A Language-Independent Proof System for Full Program Equivalence

Ștefan Ciobâcă[1], Dorel Lucanu[1], Vlad Rusu[2] and Grigore Roșu[3]

[1]Faculty Of Computer Science,"Alexandru Ioan Cuza" University, Iași, Romania,
[2]Inria Lille, France
[3]University of Illinois at Urbana-Champaign, USA

Abstract.

Two programs are fully equivalent if, for the same input, either they both diverge or they both terminate with the same result. Full 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 full equivalence, which is parametric in the operational semantics of two languages and in a state-similarity relation. The proof system is sound: a proof tree establishes the full 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 weather the sequence terminates or not; nevertheless, our proof system shows that the two programs are fully 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 **full equivalence** [**?**], which we explore in this article and which we consider to be the most appropriate notion of equivalence for general purposes. Two programs are said to be **fully 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. Full 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 $3x+1$ is x is odd. For example, the Collatz sequence that starts with 6 is: $6, 3, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots$. It is conjectured that, if we start with any positive integer, the sequence

Correspondence and offprint requests to: Ș. Ciobâcă, D. Lucanu, V. Rusu and G. Roșu

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:

```
c := n;
n := 1;
while (c != 1)
  n := n + 1;
  if (c % 2 != 0)
    then c := 3 * c + 1
    else c := c / 2
```

```
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
```

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 two[1] 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 [**?**] for anyone who would prove or disprove the conjecture. Therefore, a proof of full 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:

$$\textsc{Circularity}\ \frac{\models \varphi_1 \Rightarrow_1^+ \varphi_1' \qquad \models \varphi_2 \Rightarrow_2^+ \varphi_2' \qquad \vdash \langle \varphi_1', \varphi_2' \rangle \Downarrow^\infty E \cup \{\langle \varphi_1, \varphi_2 \rangle\}}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E}$$

Circularity is inspired from previous work [**?**], where we use a rule with the same name to prove partial correctness of programs. In previous work, that 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 [**?**], full equivalence presents additional difficulties that we have to deal with. In particular, the circularity rule for partial correctness is unsound in case the program diverges.

Our Circularity rule (introduced in Figure 9 on page 21) 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 ($\models \varphi_1 \Rightarrow_1^+ \varphi_1'$ and $\models \varphi_2 \Rightarrow_2^+ \varphi_2'$) 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 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 3) that we state and prove in Section 7.

There are other ways to show full equivalence (e.g. [**?**]). However, we adhere to the belief expressed by Meseguer and Roșu [**?**] that formal analyses and proofs of programs (proofs of partial/total correctness,

[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.

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 that are sound will not disagree on the answer for the same verification task. Moreover, 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 [**?**]. Furthermore, it is possible to define realistic languages like C [**?**], Java [**?**] and JavaScript [**?**] in a reasonable timeframe using this rewrite-based approach. 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 programming languages is to relate (basic) 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 advantage of the aggregated language is that we can use matching logic [**?**] in order to reason about pairs of programs from the initial languages directly. In the absence of aggregation, it would not be possible to reason about pairs of programs in a language-independent manner. The aggregation relies on the existence of pushouts for the signatures and on model amalgamation for the model. Section 4 is devoted to this construction.

To summarise, we formalize the notion of full equivalence and propose a logic with a deductive system for stating and proving full 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 full 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 Roșu [**?**], 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 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 pushout 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 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 synchronously, 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 full equivalence and we illustrate it on a few examples. In Section 7, we introduce our 5-rule proof system for full 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.

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 sets $A = \bigcup_{s \in S} A_s$. Note that A_s is not necessarily disjoint from $A_{s'}$ for two different sorts s and s'. We write $a{:}s$ 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 $f : A \to B$ is a collection of functions $f = \bigcup_{s \in S} f_s$ with $f_s : A_s \to B_s$. 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 write $\sigma : s_1 \times \cdots \times s_n \to s$ for $\sigma \in \Sigma_{s_1 \ldots s_n, s}$. If $n = 0$, i.e., $\sigma : \to s$, then σ is a **constant**.

Given two algebraic signatures (S, Σ) and (S', Σ'), an **algebraic signature morphism** $h : (S, \Sigma) \to (S', \Sigma')$ consists of two functions $h_{srt} : S \to S'$ and $h_{sgn} : \Sigma \to \Sigma'$ such that, for each $\sigma \in \Sigma$, if $\sigma : s_1 \times s_2 \times \ldots s_n \to s$ then $h_{sgn}(\sigma) : h_{srt}(s_1) \times h_{srt}(s_2) \times \ldots h_{srt}(s_n) \to h_{srt}(s) \in \Sigma'$. We 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

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

$\Sigma = \{\mathit{false}, \mathit{true} : \to \mathit{Bool}, \_\mathit{and}\_ : \mathit{Bool} \times \mathit{Bool} \to \mathit{Bool}\}$,

and (S', Σ') be the signature

$S' = \{\mathit{Prop}, \ldots\}$,

$\Sigma' = \{\mathit{ff}, \mathit{tt} : \to \mathit{Prop}, \_\&\_ : \mathit{Prop} \times \mathit{Prop} \to \mathit{Prop}, \ldots\}$

where ... means that (S', Σ') may include more objects (sorts and functional symbols). An example of a signature morphism $h : (S, \Sigma) \to (S', \Sigma')$ is given by $h(\mathit{Bool}) = \mathit{Prop}$, $h(\mathit{false}) = \mathit{ff}$, $h(\mathit{true}) = \mathit{tt}$, and $h(\_\mathit{and}\_) = \_\&\_$.

2.2. Matching Logic

Matching logic [**?**, **?**] 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 (first-order logic) is that it makes no distinction between function and 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.

As shown in [**?**], separation logic [**?**] can be regarded, without any syntactic change or translation, as a particular matching logic theory in the particular model of finite-domain partial maps. It is well known that any separation logic formula can be translated to an equivalent FOL formula over the map model, and that FOL can be translated to predicate logic (functional symbols can be replaced with predicate symbols). Similarly, any matching logic formula can be translated to a predicate logic formula [**?**]. Nevertheless, each of these translations adds quantifiers and increases the size of the formula, making both human and machine reasoning more difficult. Therefore, it is more convenient to work directly with the original logic instead of predicate logic. Indeed, if it were not for the notational and reasoning convenience, FOL would have never been needed, because it adds no expressiveness over predicate logic. Similarly, separation logic has established itself as a convenient notation for specifying and reasoning about programs in the presence of heap data-structures. The advantage of matching logic over separation logic, which makes it appealing for our language-independent approach, is that it uniformly provides separation at all levels in the program configuration, and not only in the heap. Additionally, prior work on language-independent verification builds upon matching logic [**?**, **?**, **?**, **?**], thus making the work in this paper easier to relate to those results.

In this section, we recall the basic concepts of matching logic, as defined in [**?**].

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

Definition 2 (Diagram). A **diagram** $D = (\mathit{Sig}_D, \mathit{Morph}_D)$ is a pair consisting of a set Sig_D of matching logic signatures and a set Morph_D of morphisms between signatures in Sig_D.

Assumption 1. In the rest of the paper, we will, implicitly or explicitly, consider only diagrams $D = (\mathit{Sig}_D, \mathit{Morph}_D)$ such that, for each signature $(S, \Sigma) \in \mathit{Sig}_D$, there exists a countably infinite S-indexed set $\mathit{Var}_{(S,\Sigma)}$ of variables with the following properties:

- for any signature $(S, \Sigma) \in \mathit{Sig}_D$, for any two distinct sorts $s, s' \in S$, we have $(\mathit{Var}_{(S,\Sigma)})_s \cap (\mathit{Var}_{(S,\Sigma)})_{s'} = \emptyset$;
- for any two distinct signatures $(S, \Sigma), (S', \Sigma') \in \mathit{Sig}_D$, $\mathit{Var}_{(S,\Sigma)} \cap \mathit{Var}_{(S',\Sigma')} = \emptyset$;
- for each morphism $h : (S, \Sigma) \to (S', \Sigma')$ in Morph_D, there exists an injection from the S-sorted set $\mathit{Var}_{(S,\Sigma)}$ to the S'-sorted set $\mathit{Var}_{(S',\Sigma')}$, denoted also by h;
- for two distinct such injections, of the form $h_i : \mathit{Var}_{(S_i,\Sigma_i)} \to \mathit{Var}_{(S',\Sigma')}$ $(i \in \{1, 2\})$, we have $h_1(x_1) \neq h_2(x_2)$ for all $x_i \in \mathit{Var}_{(S_i,\Sigma_i)}$ $(i \in \{1, 2\})$;
- for each sort $s \in S$, there are an infinite number of variables of that sort.

Remark 1. For any diagram D that is a finite directed acyclic graph, there exists a Sig_D-indexed set of variables Var that satisfies the above properties. We show how to construct such a set of variables. Let $\mathit{Var}^0_{(S,\Sigma)}$ be arbitrary, pairwise disjoint, countably infinite initial S-indexed sets of variables for each $(S, \Sigma) \in \mathit{Sig}_D$. Let $(S_1, \Sigma_1), (S_2, \Sigma_2), \ldots, (S_n, \Sigma_n)$ be a sequence of all signatures in Sig_D, considered in topological order (i.e., a morphism $h : (S_j, \Sigma_j) \to (S_i, \Sigma_i)$, with $i, j \in \{1, \ldots, n\}$, can only exist if $j < i$). For all i from 1 to n, we define the set of variables $\mathit{Var}_{(S_i,\Sigma_i)}$ associated to the signature (S_i, Σ_i) to be:

$$\mathit{Var}_{(S_i,\Sigma_i)} = \mathit{Var}^0_{(S_i,\Sigma_i)} \uplus \biguplus_{h:(S_j,\Sigma_j)\to(S_i,\Sigma_i)} \mathit{Var}_{(S_i,\Sigma_i)}.$$

Informally, $\mathit{Var}_{(S_i,\Sigma_i)}$ contains, in addition to the initial variables, a copy of all variables $\mathit{Var}_{(S_j,\Sigma_j)}$ for which there is a morphism h between (S_j, Σ_j) and (S_i, Σ_i). The injection h between $\mathit{Var}_{(S_j,\Sigma_j)}$ and $\mathit{Var}_{(S_j,\Sigma_j)}$ takes a variable to its copy in $\mathit{Var}_{(S_i,\Sigma_i)}$.

