Information and Computation 289 (2022) 104953

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Intensional Kleene and Rice theorems for abstract program semantics

Paolo Baldan [a], Francesco Ranzato [a,*], Linpeng Zhang [b]

[a] Dipartimento di Matematica, University of Padova, Italy
[b] Department of Computer Science, University College London, UK

ARTICLE INFO

Article history:
Received 14 September 2021
Received in revised form 18 August 2022
Accepted 27 August 2022
Available online 1 September 2022

Keywords:
Computability theory
Recursive function
Rice's theorem
Kleene's second recursion theorem
Program analysis
Affine program invariants

ABSTRACT

Classical results in computability theory, notably Rice's theorem, focus on the extensional content of programs, namely, on the partial recursive functions that programs compute. Later work investigated intensional generalisations of such results that take into account the way in which functions are computed, thus affected by the specific programs computing them. In this paper, we single out a novel class of program semantics based on abstract domains of program properties that are able to capture nonextensional aspects of program computations, such as their asymptotic complexity or logical invariants, and allow us to generalise some foundational computability results such as Rice's Theorem and Kleene's Second Recursion Theorem to these semantics. In particular, it turns out that for this class of abstract program semantics, any nontrivial abstract property is undecidable and every decidable over-approximation necessarily includes an infinite set of false positives which covers all the values of the semantic abstract domain.

1. Introduction

Most classical results in computability theory focus on the so-called *extensional* properties of programs, i.e., on the properties of the partial functions they compute. Notably, the renowned Rice's Theorem [33] (see also standard textbooks such as [12,28,36]) states that any nontrivial extensional property of programs is undecidable. Roughly speaking, a property is extensional when it only concerns the function computed by a program, i.e., its input/output behaviour. Despite being very general, Rice's Theorem and similar results in computability theory, due to the requirement of extensionality, leave out several *intensional* properties which are of utmost importance in the practice of programming. Essential intensional properties of programs include their asymptotic complexity of computation, their logical invariants (e.g., relations between variables at program points), or any event that might happen during the execution of the program while not affecting its output.

State-of-the-art A generalisation of well-established results of computability theory to the realm of program complexity has been put forward by Asperti [1]. A first observation is that Blum's complexity classes [3], i.e., sets of recursive functions (rather than sets of programs) with some given (lower or upper) bound on their (space and/or time) complexity, are not adequate for investigating the decidability aspects of program complexity: in fact, viewed as program properties they

* Corresponding author.
E-mail address: ranzato@math.unipd.it (F. Ranzato).

https://doi.org/10.1016/j.ic.2022.104953
0890-5401/

are trivially extensional. Thus, a key idea in [1] is to focus on the so-called *complexity cliques*, namely, sets of programs (i.e., program indices) closed with respect to their extensional input/output behaviour and their asymptotic complexity. Asperti [1] showed how this approach enables intensional versions of Rice's theorem, Rice-Shapiro theorem, and Kleene's second recursion theorem ([12,36] are standard references for these foundational results) for complexity cliques.

More recently, a different approach has been considered by Moyen and Simonsen in [25], where the classical definition of extensionality has been weakened to a notion of *partial extensionality*. Roughly, a given set of programs is partially extensional if it includes the set of all programs computing a given partial recursive function. It is shown in [25] that if a set of programs and its complement are partially extensional, then they cannot be recursive. Interestingly, this result can be further generalised by replacing the extensionality with an equivalence relation on programs satisfying some suitable structural conditions, notably, the existence of a so-called intricated switching family. Moyen and Simonsen [25] show how to derive within their framework intensional versions of Rice's Theorem — generalising Asperti's result [1] — and Rice-Shapiro Theorem.

Several results concerning the undecidability of specific intensional program properties of interest for static program analysis have been put forward. It is worth recalling the undecidability of flow-sensitive alias analysis in languages with conditional statements, loops, dynamic storage and recursive data structures [6,20], the undecidability of points-to analysis for languages restricted to use scalar variables [6], the undecidability of associativity and commutativity analysis for functions in parallel languages [7], and the undecidability of context-sensitive data-dependence analysis [32]. Notably, Müller-Olm and Seidl [26] proved that for affine programs with positive affine guards it is undecidable whether a given nontrivial affine relation holds at a given program point or not. This latter result relies on a reduction to the undecidable Post correspondence problem, inspired by earlier reductions explored in data flow analysis [13,16], and is formulated by leveraging Karr's lattice [17], a well known abstract domain in static program analysis [9,24,34] consisting of affine equalities between program variables, such as $2x - 3y = 1$.

Main contributions The results in this paper yield undecidability guarantees for general classes of intensional program properties as those of interest for program analysis. In brief, for abstract semantics satisfying suitable conditions, we show that all non-trivial properties of the semantics of programs are undecidable. In particular, our framework is instantiated to prove some undecidability results for static program verifiers in a general setting for program analysis and verification by Cousot et al. [11], and to re-discover as a particular case the undecidability of affine invariants for affine programs with positive affine guards proved by Müller-Olm and Seidl [26]. More in detail, along the lines traced by Asperti [1], we investigate whether and how some fundamental extensional results of computability theory can be systematically generalised to intensional aspects of computation, but rather than focusing on specific intensional properties we deal with generic *abstract program semantics*. More in detail, we distill two fundamental properties of abstract program semantics in our approach: the *strong smn property* and the existence of a *universal fair program*, roughly, an interpreter that preserves the abstract semantics. We show that for abstract semantics satisfying the strong smn property and admitting a universal fair program, a generalisation of Kleene's second recursion theorem can be proved. This, in turn, leads to a generalisation of Rice's theorem. As we will discuss in Section 6, the framework is general enough to be applicable to Asperti's complexity cliques of [1]. Besides relying on a general abstract program semantics, inspired by Moyen and Simonsen's approach [25], we also relax the extensionality condition to partial extensionality. This weakening provides stronger impossibility results as it allows us to show that every decidable over-approximation necessarily contains an infinite set of false positives which covers all the values of the underlying semantic abstract domain. On a different route, we establish a precise connection with Moyen and Simonsen's work [25] by showing that for any abstract program semantics satisfying the strong smn property and a structural *branching* condition (roughly, expressing some form of conditional choice), we can prove the existence of an intricated switching family, which turns out to be the crucial hypothesis in [25] for deriving an intensional version of Rice's theorem. This notion of intricated switching family is further explored by identifying a *canonical* one.

Therefore, on the one hand, we generalise the results in [1], going beyond complexity cliques, and, on the other hand, we provide an explicit characterisation of a class of program semantics that admit intricated switching families so that the results in [25] can be applied.

Finally, we show some applications of our intensional Rice's theorem that generalise some undecidability results for intensional properties used in static program analysis. In particular, we focus on program analysis in Karr's abstract domain of affine relations between program variables [17] and on the aforementioned undecidability result for affine programs with positive affine guards by Müller-Olm and Seidl [26]. Here, we first show that the class of affine programs with positive affine guards, modelled as control flow graphs, is Turing complete (something that, to the best of our knowledge, was previously unknown in the literature). Then, this allows us to derive the undecidability result in [26] as a consequence of our results.

Outline The rest of the paper is structured as follows. In Section 2, we provide some background and our basic notions. In Section 3, we introduce the strong smn property, fair universal programs, and the branching condition that will play a fundamental role in our results. In Section 4, we provide our generalisation of Kleene's second recursion theorem and use it to derive our intensional Rice's theorem. We also establish an explicit connection with the notion of intricated switching family given in [25], and discuss some immediate applications for static program verifiers in the setting of Cousot et al. [11]. In Section 5, we prove first that the class of affine control flow graphs with positive affine guards is Turing complete, and then we provide more applications of our results to the analysis of such affine programs. Section 6 discusses in detail the

relation with some of Asperti's results [1] and with Rogers' systems of indices [35,36]. Finally, Section 7 concludes and outlines some directions of future work. This is a full and revised version of the conference paper [2].

2. Basic notions

Given an n-ary partial function $f : \mathbb{N}^n \to \mathbb{N}$, we denote by $\mathrm{dom}(f)$ the domain of f and by $\mathrm{rng}(f) \triangleq \{f(\vec{x}) : \vec{x} \in \mathrm{dom}(f)\}$ its range. We write $f(\vec{x})\downarrow$ if $\vec{x} \in \mathrm{dom}(f)$ and $f(\vec{x})\uparrow$ if $\vec{x} \notin \mathrm{dom}(f)$. Moreover, $\lambda\vec{x}.\uparrow$ denotes the always undefined function. We denote by $\mathcal{F}_n \triangleq \mathbb{N}^n \to \mathbb{N}$ the class of all n-ary (possibly partial) functions and by $\mathcal{F} \triangleq \bigcup_n \mathcal{F}_n$ the class of all such functions. Additionally, $\mathcal{C}_n \subseteq \mathcal{F}_n$ denotes the subset of n-ary partial recursive functions ($\mathcal{C}$ stands for computable) and $\mathcal{C} \triangleq \bigcup_n \mathcal{C}_n$ the set of all partial recursive functions.

Assumption 2.1 *(Turing completeness).* Throughout the paper, we assume a fixed Turing complete model and we denote by $\mathcal{P}$ the corresponding set of programs. Moreover, we consider a fixed Gödel numbering for the programs in $\mathcal{P}$ and, given an index $a \in \mathbb{N}$, we write P_a for the a-th program in $\mathcal{P}$. A program can take a varying number n of inputs and we denote by $\phi_a^{(n)} \in \mathcal{C}_n$ the n-ary partial function computed by P_a. By Turing completeness of the model, $\mathcal{C} = \{\phi_a^{(n)} \mid a, n \in \mathbb{N}\}$ holds. □

The binary relation between programs that compute the same n-ary function is called *Rice's equivalence* and denoted by $\sim_R^n$, i.e.,

$$a \sim_R^n b \overset{\triangle}{\iff} \phi_a^{(n)} = \phi_b^{(n)}.$$

The classical Rice's theorem [33] compares the extension of programs, i.e., the functions they compute, and shows that unions of equivalence classes of programs computing the same function are undecidable. In Asperti's work [1], by relying on the notion of complexity clique, the asymptotic program complexity can be taken into account. Our idea here is to further generalise the approach in [1] by considering generic program semantics rather than program complexity. Additionally, an equivalence relation on program semantics allows us to further abstract and identify programs with different abstract semantics. This turns out to be worthwhile in many applications, e.g., the precise time/space program complexity is typically abstracted by considering asymptotic complexity classes.

Definition 2.2 *(Abstract semantics).* An *abstract semantics* is a pair $\langle \pi, \equiv_\pi \rangle$ where:

(1) $\pi : \mathbb{N}^2 \to \mathcal{F}$ associates a program index a and arity n with an n-ary function $\pi_a^{(n)} \in \mathcal{F}_n$, called the *semantics* of a;
(2) $\equiv_\pi \subseteq \mathcal{F} \times \mathcal{F}$ is an equivalence relation between functions.

Given $n \in \mathbb{N}$, the *n-ary program equivalence* induced by an abstract semantics $\langle \pi, \equiv_\pi \rangle$ is the equivalence $\sim_\pi^n \subseteq \mathbb{N} \times \mathbb{N}$ defined as follows: for all $a, b \in \mathbb{N}$,

$$a \sim_\pi^n b \quad \overset{\triangle}{\iff} \quad \pi_a^{(n)} \equiv_\pi \pi_b^{(n)}. \qquad \square$$

The notation for the case of arity $n = 1$ will be simplified by omitting the arity, e.g., we will write ϕ_a and $\sim_\pi$ in place of $\phi_a^{(1)}$ and $\sim_\pi^1$, respectively. Abstract semantics can be viewed as a generalisation of the notion of system of indices (or numbering), as found in standard reference textbooks [28,36]. This is discussed in detail later in Section 6.2. Let us now show how the standard extensional interpretation of programs, complexity and complexity cliques can be cast into our setting.

