff for all configurations $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$ we have that both γ_1 and γ_2 terminate (with respect to \rightarrow_1 and respectively \rightarrow_2), and there exist γ'_1 and γ'_2 such that with $\gamma_1 \rightarrow_1^* \gamma'_1$, $\gamma_2 \rightarrow_2^* \gamma'_2$ and $\langle \gamma'_1, \gamma'_2 \rangle \in \llbracket E \rrbracket$.

Example 11. Let $E = \{ \langle \langle \text{skip} \mid (\mathbf{n} \mapsto i, -) \rangle, \langle i \rangle \rangle \}$. The singleton set E denotes all pairs of IMP and respectively FUN configurations where:

1. the IMP program terminated (since it consists only of the statement `skip`) and the program variable \mathbf{n} maps to the integer i ,
2. the FUN program terminated (since it consists only of the expression i) with the same value i of the IMP variable \mathbf{n} .

By using the set E defined above as the "base equivalence relation", we express the fact that we expect the result of the IMP program to end up in the program variable \mathbf{n} , and this value should be the same as the result of the FUN program. Furthermore, the values of the rest of the variables (other than \mathbf{n}) are ignored.

We consider the following formulae φ_1 , φ_2 and φ_3 :

$$\begin{aligned} \varphi_1 &= \langle \langle \text{code}_1 \mid \mathbf{n} \mapsto n \rangle, \langle \text{exp}_1 \ n \rangle \rangle \wedge n \geq_{Int} 0 \\ \varphi_2 &= \langle \langle \text{code}_2 \mid \quad \quad \quad \rangle, \langle \text{exp}_2 \ 0 \rangle \rangle \\ \varphi_3 &= \langle \langle \text{code}_3 \mid \mathbf{n} \mapsto n \rangle, \langle \text{exp}_3 \ n \ 1 \rangle \rangle. \end{aligned}$$

where:

$$\begin{aligned} \text{code}_1 &\equiv \text{i:=n; n:=0; while i>0 do (n:=n+i; i:=i-1)} \\ \text{exp}_1 &\equiv \mu f. \lambda x. \text{if } x==1 \text{ then } 1 \text{ else } x+f \ (x-1) \\ \text{code}_2 &\equiv \text{while } 1 \text{ do skip} \\ \text{exp}_2 &\equiv (\mu f. \lambda x. f \ x) \\ \text{code}_3 &\equiv \text{PGM}_1 \\ \text{exp}_3 &\equiv \text{PGM}_2. \end{aligned}$$

The formula φ_1 denotes pairs of IMP and respectively FUN configurations where the IMP program executes `code1` starting in an environment where the program variable \mathbf{n} is mapped to the non-negative integer n and the FUN program executes the recursive function `exp1` with the same non-negative value n as argument. The IMP code `code1` compute the sum of the integers 1 up to the value of the program variable \mathbf{n} , while the FUN function `exp1` computes the sum of the integers 1 up to the value of its argument. Since both programs terminate (we have that $n \geq 0$) and they end up with the same result, we have that $\models \varphi_1 \Downarrow^\infty E$.

The formula φ_2 contains two programs that both diverge: the IMP program contains an infinite loop while the FUN program contains a non-terminating recursive function. Obviously we have that $\varphi_2 \Downarrow^\infty E$.

Finally, the φ_3 pattern contains two programs that both compute the Collatz function. Note that `PGM1` and `PGM2` are defined in Figure 8. Even though we do not know if they terminate (this is the well-known Collatz conjecture), it is still possible to establish that they are mutually equivalent, independently of the Collatz conjecture: if they both terminate they produce the same result and if one does not terminate, the other does not terminate either. Therefore we have $\varphi_3 \Downarrow^\infty E$ as well.

| | |
|---|---|
| $\begin{aligned} \text{PGM}_1 &\equiv c := n; n := 1; \text{LOOP}_1 \\ \text{LOOP}_1 &\equiv \text{while } (c \neq 1) \\ &\quad n := n + 1; \\ &\quad \text{if } (c \% 2 \neq 0) \\ &\quad \quad \text{then } c := 3 * c + 1 \\ &\quad \quad \text{else } c := c / 2 \end{aligned}$ | $\begin{aligned} \text{PGM}_2 &\equiv \mu f. \lambda n. \lambda a. \text{LOOP}_2 \\ \text{LOOP}_2 &\equiv \text{if } (n \neq 1) \\ &\quad \text{then if } (n \% 2 \neq 0) \\ &\quad \quad \text{then } f(3 * n + 1)(a + 1) \\ &\quad \quad \text{else } f(n / 2)(a + 1) \\ &\quad \text{else } a \end{aligned}$ |
|---|---|