All diagrams that we consider in this paper are finite an acyclic and therefore they satisfy Assumption 1.

Definition 3 (Matching Logic Models). Let (S, Σ) be a matching logic signature. An (S, Σ)**-model** consists of:

1. an S-sorted **carrier set** $\bigcup_{s \in S} [\![s]\!]_M$ such that, for any sort $s \in S$, the carrier set $[\![s]\!]_M$ of sort s is not empty; and
2. a function $[\![\sigma]\!]_M : [\![s_1]\!]_M \times \ldots \times [\![s_n]\!]_M \to \mathcal{P}([\![s]\!]_M)$ for each symbol $\sigma \in \Sigma_{s_1,\ldots,s_n,s}$; the function $[\![\sigma]\!]_M$ 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 $[\![\sigma]\!]_M$ such that

$$[\![\sigma]\!]_M(A_1, \ldots, A_n) = \bigcup_{a_1 \in A_1, \ldots, a_n \in A_n} [\![\sigma]\!]_M(a_1, \ldots, a_n). \tag{1}$$

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 $list : Nat \to Map$ 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: $[\![Prop]\!]_{M'} = \{0, 1\}$, $[\![ff]\!]_{M'} = \{0\}$, $[\![tt]\!]_{M'} = \{1\}$ and $[\![\_\&\_]\!]_{M'}$ is multiplication (i.e., $[\![\_\&\_]\!]_{M'}(a, b) = \{a \cdot b\}$). This model example is very simple and close to many-sorted algebra. See [**?**] for more complex matching logic models that go beyond the many-sorted algebra.

Definition 4 (Reduct of a Model). Let $h : (S, \Sigma) \to (S', \Sigma')$ be a matching logic signature morphism and M' a (S', Σ')-model. The **reduct of** M', written as $M'{\restriction_h}$, is the model of (S, Σ) defined as follows:

- for each $s \in S$, $[\![s]\!]_{M'{\restriction_h}} = [\![h(s)]\!]_{M'}$
- for each operation symbol $\sigma \in \Sigma_{s_1, \ldots, s_n, s}$, $[\![\sigma]\!]_{M'{\restriction_h}}$ is the function:

  $[\![\sigma]\!]_{M'{\restriction_h}} : [\![s_1]\!]_{M'{\restriction_h}} \times [\![s_2]\!]_{M'{\restriction_h}} \times \ldots \times [\![s_n]\!]_{M'{\restriction_h}} \to \mathcal{P}([\![s]\!]_{M'{\restriction_h}})$ such that:
  $[\![\sigma]\!]_{M'{\restriction_h}}(a_1, \ldots, a_n) = [\![h(\sigma)]\!]_{M'}(a_1, \ldots, a_n)$ for any $a_i \in [\![h(s_i)]\!]_{M'} = [\![s_i]\!]_{M'{\restriction_h}}$ $(1 \leq i \leq n)$.

  Note that $[\![h(\sigma)]\!]_{M'}(a_1, \ldots, a_n) \subseteq [\![h(s)]\!]_{M'} = [\![s]\!]_{M'{\restriction_h}}$.

Example 3. Let h be the signature morphism defined in Example 1 and M' the (S', Σ')-model defined in Example 2. Then $M'{\restriction_h}$ is the model M defined by: $[\![Bool]\!]_M = \{0, 1\}$, $[\![false]\!]_M = \{0\}$, $[\![true]\!]_M = \{1\}$, $[\![\_and\_]\!]_M(a, b) = [\![\_\&\_]\!]_{M'}(a, b) = \{a \cdot b\}$.

Example 4. We also give another example, where two sorts are collapsed by the morphism. We consider:

- a signature (S, Σ) with $S = \{Bool, Integer\}$ and $\Sigma = \{\_+\_ : Integer \times Integer \to Integer, \_\&\_ : Bool \times Bool \to Bool\}$;
- a signature (S', Σ') with $S' = \{Int\}$ and $\Sigma' = \{\_\bar{+}\_, \_\bar{\times}\_ : Int \times Int \to Int\}$;
- an (S', Σ')-model M' of (S', Σ') with $[\![Int]\!]_{M'} = \mathbb{Z}$, $[\![\_\bar{+}\_]\!]_{M'}(a, b) = \{a + b\}$ and $[\![\_\bar{\times}\_]\!]_{M'}(a, b) = \{a \cdot b\}$;
- a morphism $h : (S, \Sigma) \to (S', \Sigma')$ given by $h(Bool) = h(Integer) = Int$, $h(\_+\_) = \_\bar{+}\_$ and $h(\_\,\&\,\_) = \_\bar{\times}\_$.

The reduct $M'{\restriction_h}$ is the (S, Σ)-model such that $[\![Bool]\!]_{M'{\restriction_h}} = [\![Integer]\!]_{M'{\restriction_h}} = \mathbb{Z}$, $[\![\_+\_]\!]_{M'{\restriction_h}}(a, b) = \{a+b\}$ and $[\![\_\&\_]\!]_{M'{\restriction_h}}(a, b) = \{a \cdot b\}$, for all $a, b \in \mathbb{Z}$.

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 5 (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:

$$\begin{array}{rcll} \varphi_s & ::= & x & \text{if } x \in Var_{(S,\Sigma),s} \\ & | & \sigma(\varphi_{s_1}, \ldots, \varphi_{s_n}) & \text{if } \sigma \in \Sigma_{s_1,\ldots,s_n,s} \\ & | & \neg\varphi_s & \\ & | & \varphi_s \wedge \varphi_s & \\ & | & \exists x : s'.\varphi_s & \text{if } x \in (Var_{(S,\Sigma)})_{s'},\ s' \in S \end{array}$$

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:

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

Example 5. Let (S, Σ) be the signature

$S = \{Int, Pred\}$,
$\Sigma = \{\_ + \_, \_ * \_ : Int \times Int \rightarrow Int, \_ \leq \_, \_ < \_ : Int \times Int \rightarrow Pred\}$.

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

Definition 6 (Patterns Translation via a Signature Morphism). A matching logic signature morphism $h : (S, \Sigma) \rightarrow (S', \Sigma')$ is extended to $h : \text{PATTERNS}(S, \Sigma) \rightarrow \text{PATTERNS}(S', \Sigma')$ as follows:

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

Note that pattern translation extends the injection h defined earlier between $Var_{(S,\Sigma)}$ and $Var_{(S',\Sigma')}$.

Example 6. In the context of Example 1, $h(x \; and \; (true))$ is $h(x) \& tt$, where $x \in Var_{(S,\Sigma),Bool}$ (and therefore $h(x) \in Var_{(S',\Sigma'),Prop}$).

Definition 7 (Valuation). Let M be an (S, Σ)-model. An M-**valuation** is any function $\rho : Var_{(S,\Sigma)} \rightarrow M$ such that $\rho(x) \in [\![s]\!]_M$ for all variables $x \in Var_{(S,\Sigma),s}$ of sort s.

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

1. $\overline{\rho}(x) = \{\rho(x)\}$ for any $x \in Var_{(S,\Sigma)}$
2. $\overline{\rho}(\sigma(\varphi_1, \ldots, \varphi_n)) = [\![\sigma]\!]_M(\overline{\rho}(\varphi_1), \ldots, \overline{\rho}(\varphi_n)))$ for any $\sigma \in \Sigma$
3. $\overline{\rho}(\neg\varphi) = M_s \setminus \overline{\rho}(\varphi)$ if $\varphi \in \text{PATTERNS}_s$
4. $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$
5. $\overline{\rho}(\exists x.\varphi) = \bigcup (\overline{\rho'}(\varphi) \mid \rho' : Var_{(S,\Sigma)} \rightarrow M, \rho'|_{Var_{(S,\Sigma)} \setminus \{x\}} = \rho|_{Var_{(S,\Sigma)} \setminus \{x\}})$ for any $s' \in S, x \in Var_{(S,\Sigma),s'}$

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, $\overline{\rho}$, maps each formula into a set of elements (a subset of the domain). In particular, for variables, $\overline{\rho}$ returns the singleton set. Note that in the second case in the definition above, we use the notation in Equation (1) (in Definition 3) and therefore we might obtain several elements in the right-hand side.

Example 7. Let (S, Σ) be the signature defined in Example 5 and M be an (S, Σ)-model, where $[\![Int]\!]_M = \mathbb{Z}$, $[\![Pred]\!]_M = \{True\}$, and the operators have the usual interpretation. If $\rho(x) = 3$ and $\rho(y) = 4$, then $\overline{\rho}(x + 2) = \{5\}$, $\overline{\rho}(x \leq 5) = \{True\}$, $\overline{\rho}(y < x) = \emptyset$, and $\overline{\rho}(\exists x.x + y * 2) = \mathbb{Z}$.

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 $\rho : Var_{(S',\Sigma')} \rightarrow M'$ a valuation. The **reduct of** ρ, written $\rho{\restriction_h} : Var_{(S,\Sigma)} \rightarrow M'{\restriction_h}$, is given by $\rho{\restriction_h}(y) = \rho(h(y))$.