Example 2.3 *(Concrete semantics).* The concrete input/output semantics can be trivially seen as an abstract semantics $\langle \phi, = \rangle$ where $\phi_a^{(n)}$ is the n-ary function computed by P_a and $=$ is the equality between functions. Observe that this concrete semantics induces an n-ary program equivalence which is Rice's equivalence $\sim_R^n$. □

Example 2.4 *(Domain semantics).* For a given set of inputs $S \subseteq \mathbb{N}$, consider $\langle \phi, \equiv_S \rangle$ where $\phi_a^{(n)}$ is the n-ary function computed by P_a and for $f, g : \mathbb{N}^n \to \mathbb{N}$, we define $f \equiv_S g \overset{\triangle}{\iff} \mathrm{dom}(f) \cap S = \mathrm{dom}(g) \cap S$. □

Example 2.5 *(Blum complexity).* Let $\Phi : \mathbb{N}^2 \to \mathcal{C}$ be a Blum complexity [3], i.e., for all $a \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^n$, (1) $\Phi_a^{(n)}(\vec{x})\downarrow \Leftrightarrow \phi_a^{(n)}(\vec{x})\downarrow$ holds, and (2) for all $m \in \mathbb{N}$, the predicate $\Phi_a^{(n)}(\vec{x}) = m$ is decidable. Letting $\Theta(f)$ to denote the standard Big Theta complexity class of a function f, the pair $\langle \Phi, \equiv_\Phi \rangle$ defined by

$$\Phi_a^{(n)} \equiv_\Phi \Phi_b^{(n)} \overset{\triangle}{\iff} \Phi_a^{(n)} \in \Theta(\Phi_b^{(n)})$$

is an abstract semantics. □

Example 2.6 *(Complexity clique). Complexity cliques* as defined by Asperti in [1] can be viewed as an abstract semantics $\langle \pi, \equiv_\pi \rangle$, that we will refer to as the complexity clique semantics. For each arity n and program index a let us define:

$$\pi_a^{(n)} \triangleq \lambda \vec{y}.\langle\langle \phi_a^{(n)}(\vec{y}), \Phi_a^{(n)}(\vec{y})\rangle\rangle$$

where $\langle\langle \_, \_ \rangle\rangle : \mathbb{N}^2 \to \mathbb{N}$ is an effective bijective encoding for pairs and $\Phi : \mathbb{N}^2 \to \mathcal{C}$ is a Blum complexity. The equivalence $\equiv_\pi$ is defined as follows: for all $a, b, n \in \mathbb{N}$,

$$\pi_a^{(n)} \equiv_\pi \pi_b^{(n)} \overset{\triangle}{\iff} \phi_a^{(n)} = \phi_b^{(n)} \wedge \Phi_a^{(n)} \equiv_\Phi \Phi_b^{(n)}. \qquad \square$$

The classical Rice's theorem states the undecidability of extensional program properties. Following [25], we parameterise extensional sets by means of a generic equivalence relation.

Definition 2.7 *($\sim$-extensional set).* Let $\sim \subseteq \mathbb{N} \times \mathbb{N}$ be an equivalence relation between programs whose equivalence classes are denoted, for $a \in A$, by $[a]_\sim$. A set of indices $A \subseteq \mathbb{N}$ is called:

- *$\sim$-extensional* when for all $a, b \in \mathbb{N}$, if $a \in A$ and $a \sim b$ then $b \in A$;
- *partially $\sim$-extensional* when there exists $a \in \mathbb{N}$ such that $[a]_\sim \subseteq A$;
- *universally $\sim$-extensional* when for all $a \in \mathbb{N}$, $[a]_\sim \cap A \neq \varnothing$. $\square$

In words, a set A is $\sim$-extensional if A is a union of $\sim$-equivalence classes, partially $\sim$-extensional if A contains at least a whole $\sim$-equivalence class, and universally $\sim$-extensional if A contains at least an element from each $\sim$-equivalence class, i.e., its complement $\mathbb{N} \setminus A$ is not partially $\sim$-extensional. Notice that if A is not trivial (i.e., $A \neq \varnothing$ and $A \neq \mathbb{N}$) and $\sim$-extensional then A is partially $\sim$-extensional and not universally $\sim$-extensional. Let us observe that $\sim_R$-extensionality is the standard notion of extensionality so that the classical Rice's theorem [33] states that if A is $\sim_R$-extensional and not trivial then A is not recursive.[1]

3. Fair and strong smn semantics

In this section, we identify some fundamental properties of abstract semantics that will be later used in our intensional computability results. A first basic property stems from the fundamental smn theorem and intuitively amounts to requiring that the operation of fixing some parameters of a program is effective and preserves its abstract semantics.

Definition 3.1 *(Strong smn semantics).* An abstract semantics $\langle \pi, \equiv_\pi \rangle$ has the *strong smn* (*ssmn*) *property* if, given $m, n \geq 1$, there exists a total computable function $s : \mathbb{N}^{m+2} \to \mathbb{N}$ such that for all $a, b \in \mathbb{N}$, $\vec{x} \in \mathbb{N}^m$:

$$\lambda \vec{y}.\pi_a^{(n+1)}(\phi_b^{(m)}(\vec{x}), \vec{y}) \equiv_\pi \pi_{s(a,b,\vec{x})}^{(n)}. \tag{1}$$

In such a case, the abstract semantics $\langle \pi, \equiv_\pi \rangle$ is called *strong smn*. $\square$

It is worth noticing that the above definition requires property (1) which is slightly stronger than one would expect. In fact, the natural generalisation of the standard smn property, in the style, e.g., of [1], would amount to asking that, given $m, n \geq 1$, there exists a total computable function $s : \mathbb{N}^{m+1} \to \mathbb{N}$ such that for any program index $a \in \mathbb{N}$ and input $\vec{x} \in \mathbb{N}^m$, $\lambda \vec{y}.\pi_a^{(m+n)}(\vec{x}, \vec{y}) \equiv_\pi \pi_{s(a,\vec{x})}^{(n)}$ holds.

The concrete semantics $\langle \phi, = \rangle$ of Example 2.3 clearly satisfies this ssmn property (1). In fact, the function $\lambda a, b, \vec{y}.\pi_a^{(n+1)}(\phi_b^{(m)}(\vec{x}), \vec{y})$ is computable (by composition, relying on the existence of universal functions), hence the existence of a total computable $s : \mathbb{N}^{m+2} \to \mathbb{N}$ such that $\lambda \vec{y}.\pi_a^{(n+1)}(\phi_b^{(m)}(\vec{x}), \vec{y}) \equiv_\pi \pi_{s(a,b,\vec{x})}^{(n)}$ holds, as prescribed by Definition 3.1, follows by the standard smn theorem. It is easily seen that the same applies to the domain semantics of Example 2.4.

The reason for the stronger requirement (1) in Definition 3.1 is that, to deal with generic abstract semantics, a suitable smn definition needs to embody a condition on program composition (of a and b in Definition 3.1). Indeed, if we consider the semantics based on program complexity (i.e., Examples 2.5 and 2.6), it turns out that whenever they enjoy the smn property in [1, Definition 11] and, additionally, they satisfy the linear time composition hypothesis in [1, Section 4] relating the asymptotic complexities of a program composition to those of its components, then they are ssmn semantics according to Definition 3.1. More details on the relationship with Asperti's approach [1] will be given later in Section 6.1.

It is worth observing that for a ssmn abstract semantics $\langle \pi, \equiv_\pi \rangle$, there always exists a program whose denotation is equivalent to the always undefined function, namely,

$$\text{for any arity } n \in \mathbb{N} \text{ there exists an index } e_0 \in \mathbb{N} \text{ such that } \pi_{e_0}^{(n)} \equiv_\pi \lambda \vec{y}.\uparrow. \tag{2}$$

[1] In [25], the term "extensional" is replaced by "compatible" when one refers to generic equivalence relations $\sim$.

In fact, if b is a program index for the always undefined function $\lambda\vec{y}.\uparrow$ then, by (1), we have that $\lambda\vec{y}.\pi_0^{(n+1)}(\phi_b(0),\vec{y}) = \lambda\vec{y}.\uparrow \equiv_\pi \pi^{(n)}_{s(0,b,0)}$ holds, so that we can pick $e_0 \triangleq s(0,b,0)$.

It is also worth exhibiting an example of abstract semantics which is not ssmn. Let $\pi_a(\vec{x})$ be defined as the number of different variables accessed in a computation of the program a on the input $\vec{x}$. Then, let us observe that the mere fact that π_a is always a total function trivially makes the abstract semantics $\langle \pi, = \rangle$ non-ssmn.

To generalise Kleene's second recursion theorem, besides the ssmn property, we need to postulate the existence of so-called *fair universal programs*, namely, programs that can simulate every other program w.r.t. a given abstract semantics. This generalises the analogous notion in [1, Definition 26], where this simulation is specific to complexity cliques and must preserve both the computed function and its asymptotic complexity.

Definition 3.2 *(Fair semantics).* An index $u \in \mathbb{N}$ is a *fair universal program* for an abstract semantics $\langle \pi, \equiv_\pi \rangle$ and an arity $n \in \mathbb{N}$ if for all $a \in \mathbb{N}$:

$$\pi_a^{(n)} \equiv_\pi \lambda\vec{y}.\pi_u^{(n+1)}(a,\vec{y}).$$

An abstract semantics is *fair* if it admits a fair universal program for every arity. □

Clearly, the concrete (cf. Example 2.3) and domain (cf. Example 2.4) semantics are fair. In general, as noted in [1], the existence of a fair universal program may not only depend on the reference abstract semantics, but also on the underlying computational model. For instance, when considering program complexity, as argued by Asperti [1, Section 6] by relying on some remarks by Blum [4], multi-tape Turing machines seem not to admit fair universal programs. By contrast, single tape Turing machines do have fair universal programs, despite the fact that this is commonly considered a folklore fact and cannot be properly quoted. Hereafter, when referring to the complexity-based semantics of Examples 2.5 and 2.6, we will implicitly use the fact that they are ssmn and fair semantics.

4. Kleene's second recursion theorem and Rice's theorem

In this section, we show how some foundational results of computability theory can be extended to a general abstract semantics. The first approach relies on a generalisation of Kleene's second recursion theorem, which is then used to derive a corresponding Rice's theorem. A second approach consists in identifying conditions that ensure the existence of an intricated switching family in the sense of [25], from which Rice's theorem also follows.

4.1. Kleene's second recursion theorem

Kleene's second recursion theorem is a classical result of computability theory, originally proved in [18]. In Rogers' equivalent formulation [36], it states that for any total computable function $h : \mathbb{N} \to \mathbb{N}$ there exists a program index $n \in \mathbb{N}$ which is a fixpoint of h w.r.t. the concrete semantics, i.e., such that $\phi_n = \phi_{h(n)}$ holds. Kleene's second recursion theorem can be used to derive several other classical results, such as the undecidability of the halting problem, Rice's Theorem, or to show the existence of quines, i.e., self-reproducing programs (for more details, we refer to standard textbooks such as [12,28,36]).

We show that Kleene's second recursion theorem holds for any fair ssmn abstract semantics. This generalises the analogous result proved by Asperti [1, Section 5] for complexity cliques.