Fig. 8. An IMP program PGM_1 and a FUN program PGM_2 that both compute the Collatz function. The IMP program proceeds iteratively, incrementing a counter n at each step until 1 is reached. The FUN program is recursive. It takes two arguments: the current value n and an accumulator a that counts how many steps were already taken.

$$\begin{array}{c} \text{AXIOM} \frac{\varphi \in E}{\vdash \varphi \Downarrow^\infty E} \quad \text{CONSEQ} \frac{\models \varphi \rightarrow \exists \tilde{x}. \varphi' \quad \vdash \varphi' \Downarrow^\infty E}{\vdash \varphi \Downarrow^\infty E} \quad \text{CASE ANALYSIS} \frac{\vdash \varphi \Downarrow^\infty E \quad \vdash \varphi' \Downarrow^\infty E}{\vdash \varphi \vee \varphi' \Downarrow^\infty E} \\ \\ \text{STEP} \frac{\models \varphi_1 \rightarrow_1^* \varphi'_1 \quad \models \varphi_2 \rightarrow_2^* \varphi'_2 \quad \vdash \langle \varphi'_1, \varphi'_2 \rangle \Downarrow^\infty E}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E} \\ \\ \text{CIRCULARITY} \frac{\models \varphi_1 \rightarrow_1^+ \varphi'_1 \quad \models \varphi_2 \rightarrow_2^+ \varphi'_2 \quad \vdash \langle \varphi'_1, \varphi'_2 \rangle \Downarrow^\infty E \cup \{ \langle \varphi_1, \varphi_2 \rangle \}}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E} \end{array}$$

Fig. 9. Mutual Equivalence Proof System.

7. Proving Mutual Program Equivalence

Here we provide a language-parametric foundation for showing equivalence of programs written in possibly different languages. We assume that the set E of matching logic formulae that characterize the pairs of programs that are known to be equivalent is fixed.

7.1. 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 $\models \varphi \Downarrow^\infty E$. Figure 9 contains the 5-rule proof system for proving mutual equivalence of programs. The first rule is AXIOM. There is nothing surprising about this rule; it simply states that if an equivalence is known to be true, then it can be derived.

The second rule is CONSEQ(ueuce). This rule allows to perform domain reasoning in the formula. It states that if a formula φ implies another 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 surprising at first sight (we might expect it to be the other way around), but the intuition is that $\exists \tilde{x}. \varphi'$ is more general than φ . Moreover $\llbracket \exists \tilde{x}. \varphi' \rrbracket = \text{denotation} \varphi'$. Therefore if we are able to prove the equivalence for φ' , then φ must also hold. This rule is used in the example proof tree below (in Figure 10) to rearrange a formula of the form $(n \neq_{Int} 1 \vee n =_{Int} 1) \wedge \dots$ into $\text{true} \wedge \dots$. 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 third rule is CASE ANALYSIS. This allows to branch the proof depending on the different cases to consider. Typically, CASE ANALYSIS is used to branch the proof when the two programs also branch. In the proof tree below (in Figure 10), this rule is used to perform a case analysis between the case where both programs end (because of reaching the termination condition $n=1$) and where the programs continue ($n \neq 1$).

The fourth 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 $\langle \varphi_1, \varphi_2 \rangle$ to $\langle \varphi'_1, \varphi'_2 \rangle$, we reach a formula $\langle \varphi'_1, \varphi'_2 \rangle$ that is derivable, then we conclude that $\langle \varphi_1, \varphi_2 \rangle$ 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 [23], but any other valid reasoning will also work.

The fifth and last rule is CIRCULARITY. This rule is used to handle repetitive program structures such as loops or recursive functions. Like STEP, the CIRCULARITY rule also begins by making both configurations

take steps, but, differently from STEP, it requires that progress be made on both sides. Thereafter, the new configuration $\langle \varphi'_1, \varphi'_2 \rangle$ must be shown to reach E or the original configuration $\langle \varphi_1, \varphi_2 \rangle$. This means intuitively that $\langle \varphi_1, \varphi_2 \rangle$ will eventually reach E or that each of φ_1 and φ_2 will diverge.