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

Proof. We proceed by structural induction on φ:

$$
\begin{aligned}
\overline{\rho}(h(x{:}s)) &= \{\rho(h(x){:}h(s))\} = \{\rho{\restriction_h}(x:s)\} \\
\overline{\rho}(h(\sigma(\varphi_1,\ldots,\varphi_n))) &= [\![h(\sigma)]\!]_{M'}(\overline{\rho}(h(\varphi_1)),\ldots,\overline{\rho}(h(\varphi_n))) && \text{(def.}\overline{\rho}\text{)} \\
&= [\![h(\sigma)]\!]_{M'}(\overline{\rho}{\restriction_h}(\varphi_1),\ldots,\overline{\rho}{\restriction_h}(\varphi_n)) && \text{(ind.hyp)} \\
&= [\![\sigma]\!]_{M'{\restriction_h}}(\overline{\rho}{\restriction_h}(\varphi_1),\ldots,\overline{\rho}{\restriction_h}(\varphi_n)) && \text{(def.}M'{\restriction_h}\text{)} \\
&= \overline{\rho}{\restriction_h}(\sigma(\varphi_1,\ldots,\varphi_n)) && \text{(def.}\overline{\rho}{\restriction_h}\text{)} \\
\overline{\rho}(h(\neg\,\varphi)) &= M'_{h(s)} \setminus \overline{\rho}(h(\varphi)) && \text{(def.}\overline{\rho}\text{)} \\
&= (M'{\restriction_h})_s \setminus \overline{\rho}{\restriction_h}(\varphi) && \text{(ind.hyp)} \\
&= \overline{\rho}{\restriction_h}(\neg\,\varphi) && \text{(def.}\overline{\rho}{\restriction_h}\text{)} \\
\overline{\rho}(h(\varphi_1 \wedge \varphi_2)) &= \overline{\rho}(h(\varphi_1) \wedge h(\varphi_2)) && \text{(def.}h\text{)} \\
&= \overline{\rho}(h(\varphi_1)) \cap \overline{\rho}(h(\varphi_2)) && \text{(def.}\overline{\rho}\text{)} \\
&= \overline{\rho}{\restriction_h}(\varphi_1) \cap \overline{\rho}{\restriction_h}(\varphi_2) && \text{(ind.hyp)} \\
&= \overline{\rho}{\restriction_h}(\varphi_1 \wedge \varphi_2) && \text{(def.}\overline{\rho}{\restriction_h}\text{)} \\
\overline{\rho}(h(\exists x{:}s.\varphi)) &= \overline{\rho}(\exists h(x){:}h(s).h(\varphi)) && \text{(def.}h\text{)} \\
&= \bigcup (\overline{\rho}'(h(\varphi)) \mid \rho' : Var_{(S',\Sigma')} \to M', \\
&\qquad\qquad \rho'|_{Var_{(S',\Sigma')}\setminus\{h(x)\}} = \rho|_{Var_{(S',\Sigma')}\setminus\{h(x)\}}) && \text{(def.}\overline{\rho}\text{)} \\
&= \bigcup (\overline{\rho}'{\restriction_h}(\varphi) \mid \rho' : Var_{(S',\Sigma')} \to M', \\
&\qquad\qquad \rho'|_{Var_{(S',\Sigma')}\setminus\{h(x)\}} = \rho|_{Var_{(S',\Sigma')}\setminus\{h(x)\}}) && \text{(ind.hyp)} \\
&= \bigcup (\overline{\rho}'{\restriction_h}(\varphi) \mid \rho'{\restriction_h} : Var_{(S,\Sigma)} \to M'{\restriction_h}, \\
&\qquad\qquad \rho'{\restriction_h}|_{Var_{(S,\Sigma)}\setminus\{x\}} = \rho'{\restriction_h}|_{Var_{(S,\Sigma)}\setminus\{x\}}) && \text{(def.}\overline{\rho}'{\restriction_h}\text{)} \\
&= \overline{\rho}{\restriction_h}(\exists x.\varphi) && \text{(def.}\overline{\rho}{\restriction_h}\text{)}
\end{aligned}
$$

□

Lemma 1 is used later in proving the correctness of the aggregation. To our knowledge, this particular lemma is new. However, similar results, showing that the semantics is preserved along signature morphisms, are given in [**?**], where matching logic is organised as a stratified institution.

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

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

$\lfloor x{:}s_1 \rfloor_{s_1}^{s_2}$

for each pair of sorts $s_1, s_2 \in S$. The axiom enforces $[\![\lfloor\_\rfloor_{s_1}^{s_2}]\!]_M(m_1) = [\![s_2]\!]_M$ in all models M and for all $m_1 \in M_{s_1}$, which means that for any $\rho : Var_{(S,\Sigma)} \to M$, $\overline{\rho}(\lfloor\varphi\rfloor_{s_1}^{s_2})$ is either $\emptyset$ when $\overline{\rho}(\varphi) = \emptyset$ (i.e., φ undefined in ρ), or is $[\![s_2]\!]_M$ when $\overline{\rho}(\varphi) \neq \emptyset$ (i.e., φ defined). By convention, we take the freedom to not write the definedness 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 $(x + y * 2) \wedge \lfloor x < y \rfloor_{Pred}^{Int}$. 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

$$
\begin{aligned}
\overline{\rho}_1(x + y * 2 \wedge x < y) &= \overline{\rho}_1(x + y * 2 \wedge \lfloor x < y \rfloor_{Pred}^{Int}) \\
&= \overline{\rho}_1(x + y * 2) \cap \overline{\rho}_1(\lfloor x < y \rfloor_{Pred}^{Int}) \\
&= \{12\} \cap \mathbb{Z} && (\overline{\rho}_1(x < y) = \{True\}) \\
&= \{12\}
\end{aligned}
$$

If ρ_2 is a valuation such that $\rho_1(x) = 5$ and $\rho_1(y) = 2$. Then

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

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 $(x + y * 2) \wedge (x < y)$ 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 [**?**].

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 [**?**, **?**, **?**, **?**] 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 8. A simple example of reachability rule is the one used to bubble-sort a list ($[\_]$ takes a single integer into a singleton list, $\_,\_$ concatenates lists, L_1 and L_2 are variables of sort *List* and $\langle\_\rangle$ is a free symbol whose role is aesthetic):

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

Another example is the one specifying the sorting problem:

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

Note that there is no need for an explicit associativity rule, since the model of lists is assumed to be associative and therefore the match will succeed as expected.

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., $\rightarrow_N = \bigcup_{s \in S} \rightarrow^s_N$ with $\rightarrow^s_N \subseteq M_s \times M_s$.

Example 9. Consider a model M such that $[\![List]\!]_M$ is the set of sequences of integers $x_1, \ldots, x_n$ and the transition relation $\rightarrow_N$ consisting of pairs

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

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 $\overline{\rho}(\varphi) \times \overline{\rho}(\varphi') \subseteq \rightarrow^s_N$.

Example 10. Let $N = (M, \rightarrow_N)$ be the transition system defined in Example 9 and $\varphi_1 \Rightarrow \varphi'_1$ be the reachability formula expressing a list sorting step given in Example 8. Since the transitions $\overline{\rho}(\varphi_1) \times \overline{\rho}(\varphi'_1)$ are of the form

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

```
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
Map{Var,Int}  ::=  Var ↦ Int | Map{Var,Int},Map{Var,Int} | .
CfgI          ::=  ⟨Code : Map{Var,Int}⟩
```

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

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 8 because $\rightarrow_N$ does not always include 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.

Note that all of the reachability rules in $\mathcal{A}$ need to have the same sort s: the sort of configurations of that language. The intuition is that reachability rules describe how program configurations change with each step of the program. Section 3 contains examples of reachability rules for two different languages.

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:

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

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

$$\rightarrow_N^s = \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho: Var_{(S,\Sigma)} \rightarrow M}} \overline{\rho}(\varphi) \times \overline{\rho}(\varphi')$$

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

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

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

The sorts S_i are the corresponding sets of non-terminals. Each function symbol $\sigma \in \Sigma_i$ is represented by a production in the grammar.

Some productions have no visible syntactic representation (e.g. the first two productions for `ExpI`). For example, the program expression `1 + 2` is represented as the following term: `___(+, _(1), _(2))`. Note that we adopt a Maude-like notation, where the function symbols are written in prefix notation, with arguments replaced by `_`. The `___` function symbol takes one `Op` and two `ExpI`s to an `ExpI`. The function symbol `_` (a single underscore, used in the terms `_(1)` and `_(2)`) takes an `Int` into an `ExpI`. From this point on, we do not explicitly write these silent function symbols; note however that they are important because they allows us to remain within a many-sorted algebra.

The program variables `x`, `y`, etc. are constants of sort `Var` in the term algebra. The construction `Map{Var, Int}` denotes maps from `Var` to `Int` (note that `Map` is not a polymorphic construction; we just use the syntax `Map{Var, Int}` to recall more easily the domain and codomain of the map). 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. The empty map is denoted by ".". Incorrect maps are allowed syntactically: `x` $\mapsto$ 10, `x` $\mapsto$ 9, but they are rejected semantically (by having an interpretation that is the empty set) 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 `op` $\in$ `Op` in the syntax in Figure 1, Σ_i also contains a function symbol $\texttt{op}_{Int}$ that is interpreted by M_i as the corresponding integer operation. Note that $\texttt{op}_{\texttt{Int}}$ is not part of the syntax of the language, but is only useful in the evaluation of programs. For example, while $[\![\texttt{<=}]\!]_{M_i}(4, 11)$ is `<=`$(4, 11)$ (`<=` is interpreted syntactically), we have that $[\![\texttt{<=}_{Int}]\!]_{M_i}(4, 11) = \{1\}$ (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 set $\mathcal{A}_1$ includes all rules that conform to the rule schema given in the second to last line of Figure 3 (i.e., there is a rewrite rule for each context C and for each other rewrite rule $\langle c \,:\, h\rangle \Rightarrow \langle c' \,:\, h'\rangle$). The second line also contains a rule schema: there is one rule for each possible `op`. In the rule schema for contexts (last rule), there are two invisible conversions that can take place: from `Var` to `ExpI` and from `ExpI` to `Code`. Keeping to our previous convention, we do not write these conversions explicitly.

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 `Map{Var, 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 $h[x \mapsto i]$ we denote the environment obtained from h by setting x to i. Note that $\_[\_ \mapsto \_]$ is a function symbol that takes a map, a key and a value. The interpretation of $\_[\_ \mapsto \_]$ is to update the map so that the key is associated with this value. Note that $\_[\_ \mapsto \_]$ is not part of the syntax of the language, but it is used only during evaluation. In a model of IMP, the transition system will intuitively contain all concrete instantiations $\rho(\varphi) \rightarrow_{IMP} \rho(\varphi')$ of reachability formulae in $\mathcal{A}_1$.