Theorem 4.1 *(Intensional Second Recursion Theorem). Let $\langle \pi, \equiv_\pi \rangle$ be a fair ssmn abstract semantics. For any total computable function $h : \mathbb{N} \to \mathbb{N}$ and arity $n \in \mathbb{N}$, there exists an index $a \in \mathbb{N}$ such that $a \sim^n_\pi h(a)$.*

Proof. Since $\langle \pi, \equiv_\pi \rangle$ is a fair semantics (Definition 3.2), there exists $u, n \in \mathbb{N}$ such that u is an abstract universal program for n-ary functions. Hence, for all $x \in \mathbb{N}$:

$$\pi^{(n)}_{h(\phi_x(x))} \equiv_\pi \lambda\vec{y}.\pi_u^{(n+1)}(h(\phi_x(x)),\vec{y}) \equiv_\pi \lambda\vec{y}.\pi_u^{(n+1)}(h(\psi_U(x,x)),\vec{y}),$$

where ψ_U is the standard unary universal function for the concrete semantics ϕ, i.e., $\forall p \in \mathbb{N}.\ \lambda y.\psi_U(p,y) = \phi_p$. Note that $h \circ \lambda z.\psi_U(z,z)$ is computable by composition of computable functions. Hence, there exists e such that $\phi_e = h \circ \lambda z.\psi_U(z,z)$. Since $\langle \pi, \equiv_\pi \rangle$ is a ssmn semantics (Definition 3.1), there exists a total computable function $s : \mathbb{N}^3 \to \mathbb{N}$ such that for all $x \in \mathbb{N}$:

$$\lambda\vec{y}.\pi_u^{(n+1)}(h(\psi_U(x,x)),\vec{y}) \equiv_\pi \lambda\vec{y}.\pi_u^{(n+1)}(\phi_e(x),\vec{y}) \equiv_\pi \pi^{(n)}_{s(u,e,x)}.$$

Since s is computable, by standard smn theorem, there exists $m \in \mathbb{N}$ such that $\phi_m = \lambda x.\ s(u,e,x)$. Hence, for all $x \in \mathbb{N}$:

$$\pi^{(n)}_{\phi_m(x)} \equiv_\pi \pi^{(n)}_{h(\phi_x(x))}.$$

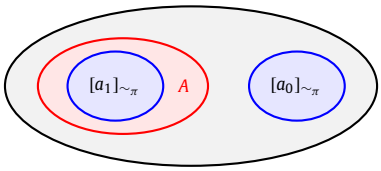

Fig. 1. A graphical representation of Theorem 4.3. Here, a_1 is a program index whose $\equiv_\pi$-equivalence class $[a_1]_{\sim_\pi}$ is overapproximated by a set A of programs, i.e., A includes all the programs that are $\equiv_\pi$-equivalent to a_1. For the program index a_0, its $\equiv_\pi$-equivalence class $[a_0]_{\sim_\pi}$ is disjoint with A, i.e., all the programs in A are not $\equiv_\pi$-equivalent to a_0. Whenever such conditions are met for a fair ssmn semantics π, it turns out that the set A is not recursive.

If we set $x = m$ we obtain:

$$\pi^{(n)}_{\phi_m(m)} \equiv_\pi \pi^{(n)}_{h(\phi_m(m))}.$$

Because $\phi_m = \lambda x.\ s(u, e, x)$ is total, we can consider $a = \phi_m(m)$ and obtain:

$$\pi^{(n)}_{a} \equiv_\pi \pi^{(n)}_{h(a)}$$

which amounts to $a \sim^n_\pi h(a)$. □

As an example, this result, instantiated to the complexity semantics of Example 2.5, entails the impossibility of designing a program transformation that systematically modifies the asymptotic complexity of every program, even without preserving its input-output behaviour. The details are discussed below.

Example 4.2 *(Fixpoints of Blum complexity semantics).* Let $\langle \Phi, \equiv_\Phi \rangle$ be the Blum complexity semantics of Example 2.5. A program transformation $h : \mathbb{N} \to \mathbb{N}$ is a total computable function which maps indices of programs into indices of transformed programs. By applying Theorem 4.1, for any arity $n \in \mathbb{N}$, we know that there exists a program index a such that $a \sim^n_\pi h(a)$ holds. This means that the program transform h does not alter the asymptotic complexity of, at least, the program a. □

Our second recursion theorem allows us to obtain an intensional version of Rice's theorem for fair and ssmn abstract semantics. Inspired by [25], we generalise the statement to cover partially extensional properties.

Theorem 4.3 *(Rice by fair and ssmn semantics). Let $\langle \pi, \equiv_\pi \rangle$ be a fair and ssmn semantics. If $A \subseteq \mathbb{N}$ is partially $\sim^n_\pi$-extensional and not universally $\sim^n_\pi$-extensional, for some arity $n \in \mathbb{N}$, then A is not recursive.*

Proof. Since A is partially $\sim^n_\pi$-extensional and not universally $\sim^n_\pi$-extensional, there are $x_0, x_1 \in \mathbb{N}$ such that $[x_0]_{\sim^n_\pi} \cap A = \varnothing$ and $[x_1]_{\sim^n_\pi} \subseteq A$. Assume A is recursive, hence its characteristic function χ_A is computable. Then, we can define a function $f : \mathbb{N} \to \mathbb{N}$ defined as follows:

$$f(x) \triangleq \begin{cases} x_0 & \text{if } x \in A \\ x_1 & \text{otherwise} \end{cases} = x_0 \cdot \chi_A(x) + x_1 \cdot (1 - \chi_A(x)).$$

Observe that f is clearly total and computable. We can now apply our intensional second recursion Theorem 4.1, and obtain that there exists $a \in \mathbb{N}$ such that $f(a) \sim_\pi a$. This easily leads to a contradiction that closes the proof. In fact, there are two cases, either $a \in A$ or $a \notin A$.

1. If $a \in A$ then $f(a) = x_0 \sim_\pi a$ and thus, since $[x_0]_{\sim_\pi} \cap A = \varnothing$, we have the contradiction $a \notin A$.
2. Similarly, if $a \notin A$ then $f(a) = x_1 \sim_\pi a$ and thus, since $[x_1]_{\sim} \subseteq A$, we deduce the contradiction $a \in A$. □

Fig. 1 provides a graphical representation of this result: if we can find two program indices $a_0, a_1 \in \mathbb{N}$ such that A overapproximates the $\equiv_\pi$-equivalence class $[a_1]_{\sim_\pi}$ and A does not intersect $[a_0]_{\sim_\pi}$, then A cannot be recursive. Let us illustrate some applications of Theorem 4.3.

Example 4.4 *(Halting set).* Let $\langle \phi, \equiv_{\mathbb{N}} \rangle$ be the domain semantics of Example 2.4 with $S = \mathbb{N}$, hence $f \equiv_{\mathbb{N}} g$ when $\mathrm{dom}(f) = \mathrm{dom}(g)$. The halting set $K \triangleq \{a \in \mathbb{N} \mid \phi_a(a)\downarrow\}$ can be proved to be non-recursive by resorting to Theorem 4.3 for $\langle \phi, \equiv_{\mathbb{N}} \rangle$. Let $e_0, e_1 \in \mathbb{N}$ be such that $\phi_{e_0} = \lambda x.\uparrow$ and $\phi_{e_1} = \lambda x.1$. Since $[e_1]_{\equiv_{\mathbb{N}}}$ is the set of programs that compute total functions, we have that $[e_1]_{\equiv_{\mathbb{N}}} \subseteq K$. Moreover, $[e_0]_{\equiv_{\mathbb{N}}}$ is the set of nonterminating programs for any input, so that $[e_0]_{\equiv_{\mathbb{N}}} \cap K = \varnothing$. This means that $\langle \phi, \equiv_{\mathbb{N}} \rangle$ satisfies the hypotheses of Theorem 4.3, thus entailing that K is not recursive. □

Example 4.5 *(Complexity sets).* Let $\langle \phi, = \rangle$, $\langle \Phi, \equiv_\Phi \rangle$ be, resp., the semantics of Examples 2.3 and 2.5. As observed in Section 3, on a suitable computational model such as single tape Turing machines, these are fair ssmn semantics, so that Theorem 4.3 applies.

Let $sort : \mathbb{N} \to \mathbb{N}$ be a total function that takes as input an encoded sequence of numbers and outputs the encoding of the corresponding sorted sequence. It turns out that by applying Theorem 4.3, the following sets can be proved to be non-recursive:

(1) $A \triangleq \{a \mid \Phi_a \in \Theta(n \log n) \wedge \phi_a = sort\}$;
(2) $B \triangleq \{a \mid \Phi_a \in \mathcal{O}(n \log n)\}$;
(3) $C \triangleq \{a \mid \Phi_a \in \Omega(n \log n)\}$.

Let *is*, *ms* be different implementations of *sort*, i.e., $\phi_{is} = \phi_{ms} = sort$, such that $\Phi_{is} \in \Theta(n^2)$ and $\Phi_{ms} \in \Theta(n \log n)$ – *is* and *ms* could be, resp., insertion and merge sort. Recall that $\sim_R$ denotes Rice's equivalence induced by $\langle \phi, = \rangle$ (i.e., $a \sim_R b \Leftrightarrow \phi_a = \phi_b$), and, in turn, let $\sim_{\Phi R} = \sim_\Phi \cap \sim_R$ be the equivalence induced by the complexity clique semantics of Example 2.6, which is a fair ssmn semantics. Then, we have that:

(1) since $[is]_{\sim_{\Phi R}} \cap A = \varnothing$ and $[ms]_{\sim_{\Phi R}} \subseteq A$, by Theorem 4.3, we have that A is non-recursive;
(2) since $[is]_{\sim_\Phi} \cap B = \varnothing$ and $[ms]_{\sim_\Phi} \subseteq B$, by Theorem 4.3, we have that B is non-recursive;
(3) let e be any program index such that $\Phi_e \in \Theta(1)$. Since $[e]_{\sim_\Phi} \cap C = \varnothing$ and $[is]_{\sim_\Phi} \subseteq C$, by Theorem 4.3, we have that C is non-recursive. □

It is worth remarking that in Example 4.5, $n \log n$ could be replaced by any function, thus showing the undecidability of the asymptotic complexities "big O" (case (2)) and "big Omega" (case (3)). Let us also point out that Example 4.4 shows how easily the halting set K can be proved to be non-recursive by applying Theorem 4.3.

4.2. Branching semantics

Let us investigate the connection between our results and the key notion of intricated switching family used by Moyen and Simonsen [25] for proving their intensional version of Rice's theorem. Firstly, we argue that every ssmn abstract semantics admits an intricated switching family whenever it is able to express a suitable form of *conditional branching*. This allows us to derive an intensional Rice's theorem. Moreover, we show that for fair and ssmn semantics, the identity can always play the role of intricated switching family.

Definition 4.6 *(Branching and discharging semantics).* An abstract semantics $\langle \pi, \equiv_\pi \rangle$ is *branching* if, given $n \geq 1$, there exists a total computable function $r : \mathbb{N}^4 \to \mathbb{N}$ such that $\forall a, b, c_1, c_2, x \in \mathbb{N}$ with $c_1 \neq c_2$:

$$\lambda \vec{y}.\pi^{(n)}_{r(a,b,c_1,c_2)}(x, \vec{y}) \equiv_\pi \begin{cases} \lambda \vec{y}.\pi^{(n)}_a(x, \vec{y}) & \text{if } x = c_1 \\ \lambda \vec{y}.\pi^{(n)}_b(x, \vec{y}) & \text{if } x = c_2 \\ \lambda \vec{y}.\uparrow & \text{otherwise.} \end{cases}$$

Moreover, $\langle \pi, \equiv_\pi \rangle$ is (variable) *discharging* if, for all $n \geq 1$, there exists a total computable function $t : \mathbb{N} \to \mathbb{N}$ such that for all $a, x \in \mathbb{N}$:

$$\pi^{(n)}_a \equiv_\pi \lambda \vec{y}.\pi^{(n+1)}_{t(a)}(x, \vec{y}).$$ □

Hence, intuitively, an abstract semantics is branching when it is able to model the branching structure of conditional statements with multiple positive guards, while the property of being variable discharging holds when one can freely add fresh and unused variables without altering the abstract semantics. Let us first recall the notion of recursive inseparability [37, Section 3] and of intricated switching family from [25, Definition 5].[2]

Definition 4.7 *(Recursively inseparable sets).* Two sets $A, B \subseteq \mathbb{N}$ of program indices are *recursively inseparable* if there exists no decidable set $C \subseteq \mathbb{N}$ such that $A \subseteq C$ and $B \cap C = \varnothing$. □

Definition 4.8 *(Intricated switching family [25, Definition 5]).* Let $\sim \subseteq \mathbb{N} \times \mathbb{N}$ be an equivalence relation on program indices. An *intricated switching family* (ISF) w.r.t. $\sim$ is an indexed set of total computable functions $\{\sigma_{a,b}\}_{a,b \in \mathbb{N}}$, with $\sigma_{a,b} : \mathbb{N} \to \mathbb{N}$, such that for all $a, b \in \mathbb{N}$, the sets $A_{a,b} = \{x \in \mathbb{N} \mid \sigma_{a,b}(x) \sim a\}$ and $B_{a,b} = \{x \in \mathbb{N} \mid \sigma_{a,b}(x) \sim b\}$ are recursively inseparable. □

[2] Definition 5 in [25] is instantiated to the case of recursive sets and equivalence relations over program indices. This is the case of interest for this paper and such restriction simplifies the presentation. More precisely, Definition 4.8 is obtained from [25, Definition 5] by taking: (1) the sets $S = T = \mathbb{N}$ (intuitively corresponding to sets of program indices); (2) the sets $\mathcal{S} = \mathcal{T} = REC$, where $REC \subseteq \mathcal{P}(\mathbb{N})$ is the set of decidable sets; (3) F and G such that $\cup F = \cup G = \mathbb{N}$. The requirement of *REC*-*REC*-continuity of each total function $\sigma_{a,b}$ is replaced with the stronger condition that each $\sigma_{a,b}$ is also computable.