Another way to look at CIRCULARITY is that it allows to *postulate* that the equivalence being proven ($\langle \varphi_1, \varphi_2 \rangle$) holds, make progress ($\varphi_1 \rightarrow_1^+ \varphi'_1, \varphi_2 \rightarrow_2^+ \varphi'_2$) in both programs that we want to show equivalent, and then derive $\langle \varphi'_1, \varphi'_2 \rangle$ possibly using $\langle \varphi_1, \varphi_2 \rangle$ as an axiom, i.e., $\vdash \langle \varphi'_1, \varphi'_2 \rangle \Downarrow^\infty E \cup \{\langle \varphi_1, \varphi_2 \rangle\}$. We use this rule in the proof tree in Figure 10 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 before reaching the original configuration again. As for the STEP rule, an oracle that decides $\models \varphi_i \rightarrow_i^+ \varphi'_i$ is needed. This oracle can be implemented again using the reachability logic proof system in [23], but any other type of sound reasoning suffices.

This intuition is formalized in the soundness proof in Section 7.2.

7.2. Soundness

In this subsection we prove the main result of the paper:

Theorem 6 (Soundness). For any set of patterns E and for any pattern φ , if the sequent $\vdash \varphi \Downarrow^\infty E$ is derivable using the proof system given in Figure 9 then $\models \varphi \Downarrow^\infty E$.

We extend the definition of $\models \varphi \Downarrow^\infty E$ to sets of matching logic formulae of sort s as expected:

Definition 21. If F is a set of matching logic formulae of sort s , 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 $G \subseteq \llbracket s \rrbracket_M$ denote a set of configurations in the aggregated language. By the definition of $\llbracket s \rrbracket_M$, we have that $G \subseteq \llbracket h'_1(s_1) \rrbracket_M \times \llbracket h'_2(s_2) \rrbracket_M$.

Definition 22. We say that a configuration $\langle \gamma_1, \gamma_2 \rangle \in \llbracket s \rrbracket_M$ in the aggregated language reaches G , written $\langle \gamma_1, \gamma_2 \rangle \rightarrow^* G$, if there exist configurations $\gamma'_1 \in \llbracket h'_1(s_1) \rrbracket_M$ and $\gamma'_2 \in \llbracket h'_2(s_2) \rrbracket_M$ such that $\gamma_1 \rightarrow_1^* \gamma'_1, \gamma_2 \rightarrow_2^* \gamma'_2$ and $\langle \gamma'_1, \gamma'_2 \rangle \in G$.

Definition 23. We say that a configuration $\langle \gamma_1, \gamma_2 \rangle \in \llbracket s \rrbracket_M$ diverges, written $\langle \gamma_1, \gamma_2 \rangle \uparrow^\infty$, if both γ_1 and γ_2 diverge (in \rightarrow_1 and respectively \rightarrow_2).

Definition 24. We say that a configuration $\langle \gamma_1, \gamma_2 \rangle \in \llbracket s \rrbracket_M$ of the aggregated language co-reaches G , written $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} G$, if at least one of the following conditions holds:

1. $\langle \gamma_1, \gamma_2 \rangle \uparrow^\infty$ (the configuration diverges), or
2. $\langle \gamma_1, \gamma_2 \rangle \rightarrow^* G$ (the configuration reaches G).

The following utility lemma establishes a link between models of matching logic formulae of the aggregated language and the notion of co-reachability defined above:

Lemma 3. For all sets of matching logic formulae E of sort s and for any matching logic formula φ of sort s , we have that:

$$\models \varphi \Downarrow^\infty E \text{ iff for all } \langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket, \langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket.$$

Proof. Immediately, by unfolding the definitions. \square

The next lemma is the core of our soundness proof.

Lemma 4 (Circularity Principle). Let F be a set of matching formulae of sort s . If for each $\langle \gamma_1, \gamma_2 \rangle \in \llbracket F \rrbracket$ there exist $\gamma'_1 \in \llbracket h'_1(s_1) \rrbracket_M, \gamma'_2 \in \llbracket h'_2(s_2) \rrbracket_M$ such that

$$\gamma_1 \rightarrow_1^+ \gamma'_1, \gamma_2 \rightarrow_2^+ \gamma'_2 \text{ and } \langle \gamma'_1, \gamma'_2 \rangle \rightarrow^{*,\infty} \llbracket E \cup F \rrbracket,$$

then

$$\models F \Downarrow^\infty E.$$

Proof. By Lemma 3, to prove $\models F \Downarrow^\infty E$ it is enough to show that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$ for all $\langle \gamma_1, \gamma_2 \rangle \in \llbracket F \rrbracket$. Let $\langle \gamma_1, \gamma_2 \rangle \in \llbracket F \rrbracket$ be arbitrarily chosen.