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 all of the reachability rules generated by the rule schema (i.e., one reachability rule for each context C and for each reachability rule $\langle e\rangle \Rightarrow \langle e'\rangle$). The variables i, i_1, i_2 are of sort `Int`, `op` ranges over all binary operators (`<=`,

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

Fig. 3. Reachability Rules for IMP.

$$\langle i_1 \ \texttt{op} \ i_2\rangle \Rightarrow \langle i_1 \ \texttt{op}_{Int} \ i_2\rangle$$
$$\langle \texttt{if i then } e_1 \texttt{ else } e_2\rangle \wedge i \ \texttt{!=}_{Int} \ 0 \Rightarrow \langle e_1\rangle$$
$$\langle \texttt{if } 0 \texttt{ then } e_1 \texttt{ else } e_2\rangle \Rightarrow \langle e_2\rangle$$
$$\langle (\lambda x.e) \ v\rangle \Rightarrow \langle e[x \mapsto v]\rangle$$
$$\langle \mu x.e\rangle \Rightarrow \langle e[x \mapsto (\mu x.e)]\rangle$$
$$\langle C[e]\rangle \Rightarrow \langle C[e']\rangle \qquad \text{if } \langle e\rangle \Rightarrow \langle e'\rangle$$
$$\text{where } C \ \texttt{::=} \ \_ \mid C \ \texttt{op} \ e \mid \texttt{if } C \texttt{ then } e_1 \texttt{ else } e_2 \mid C \ e \mid v \ C$$

Fig. 4. Reachability Rules for FUN.

+, 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'. As for IMP, the transition system of FUN intuitively contains all ground instantiations of the reachability rules in $\mathcal{A}_2$. Therefore, even if apparently the configuration $\langle e[x \mapsto v]\rangle$ is stuck when e is a variable of sort `ExpF` (because $\_[\_ \mapsto \_]$ cannot make any replacement in the atomic expression e), in fact the interpretation of $\_[\_ \mapsto \_]$ is applied on ground instantiations of the term $\langle e[x \mapsto v]\rangle$. In such a ground instantiation, the variable e will be replaced by a ground expression $\rho(e)$ where the replacement $\rho(x) \mapsto \rho(v)$ can take place. For example, in the model of `FUN`, the interpretation of $(\lambda x.x + 2)\ 1$ reduces to the interpretation of $(x + 2)[x \mapsto 1]$, which is the same as the interpretation of $1 + 2$, which in turn reduces to the interpretation of $1 \ \texttt{+}_{Int} \ 2$, which is the same as the interpretation of 3.

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 $h_1 : (S_0, \Sigma_0) \to (S_1, \Sigma_1)$ and $h_2 : (S_0, \Sigma_0) \to (S_2, \Sigma_2)$. Furthermore, we assume that the models M_1 and M_2 agree on the common signature: $M_1{\restriction_{h_1}} = M_2{\restriction_{h_2}}$. We call this common model M_0 ($M_0 = M_1{\restriction_{h_1}} = M_2{\restriction_{h_2}}$).

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 h_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 unique morphism h from (S', Σ') to (S'', Σ'').

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

Fig. 5. pushout diagram assumed throughout the paper.

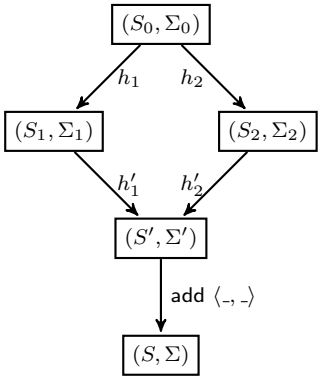

Fig. 6. The aggregation of the two signatures.

Furthermore, the pushout is unique (up to renaming). For a proof of this theorem (including uniqueness), see, e.g., [**?**]. The pushout is summarised in Figure 5, which is used throughout the paper.

One of the most important properties of the pushout construction is captured by the following proposition:

Proposition 1. In the pushout 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 $h_1(o_0) = o_1$ and $h_2(o_0) = o_2$.

It says that if an object $o' \in S' \cup \Sigma'$ 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 $\langle \_, \_ \rangle : h'_1(s_1) \times h'_2(s_2) \to s$. Formally:

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

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). In the context of Figure 5, if M_1, M_2 and M_0 are models of (S_1, Σ_1), (S_2, Σ_2) and respectively (S_0, Σ_0) such that $M_1{\restriction_{h_1}} = M_2{\restriction_{h_2}} = M_0$, there exists a unique model M' of (S', Σ') such that $M'{\restriction_{h'_1}} = M_1$ and $M'{\restriction_{h'_2}} = M_2$.

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

1. Let $s' \in S'$ be any sort. By the minimality condition in the pushout, we have that $s' = h'_1(s_1)$ for some

s_1 or that $s' = h'_2(s_2)$ for some s_2. We let

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

Note that $[\![s']\!]_{M'}$ 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, $[\![s_1]\!]_{M_1} = [\![s_0]\!]_{M_0} = [\![s_2]\!]_{M_2}$.

2. Let $\sigma' \in \Sigma'$ be any function symbol. By the minimality condition in the pushout, we have that $\sigma' = h'_1(\sigma_1)$ for some σ_1 or that $\sigma' = h'_2(\sigma_2)$ for some σ_2. We let

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

By the same reasoning as above, we have that $[\![\sigma']\!]_{M'}$ is well defined.

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

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

1. Let $s' \in S'$ be an arbitrary sort. We will show that $[\![s']\!]_{M'} = [\![s']\!]_{M''}$. Since $s' \in S'$, it follows from the minimality condition of the pushout theorem 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 $M''{\restriction}_{h'_1} = M_1$, it follows that $[\![s']\!]_{M''} = [\![s_1]\!]_{M_1}$; by the definition of M', we also have that $[\![s']\!]_{M'} = [\![s_1]\!]_{M_1}$ and therefore $[\![s']\!]_{M''} = [\![s']\!]_{M'}$.
   In the second case, because $M''{\restriction}_{h'_2} = M_2$, it follows that $[\![s']\!]_{M''} = [\![s_2]\!]_{M_2}$; by the definition of M', we also have that $[\![s']\!]_{M'} = [\![s_2]\!]_{M_2}$ and therefore $[\![s']\!]_{M''} = [\![s']\!]_{M'}$.
2. Let $\sigma' \in \Sigma'$ be an arbitrary function symbol. Since $\sigma' \in \Sigma'$, it follows from the minimality condition in the pushout theorem 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 $M''{\restriction}_{h'_1} = M_1$, it follows that $[\![\sigma']\!]_{M''}(\tilde{a}) = [\![\sigma_1]\!]_{M_1}(\tilde{a})$ for any well-sorted arguments $\tilde{a} = a_1, \ldots, a_n$ of the functions; by the definition of M', we also have that $[\![\sigma']\!]_{M'}(\tilde{a}) = [\![\sigma_1]\!]_{M_1}(\tilde{a})$ and therefore $[\![\sigma']\!]_{M''}(\tilde{a}) = [\![\sigma']\!]_{M'}(\tilde{a})$ for any well-sorted $\tilde{a}$, which is what we had to show.
   In the second case, because $M''{\restriction}_{h'_2} = M_2$, it follows that $[\![\sigma']\!]_{M''}(\tilde{a}) = [\![\sigma_2]\!]_{M_2}(\tilde{a})$ for any well-sorted arguments $\tilde{a} = a_1, \ldots, a_n$ of the function; by the definition of M', we also have that $[\![\sigma']\!]_{M'}(\tilde{a}) = [\![\sigma_2]\!]_{M_2}(\tilde{a})$ and therefore $[\![\sigma']\!]_{M''}(\tilde{a}) = [\![\sigma']\!]_{M'}(\tilde{a})$ for any well-sorted $\tilde{a}$, which is what we had to show.

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

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 $\langle \_, \_ \rangle : h'_1(s_1) \times h'_2(s_2) \to s$, 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. $[\![s]\!]_M = [\![h'_1(s_1)]\!]_{M'} \times [\![h'_2(s_2)]\!]_{M'}$ (the set of configurations is the cartesian product of the sets of configurations of the individual programs) and
2. $[\![\langle \_, \_ \rangle]\!]_M(a, b) = \{\langle [\![a]\!]_{M'}, [\![b]\!]_{M'} \rangle\}$ (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).

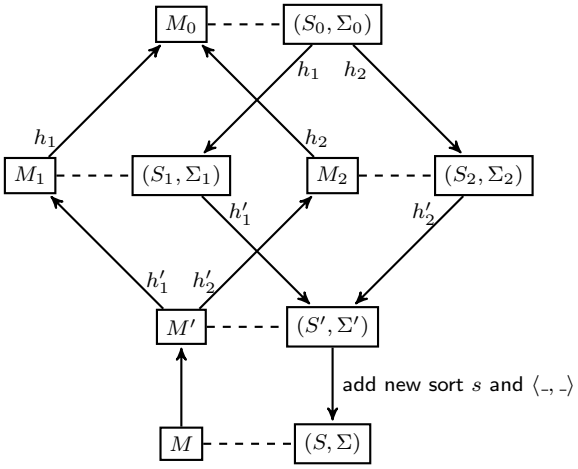

Fig. 7. Construction of the aggregated model M of (S, Σ)

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.

Remark 2. We note that $M\restriction_{h'_i} = M'\restriction_{h'_i}$ (for $i \in \{1, 2\}$), since the only difference between M and M' is that M interprets the additional sort and pairing symbol.

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 $h_1 : (S_0, \Sigma_0) \to (S_1, \Sigma_1)$ and $h_2 : (S_0, \Sigma_0) \to (S_2, \Sigma_2)$, and the models M_1 and M_2 agree on the common signature: $M_1\restriction_{h_1} = M_2\restriction_{h_2}$. We further consider the standard model $N_i = (M_i, \to_i)$ 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 agregated matching logic semantic domain (S, Σ, M), we now show how the transition relation $\to$ and the reachability rules $\mathcal{A}$ of the aggregated language are defined from from the relations $\to_1$, $\to_2$, and respectively the reachability rules $\mathcal{A}_1$, $\mathcal{A}_2$ of the initial languages. Let s denote the newly added sort in S and let $\to_i^{h'_i(s_i)} \subseteq M_{h'_i(s_i)} \times M_{h'_i(s_i)}$ denote the relation $\to_i^{h'_i(s_i)} = \to_i^{s_i} \subseteq (M_i)_{s_i} \times (M_i)_{s_i}$ for $i \in \{1, 2\}$ (note that $M_{h'_i(s_i)} = M'_{h'_i(s_i)} = (M_i)_{s_i}$ for $i \in \{1, 2\}$).