Moyen and Simonsen [25, Theorem 3] show that if an equivalence $\sim$ admits an ISF, then every partially $\sim$-extensional and not universally $\sim$-extensional set is not recursive. A simplified version of their intensional result, tailored for our setting, can be stated as follows.

Theorem 4.9 *([25, Theorem 3]). Let $\sim \subseteq \mathbb{N} \times \mathbb{N}$ be an equivalence relation. If $A \subseteq \mathbb{N}$ is partially $\sim$-extensional, not universally $\sim$-extensional and there exists an ISF w.r.t. $\sim$ then A is not recursive.*

Branching discharging and ssmn semantics can be shown to admit an intricated switching family, in a way that, relying on Theorem 4.9 we can derive the following intensional version of Rice's Theorem.

Theorem 4.10 *(Rice by branching, discharging and ssmn semantics). Let $\langle \pi, \equiv_\pi \rangle$ be a branching, discharging and ssmn semantics. If $A \subseteq \mathbb{N}$ is partially $\sim^n_\pi$-extensional and not universally $\sim^n_\pi$-extensional for some arity $n \in \mathbb{N}$, then A is not recursive.*

Proof. Let $u \in \mathbb{N}$ be an index for the standard unary universal program. Consider the total computable functions $r : \mathbb{N}^4 \to \mathbb{N}$ and $t : \mathbb{N} \to \mathbb{N}$ of, resp., the branching and variable discharging properties. By the ssmn property, there exists a total computable function $s : \mathbb{N}^4 \to \mathbb{N}$ such that $\forall a, b, x \in \mathbb{N}$:

$$
\begin{aligned}
\pi^{(n)}_{s(r(t(a),t(b),0,1),u,x,0)} &\equiv_\pi \lambda\vec{y}.\pi^{(n+1)}_{r(t(a),t(b),0,1)}(\phi^{(2)}_u(x,0),\vec{y}) && \text{[by the ssmn property]}\\
&= \lambda\vec{y}.\pi^{(n+1)}_{r(t(a),t(b),0,1)}(\phi_x(0),\vec{y})\\
&\equiv_\pi \begin{cases} \lambda\vec{y}.\pi^{(n+1)}_{t(a)}(0,\vec{y}) & \text{if } \phi_x(0)=0 \\ \lambda\vec{y}.\pi^{(n+1)}_{t(b)}(1,\vec{y}) & \text{if } \phi_x(0)=1 \\ \lambda\vec{y}.\uparrow & \text{otherwise} \end{cases} && \text{[by the branching property]}\\
&\equiv_\pi \begin{cases} \pi^{(n)}_a & \text{if } \phi_x(0)=0 \\ \pi^{(n)}_b & \text{if } \phi_x(0)=1 \\ \lambda\vec{y}.\uparrow & \text{otherwise} \end{cases} && \text{[by the variable discharging property]}
\end{aligned}
$$

For all $a, b \in \mathbb{N}$, we define the total computable function

$$\sigma_{a,b}(x) \triangleq s(r(t(a),t(b),0,1),u,x,0).$$

We claim that the family of functions $\{\sigma_{a,b}\}_{a,b\in\mathbb{N}}$ is intricated w.r.t. $\sim^n_\pi$ (cf. Definition 4.8). In fact, for all $a, b \in \mathbb{N}$, let $A_{a,b} \triangleq \{x \in \mathbb{N} \mid \sigma_{a,b}(x) \sim^n_\pi a\}$ and $B_{a,b} \triangleq \{x \in \mathbb{N} \mid \sigma_{a,b}(x) \sim^n_\pi b\}$. We have four cases:

1. if $\pi^{(n)}_a \equiv_\pi \pi^{(n)}_b$, then $A_{a,b} = B_{a,b}$ and therefore they are trivially recursively inseparable;
2. if $\pi^{(n)}_a \not\equiv_\pi \pi^{(n)}_b$ and $\pi^{(n)}_a \not\equiv_\pi \lambda\vec{x}.\uparrow \not\equiv_\pi \pi^{(n)}_b$, we have that $A_{a,b} = \{x \in \mathbb{N} \mid \phi_x(0) = 0\}$ and $B_{a,b} = \{x \in \mathbb{N} \mid \phi_x(0) = 1\}$. Hence, the sets $A_{a,b}$ and $B_{a,b}$ are recursively inseparable (cf. [29, Section 3.3]);
3. if $\pi^{(n)}_b \not\equiv_\pi \pi^{(n)}_a \equiv_\pi \lambda\vec{x}.\uparrow$, we have that $B_{a,b} = \{x \in \mathbb{N} \mid \phi_x(0) = 1\}$ and $A_{a,b} = \{x \in \mathbb{N} \mid \phi_x(0) \neq 1\} = \overline{B_{a,b}}$. The mere fact that $B_{a,b}$ is not recursive (by the classical Rice's Theorem) thus implies that $A_{a,b}$ and $B_{a,b}$ are not recursively separated;
4. if $\pi^{(n)}_a \not\equiv_\pi \pi^{(n)}_b \equiv_\pi \lambda\vec{x}.\uparrow$, then we can take $a' = b$ and $b' = a$ and conclude by case 3.

Since in all cases $A_{a,b}$ and $B_{a,b}$ are recursively inseparable, it turns out that $\{\sigma_{a,b}\}_{a,b\in\mathbb{N}}$ is an ISF w.r.t. $\sim^n_\pi$ and thus we conclude by Theorem 4.9. □

Let us discuss more in detail the relationship with the approach in [25]. Firstly, let us show a lemma which will be fundamental to prove the following results.

Lemma 4.11. *Let $\sim$ be an equivalence relation on program indices. If every set A partially $\sim$-extensional and not universally $\sim$-extensional is non-recursive then the identity* Id *is an ISF w.r.t. $\sim$.*

Proof. Clearly, the identity $\mathrm{Id} \triangleq \{(\lambda x.x)_{a,b}\}_{a,b\in\mathbb{N}}$ is a family of total computable functions. Moreover, for $a, b \in \mathbb{N}$ we have $A_{a,b} = \{x \in \mathbb{N} : x \sim a\} = [a]_\sim$ and $B_{a,b} = \{x \in \mathbb{N} : x \sim b\} = [b]_\sim$. Therefore, every set $C \subseteq \mathbb{N}$ such that $A_{a,b} \subseteq C$ and $B_{a,b} \cap C = \varnothing$, is partially $\sim$-extensional and not universally $\sim$-extensional and thus, by hypothesis, not recursive. Hence, $A_{a,b}$ and $B_{a,b}$ are recursively inseparable. □

It turns out that a fair ssmn semantics always admits a canonical ISF, namely, the identity $\mathrm{Id} \triangleq \{(\lambda x.x)_{a,b}\}_{a,b\in\mathbb{N}}$.

Proposition 4.12. *Let $\langle \pi, \equiv_\pi \rangle$ be a fair and ssmn semantics. Then, the identity* Id *is an ISF w.r.t. $\sim_\pi^n$, for all $n \geq 1$.*

Proof. Since $\langle \pi, \equiv_\pi \rangle$ is a fair ssmn semantics, by Theorem 4.3, every partially $\sim_\pi^n$-extensional and not universally $\sim_\pi^n$-extensional set A is non-recursive. Therefore, we conclude by applying Lemma 4.11. □

Let us point out that the identity function has not been exploited in [25], that instead focuses on the standard switching family. It turns out that the identity function plays a key role as ISF.

Theorem 4.13. *Let $\sim \subseteq \mathbb{N} \times \mathbb{N}$ be an equivalence relation. The following statements are equivalent:*

(1) *Every set $A \subseteq \mathbb{N}$ partially $\sim$-extensional and not universally $\sim$-extensional is non-recursive.*
(2) *The identity* Id *is an ISF w.r.t. $\sim$.*
(3) *There exists an ISF w.r.t. $\sim$.*

Proof. $(1 \Rightarrow 2)$: by Lemma 4.11;
$(2 \Rightarrow 3)$: trivial;
$(3 \Rightarrow 1)$: by Theorem 4.9. □

Therefore, the above result roughly states that the identity function is the "canonical" ISF, meaning that if an ISF exists, then Id is an ISF as well. Moreover, the intensional Rice's Theorem 4.9 of [25] provides a sufficient condition (i.e., the existence of an ISF) for a partially and not universally extensional set to be undecidable. Theorem 4.13 enhances Theorem 4.9 by showing that such a sufficient condition is necessary as well, or, equivalently, that a partially and not universally extensional set is undecidable iff there exists an ISF.

We conclude this section by discussing an alternative notion of branching, which requires the preservation of a full conditional statement with positive and negative guards.

Definition 4.14 *(Strongly branching semantics).* An abstract semantics $\langle \pi, \equiv_\pi \rangle$ is *strongly branching* if, given $n \geq 1$, there exists a total computable function $r : \mathbb{N}^3 \to \mathbb{N}$ such that for all $a, b, c, x \in \mathbb{N}$:

$$\lambda \vec{y}.\pi^{(n)}_{r(a,b,c)}(x, \vec{y}) \equiv_\pi \begin{cases} \lambda \vec{y}.\pi^{(n)}_a(x, \vec{y}) & \text{if } x = c \\ \lambda \vec{y}.\pi^{(n)}_b(x, \vec{y}) & \text{otherwise.} \end{cases}$$

□

The condition above is an adaptation to our framework of a property that is needed in order to exploit a so-called standard switching family as defined in [25, Example 1]. Despite appearing to be more natural, the preservation of conditionals with positive and negative conditions is a stronger requirement than the one we considered in Definition 4.6. Indeed, it turns out that every ssmn and strongly branching semantics is a branching semantics.

Proposition 4.15 *(Strongly branching implies branching). If $\langle \pi, \equiv_\pi \rangle$ is a ssmn and strongly branching semantics, then $\langle \pi, \equiv_\pi \rangle$ is a branching semantics.*

Proof. Given an arity n, let r be the function of the strongly branching property of Definition 4.14. By (2) there exists an index $e_0 \in \mathbb{N}$ such that $\pi^{(n)}_{e_0} \equiv_\pi \lambda \vec{y}.\uparrow$. Now, we define the function $\sigma : \mathbb{N}^4 \to \mathbb{N}$ such that for all $a, b, c_1, c_2 \in \mathbb{N}$ we have $\sigma(a, b, c_1, c_2) = r(a, r(b, e_0, c_2), c_1)$. Note that σ is a total computable function, by composition, and for all $a, b, c_1, c_2, x \in \mathbb{N}$ with $c_1 \neq c_2$:

$$\begin{aligned}
\lambda \vec{y}.\pi^{(n)}_{\sigma(a,b,c_1,c_2)}(x, \vec{y}) &= \lambda \vec{y}.\pi^{(n)}_{r(a,r(b,e_0,c_2),c_1)}(x, \vec{y}) \\
&\equiv_\pi \begin{cases} \lambda \vec{y}.\pi^{(n)}_a(x, \vec{y}) & \text{if } x = c_1 \\ \lambda \vec{y}.\pi^{(n)}_{r(b,e_0,c_2)}(x, \vec{y}) & \text{otherwise} \end{cases} && \text{[by the branching property]} \\
&\equiv_\pi \begin{cases} \lambda \vec{y}.\pi^{(n)}_a(x, \vec{y}) & \text{if } x = c_1 \\ \lambda \vec{y}.\pi^{(n)}_b(x, \vec{y}) & \text{if } x \neq c_1 \wedge x = c_2 \\ \lambda \vec{y}.\pi^{(n)}_{e_0}(x, \vec{y}) & \text{if } x \neq c_1 \wedge x \neq c_2 \end{cases} && \text{[by the branching property]} \\
&\equiv_\pi \begin{cases} \lambda \vec{y}.\pi^{(n)}_a(x, \vec{y}) & \text{if } x = c_1 \\ \lambda \vec{y}.\pi^{(n)}_b(x, \vec{y}) & \text{if } x = c_2 \\ \lambda \vec{y}.\uparrow & \text{otherwise} \end{cases}
\end{aligned}$$

Thus, σ is the desired function for the branching property. □

4.3. An application to static program verifiers

We adapt the general definition of static program verifier of Cousot et al. [11, Definition 4.3] to our framework. Given a program property $P \subseteq \mathbb{N}$ to check, a static program verifier is a total recursive function $\mathcal{V} : \mathbb{N} \to \{0, 1\}$. It is *sound* when for all $p \in \mathbb{N}$, $\mathcal{V}(p) = 1 \Rightarrow p \in P$, while $\mathcal{V}$ is *precise* if the reverse implication also holds, i.e., when $\mathcal{V}(p) = 1 \Leftrightarrow p \in P$ holds. Informally, soundness guarantees that only false negatives are allowed, i.e., $\mathbb{N} \setminus P$ is possibly a proper subset of $\{p \in \mathbb{N} : \mathcal{V}(p) = 0\}$, while precise verifiers output true positives and true negatives only (i.e., they decide P).