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

$$\begin{aligned} P(i) = & \text{ there exist configurations } \gamma_1^1, \dots, \gamma_1^i, \gamma_2^1, \dots, \gamma_2^i \text{ such that} \\ & \gamma_1 \rightarrow_1^+ \gamma_1^1 \rightarrow_1^+ \gamma_1^2 \rightarrow_1^+ \dots \rightarrow_1^+ \gamma_1^i, \\ & \gamma_2 \rightarrow_2^+ \gamma_2^1 \rightarrow_2^+ \gamma_2^2 \rightarrow_2^+ \dots \rightarrow_2^+ \gamma_2^i \text{ and} \\ & \langle \gamma_1^1, \gamma_2^1 \rangle \rightarrow^{*,\infty} E \cup F, \dots, \langle \gamma_1^i, \gamma_2^i \rangle \rightarrow^{*,\infty} E \cup F \\ \text{or } & \langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket \end{aligned}$$

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

1. for $i = 1$, we have that by hypothesis that there exist $\gamma_1^1 = \gamma'_1$, $\gamma_2^1 = \gamma'_2$ such that $\gamma_c \rightarrow_c^+ \gamma_c^1$ (for all $c \in \{1, 2\}$), and $\langle \gamma_1^1, \gamma_2^1 \rangle \rightarrow^{*,\infty} \llbracket E \cup F \rrbracket$.
2. for $i > 1$, we assume that $P(i-1)$ holds and we prove that $P(i)$ holds. If $P(i-1)$ holds because $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$, then by definition $P(i)$ holds as well.

Otherwise, we have that there exist configurations $\gamma_1^1, \dots, \gamma_1^{i-1}, \gamma_2^1, \dots, \gamma_2^{i-1}$ such that:

- (a) $\gamma_1 \rightarrow_1^+ \gamma_1^1 \rightarrow_1^+ \gamma_1^2 \rightarrow_1^+ \dots \rightarrow_1^+ \gamma_1^{i-1}$,
- (b) $\gamma_2 \rightarrow_2^+ \gamma_2^1 \rightarrow_2^+ \gamma_2^2 \rightarrow_2^+ \dots \rightarrow_2^+ \gamma_2^{i-1}$ and
- (c) $\langle \gamma_1^1, \gamma_2^1 \rangle \rightarrow^{*,\infty} E \cup F, \dots, \langle \gamma_1^{i-1}, \gamma_2^{i-1} \rangle \rightarrow^{*,\infty} E \cup F$

As $\langle \gamma_1^{i-1}, \gamma_2^{i-1} \rangle \rightarrow^{*,\infty} \llbracket E \cup F \rrbracket$, we distinguish between the following two cases:

- (a) either $\langle \gamma_1^{i-1}, \gamma_2^{i-1} \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$ and, as $\gamma_1 \rightarrow_1^+ \gamma_1^{i-1}$ and $\gamma_2 \rightarrow_2^+ \gamma_2^{i-1}$, we have that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$. This implies that $P(i)$ holds (by the second condition in definition of the predicate).
- (b) or $\langle \gamma_1^{i-1}, \gamma_2^{i-1} \rangle \rightarrow^{*,\infty} \llbracket F \rrbracket$. By the hypothesis, there exist $\gamma_1^i (= \gamma'_1)$ and $\gamma_2^i (= \gamma'_2)$ such that $\gamma_1^{i-1} \rightarrow_1^+ \gamma_1^i$, $\gamma_2^{i-1} \rightarrow_2^+ \gamma_2^i$ and $\langle \gamma_1^i, \gamma_2^i \rangle \rightarrow^{*,\infty} \llbracket E \cup F \rrbracket$. 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 $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$. We will assume by contradiction that $\langle \gamma_1, \gamma_2 \rangle \not\rightarrow^{*,\infty} \llbracket E \rrbracket$. As $P(i)$ holds for all naturals $i > 0$, it follows that for each $i > 0$, there exist $\gamma_1^1, \dots, \gamma_1^i, \gamma_2^1, \dots, \gamma_2^i$ such that $\gamma_1 \rightarrow_1^+ \gamma_1^1 \rightarrow_1^+ \dots \rightarrow_1^+ \gamma_1^i$ and $\gamma_2 \rightarrow_2^+ \gamma_2^1 \rightarrow_2^+ \dots \rightarrow_2^+ \gamma_2^i$.