We identify three types of language aggregations, depending on how the $\to$ transition relation is defined from $\to_1$ and $\to_2$. Each of the three constructions could be useful in various contexts:

1. $\to_1 \otimes_p \to_2 = \bigcup_{s' \in S} (\to_1 \otimes_p \to_2)_{s'}$ is the synchronous **p**roduct of the two transition relations, i.e.,

$$(\to_1 \otimes_p \to_2)_s = \{\langle \gamma_1, \gamma_2 \rangle \to \langle \gamma'_1, \gamma'_2 \rangle \mid \gamma_1 \to_1^{h'_1(s_1)} \gamma'_1 \text{ and } \gamma_2 \to_2^{h'_2(s_2)} \gamma'_2\}$$
$$(\to_1 \otimes_p \to_2)_{s'} = \emptyset \text{ for } s' \neq s$$

2. $\to_1 \otimes_a \to_2 = \bigcup_{s' \in S} (\to_1 \otimes_a \to_2)_{s'}$ is the **a**synchronous interleaving product of the two transition relations, i.e.,

$$(\to_1 \otimes_a \to_2)_s = (id^{h'_1(s_1)} \otimes_p \to_2^{h'_2(s_2)}) \cup (\to_1^{h'_1(s_1)} \otimes_p id^{h'_2(s_2)})$$
$$(\to_1 \otimes_a \to_2)_{s'} = \emptyset \text{ for } s' \neq s$$

   where $id = \bigcup_{s' \in S} id^{s'}$ is the identity relation.
3. finally, $\to_1 \otimes \to_2 = (\to_1 \otimes_a \to_2) \cup (\to_1 \otimes_p \to_2)$ is the (general) product of $\to_1$ and $\to_2$.

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 synchronous 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 reachability rules A for the aggregated language from the set of reachability rules 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 reachability rules. 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 Reachability Rules for the Synchronous 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 $(M, \to_1 \otimes_p \to_2)$:

Theorem 3 (correctness for synchronous product). Let $\mathcal{A} = \mathcal{A}_1 \otimes_p \mathcal{A}_2$ and $\to = \to_1 \otimes_p \to_2$. Then $(M, \to)$ is the standard model for $L_1 \otimes_p L_2 = (S, \Sigma, M, \mathcal{A})$.

Proof. We have $(\to_1 \otimes_p \to_2)_{s'} = \emptyset$ for $s' \neq s$ by the definition of $\to_1 \otimes_p \to_2$, so we have only to show that

$$(\to_1 \otimes_p \to_2)_s = \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}(\varphi) \times \overline{\rho}(\varphi').$$

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 $\varphi_1 \Rightarrow \varphi_1' \in \mathcal{A}_1$, $\varphi_2 \Rightarrow \varphi_2' \in \mathcal{A}_2$, $\rho_1, \rho_2 : Var_{(S,\Sigma)} \to M$, there exists $\rho : Var_{(S,\Sigma)} \to M$ such that

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

We have:

$$\begin{aligned}
(\to_1 \otimes_p \to_2)_s &= \left\{\langle \gamma_1, \gamma_2\rangle \to \langle \gamma_1', \gamma_2'\rangle \mid \gamma_1 \to_1^{h_1'(s_1)} \gamma_1', \gamma_2 \to_2^{h_2'(s_2)} \gamma_2'\right\} && \text{(def.}\otimes_p\text{)} \\
&= \left\{\langle \gamma_1, \gamma_2\rangle \to \langle \gamma_1', \gamma_2'\rangle \mid \gamma_1 \to_1^{s_1} \gamma_1', \gamma_2 \to_2^{s_2} \gamma_2'\right\} && \text{(def. } \to_i^{h_i'(s_i)}\text{)} \\
&= \Big\{\langle \gamma_1, \gamma_2\rangle \to \langle \gamma_1', \gamma_2'\rangle \mid \gamma_1 \to \gamma_1' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho_1 : Var_{(S_1,\Sigma_1)} \to M_1}} \overline{\rho}_1(h_1'(\varphi)) \times \overline{\rho}_1(h_1'(\varphi')), \\
&\qquad \gamma_2 \to \gamma_2' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho_2 : Var_{(S_2,\Sigma_2)} \to M_2}} \overline{\rho}_2(h_2'(\varphi)) \times \overline{\rho}_2(h_2'(\varphi'))\Big\} && ((M_i, \to_i) \text{ std.}) \\
&= \Big\{\langle \gamma_1, \gamma_2\rangle \to \langle \gamma_1', \gamma_2'\rangle \mid \gamma_1 \to \gamma_1' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}\restriction_{h_1'}(\varphi) \times \overline{\rho}\restriction_{h_1'}(\varphi'), \\
&\qquad \gamma_2 \to \gamma_2' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}\restriction_{h_2'}(\varphi) \times \overline{\rho}\restriction_{h_2'}(\varphi')\Big\} && \text{(Lemma 1)}
\end{aligned}$$

$$= \bigcup_{\substack{\varphi_1 \Rightarrow \varphi_1' \in \mathcal{A}_1 \\ \varphi_2 \Rightarrow \varphi_2' \in \mathcal{A}_2 \\ \rho : Var_{(S,\Sigma)} \to M}} \left\{ \begin{array}{r} \langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1', \gamma_2' \rangle \mid \gamma_1 \to \gamma_1' \in \overline{\rho}\restriction_{h_1'}(\varphi_1) \times \overline{\rho}\restriction_{h_1'}(\varphi_1'), \\ \gamma_2 \to \gamma_2' \in \overline{\rho}\restriction_{h_2'}(\varphi_2) \times \overline{\rho}\restriction_{h_1'}(\varphi_2')) \end{array} \right\} \quad \text{(cf. (2))}$$

$$= \bigcup_{\substack{\varphi_1 \Rightarrow \varphi_1' \in \mathcal{A}_1 \\ \varphi_2 \Rightarrow \varphi_2' \in \mathcal{A}_2 \\ \rho : Var_{(S,\Sigma)} \to M}} \left\{ \begin{array}{r} \langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1', \gamma_2' \rangle \mid \gamma_1 \to \gamma_1' \in \overline{\rho}(h_1'(\varphi_1)) \times \overline{\rho}(h_1'(\varphi_1')), \\ \gamma_2 \to \gamma_2' \in \overline{\rho}(h_2'(\varphi_2)) \times \overline{\rho}(h_2'(\varphi_2')) \end{array} \right\} \quad \text{(Lemma 1)}$$

$$= \bigcup_{\substack{\varphi_1 \Rightarrow \varphi_1' \in \mathcal{A}_1 \\ \varphi_2 \Rightarrow \varphi_2' \in \mathcal{A}_2 \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}(\langle h_1'(\varphi_1), h_2'(\varphi_2) \rangle) \times \overline{\rho}(\langle h_1'(\varphi_1'), h_2'(\varphi_2') \rangle) \quad \text{(reorg.)}$$

$$= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}(\varphi) \times \overline{\rho}(\varphi') \quad \text{(def.}\mathcal{A}\text{)}$$

□

5.2. The Reachability Rules for the Asynchronous Product with Interleaving Semantics

We let

$$\mathcal{A}_1 \otimes_a \mathcal{A}_2 = \bigcup_{\varphi \Rightarrow \varphi' \in \mathcal{A}_1} \{\langle h_1'(\varphi), y \rangle \Rightarrow \langle h_1'(\varphi'), y \rangle\} \cup \bigcup_{\varphi \Rightarrow \varphi' \in \mathcal{A}_2} \{\langle x, h_2'(\varphi) \rangle \Rightarrow \langle x, h_2'(\varphi') \rangle\}$$

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 $A_1 \otimes_a A_2$ is indeed a formal specification of the language $(M, \to_1 \otimes_a \to_2)$:

Theorem 4 (correctness for asynchronous product). Let $\mathcal{A} = \mathcal{A}_1 \otimes_a \mathcal{A}_2$ and $\to = \to_1 \otimes_a \to_2$. We have that $(M, \to)$ 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 $(\to_1 \otimes_a \to_2)_{s'} = \emptyset$ for $s' \neq s$ by the definition of $\to_1 \otimes_a \to_2$, so we have only to show that

$$(\to_1 \otimes_a \to_2)_s = \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}(\varphi) \times \overline{\rho}(\varphi').$$