The classical Rice's theorem clearly entails the impossibility of designing a precise verifier for a nontrivial extensional property. However, one may wonder whether there exist sound verifiers with "few" false negatives. By applying our intensional Theorem 4.3, we are able to show that sound but imprecise verifiers necessarily have at least one false negative for each equivalence class of programs, even for intensional properties.

Example 4.16 *(Constant value verifier).* Assume we are interested in checking if a program can output a given constant value, for instance, zero with the aim of statically detecting division-by-zero bugs. Let $\mathcal{V}$ be a sound static verifier for the set $P_{=0} \triangleq \{p \in \mathbb{N} \mid 0 \in \mathrm{rng}(\phi_p)\}$ of programs that output zero for some input. The set $N \triangleq \{p \in \mathbb{N} \mid \mathcal{V}(p) = 0\}$ is recursive since $\mathcal{V}$ is assumed to be a total computable function. By soundness of $\mathcal{V}$, we have that $\mathbb{N} \setminus P_{=0} \subseteq N$, so that N includes, for example, the set of programs computing the constant function $\lambda x.1$. Therefore, N is partially extensional, and, by Theorem 4.3, N has to be universally extensional. This means that for any computable function $f \in \mathcal{C}$ there exists a program $p \in \mathbb{N}$ that computes f such that $\mathcal{V}(p) = 0$. Thus, when $0 \in \mathrm{rng}(f)$ holds (e.g., for $f = \lambda x.0$), $\mathcal{V}$ necessarily outputs a false negative for p. Hence, $\mathcal{V}$ outputs infinitely many false negatives. □

Example 4.17 *(Complexity verifier).* Consider a speculative sound static verifier $\mathcal{V}$ for recognizing programs that meet some lower bound, for instance, programs having a cubic lower bound $P_{\Omega(n^3)} \triangleq \{p \in \mathbb{N} \mid \Phi_p = \Omega(n^3)\}$. Thus, $N \triangleq \{p \in \mathbb{N} \mid \mathcal{V}(p) = 0\}$ has to be recursive and if $\sim_\Phi$ is the program equivalence induced by the Blum complexity semantics $\langle \Phi, \equiv_\Phi \rangle$ of Example 2.5 then, by soundness of $\mathcal{V}$, we have, for example, $\{p \in \mathbb{N} \mid \Phi_p = \Theta(1)\} \subseteq N$. This means that N is partially $\sim_\Phi$-extensional and, by Theorem 4.3, N is universally extensional, namely, $\mathcal{V}$ will output 0 for at least a program in each Blum complexity class. For instance, even some programs with an exponential lower bound will be wrongly classified by $\mathcal{V}$ as programs that do not meet a cubic lower bound. □

As shown by Cousot et al. [11, Theorem 5.4], precise static verifiers cannot be designed (unless for trivial program properties). The examples above prove that, additionally, we cannot have any certain information on an input program p whenever the output of a sound (and imprecise) verifier for p is 0. In fact, when this happens, p could compute any partial function (cf. Example 4.16) or have any complexity (cf. Example 4.17).

5. On the decidability of affine program invariants

Karr [17] put forward an algorithm that infers for each program point q of a control flow graph modelling an affine program P (i.e., an unguarded program with non-deterministic branching and affine assignments) a set of affine equalities that hold among the variables of P when the control reaches q, namely, an *affine invariant* for P. Müller-Olm and Seidl [26] show that Karr's algorithm actually computes the strongest affine invariant for affine programs (this result has been extended to a slightly larger class of affine programs in [30, Theorem 5.1]). Moreover, they design a more efficient algorithm implementing this static analysis and they extend in [27] this algorithm for computing bounded polynomial invariants, i.e., the strongest polynomial equalities of degree at most a given $d \in \mathbb{N}$. Later, Hrushovski et al. [15] put forward a sophisticated algorithm for computing the strongest unbounded polynomial invariants of affine programs, by relying on the Zariski closure of semigroups.

On the impossibility side, Müller-Olm and Seidl [26, Section 7] prove that for affine programs allowing positive affine guards it is undecidable whether a given nontrivial affine equality holds at a given program point or not. In practical applications, static analyses on Karr's abstract domain of guarded affine programs ignore non-affine Boolean guards, while for an affine guard b, the current affine invariant i is propagated through the positive branch of b by the intersection $i \cap b$, that remains an affine subspace. By the aforementioned undecidability result [26, Section 7], this latter analysis algorithm for guarded affine programs turns out to be sound but necessarily imprecise, thus inferring affine invariants that, in general, might not be the strongest ones. Müller-Olm and Seidl [26, Section 7] prove their undecidability result by exploiting an acute reduction to the undecidable Post correspondence problem, inspired by early reductions studied in data flow analysis [13, 16]. In this section, we show that our Theorem 4.10 allows us to derive and extend this undecidability result by exploiting an orthogonal intensional approach. More precisely, we prove that any nontrivial (and not necessarily affine) relation on the states of control flow graphs of programs allowing: (1) zero, variable and successor assignments, resp., $x := 0$, $x := y$ and $x := y + 1$, and (2) positive equality guards $x = y?$ and $x = v?$, turns out to be undecidable. Since these control flow graphs form a subclass of affine programs with positive affine guards, the undecidability result of Müller-Olm and Seidl [26, Section 7] is retrieved as a consequence.

Following the standard approach, we consider control flow graphs that consist of program points connected by edges labelled by assignments and guards. Variables are denoted by x_i, with $i \in \mathbb{N}$, and store values ranging in $\mathbb{N}$, while Karr's

abstract domain is designed for variables assuming values in $\mathbb{Q}$. Clearly, from a computability perspective, this is not a restriction since one can consider a computable bijection between $\mathbb{N}$ and $\mathbb{Q}$.

Definition 5.1 *(Basic affine control flow graph).* A *basic affine control flow graph* (BACFG) is a tuple $G = (N, E, s, e)$, where N is a finite set of nodes, $s, e \in N$ are the start and end nodes, and $E \subseteq N \times \mathrm{Com} \times N$ is a set of labelled edges, and the set Com of commands consists of assignments of type $x_n := 0$, $x_n := x_m$, $x_n := x_m + 1$, and equality guards of type $x_n = x_m?$, $x_n = v?$, with $v \in \mathbb{N}$. □

Let us remark that BACFGs only include basic affine assignments and positive affine guards, in particular inequality checks such as $x_n \neq x_m?$ and $x_n \neq v?$ are not allowed. Thus, BACFGs are a subclass of affine programs with positive affine guards considered in [26, Section 7].

As in dataflow analysis and abstract interpretation [9,10,13,34], BACFGs have a *collecting semantics* where, given a set of input states *In*, each program point is associated with the set of states that occur in some program execution from some state in *In*. A finite number of variables may occur in a BACFG, so that a state of a BACFG G is a tuple $(x_1, \ldots, x_k) \in \mathbb{N}^k$, where k is the maximum variable index occurring in G and $k = 0$ is a degenerate case for trivial BACFGs with $\mathbb{N}^0 = \{\bullet\}$. The *collecting transfer function* $f_{(\cdot)}(\cdot) : \mathrm{Com} \to \wp(\mathbb{N}^k) \to \wp(\mathbb{N}^k)$ for $k \in \mathbb{N}$ variables and with $n, m \in [1, k]$ is defined as follows:

$$
\begin{aligned}
f_{x_n:=0}(S) &\triangleq \{(x_1, \ldots, x_{n-1}, 0, x_{n+1}, \ldots, x_k) \mid \vec{x} \in S\},\\
f_{x_n:=x_m}(S) &\triangleq \{(x_1, \ldots, x_{n-1}, x_m, x_{n+1}, \ldots, x_k) \mid \vec{x} \in S\},\\
f_{x_n:=x_m+1}(S) &\triangleq \{(x_1, \ldots, x_{n-1}, x_m + 1, x_{n+1}, \ldots, x_k) \mid \vec{x} \in S\},\\
f_{x_n=v?}(S) &\triangleq \{\vec{x} \in S \mid x_n = v\},\\
f_{x_n=x_m?}(S) &\triangleq \{\vec{x} \in S \mid x_n = x_m\}.
\end{aligned}
$$

A no-op command denoted by ϵ is syntactic sugar for $x_1 := x_1$, i.e., $f_\epsilon \triangleq f_{x_1:=x_1} = \lambda S.S$. Given $k, k' \in \mathbb{N}$ and $S \in \wp(\mathbb{N}^{k'})$, the projection $S \restriction_k \in \wp(\mathbb{N}^k)$ is defined as follows:

$$
S \restriction_k \triangleq \begin{cases} S \times \mathbb{N}^{k-k'} & \text{if } 0 \leq k' < k \\ S & \text{if } k' = k \\ \{(x_1, \ldots, x_k) \mid \vec{x} \in S\} & \text{if } k < k' \end{cases}
$$

Definition 5.2 *(Collecting semantics of BACFGs).* Given a BACFG $G = (N, E, s, e)$ with $k \in \mathbb{N}$ variables and a set of input states $S \subseteq \mathbb{N}^{k'}$, with $k' \leq k$, the *collecting semantics* $[\![G]\!]_S : N \to \wp(\mathbb{N}^k)$ is the least, w.r.t. pointwise set inclusion, solution in $\wp(\mathbb{N}^k)^{|N|}$ of the following system of constraints:

$$
\begin{cases} [\![G]\!]_S[s] \supseteq S \restriction_k & \text{for the start node } s \\ [\![G]\!]_S[v] \supseteq f_c([\![G]\!]_S[u]) & \text{for each edge } (u, c, v) \in E. \end{cases}
$$

□

Let us observe that, since the collecting transfer functions f_c are additive on the complete lattice $\langle \wp(\mathbb{N}^k), \subseteq \rangle$, by Knaster-Tarski fixpoint theorem, $[\![G]\!]_S$ is well defined. For $\vec{x} \in \mathbb{N}^{k'}$, we write $[\![G]\!]_{\vec{x}}$ instead of $[\![G]\!]_{\{\vec{x}\}}$. Notice that $[\![G]\!]_{(\cdot)}$ is an additive function, so that, for any program point $u \in N$, $[\![G]\!]_S[u] = \bigcup_{\vec{x} \in S} [\![G]\!]_{\vec{x}}[u]$ holds.

5.1. Turing completeness of BACFGs

Let us recall that a ssmn abstract semantics needs an underlying Turing complete concrete semantics of programs (cf. Assumption 2.1). A crucial observation is that BACFGs are Turing complete despite not including full (both positive and negative) Boolean tests. This is proved by showing that any program of an Unlimited Register Machine (URM), which is a well-known Turing complete computational model [12], can be simulated by a BACFG.

Theorem 5.3 *(Turing completeness of BACFGs).* BACFGs *are a Turing complete computational model.*

Before getting into the technical details, it is worth providing, first, an intuition of the proof of Theorem 5.3. Using the definition and notation of Cutland [12, Section 1.2], let us recall the four types of instructions of URMs:

- $z(n)$: sets register r_n to 0 ($r_n \leftarrow 0$) and transfers the control to the next instruction;
- $s(n)$: increments register r_n by 1 ($r_n \leftarrow r_n + 1$) and transfers the control to the next instruction;
- $t(m, n)$: sets register r_n to r_m ($r_n \leftarrow r_m$) and transfers the control to the next instruction;

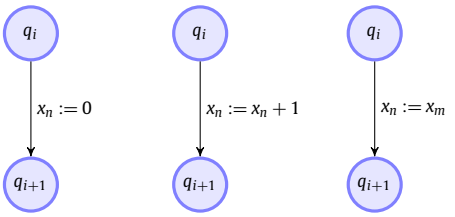

Fig. 2. BACFGs simulating: $z(n)$ (left), $s(n)$ (center), $t(m,n)$ (right).