This means that both γ_1 and γ_2 diverge (in \rightarrow_1 and respectively \rightarrow_2) and therefore $\langle \gamma_1, \gamma_2 \rangle \uparrow^\infty$. But this implies $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$, which yields a contradiction. As our assumption $\langle \gamma_1, \gamma_2 \rangle \not\rightarrow^{*,\infty} \llbracket E \rrbracket$ led to a contradiction, it follows that it must hold and therefore $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$, which is what we had to show. \square

The circularity principle lies at the core of the proof for Theorem 6. We recall Theorem 6 and we are now ready to prove it:

Theorem 6 (Soundness). For any set of patterns E and for any pattern φ , if the sequent $\vdash \varphi \Downarrow^\infty E$ is derivable using the proof system given in Figure 9 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 $\models \varphi \Downarrow^\infty E$. Let $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$ be an arbitrary configuration. Let $\gamma_1' = \gamma_1$ and $\gamma_2' = \gamma_2$. We have that $\gamma_1 \rightarrow_1^* \gamma_1'$, $\gamma_2 \rightarrow_2^* \gamma_2'$ and $\langle \gamma_1', \gamma_2' \rangle \in \llbracket E \rrbracket$. Therefore, by the second case in Definition 19, we have that $\models \varphi \Downarrow^\infty E$.
2. if the last rule to be applied is **Step**, we have by the induction hypothesis that there exists an aggregated configuration $\langle \varphi_1', \varphi_2' \rangle$ such that $\models \varphi_1 \rightarrow_1^* \varphi_1'$, that $\models \varphi_2 \rightarrow_2^* \varphi_2'$ and that $\models \langle \varphi_1', \varphi_2' \rangle \Downarrow^\infty E$. We show that $\models \langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E$. Let $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \langle \varphi_1, \varphi_2 \rangle \rrbracket$ be an arbitrary configuration. By Lemma 3, it is sufficient to show that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$.

As $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \langle \varphi_1, \varphi_2 \rangle \rrbracket$, we have that there exists $\rho : \text{Var} \rightarrow M$ such that $\gamma_1 \in \bar{\rho}(\varphi_1)$ and $\gamma_2 \in \bar{\rho}(\varphi_2)$. Since $\models \varphi_1 \rightarrow_1^* \varphi'_1$, there exists γ'_1 such that $\gamma_1 \rightarrow_1^* \gamma'_1$ and $\gamma'_1 \in \bar{\rho}(\varphi'_1)$. Similarly, there exists γ'_2 such that $\gamma_2 \rightarrow_2^* \gamma'_2$ and $\gamma'_2 \in \bar{\rho}(\varphi'_2)$. Therefore $\langle \gamma'_1, \gamma'_2 \rangle \in \bar{\rho}(\langle \varphi'_1, \varphi'_2 \rangle)$.

We also have that $\models \langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E$ and therefore, by Lemma 3, $\langle \gamma'_1, \gamma'_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$. But $\gamma_1 \rightarrow_1^* \gamma'_1$ and $\gamma_2 \rightarrow_2^* \gamma'_2$. Therefore $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$ as well, which is what we had to show.

3. if the last rule to be applied is **Consequence**, we have by the induction hypothesis that there exists φ' such that $\models \varphi \rightarrow \exists \tilde{x}. \varphi'$ and $\models \varphi' \Downarrow^\infty E$. We show that $\models \varphi \Downarrow^\infty E$.

Let $\langle \gamma_1, \gamma_2 \rangle$ be an arbitrary configuration such that $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$. By Lemma 3, it is sufficient to show that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$.

As $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$ and $\models \varphi \rightarrow \exists \tilde{x}. \varphi'$, we have that $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \exists \tilde{x}. \varphi' \rrbracket = \llbracket \varphi' \rrbracket$. But we have that $\models \varphi' \Downarrow^\infty E$, and therefore, by Lemma 3, $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$, 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 $\langle \gamma_1, \gamma_2 \rangle$ be an arbitrary configuration such that $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \vee \varphi' \rrbracket$. By Lemma 3, it is sufficient to show that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$.