We have:

$$\begin{aligned} (\to_1 \otimes_a \to_2)_s &= (id^{h_1'(s_1)} \otimes_p \to_2^{h_2'(s_2)}) \cup (\to_1^{h_1'(s_1)} \otimes_p id^{h_2'(s_2)}) && \text{(def.}\otimes_a\text{)} \\ &= \{\langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1, \gamma_2' \rangle \mid \gamma_2 \to_2^{s_2} \gamma_2'\} \cup \\ &\quad \{\langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1', \gamma_2 \rangle \mid \gamma_1 \to_1^{s_1} \gamma_1'\} && \text{(def.}\otimes_p\text{)} \end{aligned}$$

$$= \Big\{ \langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1, \gamma_2' \rangle \mid \gamma_2 \to \gamma_2' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho_2 : Var_{(S_2,\Sigma_2)} \to M_2}} \overline{\rho}_2(h_2'(\varphi)) \times \overline{\rho}_2(h_2'(\varphi')) \Big\} \cup$$

$$\Big\{ \langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1', \gamma_2 \rangle \mid \gamma_1 \to \gamma_1' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho_1 : Var_{(S_1,\Sigma_1)} \to M_1}} \overline{\rho}_1(h_1'(\varphi)) \times \overline{\rho}_1(h_1'(\varphi')) \Big\} \qquad (\to_i \text{ is std.})$$

$$= \Big\{ \langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1, \gamma_2' \rangle \mid \gamma_2 \to \gamma_2' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}\restriction_{h_2'}(\varphi) \times \overline{\rho}\restriction_{h_2'}(\varphi') \Big\} \cup$$

$$\Big\{ \langle \gamma_1, \gamma_2 \rangle \to \langle \gamma_1', \gamma_2 \rangle \mid \gamma_1 \to \gamma_1' \in \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}\restriction_{h_1'}(\varphi) \times \overline{\rho}\restriction_{h_1'}(\varphi') \Big\} \qquad \text{(Lemma 1)}$$

$$= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho : Var_{(S,\Sigma)} \to M}} \Big\{ \langle \rho(x), \gamma_2 \rangle \to \langle \rho(x), \gamma_2' \rangle \mid \gamma_2 \to \gamma_2' \in \overline{\rho}(h_2'(\varphi)) \times \overline{\rho}(h_2'(\varphi')) \Big\} \cup$$

$$\bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho : Var_{(S,\Sigma)} \to M}} \Big\{ \langle \gamma_1, \rho(y) \rangle \to \langle \gamma_1', \rho(y) \rangle \mid \gamma_1 \to \gamma_1' \in \overline{\rho}(h_1'(\varphi)) \times \overline{\rho}(h_1'(\varphi')) \Big\} \qquad \text{(reorg.)}$$

$$= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_2 \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}(\langle x, h_2'(\varphi) \rangle) \times \overline{\rho}(\langle x, h_2'(\varphi') \rangle) \cup$$

$$\bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A}_1 \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}(\langle h_1'(\varphi), y \rangle) \times \overline{\rho}(\langle h_1'(\varphi'), y \rangle) \qquad \text{(reorg.)}$$

$$= \bigcup_{\substack{\varphi \Rightarrow \varphi' \in \mathcal{A} \\ \rho : Var_{(S,\Sigma)} \to M}} \overline{\rho}(\varphi) \times \overline{\rho}(\varphi') \qquad \text{(def.}\mathcal{A}\text{)}$$

where $\overline{\rho}\restriction_{h_2'}(x) = \gamma_1$, $\overline{\rho}\restriction_{h_1'}(y) = \gamma_2$. $\square$

5.3. The Reachability Rules for the General Product

For the general product, which allows for both interleaving and synchronous 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 $\to = \to_1 \otimes \to_2$. We have that $(M, \to)$ 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 $L_2 = (S_2, \Sigma_2, M_2, \mathcal{A}_2)$, 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 $\gamma \to_i \gamma'$ the one-step transition relation of language L_i (for $i \in \{1,2\}$)). We denote by $\to_i^*$ and $\to_i^+$ 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. $\models \varphi \Rightarrow_i \varphi'$ means that for any ρ and for any γ such that $\gamma \in \overline{\rho}(\varphi)$, there exists γ' such that $\gamma \to_i \gamma'$ and $\gamma' \in \overline{\rho}(\varphi')$ (any configuration γ matching φ moves in one step into a configuration γ' matching φ')
2. $\models \varphi \Rightarrow_i^+ \varphi'$ means that for any ρ and for any γ such that $\gamma \in \overline{\rho}(\varphi)$, there exists γ' such that $\gamma \to_i^+ \gamma'$ and $\gamma' \in \overline{\rho}(\varphi')$ (any configuration γ matching φ moves in one or more steps into a configuration γ' matching φ')
3. $\models \varphi \Rightarrow_i^* \varphi'$ means that for any ρ and for any γ such that $\gamma \in \overline{\rho}(\varphi)$, there exists γ' such that $\gamma \to_i^* \gamma'$ and $\gamma' \in \overline{\rho}(\varphi')$ (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 $[\![\varphi]\!]$), is the set of all concrete configurations that satisfy it:

$$[\![\varphi]\!] = \bigcup_{\rho: Var_{(S,\Sigma)} \to M} \overline{\rho}(\varphi).$$

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

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

Example 11. The following set

$$E = \{\langle\langle \texttt{skip} : (\texttt{x} \mapsto i, \_)\rangle, \langle j\rangle\rangle \wedge i =_{Int} j\} \tag{3}$$

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 $[\![E]\!]$ of set E above holds exactly the set of pairs of terminal configurations in which the two programs should end in order for them to compute the same result.

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

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

All definitions for program equivalence contain a part of the form "executions of P_1 and P_2 on equal inputs". This contraint 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 $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$ such that γ_1 terminates in $\rightarrow_1$ with γ_1' and γ_2 terminates in $\rightarrow_2$ with γ_2', we have that $\langle \gamma_1', \gamma_2' \rangle \in [\![E]\!]$.

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 $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$, γ_1 terminates in $\rightarrow_1$ if and only if γ_2 terminates in $\rightarrow_2$.

Full equivalence is defined as "partially equivalent and mutually terminate" and is formally expressed as follows:

Definition 19 (Full Equivalence). We write $\models \varphi \Downarrow^{\infty} E$, and say that φ **reaches** E, iff for all configurations $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$ 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 $\langle \gamma_1', \gamma_2' \rangle \in [\![E]\!]$.

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 $\models \varphi \Downarrow_! E$, and say that φ **totally reaches** E, iff for all configurations $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$ 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 $\langle \gamma_1', \gamma_2' \rangle \in [\![E]\!]$.

Example 12. Let $E = \{\langle \langle \texttt{skip} : (\texttt{n} \mapsto i, \_) \rangle, \langle i \rangle \rangle\}$. 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 `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 `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 `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 `n`) are ignored.

We consider the formulae φ_1, φ_2 and φ_3 (note that $\texttt{PGM}_1$ and $\texttt{PGM}_2$ are defined in Figure 8) to be defined as follows:

$$
\begin{array}{rcl}
\varphi_1 & = & \langle \langle \texttt{code}_1 \ : \ \texttt{n} \mapsto n \rangle, \langle \texttt{exp}_1 \ n \rangle \rangle \wedge n \ \texttt{>=}_{Int} \ 0 \\
\varphi_2 & = & \langle \langle \texttt{code}_2 \ : \qquad \rangle, \langle \texttt{exp}_2 \ 0 \rangle \rangle \\
\varphi_3 & = & \langle \langle \texttt{code}_3 \ : \ \texttt{n} \mapsto n \rangle, \langle \texttt{exp}_3 \ n \ 1 \rangle \rangle.
\end{array}
$$

where:

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

The formula φ_1 denotes pairs of IMP and respectively FUN configurations where the IMP program executes $\texttt{code}_1$ starting in an environment where the program variable `n` is mapped to the non-negative integer n and the FUN program executes the recursive function $\texttt{exp}_1$ with the same non-negative value n as argument. The IMP code $\texttt{code}_1$ compute the sum of the integers 1 up to the value of the program variable `n`, while the FUN function $\texttt{exp}_1$ 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 $\models \varphi_2 \Downarrow^{\infty} E$.

Finally, the φ_3 pattern contains two programs that both compute the Collatz function. 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 fully equivalent, independently of the conjecture: if they both terminate they produce the same

```
PGM1  ≡  c := n; n := 1; LOOP1
LOOP1 ≡  while (c != 1)
           n := n + 1;
           if (c % 2 != 0)
             then c := 3 * c + 1
             else c := c / 2
```

```
PGM2  ≡  μf.λn.λa.LOOP2
LOOP2 ≡  if (n != 1)
           then if (n % 2 != 0)
             then f (3 * n + 1) (a + 1)
             else f (n / 2) (a + 1)
           else a
```

Fig. 8. An IMP program $\texttt{PGM}_1$ and a FUN program $\texttt{PGM}_2$ 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.

$$\textsc{Axiom}\ \frac{\varphi \in E}{\vdash \varphi \Downarrow^{\infty} E} \qquad \textsc{Conseq}\ \frac{\models \varphi \to \exists \tilde{x}.\varphi' \qquad \vdash \varphi' \Downarrow^{\infty} E}{\vdash \varphi \Downarrow^{\infty} E} \qquad \textsc{Case Analysis}\ \frac{\vdash \varphi \Downarrow^{\infty} E \qquad \vdash \varphi' \Downarrow^{\infty} E}{\vdash \varphi \vee \varphi' \Downarrow^{\infty} E}$$

$$\textsc{Step}\ \frac{\models \varphi_1 \Rightarrow_1^* \varphi_1' \qquad \models \varphi_2 \Rightarrow_2^* \varphi_2' \qquad \vdash \langle \varphi_1', \varphi_2' \rangle \Downarrow^{\infty} E}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^{\infty} E}$$

$$\textsc{Circularity}\ \frac{\models \varphi_1 \Rightarrow_1^+ \varphi_1' \qquad \models \varphi_2 \Rightarrow_2^+ \varphi_2' \qquad \vdash \langle \varphi_1', \varphi_2' \rangle \Downarrow^{\infty} E \cup \{\langle \varphi_1, \varphi_2 \rangle\}}{\vdash \langle \varphi_1, \varphi_2 \rangle \Downarrow^{\infty} E}$$

Fig. 9. Full Equivalence Proof System.

result and if one does not terminate, the other does not terminate either. Therefore we have $\models \varphi_3 \Downarrow^{\infty} E$ as well.

7. Proving Full 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 full 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 full equivalence of programs.

Note that although we used proof rule names similar to those in prior work on language-independent program verification [**?**, **?**, **?**, **?**], our rules in Figure 9 are new. Unlike in prior work where the proof rules were used to derive program reachability claims in one object language, our proof rules are used for establishing binary relationships between programs in two different languages. This is reminiscent of bisimulation [**?**], although note that two related programs are allowed to execute any number of steps in their corresponding semantics before other related programs are reached again. The main challenge in designing the proof system in Figure 9 was to properly capture the mutual termination of the two programs involved in a relation pair.