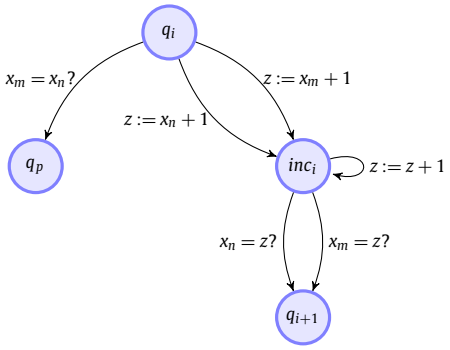

Fig. 3. BACFG simulating a jump instruction $j(m,n,p)$.

- $j(m,n,p)$: if $r_m = r_n$ and I_p is a proper instruction, then it jumps to the instruction I_p; otherwise, it skips to the next instruction;

It turns out that all these URM instructions can be simulated by the BACFGs depicted in Figs. 2 and 3. While the BACFGs in Fig. 2 are trivial, let us describe more in detail how the BACFG in Fig. 3 simulates a jump instruction $j(m,n,p)$. Intuitively, a difficulty arises for simulating the negative branch $x_n \neq x_m?$. Here, the BACFG at node q_i initialises a fresh unused variable z with both x_n+1 and x_m+1 and transfers the control to a node inc_i where z is incremented infinitely many times. Thus, in the least fixpoint solution, at node inc_i the variable z stores any value $v > \min(x_m, x_n)$, including $z = \max(x_m, x_n)$. Suppose now that $x_n > x_m$ holds: in this case, the guard $x_n = z?$ between nodes inc_i and q_{i+1} eventually will be made true and at the node q_{i+1} the store will retain the original values of all variables (x_m and x_n included), except for the new variable z which will be ignored by the remaining nodes. The case $x_m > x_n$ is analogous. Therefore, it turns out that the node q_{i+1} will be reached if and only if $x_m \neq x_n$ holds, while q_p will be reached if and only if $x_m = x_n$ holds, thus providing a simulation for the jump instruction $j(m,n,p)$.

We next give a precise definition of a model of computation for BACFGs which is able to simulate URMs. Firstly, let us formalise the operational semantics of URMs. Given a URM program $P = (I_1, \ldots, I_t)$ consisting of a sequence of t instructions I_j, we denote its states by vectors $\vec{x} \in \mathbb{N}^{k_P}$, where k_P is the largest index of registers used by P (which is finite). A configuration of a URM is a pair $\langle \vec{x}, c \rangle \in \mathbb{N}^{k_P} \times \mathbb{N}$ representing the state of the (possibly used) registers, and the current instruction I_c. Then, the operational semantics is as follows:

Definition 5.4 *(Operational semantics $\Rightarrow$ of URMs).* Given a URM program $P = (I_1, \ldots, I_t)$, its operational semantics is given by the transition function $\Rightarrow : (\mathbb{N}^{k_P} \times \mathbb{N}) \to (\mathbb{N}^{k_P} \times \mathbb{N})$ defined as follows: for all $\vec{x} \in \mathbb{N}^{k_P}, 1 \leq c \leq t$,

$$\langle \vec{x}, c \rangle \Rightarrow \begin{cases} \langle (x_1, \ldots, x_{n-1}, 0, x_{n+1}, \ldots, x_{k_P}), c+1 \rangle & \text{if } I_c = z(n) \\ \langle (x_1, \ldots, x_{n-1}, x_n+1, x_{n+1}, \ldots, x_{k_P}), c+1 \rangle & \text{if } I_c = s(n) \\ \langle (x_1, \ldots, x_{m-1}, x_n, x_{m+1}, \ldots, x_{k_P}), c+1 \rangle & \text{if } I_c = t(m,n) \\ \langle \vec{x}, q \rangle & \text{if } I_c = j(m,n,q) \wedge x_m = x_n \\ \langle \vec{x}, c+1 \rangle & \text{if } I_c = j(m,n,q) \wedge x_m \neq x_n \end{cases}$$

The URM halts when it reaches a configuration $\langle \vec{x}, t+1 \rangle$. □

Getting back to control flow graphs, let us point out that the collecting semantics of BACFGs of Definition 5.2 can be expressed in terms of Kleene's iterates as follows.

Definition 5.5 *(Kleene's iterates of BACFGs).* Let $G = (N, E, s, e)$ be a BACFG with k_G variables. The corresponding initial state $\bot^s_{\vec{x}} : N \to \wp(\mathbb{N}^{k_G})$, with $\vec{x} \in \mathbb{N}^{k_G}$, and transformer $F_G : (N \to \wp(\mathbb{N}^{k_G})) \to (N \to \wp(\mathbb{N}^{k_G}))$ are defined as follows: for all $v \in N$ and $\mathcal{X} \in N \to \wp(\mathbb{N}^{k_G})$,

$$\bot^s_{\vec{x}}[v] \triangleq \begin{cases} \{\vec{x}\} & \text{if } v = s \\ \varnothing & \text{otherwise} \end{cases}$$

$$F_G(\mathcal{X})[v] \triangleq \bigcup_{(u,c,v)\in E} f_c(\mathcal{X}[u]) \cup \mathcal{X}[v]$$

The sequence of *Kleene's iterates of* G starting from $\bot^s_{\vec{x}}$ is the infinite (pointwise) ascending chain $\{F^i_G(\bot^s_{\vec{x}})\}_{i\in\mathbb{N}} \subseteq N \to \wp(\mathbb{N}^{k_G})$, where the powers of the function F_G are inductively defined in the usual way: $F^0_G(\mathcal{X}) \triangleq \mathcal{X}$ and $F^{i+1}_G(\mathcal{X}) \triangleq F_G(F^i_G(\mathcal{X}))$. □

Observe that the collecting semantics of Definition 5.2 coincides with the least fixed point of F_G above $\bot^s_{\vec{x}}$ w.r.t. the pointwise inclusion order of the complete lattice $N \to \wp(\mathbb{N}^{k_G})$ obtained by lifting $\langle \wp(\mathbb{N}^{k_G}), \subseteq\rangle$. Moreover, since F_G is a Scott-continuous function (even more, F_G preserves arbitrary least upper bounds), by Kleene's fixpoint theorem, it turns out that

$$\cup_{i\in\mathbb{N}} F^i_G(\bot^s_{\vec{x}})[v] = [\![G]\!]_{\vec{x}}[v].$$

Our key insight is that the states of our abstract computational model can be represented as "differences" between consecutive Kleene's iterates of F_G.

Definition 5.6 *(Operational semantics Δ of BACFGs).* Given a BACFG $G = (N, E, s, e)$, its operational semantics is given by the function $\Delta_G : (N \to \wp(\mathbb{N}^{k_G})) \to (N \to \wp(\mathbb{N}^{k_G}))$ defined as follows: for all $\mathcal{X} : N \to \wp(\mathbb{N}^{k_G})$ and $v \in N$,

$$\Delta_G(\mathcal{X})[v] \triangleq \bigcup_{(u,c,v)\in E} f_c(\mathcal{X}[u]).$$ □

Therefore, $\Delta_G(\mathcal{X})[v]$ is the standard "meet-over-paths" of classical dataflow analysis, namely, the join of the transfer functions $f_c(X)$ over all the edges (u, c, v) of G.

Lemma 5.7. *Let $G = (N, E, s, e)$ be a BACFG. For all $n \in \mathbb{N}$, $\mathcal{X} : N \to \wp(\mathbb{N}^{k_G})$, $v \in N$, we have that $F^n_G(\mathcal{X})[v] = \cup_{0\le i\le n} \Delta^i_G(\mathcal{X})[v]$.*

Proof. We proceed by induction on $n \in \mathbb{N}$.

- $n = 0$: $F^0_G(\mathcal{X})[v] = \mathcal{X}[v] = \Delta^0_G(\mathcal{X})[v] = \cup_{0\le i\le 0}\Delta^i_G(\mathcal{X})[v]$;
- $n > 0$:

$$\begin{aligned}
F^n_G(\mathcal{X})[v] &= F_G(F^{n-1}_G(\mathcal{X}))[v] \\
&= \bigcup_{(u,c,v)\in E} f_c(F^{n-1}_G(\mathcal{X})[u]) \cup F^{n-1}_G(\mathcal{X})[v] && \text{[by ind. hyp.]} \\
&= \Delta_G(\cup_{0\le i\le n-1}\Delta^i_G(\mathcal{X}))[v] \cup (\cup_{0\le i\le n-1}\Delta^i_G(\mathcal{X})[v]) && \text{[by additivity of } \Delta_G\text{]} \\
&= \cup_{1\le i\le n}\Delta^i_G(\mathcal{X})[v] \cup (\cup_{0\le i\le n-1}\Delta^i_G(\mathcal{X})[v]) \\
&= \cup_{0\le i\le n}\Delta^i_G(\mathcal{X})[v]
\end{aligned}$$

This closes the proof. □

In the following, we describe an effective procedure τ to translate a URM program P into a BACFG which simulates P.

Definition 5.8 *(Transformer τ).* Given a URM $P = (I_1, \ldots, I_t)$, the procedure $\tau(P)$ starts from $N_0 = \{q_1, \ldots, q_t, q_{t+1}\}$ and $E_0 = \varnothing$ as, resp., sets of nodes and edges. Then, for all the instructions I_i of P:

(i) If $I_i \in \{z(n), s(n), t(m, n) \mid n, m \in \mathbb{N}\}$ then $\tau(P)$ adds an edge between the nodes q_i and q_{i+1} as depicted by the diagrams in Fig. 2. For instance, if $I_i = z(n)$ the edge $(q_i, x_n := 0, q_{i+1})$ is added to the set E; the other cases are analogous.
(ii) If $I_i = j(m, n, q)$, for some m, n, q, then $\tau(P)$ adds a new node inc_i and the edges depicted by the diagram in Fig. 3. We shall use the variable z as a syntactic shorthand for x_{k_P+1}, which is a fresh variable not used in P.

Let N and E denote the final sets of, resp., nodes and edges obtained by applying the above two steps (i)–(ii) for all the instructions of P. Then, $\tau(P)$ returns a set of BACFGs $\{(N, E, q_s, q_e) \mid q_s, q_e \in N_0\}$, where start and end nodes freely range in N_0 and each BACFG has $k_G \in \{k_P, k_P + 1\}$ variables. Without loss of generality, we assume $k_G \triangleq k_P + 1$: In fact, if the program P contains no jump and the extra-variable z is actually not used, then we can add a useless edge involving the extra-variable z. □

In the rest of this section, we prove that the BACFG $G = (N, E, q_1, q_{t+1}) \in \tau(P)$ simulates the original URM program P. To prove our claim, we define an equivalence relation between sets of states of a BACFG in $\tau(P)$. Intuitively, two sets $\mathcal{X}$ and $\mathcal{X}'$ are deemed equivalent if, for each node, $\mathcal{X}$ and $\mathcal{X}'$ induce the same invariant on the first k_P variables, except for the states inc_i whose variable z is already greater than the variables occurring in the outgoing guards.

Definition 5.9 *(Equivalence $\approx$).* Let $P = (I_1, \ldots, I_t)$ be a URM program and $G = (N, E, q_s, q_e) \in \tau(P)$. Then, given $\mathcal{X}, \mathcal{X}' : N \to \wp(\mathbb{N}^{k_G})$, the relation $\mathcal{X} \approx \mathcal{X}'$ is defined as follows:

(1) $\forall i \in [1, t+1].\ \mathcal{X}[q_i] \restriction_{k_P} = \mathcal{X}'[q_i] \restriction_{k_P}$;
(2) $\forall i \in [1, t], \forall m \in [1, k_P], \forall (inc_i, x_m = z?, q_{i+1}) \in E.\ \{\vec{x} \in \mathcal{X}[inc_i] \mid z \leq x_m\} = \{\vec{x} \in \mathcal{X}'[inc_i] \mid z \leq x_m\}$. □

Let us point out that condition (2) is motivated by the observation that for nodes of type inc_i, the states containing values of x_m below z do not matter. Clearly, observe that $\approx$ is an equivalence relation. Moreover, it turns out that the operational semantic function Δ_G of Definition 5.6 preserves this equivalence $\approx$.