As $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \vee \varphi' \rrbracket$, we have that $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$ or that $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi' \rrbracket$:

- (a) in the first case, we have that $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi \rrbracket$. We also have that $\models \varphi \Downarrow^\infty E$ (by the induction hypothesis). By Lemma 3, we obtain that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$.
- (b) in the second case, we have that $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \varphi' \rrbracket$. We also have that $\models \varphi' \Downarrow^\infty E$ (by the induction hypothesis). By Lemma 3, we obtain that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$.

In either case we have obtained that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} \llbracket E \rrbracket$, which is what we had to prove.

5. if the last rule to be applied is **Circularity**, then, by the induction hypothesis, there exists a configuration $\langle \varphi'_1, \varphi'_2 \rangle$ such that:

$$\langle \varphi'_1, \varphi'_2 \rangle \Downarrow^\infty E \cup \{ \langle \varphi_1, \varphi_2 \rangle \}, \quad (4)$$

$$\models \varphi_1 \rightarrow_1^+ \varphi'_1 \text{ and} \quad (5)$$

$$\models \varphi_2 \rightarrow_2^+ \varphi'_2. \quad (6)$$

Let $\langle \gamma_1, \gamma_2 \rangle \in \llbracket \langle \varphi_1, \varphi_2 \rangle \rrbracket$ be an arbitrary configuration. By definition, there exists $\rho : \text{Var} \rightarrow M$ such that $\gamma_1 \in \bar{\rho}(\varphi_1)$ and $\gamma_2 \in \bar{\rho}(\varphi_2)$.

By Equation (5) and (6), we obtain that there exist γ'_1 and γ'_2 such that

$$\gamma_1 \rightarrow_1^+ \gamma'_1, \quad (7)$$

$$\gamma_2 \rightarrow_2^+ \gamma'_2 \quad (8)$$

and $\gamma'_1 \in \bar{\rho}(\varphi'_1)$ and $\gamma'_2 \in \bar{\rho}(\varphi'_2)$. From $\gamma'_1 \in \bar{\rho}(\varphi'_1)$ and $\gamma'_2 \in \bar{\rho}(\varphi'_2)$, we obtain by Equation (4) and Lemma 3 that

$$\langle \gamma'_1, \gamma'_2 \rangle \rightarrow^{*,\infty} E \cup \{ \langle \varphi_1, \varphi_2 \rangle \}. \quad (9)$$

Let $F = \{ \langle \varphi_1, \varphi_2 \rangle \}$. We have shown that the three conditions of the Circularity Principle (Lemma 4) are met (Equations 7, 8 and 9). Therefore we can apply Lemma 4 and obtain $\models F \Downarrow^\infty E$. But $F = \{ \langle \varphi_1, \varphi_2 \rangle \}$ and therefore $\langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E$, which is exactly what we had to show.

□

7.3. Example

We discuss here the proof tree in Figure 10 that shows that the two Collatz programs in Figure 8 are equivalent.

As we have already shown, 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 program variable n . This base equivalence is captured by the set $E = \{ \langle \text{skip} \mid (n \mapsto i, -), \langle i \rangle \}$.