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

The second rule is Conseq(uence). This rule allows to perform domain reasoning in the formula. It states that if a formula φ implies another formula $\exists \tilde{x}.\varphi'$ (which means that φ' is more general than φ) and the sequent $\vdash \varphi' \Downarrow^{\infty} E$ is derivable, then $\vdash \varphi \Downarrow^{\infty} E$ must also be derivable. Recall that the judgement $\vdash \varphi \Downarrow^{\infty} E$ holds if then programs denoted by $[\![\varphi]\!]$ reach $[\![E]\!]$. 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 in the sense that $[\![\varphi]\!] \subseteq [\![\varphi']\!]$. Moreover $[\![\exists \tilde{x}.\varphi']\!] = [\![\varphi']\!]$. The existential quantifier allows to hide certain variables. Therefore if we are able to prove the equivalence for φ', then φ must also hold (since fewer programs "fit" φ). This rule is used in the example proof tree below (in Figure 10) to rearrange a formula of

the form $(n \texttt{!=}_{Int} 1 \vee n =_{Int} 1) \wedge \ldots$ into $\texttt{true} \wedge \ldots$. Another possible use of Conseq would be, for example, to transform a more particular case, like "$n = 20$", into a more general case "n is even" in order to be able to apply other rules. In the implementation of this rule, an oracle for deciding matching logic implication is needed.

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!=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 [**?**], 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 ($\models \varphi_1 \Rightarrow_1^+ \varphi_1'$, $\models \varphi_2 \Rightarrow_2^+ \varphi_2'$) 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 behaviour (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 [**?**], 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

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

The following definitions will be useful in the proof of soundness. Let $G \subseteq [\![s]\!]_M$ denote a set of configurations in the aggregated language. By the definition of $[\![s]\!]_M$, we have that $G \subseteq [\![h_1'(s_1)]\!]_M \times [\![h_2'(s_2)]\!]_M$.

Definition 22. We say that a configuration $\langle \gamma_1, \gamma_2 \rangle \in [\![s]\!]_M$ in the aggregated language **reaches** G, written $\langle \gamma_1, \gamma_2 \rangle \rightarrow^* G$, if there exist configurations $\gamma_1' \in [\![h_1'(s_1)]\!]_M$ and $\gamma_2' \in [\![h_2'(s_2)]\!]_M$ 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 $\langle \gamma_1, \gamma_2 \rangle \in [\![s]\!]_M$ **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 $\langle \gamma_1, \gamma_2 \rangle \in [\![s]\!]_M$ 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 2. For all sets of matching logic formulae E of sort s and for any matching logic formula φ of sort s, we have that:

$\models \varphi \Downarrow^{\infty} E$ iff for all $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!], \langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} [\![E]\!]$.

Proof. Immediately, by unfolding the definitions. □

The next lemma is the core of our soundness proof.

Lemma 3 (Circularity Principle). Let F be a set of matching formulae of sort s. If for each $\langle \gamma_1, \gamma_2 \rangle \in [\![F]\!]$ there exist $\gamma_1' \in [\![h_1'(s_1)]\!]_M$, $\gamma_2' \in [\![h_2'(s_2)]\!]_M$ such that

$\gamma_1 \rightarrow_1^+ \gamma_1', \gamma_2 \rightarrow_2^+ \gamma_2'$ and $\langle \gamma_1', \gamma_2' \rangle \rightarrow^{*,\infty} [\![E \cup F]\!]$,

then

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

Proof. By Lemma 2, to prove $\models F \Downarrow^{\infty} E$ it is enough to show that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} [\![E]\!]$ for all $\langle \gamma_1, \gamma_2 \rangle \in [\![F]\!]$. Let $\langle \gamma_1, \gamma_2 \rangle \in [\![F]\!]$ be arbitrarily chosen.

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

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

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

1. for $i = 1$, we have that by 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 $\langle \gamma_1^1, \gamma_2^1 \rangle \rightarrow^{*,\infty} [\![E \cup F]\!]$, which implies that $P(1)$ holds.
2. for $i > 1$, we assume that $P(i-1)$ holds and we prove that $P(i)$ holds. If $P(i-1)$ holds because $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} [\![E]\!]$, then by definition $P(i)$ holds as well.
   Otherwise, we have that there exist configurations $\gamma_1^1, \ldots, \gamma_1^{i-1}, \gamma_2^1, \ldots, \gamma_2^{i-1}$ such that:
   (a) $\gamma_1 \rightarrow_1^+ \gamma_1^1 \rightarrow_1^+ \gamma_1^2 \rightarrow_1^+ \ldots \rightarrow_1^+ \gamma_1^{i-1}$,
   (b) $\gamma_2 \rightarrow_2^+ \gamma_2^1 \rightarrow_2^+ \gamma_2^2 \rightarrow_2^+ \ldots \rightarrow_2^+ \gamma_2^{i-1}$ and
   (c) $\langle \gamma_1^{i-1}, \gamma_2^{i-1} \rangle \rightarrow^{*,\infty} [\![F]\!]$

   As $\langle \gamma_1^{i-1}, \gamma_2^{i-1} \rangle \rightarrow^{*,\infty} [\![F]\!]$, we have by hypothesis that there exist γ_1^i $(= \gamma_1')$ and γ_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 $\langle \gamma_1^i, \gamma_2^i \rangle \rightarrow^{*,\infty} [\![E \cup F]\!]$. But this implies that $P(i)$ holds.

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

We will now show by contradiction that $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} [\![E]\!]$. We will assume by contradiction that $\langle \gamma_1, \gamma_2 \rangle \not\rightarrow^{*,\infty} [\![E]\!]$. As $P(i)$ holds for all naturals $i > 0$, it follows that for each $i > 0$, there exist $\gamma_1^1, \ldots, \gamma_1^i, \gamma_2^1, \ldots, \gamma_2^i$ such that $\gamma_1 \rightarrow_1^+ \gamma_1^1 \rightarrow_1^+ \ldots \rightarrow_1^+ \gamma_1^i$ and $\gamma_2 \rightarrow_2^+ \gamma_2^1 \rightarrow_2^+ \ldots \rightarrow_2^+ \gamma_2^i$.

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 $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} [\![E]\!]$, which yields a contradiction. As our assumption $\langle \gamma_1, \gamma_2 \rangle \not\rightarrow^{*,\infty} [\![E]\!]$ led to a contradiction, it follows that it must hold and therefore $\langle \gamma_1, \gamma_2 \rangle \rightarrow^{*,\infty} [\![E]\!]$, which is what we had to show. □

If we use a coinductive definition for paths [**?**], then Lemma 3 states that $[\![F]\!]$ satisfies the coinduction principle [**?**], which, for the case of lists, is also known as Park's principle [**?**, p. 371].

We need one additional helper lemma:

Lemma 4. For any formulae φ, φ' of some sort s, we have that $\models \varphi \rightarrow \varphi'$ implies $[\![\varphi]\!] \subseteq [\![\varphi']\!]$.

Proof. By definition, $\models \varphi \to \varphi'$ iff $\overline{\rho}(\varphi \to \varphi') = M_s$ for every valuation ρ.
Therefore, for any valuation ρ, we have that:

$$\begin{aligned}
M_s =& \overline{\rho}(\varphi \to \varphi') \\
=& \overline{\rho}(\neg\varphi \vee \varphi') \\
=& \overline{\rho}(\neg(\neg\neg\varphi \wedge \neg\varphi')) \\
=& \overline{\rho}(\neg(\varphi \wedge \neg\varphi')) \\
=& M_s \setminus \overline{\rho}(\varphi \wedge \neg\varphi') \\
=& M_s \setminus (\overline{\rho}(\varphi) \cap \overline{\rho}(\neg\varphi')) \\
=& M_s \setminus (\overline{\rho}(\varphi) \cap (M_s \setminus \overline{\rho}(\varphi'))).
\end{aligned}$$

Therefore $\overline{\rho}(\varphi) \cap (M_s \setminus \overline{\rho}(\varphi')) = \emptyset$ for any valuation ρ. This implies $\overline{\rho}(\varphi) \subseteq \overline{\rho}(\varphi')$ for any valuation ρ. We conclude that $[\![\varphi]\!] = \bigcup_\rho \overline{\rho}(\varphi) \subseteq \bigcup_\rho \overline{\rho}(\varphi') = [\![\varphi']\!]$. $\square$

The circularity principle (Lemma 3) 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 $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$ be an arbitrary configuration. Let $\gamma_1' = \gamma_1$ and $\gamma_2' = \gamma_2$. We have that $\gamma_1 \to_1^* \gamma_1'$, $\gamma_2 \to_2^* \gamma_2'$ and $\langle \gamma_1', \gamma_2' \rangle \in [\![E]\!]$. 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 $\langle \gamma_1, \gamma_2 \rangle \in [\![\langle \varphi_1, \varphi_2 \rangle]\!]$ be an arbitrary configuration. By Lemma 2, it is sufficient to show that $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$.
   As $\langle \gamma_1, \gamma_2 \rangle \in [\![\langle \varphi_1, \varphi_2 \rangle]\!]$, we have that there exists $\rho : Var_{(S,\Sigma)} \to M$ such that $\gamma_1 \in \overline{\rho}(\varphi_1)$ and $\gamma_2 \in \overline{\rho}(\varphi_2)$. Since $\models \varphi_1 \Rightarrow_1^* \varphi_1'$, there exists γ_1' such that $\gamma_1 \to_1^* \gamma_1'$ and $\gamma_1' \in \overline{\rho}(\varphi_1')$. Similarly, there exists γ_2' such that $\gamma_2 \to_2^* \gamma_2'$ and $\gamma_2' \in \overline{\rho}(\varphi_2')$. Therefore $\langle \gamma_1', \gamma_2' \rangle \in \overline{\rho}(\langle \varphi_1', \varphi_2' \rangle)$.
   We also have that $\models \langle \varphi_1, \varphi_2 \rangle \Downarrow^\infty E$ and therefore, by Lemma 2, $\langle \gamma_1', \gamma_2' \rangle \to^{*,\infty} [\![E]\!]$. But $\gamma_1 \to_1^* \gamma_1'$ and $\gamma_2 \to_2^* \gamma_2'$. Therefore $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$ 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 $\models \varphi \to \exists \tilde{x}.\varphi'$ 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 $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$. By Lemma 2, it is sufficient to show that $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$.
   As $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$ and $\models \varphi \to \exists \tilde{x}.\varphi'$, we have by Lemma 4 that $\langle \gamma_1, \gamma_2 \rangle \in [\![\exists \tilde{x}.\varphi']\!]$. From the definition of $[\![\_]\!]$, we immediately have that $[\![\exists \tilde{x}.\varphi']\!] = [\![\varphi']\!]$ and therefore $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi']\!]$. But we also have that $\models \varphi' \Downarrow^\infty E$, and therefore, by Lemma 2, $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$, which is what we had to show.
4. if the last rule to be applied is Case Analysis, we have by the induction hypothesis that $\models \varphi \Downarrow^\infty E$ and that $\models \varphi' \Downarrow^\infty E$. We show that $\models \varphi \vee \varphi' \Downarrow^\infty E$.
   Let $\langle \gamma_1, \gamma_2 \rangle$ be an arbitrary configuration such that $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi \vee \varphi']\!]$. By Lemma 2, it is sufficient to show that $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$.
   As $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi \vee \varphi']\!]$, we have that $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$ or that $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi']\!]$:
   (a) in the first case, we have that $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi]\!]$. We also have that $\models \varphi \Downarrow^\infty E$ (by the induction hypothesis). By Lemma 2, we obtain that $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$.
   (b) in the second case, we have that $\langle \gamma_1, \gamma_2 \rangle \in [\![\varphi']\!]$. We also have that $\models \varphi' \Downarrow^\infty E$ (by the induction hypothesis). By Lemma 2, we obtain that $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$.