Lemma 5.10. *Let $P = (I_1, \ldots, I_t)$ be a URM program and $G = (N, E, q_s, q_e) \in \tau(P)$. Then, for all $\mathcal{X}, \mathcal{X}' : N \to \wp(\mathbb{N}^{k_G})$, $\mathcal{X} \approx \mathcal{X}' \Rightarrow \Delta_G(\mathcal{X}) \approx \Delta_G(\mathcal{X}')$.*

Proof. Assume that $\mathcal{X} \approx \mathcal{X}'$. For all $i \in [1, t+1]$ we have:

$$
\begin{aligned}
&\Delta_G(\mathcal{X})[q_i] \restriction_{k_P} \\
&= \bigcup_{(u,c,q_i) \in E} f_c(\mathcal{X}[u]) \restriction_{k_P} \\
&= \bigcup_{(q_u,c,q_i) \in E} f_c(\mathcal{X}[q_u]) \restriction_{k_P} \cup \bigcup_{(inc_{i-1}, x_m = z?, q_i) \in E} f_{x_m = z?}(\mathcal{X}[inc_{i-1}]) \restriction_{k_P} \\
&= \bigcup_{(q_u,c,q_i) \in E} f_c(\mathcal{X}[q_u]) \restriction_{k_P} \cup \bigcup_{(inc_{i-1}, x_m = z?, q_i) \in E} f_{x_m = z?}(\{\vec{x} \in \mathcal{X}[inc_{i-1}] \mid z \leq x_m\}) \restriction_{k_P} && \text{[as } \mathcal{X} \approx \mathcal{X}'\text{]} \\
&= \bigcup_{(q_u,c,q_i) \in E} f_c(\mathcal{X}'[q_u]) \restriction_{k_P} \cup \bigcup_{(inc_{i-1}, x_m = z?, q_i) \in E} f_{x_m = z?}(\{\vec{x} \in \mathcal{X}'[inc_{i-1}] \mid z \leq x_m\}) \restriction_{k_P} \\
&= \bigcup_{(q_u,c,q_i) \in E} f_c(\mathcal{X}'[q_u]) \restriction_{k_P} \cup \bigcup_{(inc_{i-1}, x_m = z?, q_i) \in E} f_{x_m = z?}(\mathcal{X}'[inc_{i-1}]) \restriction_{k_P} \\
&= \Delta_G(\mathcal{X}')[q_i] \restriction_{k_P}.
\end{aligned}
$$

Moreover, for all $i \in [1, t]$, $m \in [1, k_P]$ such that $(inc_i, x_m = z?, q_{i+1}) \in E$:

$$
\begin{aligned}
&\{\vec{x} \in \Delta_G(\mathcal{X})[inc_i] \mid z \leq x_m\} \\
&= \{\vec{x} \in \bigcup_{(u,c,inc_i) \in E} f_c(\mathcal{X}[u]) \mid z \leq x_m\} \\
&= \{\vec{x} \in \bigcup_{(q_i,c,inc_i) \in E} f_c(\mathcal{X}[q_i]) \mid z \leq x_m\} \cup \{\vec{x} \in f_{z:=z+1}(\mathcal{X}[inc_i]) \mid z \leq x_m\} \\
&= \{\vec{x} \in f_{z:=x_n+1}(\mathcal{X}[q_i]) \cup f_{z:=x_m+1}(\mathcal{X}[q_i]) \mid z \leq x_m\} \cup \\
&\qquad\qquad \{\vec{x} \in f_{z:=z+1}(\mathcal{X}[inc_i]) \mid z \leq x_m\} \\
&= \{\vec{x} \in f_{z:=x_n+1}(\mathcal{X}[q_i]) \mid z \leq x_m\} \cup \{\vec{x} \in f_{z:=z+1}(\mathcal{X}[inc_i]) \mid z \leq x_m\},
\end{aligned}
$$

for some $n \neq m$. Since $\mathcal{X}[q_i] \restriction_{k_P} = \mathcal{X}'[q_i] \restriction_{k_P}$ it follows that $f_{z:=x_n+1}(\mathcal{X}[q_i]) = f_{z:=x_n+1}(\mathcal{X}'[q_i])$. Also note that:

$$
\begin{aligned}
&\{\vec{x} \in f_{z:=z+1}(\mathcal{X}[inc_i]) \mid z \leq x_m\} \\
&= \{\vec{x} \in f_{z:=z+1}(\{\vec{x} \in \mathcal{X}[inc_i] \mid z \leq x_m\}) \mid z \leq x_m\} && \text{[as } \mathcal{X} \approx \mathcal{X}'\text{]} \\
&= \{\vec{x} \in f_{z:=z+1}(\{\vec{x} \in \mathcal{X}'[inc_i] \mid z \leq x_m\}) \mid z \leq x_m\} \\
&= \{\vec{x} \in f_{z:=z+1}(\mathcal{X}'[inc_i]) \mid z \leq x_m\}.
\end{aligned}
$$

Hence,

$$\begin{aligned}
&\{\vec{x} \in \Delta_G(\mathcal{X})[inc_i] \mid z \leq x_m\} \\
&= \{\vec{x} \in f_{z:=x_n+1}(\mathcal{X}[q_i]) \mid z \leq x_m\} \cup \{\vec{x} \in f_{z:=z+1}(\mathcal{X}[inc_i]) \mid z \leq x_m\} \\
&= \{\vec{x} \in f_{z:=x_n+1}(\mathcal{X}'[q_i]) \mid z \leq x_m\} \cup \{\vec{x} \in f_{z:=z+1}(\mathcal{X}'[inc_i]) \mid z \leq x_m\} \\
&= \{\vec{x} \in \Delta_G(\mathcal{X}')[inc_i] : z \leq x_m\}.
\end{aligned}$$

This therefore shows that $\Delta_G(\mathcal{X}) \approx \Delta_G(\mathcal{X}')$. □

Let us now show that each transition of a URM program can be simulated by a finitely many applications, say k, of the function Δ. Moreover, whenever Δ is applied less than k times, we obtain the empty set of states for all the nodes. Let us define the following concatenation operation for sequences: $(a_1, \ldots, a_k) : a \triangleq (a_1, \ldots, a_k, a)$. Concatenation will be used to deal with the fact that our transformed BACFG has an additional variable w.r.t. the original URM program.

Lemma 5.11. *Let $P = (I_1, \ldots, I_t)$ be a URM program. For all BACFGs $G = (N, E, q_s, q_e) \in \tau(P)$, $\vec{x}, \vec{x}' \in \mathbb{N}^{k_P}$, $s' \in \mathbb{N}$, if $\langle \vec{x}, s \rangle \Rightarrow \langle \vec{x}', s' \rangle$ then there exists $k \in \mathbb{N}$ such that:*

(1) $\Delta_G^k(\bot_{\vec{x}:0}^{q_s}) \approx \bot_{\vec{x}':0}^{q_{s'}}$;
(2) $\forall i \in [1, k-1], \forall j \in [1, t+1].\ \Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_j] = \varnothing$.

Proof. Assume that $\langle \vec{x}, s \rangle \Rightarrow \langle \vec{x}', s' \rangle$. We distinguish three cases.

(i) Let $I_s \in \{z(n), s(n), t(m,n) \mid n, m \in \mathbb{N}\}$. Consider the case $I_s = z(n)$ for some n (the remaining cases are analogous), so that $s' = s + 1$. For $k = 1$ we have that:

$$\begin{aligned}
\Delta_G(\bot_{\vec{x}:0}^{q_s}) &= \lambda v.\ \bigcup_{(u,c,v) \in E} f_c(\bot_{\vec{x}:0}^{q_s}[u]) \\
&= \lambda v. \begin{cases} f_{x_n:=0}(\{\vec{x} : 0\}) & \text{if } v = q_{s+1} \\ \varnothing & \text{otherwise} \end{cases} && \text{[by def. of } G\text{]} \\
&= \lambda v. \begin{cases} \vec{x}' : 0 & \text{if } v = q_{s+1} \\ \varnothing & \text{otherwise} \end{cases} \\
&= \bot_{\vec{x}':0}^{q_{s'}} && \text{[as } s' = s + 1\text{]}
\end{aligned}$$

Thus, $\Delta_G(\bot_{\vec{x}:0}^{q_s}) \approx \bot_{\vec{x}':0}^{q_{s'}}$, i.e., property (1) holds with $k = 1$. Property (2) trivially holds since for $k = 1$, $[1, k-1]$ is the empty set.

(ii) Let $I_s = j(m, n, p)$ and assume that $x_m = x_n$ holds, so that the next instruction to execute is I_q, i.e., $s' = p$. For $k = 1$ we have that:

$$\begin{aligned}
\Delta_G(\bot_{\vec{x}:0}^{q_s}) &= \lambda v.\ \bigcup_{(u,c,v) \in E} f_c(\bot_{\vec{x}:0}^{q_s}[u]) \\
&= \lambda v. \begin{cases} f_{x_m = x_n?}(\{\vec{x} : 0\}) & \text{if } v = q_p \\ f_{z:=x_m+1}(\{\vec{x} : 0\}) \cup f_{z:=x_n+1}(\{\vec{x} : 0\}) & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases} && \text{[by def. of } G\text{]} \\
&= \lambda v. \begin{cases} \{\vec{x} : 0\} & \text{if } v = q_p \\ \{\vec{x} : x_m + 1\} & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases} && \text{[as } x_m = x_n\text{]}
\end{aligned}$$

Since $s' = p$ and $\vec{x} = \vec{x}'$, we have that:

- for all $i \in [1, t+1]$, $\Delta_G(\bot_{\vec{x}:0}^{q_s})[q_i] = \bot_{\vec{x}:0}^{q_p}[q_i] = \bot_{\vec{x}':0}^{q_{s'}}[q_i]$;
- for all $i \in [1, t+1]$ and $m \in [1, k_P]$ such that $(inc_i, x_m = z?, q_{i+1}) \in E$:

$$\{\vec{x} \in \Delta_G(\bot_{\vec{x}:0}^{q_s})[inc_i] \mid z \leq x_m\} = \varnothing = \{\vec{x} \in \bot_{\vec{x}':0}^{q_{s'}}[inc_i] \mid z \leq x_m\}.$$

Thus, $\Delta_G(\bot^{q_s}_{\vec{x}:0}) \approx \bot^{q_{s'}}_{\vec{x}':0}$ holds, i.e., property (1) holds with $k=1$. Moreover, once again property (2) trivially holds because $[1, k-1]$ is empty.