| | | |
|-----|--|---|
| | $\varphi := n \neq_{Int} 1 \wedge \langle \langle \text{LOOP}_1 \mid n \mapsto i, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ i \rangle \rangle$ | |
| 1. | $\vdash \langle \langle \text{skip} \mid n \mapsto i, - \rangle, \langle i \rangle \rangle$ | $\Downarrow^\infty E$ AXIOM |
| 2. | $\vdash \langle \langle \text{skip} \mid n \mapsto i, - \rangle, \langle i \rangle \rangle$ | $\Downarrow^\infty E \cup \{\varphi\}$ AXIOM |
| 3. | $\vdash \langle \langle \text{skip} \mid n \mapsto i, c \mapsto 1 \rangle, \langle i \rangle \rangle$ | $\Downarrow^\infty E$ CONSEQ(1) |
| 4. | $\vdash \langle \langle \text{skip} \mid n \mapsto i, c \mapsto 1 \rangle, \langle i \rangle \rangle$ | $\Downarrow^\infty E \cup \{\varphi\}$ CONSEQ(2) |
| 5. | $\vdash n =_{Int} 1 \wedge \langle \langle \text{LOOP}_1 \mid n \mapsto i, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ i \rangle \rangle$ | $\Downarrow^\infty E$ STEP(3) |
| 6. | $\vdash n =_{Int} 1 \wedge \langle \langle \text{LOOP}_1 \mid n \mapsto i, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ i \rangle \rangle$ | $\Downarrow^\infty E \cup \{\varphi\}$ STEP(4) |
| 7. | $\vdash n \neq_{Int} 1 \wedge \langle \langle \text{LOOP}_1 \mid n \mapsto i, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ i \rangle \rangle$ | $\Downarrow^\infty E \cup \{\varphi\}$ AXIOM |
| 8. | $\vdash \langle \langle \text{LOOP}_1 \mid n \mapsto i, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ i \rangle \rangle$ | $\Downarrow^\infty E \cup \{\varphi\}$ CONSEQ(CA(6, 7)) |
| 9. | $\vdash n \neq_{Int} 1 \wedge \langle \langle \text{LOOP}_1 \mid n \mapsto i, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ i \rangle \rangle$ | $\Downarrow^\infty E$ CIRCULARITY (8) |
| 10. | $\vdash \langle \langle \text{LOOP}_1 \mid n \mapsto i, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ i \rangle \rangle$ | $\Downarrow^\infty E$ CONSEQ(CA(5, 9)) |
| 11. | $\vdash \langle \langle \text{LOOP}_1 \mid n \mapsto 1, c \mapsto n \rangle, \langle \text{PGM}_2 \ n \ 1 \rangle \rangle$ | $\Downarrow^\infty E$ CONSEQ(10) |
| 12. | $\vdash \langle \langle \text{PGM}_1 \mid n \mapsto n \rangle, \langle \text{PGM}_2 \ n \ 1 \rangle \rangle$ | $\Downarrow^\infty E$ STEP (11) |

Fig. 10. Formal proof showing that the two Collatz programs are mutually equivalent. **CA** stands for CASE ANALYSIS. The main use of CONSEQ in the proof tree above is to write **true** as $n \neq_{Int} 0 \vee n =_{Int} 0$ in order to be able to apply **CA**. We also make use of CONSEQ liberally, without making it explicit in the proof tree, in order to rewrite the pattern in the appropriate form.

It says that an IMP configuration $\langle \text{skip} \mid (n \mapsto i, -) \rangle$ (describing programs that stopped (because the code cell contains **skip**) and that have the integer i in the n memory cell) is equivalent to a FUN configuration that contains exactly the integer i . The proof tree in Figure 10 shows that the two programs are equivalent.

The proof tree should be read starting with line 12, which is the proof goal. We perform STEP (line 11) to enter the loops, CONSEQ (line 10 to rearrange the formula) and CASE ANALYSIS to see whether we have to immediately quit the loop or not. The case where we immediately quit is solved quickly by AXIOM (lines 5, 3, 1). In the other case we postulate a CIRCULARITY (line 8) and we branch again into two parts: the case where we immediately quit the loops (solved by lines 6, 4, 2 – similarly to lines 5, 3, 1 but there is an additional φ next to E) and the case where the postulated circularity is actually used as an axiom (line 7).

8. 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 (the model of the shared domain is also an input to the aggregation operation) 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), the initial configuration is parametric in n (the input to the two programs).

Related Work. It was first remarked by Hoare in [13] that program equivalence might be easier than program correctness. Among the recent works on equivalence we mention [9, 10, 1]. 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. Symbolic programs are considered in [17] but for a different notion of bisimulation-based program equivalence. In [9] 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 [9] proves equivalence for LPL programs. On the other hand, [9] propose

deductive systems for several kinds of equivalences, whereas we focus on mutual (a.k.a. full) equivalence only. In [14], 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 [16]), 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 [19] (we only cite two of the most relevant recent works). Other work targets specific classes of languages: functional [20], microcode [8], CLP [5]. In order to be less language-specific some approaches advocate the use of intermediate languages, such as [15], 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. And finally, only a few approaches, among which [16, 8], deal with real-life language and industrial-size programs in those languages. This is in contrast to the equivalence checking of hardware circuits, which has entered the mainstream industrial practice (see, e.g., [28] for a survey on this topic). Finally, our own related work [17] 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 is more natural for certain classes of applications such as proving compilers. In [6] we have investigated the theoretical foundations of language aggregation, including signature aggregation and model amalgamation. Unfortunately, to perform the aggregation, we have used a restricted version of matching logic (we used top-most matching logic), which complicates the presentation. This article contains a better and simpler explanation of the aggregation process, even if we had to prove our own amalgamation theorem to deal with the models of matching logic in their full generality.