   In either case we have obtained that $\langle \gamma_1, \gamma_2 \rangle \to^{*,\infty} [\![E]\!]$, which is what we had to prove.
5. if the last rule to be applied is Circularity, then, by the induction hypothesis, there exists a configuration

$\langle \varphi_1', \varphi_2' \rangle$ such that:

$$\langle \varphi_1', \varphi_2' \rangle \Downarrow^{\infty} E \cup \{\langle \varphi_1, \varphi_2 \rangle\}, \tag{4}$$

$$\models \varphi_1 \Rightarrow_1^+ \varphi_1' \text{ and} \tag{5}$$

$$\models \varphi_2 \Rightarrow_2^+ \varphi_2'. \tag{6}$$

Let $\langle \gamma_1, \gamma_2 \rangle \in [\![\langle \varphi_1, \varphi_2 \rangle]\!]$ be an arbitrary configuration. By definition, there exists $\rho : Var_{(S,\Sigma)} \to M$ such that $\gamma_1 \in \overline{\rho\restriction_{h_1'}}(\varphi_1)$ and $\gamma_2 \in \overline{\rho\restriction_{h_2'}}(\varphi_2)$.
By Equation (5) and (6), we obtain that there exist γ_1' and γ_2' such that

$$\gamma_1 \to_1^+ \gamma_1', \tag{7}$$

$$\gamma_2 \to_2^+ \gamma_2' \tag{8}$$

and $\gamma_1' \in \overline{\rho\restriction_{h_1'}}(\varphi_1')$ and $\gamma_2' \in \overline{\rho\restriction_{h_2'}}(\varphi_2')$. From $\gamma_1' \in \overline{\rho\restriction_{h_1'}}(\varphi_1')$ and $\gamma_2' \in \overline{\rho\restriction_{h_2'}}(\varphi_2')$, we obtain by Equation (4) and Lemma 2 that

$$\langle \gamma_1', \gamma_2' \rangle \to^{*,\infty} E \cup \{\langle \varphi_1, \varphi_2 \rangle\}. \tag{9}$$

Let $F = \{\langle \varphi_1, \varphi_2 \rangle\}$. We have shown that the three conditions of the Circularity Principle (Lemma 3) are met (Equations 7, 8 and 9). Therefore we can apply Lemma 3 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 full 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 $E = \{\langle \langle \texttt{skip : (n} \mapsto i, \_) \rangle, \langle i \rangle \rangle\}$.

It says that an IMP configuration $\langle \texttt{skip : (n} \mapsto i, \_) \rangle$ (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 so that the tautology $(n \texttt{ !=}_{Int} 1) \vee (n \texttt{ =}_{Int} 1)$ appears explicitly in it) and Case Analysis to see whether we have to immediately quit the loop $(n \texttt{ =}_{Int} 1)$ or not $(n \texttt{ !=}_{Int} 1)$. 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). Note that the Circularity step needs to ensure strict progress of the two programs, which is why the $n \texttt{ !=}_{Int} 1$ condition disappears between steps 9 and 8.

8. Discussion, Related Work and Conclusion

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

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

$$\varphi := n \texttt{ !=}_{Int} 1 \wedge \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto i, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ i\rangle\rangle$$

1.	$\vdash \langle\langle \texttt{skip : n} \mapsto i, \_\rangle, \langle i\rangle\rangle$	$\Downarrow^{\infty}$	E	Axiom
2.	$\vdash \langle\langle \texttt{skip : n} \mapsto i, \_\rangle, \langle i\rangle\rangle$	$\Downarrow^{\infty}$	$E \cup \{\varphi\}$	Axiom
3.	$\vdash \langle\langle \texttt{skip : n} \mapsto i, \texttt{c} \mapsto 1\rangle, \langle i\rangle\rangle$	$\Downarrow^{\infty}$	E	Conseq(1)
4.	$\vdash \langle\langle \texttt{skip : n} \mapsto i, \texttt{c} \mapsto 1\rangle, \langle i\rangle\rangle$	$\Downarrow^{\infty}$	$E \cup \{\varphi\}$	Conseq(2)
5.	$\vdash n =_{Int} 1 \wedge \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto i, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ i\rangle\rangle$	$\Downarrow^{\infty}$	E	Step(3)
6.	$\vdash n =_{Int} 1 \wedge \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto i, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ i\rangle\rangle$	$\Downarrow^{\infty}$	$E \cup \{\varphi\}$	Step(4)
7.	$\vdash n \texttt{ !=}_{Int} 1 \wedge \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto i, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ i\rangle\rangle$	$\Downarrow^{\infty}$	$E \cup \{\varphi\}$	Axiom
8.	$\vdash \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto i, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ i\rangle\rangle)$	$\Downarrow^{\infty}$	$E \cup \{\varphi\}$	Conseq(CA(6, 7))
9.	$\vdash n \texttt{ !=}_{Int} 1 \wedge \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto i, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ i\rangle\rangle$	$\Downarrow^{\infty}$	E	Circularity (8)
10.	$\vdash \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto i, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ i\rangle\rangle$	$\Downarrow^{\infty}$	E	Conseq(CA(5, 9))
11.	$\vdash \langle\langle \texttt{LOOP}_1 \texttt{ : n} \mapsto 1, \texttt{c} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ 1\rangle\rangle$	$\Downarrow^{\infty}$	E	Conseq(10)
12.	$\vdash \langle\langle \texttt{PGM}_1 \texttt{ : n} \mapsto n\rangle, \langle \texttt{PGM}_2\ n\ 1\rangle\rangle$	$\Downarrow^{\infty}$	E	Step (11)

Fig. 10. Formal proof showing that the two Collatz programs are fully equivalent. CA stands for Case Analysis. The main use of Conseq in the proof tree above is to write true as $\texttt{n !=}_{Int} 0 \vee \texttt{n =}_{Int} 0$ 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.

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 [**?**] that program equivalence might be easier than program correctness. Among the recent works on equivalence we mention [**?**, **?**, **?**]. 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 [**?**] but for a different notion of program equivalence. In [**?**] a classification of equivalence relations used in program-equivalence research is given, one of which is full equivalence. 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 [**?**] proves equivalence for LPL programs. On the other hand, [**?**] propose deductive systems for several kinds of equivalences, whereas we focus on full equivalence only. In [**?**], 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 [**?**]), 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 [**?**]. Other results target specific classes of languages: functional [**?**], microcode [**?**], CLP [**?**]. In order to be less language-specific some approaches advocate the use of intermediate languages, such as [**?**], 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 [**?**, **?**], 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., [**?**] for a survey on this topic).

Bisimulation techniques such as [**?**] are also highly language dependent, work on a single language and a more formal general framework is left as future work. Our work has more in common with the approach of logical relations started by the notion of representation independence of Mitchell [**?**] and continued by others [**?**, **?**, **?**]. Like logical relations, our Circularity rule allows to postulate synchronisation points between the two programs. Unlike in logical relations, our "synchronisation points" are given incrementally, during the construction of the proof tree. Also unlike logical relations, our synchronisation points are specified directly, by a matching logic formula, and not by a logical relation between the two states. This is possible because the two languages have been aggregated and the common sorts are identified in the aggregated language and has the advantage of being naturally language-independent. In fact, our work can be seen as

a relational Hoare logic like in [?], but parametric in the operational semantics of the two programming languages. This is reminiscent of bisimulation; however, in our system, the synchronisation points (captured by the formulae φ_1' and φ_2' in CIRCULARITY) are not given at the level of the transition system, but as matching logic formulae. A line of work by Hur and others [?, ?] aims at reconciling the advantages of bisimulations with those of Kripke logical relations. Compared to their approach, our work is inherently language-parametric; we handle all language-dependent constructs, including higher-order state, by domain reasoning. In contrast to [?], we have not yet studied the practicality of our approach in reasoning about equivalence in higher-order languages and we leave this study for further work.

Finally, our own related work [?] gives a proof system for another equivalence relation parameterized by an observation relation and uses other technical mechanisms. The equivalence relation is based on bisimulation, with each process having some part of its state observable. Therefore, the equivalence relation in [?] is better suited for interactive processes. Also, the proof system presented in [?] is based on derivatives and is more operational than the proof system here. The approach we propose in the present article is tailored to proving the equivalence of two programs from two distinct languages, whereas that of [?] is limited to program equivalence within a single language. In [?] 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 full 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 = MAX_INT`, the test `i <= 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 [?], 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 [?]. Domain reasoning brings, as usual, additional complexity in practice. Relative completeness results assume an oracle for deciding domain theorems (introduced by $\models$ in our proof system in Figure 9). Such oracles are usually unrealistic and, in practice, procedures meant to replace them can be arbitrarily complex, such as ones for reasoning about the heap or about higher-order values or state. 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 and study its practicality for reasoning about equivalence in larger languages.