Further Work. Our definition (Definition 19) 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.

Our approach handles programs with parameters, but it cannot handle **symbolic statements**. For example, assume that we have the following two programs:

```

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

```

The two programs are equivalent for any particular statement s , assuming that n starts with a non-negative value and that the variables hold mathematical integers (otherwise, if $n = \text{MAX_INT}$, the test $i \leq n$ would always yield true, leaving an infinite loop in the first program and a finite loop in the second). It would be interesting to adapt our approach to handle such symbolic statements s in order to prove equivalence of programs that are parametric not only in data, but also code.

Another issue is completeness. Although relative completeness results have been proven for matching logic based proof systems for partial correctness [23], 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 [21]. 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. We also plan to implement a semi-automated version of the proof system.

References

- [1] S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *VMCAI 2012*, volume 7148 of *LNCS*, pages 119–135, 2012.
- [2] C. E. Chiriță and T. F. Șerbănuță. An institutional foundation for the \mathbb{K} semantic framework. In *WADT 2014*, *LNCS*, to appear.
- [3] Ș. Ciobăcă. Reducing partial equivalence to partial correctness. In *SYNASC 2014*, pages 164–171. IEEE, 2014.
- [4] Ș. Ciobăcă, D. Lucanu, V. Rusu, and G. Roșu. A language-independent proof system for mutual a equivalence. In *ICFEM 2014*, volume 8829 of *LNCS*, pages 75–90, 2014.

- [5] S. Crăciunescu. Proving the equivalence of CLP programs. In *ICLP 2002*, volume 2401 of *LNCS*, pages 287–301, 2002.
- [6] Ș. Ciobăcă, D. Lucanu, V. Rusu, and G. Roșu. A theoretical foundation for programming language aggregation. In *WADT 2014*, LNCS, to appear.
- [7] A. Ștefănescu, Ș. Ciobăcă, R. Mereuță, B. M. Moore, T. F. Șerbănuță, and G. Roșu. All-path reachability logic. In *RTA-TLCA '14*, volume 8560 of *LNCS*, pages 425–440, July 2014.
- [8] T. A. et al. Formal verification of backward compatibility of microcode. In *CAV 2005*, volume 3576 of *LNCS*, pages 185–198, 2005.
- [9] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.
- [10] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability*, 23(3):241–258, 2013.
- [11] R. K. Guy. Don't try to solve these problems. *The American Mathematical Monthly*, 90(1):pp. 35–38+39–41, 1983.
- [12] A. E. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In *AMAST 1996*, pages 132–147. Springer-Verlag, 1996.
- [13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
- [14] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI 2009*, pages 327–337. ACM, 2009.
- [15] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In *CAV 2012*, volume 7358 of *LNCS*, pages 712–717, 2012.
- [16] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [17] D. Lucanu and V. Rusu. Program equivalence by circular reasoning. *Formal Aspects of Computing*, pages 1–26, 2014.
- [18] J. Meseguer and G. Roșu. Rewriting logic semantics: From language specifications to formal analysis tools. In *IJCAR 2004*, volume 3097 of *LNCS*, pages 1–44, 2004.
- [19] G. Necula. Translation validation for an optimizing compiler. In *PLDI 2000*, pages 83–94. ACM, 2000.
- [20] A. Pitts. Operational semantics and program equivalence. In *Applied Semantics Summer School*, volume 2395 of *LNCS*, pages 378–412, 2002.
- [21] G. Roșu. Equality of streams is a Π_2^0 -complete problem. In *ICFP 2006*, pages 184–191. ACM, 2006.
- [22] G. Roșu. Matching logic. In *RTA 2015*, LNCS, to appear.
- [23] G. Roșu and A. Ștefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.
- [24] G. Roșu and A. Ștefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP 2012*, volume 7392 of *LNCS*, pages 351–363, 2012.
- [25] G. Roșu, A. Ștefănescu, Ș. Ciobăcă, and B. M. Moore. One-path reachability logic. In *LICS 2013*, pages 358–367. IEEE, 2013.
- [26] G. Roșu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST 2010*, volume 6486 of *LNCS*, pages 142–162, 2010.
- [27] T.-F. Șerbănuță, G. Roșu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009.
- [28] F. Somenzi and A. Kuehlmann. *Electronic Design Automation For Integrated Circuits Handbook*, volume 2, chapter 4: Equivalence Checking. CRC Press, 2006.