(iii) The last possible case is $I_s = j(m,n,q)$ with $x_m \neq x_n$, so that the next instruction to execute is I_{s+1}, i.e., $s' = s+1$. We first prove, by induction, that for all $i \geq 1$, the following implication holds:

$$i \leq |x_m - x_n| \Rightarrow \Delta^i_G(\bot^{q_s}_{\vec{x}:0}) = \lambda v. \begin{cases} f_{z:=x_n+i}(\{\vec{x}:0\}) \cup f_{z:=x_m+i}(\{\vec{x}:0\}) & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases} \qquad (*)$$

For the base case $i=1$, we have that:

$$\begin{aligned} &\Delta_G(\bot^{q_s}_{\vec{x}:0}) \\ &= \lambda v. \bigcup_{(u,c,v)\in E} f_c(\bot^{q_s}_{\vec{x}:0}[u]) \\ &= \lambda v. \begin{cases} f_{z:=x_n+1}(\{\vec{x}:0\}) \cup f_{z:=x_m+1}(\{\vec{x}:0\}) & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases} && \text{[by def. of } G\text{]} \end{aligned}$$

For the inductive case $i > 1$, assume that $i \leq |x_m - x_n|$ (if $i > |x_m - x_n|$ the implication $(*)$ trivially holds). We have that:

$$\begin{aligned} &\Delta^i_G(\bot^{q_s}_{\vec{x}:0}) \\ &= \Delta_G(\Delta^{i-1}_G(\bot^{q_s}_{\vec{x}:0})) \\ &= \Delta_G\left(\lambda v. \begin{cases} f_{z:=x_n+i-1}(\{\vec{x}:0\}) \cup f_{z:=x_m+i-1}(\{\vec{x}:0\}) & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases}\right) && \text{[by ind. hyp. for } i-1 \leq |x_m - x_n|\text{]} \\ &= \lambda v. \begin{cases} f_{z:=z+1}\big(f_{z:=x_n+i-1}(\{\vec{x}:0\}) \cup f_{z:=x_m+i-1}(\{\vec{x}:0\})\big) & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases} \\ &&& \text{[as } (inc_s, z:=z+1, inc_s) \text{ is an edge of } G \text{ and} \\ &&& x_m \neq x_n + i - 1 \text{ and } x_n \neq x_m + i - 1 \text{ since } i - 1 < |x_m - x_n|\text{]} \\ &= \lambda v. \begin{cases} f_{z:=x_n+i}(\{\vec{x}:0\}) \cup f_{z:=x_m+i}(\{\vec{x}:0\}) & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases} \end{aligned}$$

We have therefore shown the implication $(*)$. Now, note that for $k = |x_m - x_n| + 1$ we have that:

$$\begin{aligned} &\Delta^{|x_m-x_n|+1}_G(\bot^{q_s}_{\vec{x}:0}) \\ &= \Delta_G(\Delta^{|x_m-x_n|}_G(\bot^{q_s}_{\vec{x}:0})) \\ &= \Delta_G\left(\lambda v. \begin{cases} f_{z:=x_n+|x_m-x_n|}(\{\vec{x}:0\}) \cup f_{z:=x_m+|x_m-x_n|}(\{\vec{x}:0\}) & \text{if } v = inc_s \\ \varnothing & \text{otherwise} \end{cases}\right) && \text{[by } (*)\text{]} \\ &= \lambda v. \begin{cases} f_{z:=x_n+|x_m-x_n|+1}(\{\vec{x}:0\}) \cup f_{z:=x_m+|x_m-x_n|+1}(\{\vec{x}:0\}) & \text{if } v = inc_s \\ \{\vec{x} : \max(x_m, x_n)\} & \text{if } v = q_{s+1} \\ \varnothing & \text{otherwise} \end{cases} \\ &&& \text{[because } \max(x_m,x_n) = \min(x_m,x_n) + |x_m - x_n|\text{]} \end{aligned}$$

Since $s' = s+1$ and $\vec{x} = \vec{x}'$, we have that:

- for all $i \in [1, t+1]$, $\Delta^{|x_m-x_n|+1}_G(\bot^{q_s}_{\vec{x}:0})[q_i]\restriction_{k_P} = \bot^{q_{s+1}}_{\vec{x}:0}[q_i]\restriction_{k_P} = \bot^{q_{s'}}_{\vec{x}':0}[q_i]\restriction_{k_P}$;
- for all $i \in [1, t+1]$ and $m \in [1, k_P]$ such that $(inc_i, x_m = z?, q_{i+1}) \in E$:

$$\{\vec{x} \in \Delta^{|x_m-x_n|+1}_G(\bot^{q_s}_{\vec{x}:0})[inc_i] \mid z \leq x_m\} = \varnothing = \{\vec{x} \in \bot^{q_{s'}}_{\vec{x}':0}[inc_i] : z \leq x_m\}.$$

Therefore, $\Delta_G(\bot^{q_s}_{\vec{x}:0}) \approx \bot^{q_{s'}}_{\vec{x}':0}$ holds. Furthermore, for all $i \in [1, |x_m - x_n|]$, by applying the implication $(*)$ we obtain:

$$\Delta^i_G(\bot^{q_s}_{\vec{x}:0}) = \begin{cases} f_{z:=x_n+i}(\{\vec{x}:0\}) \cup f_{z:=x_m+i}(\{\vec{x}:0\}) & \text{if } v = inc_s \\ \varnothing & \text{otherwise.} \end{cases}$$

Thus, for all $j \in [1, t+1]$, $\Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_j] = \varnothing$ holds and this concludes the proof. □

Let us now generalise Lemma 5.11 to any number of execution steps $\Rightarrow^n$ performed by a URM program. In particular, we show that if the URM halts then our abstract model will reach, after finitely many steps, a state that stores the URM output in its end node. Likewise, whenever the URM diverges, the state of the end node will be empty.

Proposition 5.12. *Let $P = (I_1, \ldots, I_t)$ be a URM program. Then, for all $G = (N, E, q_s, q_e) \in \tau(P)$, $\vec{x}, \vec{x}' \in \mathbb{N}^{k_P}$, $n \in \mathbb{N}$, if $\langle \vec{x}, s\rangle \Rightarrow^n \langle \vec{x}', t+1\rangle$ then there exists $n' \in \mathbb{N}$ such that:*

(1) $\Delta_G^{n'}(\bot_{\vec{x}:0}^{q_s}) \approx \bot_{\vec{x}':0}^{q_{t+1}}$;
(2) $\forall i \in [0, n'-1].\ \Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] = \varnothing$.

Proof. We proceed by induction on $n \in \mathbb{N}$.

- Base case $n = 0$, so that $\langle \vec{x}, s\rangle = \langle \vec{x}', t+1\rangle$. Therefore, for $n' = 0$ the property (1) holds because:

$$\begin{aligned} \Delta_G^{n'}(\bot_{\vec{x}:0}^{q_s}) &= \Delta_G^0(\bot_{\vec{x}':0}^{q_{t+1}}) && [\text{as } n' = 0, t+1 = s, \vec{x}' = \vec{x}] \\ &= \bot_{\vec{x}':0}^{q_{t+1}} && [\text{as } \Delta_G^0 = \lambda x.x] \end{aligned}$$

  Moreover, the property (2) trivially holds because $[0, n'-1]$ is empty.
- Inductive case $n > 0$, so that $\langle \vec{x}, s\rangle \Rightarrow \langle \vec{x}'', s''\rangle \Rightarrow^{n-1} \langle \vec{x}', t+1\rangle$. We have that:
  – by Lemma 5.11, and observing that $s \neq t+1$, we know that there exists $m \in \mathbb{N}$ such that: (1) $\Delta_G^m(\bot_{\vec{x}:0}^{q_s}) \approx \bot_{\vec{x}'':0}^{q_{s''}}$; (2) $\forall i \in [0, m-1].\ \Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] = \varnothing$.
  – by inductive hypothesis there exists $n'' \in \mathbb{N}$ such that: (i) $\Delta_G^{n''}(\bot_{\vec{x}'':0}^{q_{s''}}) \approx \bot_{\vec{x}':0}^{q_{t+1}}$; (ii) $\forall i \in [0, n''-1].\ \Delta_G^i(\bot_{\vec{x}'':0}^{q_{s''}})[q_{t+1}] = \varnothing$.
  Therefore, it turns out that:

$$\begin{aligned} \Delta_G^{n''+m}(\bot_{\vec{x}:0}^{q_s}) &= \Delta_G^{n''}(\Delta_G^m(\bot_{\vec{x}:0}^{q_s})) \\ &\approx \Delta_G^{n''}(\bot_{\vec{x}'':0}^{q_{s''}}) && [\text{as } \Delta_G^m(\bot_{\vec{x}:0}^{q_s}) \approx \bot_{\vec{x}'':0}^{q_{s''}}, \text{by Lemma 5.10}] \\ &\approx \bot_{\vec{x}':0}^{q_{t+1}} && [\text{by ind. hyp.}] \end{aligned}$$

  thus showing (1) for $n'' + m$. Moreover, for all $i \in [0, n''-1]$:

$$\begin{aligned} \Delta_G^{i+m}(\bot_{\vec{x}:0}^{q_s}) &= \Delta_G^i(\Delta_G^m(\bot_{\vec{x}:0}^{q_s})) \\ &\approx \Delta_G^i(\bot_{\vec{x}'':0}^{q_{s''}}). && [\text{as } \Delta_G^m(\bot_{\vec{x}:0}^{q_s}) \approx \bot_{\vec{x}'':0}^{q_{s''}}, \text{by Lemma 5.10}] \end{aligned}$$

  Recall that, by inductive hypothesis, $\Delta_G^i(\bot_{\vec{x}'':0}^{q_{s''}})[q_{t+1}] = \varnothing$, so that we obtain that for all $i \in [m, n''+m-1]$, $\Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] = \varnothing$ holds. Since, by Lemma 5.11, we have that for all $i \in [0, m-1]$, $\Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] = \varnothing$ holds, we conclude that for all $i \in [0, n''+m-1]$, $\Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] = \varnothing$, thus showing (2) for $n''+m$. □

Proposition 5.13. *Let $P = (I_1, \ldots, I_t)$ be a URM program. Then, for all $G = (N, E, q_s, q_e) \in \tau(P)$, $\vec{x} \in \mathbb{N}^{k_P}$, $n \in \mathbb{N}$:*

$$\text{if } \Delta_G^n(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] \neq \varnothing \text{ then } \exists \vec{x}' \in \mathbb{N}^{k_P}, \exists n' \in \mathbb{N}.\ \langle \vec{x}, s\rangle \Rightarrow^{n'} \langle \vec{x}', t+1\rangle.$$

Proof. We proceed by induction on $n \in \mathbb{N}$:

- $n = 0$: by hypothesis, $\Delta_G^0(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] = \bot_{\vec{x}:0}^{q_s}[q_{t+1}] \neq \varnothing$, so that $s = t+1$ and, in turn, $\langle \vec{x}, s\rangle \Rightarrow^0 \langle \vec{x}, t+1\rangle$.
- $n > 0$: by hypothesis, we have that $\Delta_G^n(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] \neq \varnothing$. We consider $s \neq t+1$, otherwise, one can trivially pick $n' = 0$. By construction, there exist $\vec{x}'', s''$ such that $\langle \vec{x}, s\rangle \Rightarrow \langle \vec{x}'', s''\rangle$, and, by Lemma 5.11, there exists m such that $\Delta_G^m(\bot_{\vec{x}:0}^{q_s}) \approx \bot_{\vec{x}'':0}^{q_{s''}}$. Note that $n \geq m$ holds, since for all $i \in [1, m-1]$, $\Delta_G^i(\bot_{\vec{x}:0}^{q_s})[q_{t+1}] = \varnothing$ holds. By Lemma 5.10, it follows that $\Delta_G^{n-m}(\Delta_G^m(\bot_{\vec{x}:0}^{q_s})) \approx \Delta_G^{n-m}(\bot_{\vec{x}'':0}^{q_{s''}})$, By hypothesis and Definition 5.9, we have that $\Delta_G^{n-m}(\Delta_G^m(\bot_{\vec{x}:0}^{q_s}))[q_{t+1}] = \Delta_G^{n-m}(\bot_{\vec{x}'':0}^{q_{s''}})[q_{t+1}] \neq \varnothing$. We conclude by applying the inductive hypothesis, that entails the existence of m' such that $\langle \vec{x}, s\rangle \Rightarrow \langle \vec{x}'', s''\rangle \Rightarrow^{m'} \langle \vec{x}', s'\rangle$. □

The next two results show that for a given URM program $P = (I_1, \ldots, I_t)$, the BACFG $G = (N, E, q_1, q_t) \in \tau(P)$ simulates the operational semantics of P starting from its first instruction I_1.

Proposition 5.14. *Let $P = (I_1, \ldots, I_t)$ be a given URM program and $G = (N, E, q_1, q_{t+1}) \in \tau(P)$. Then, for all $\vec{x}, \vec{x}' \in \mathbb{N}^{k_P}$ and $n \in \mathbb{N}$:*

if $\langle \vec{x}, 1\rangle \Rightarrow^n \langle \vec{x}', t+1\rangle$ then $[\![G]\!]_{\vec{x}:0}[q_{t+1}] \restriction_{k_P} = \{\vec{x}'\}$.

Proof. By Proposition 5.12 there exists n' such that $\Delta_G^{n'}(\bot^{q_1}_{\vec{x}:0}) \approx \bot^{q_{t+1}}_{\vec{x}':0}$ and for all $i \in [0, n'-1]$, $\Delta_G^i(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] = \varnothing$. Let us prove, by induction on i, that for all $i > n'$, $\Delta_G^i(\bot^{q_1}_{\vec{x}:0}) \approx \lambda v.\varnothing$.

- $i = n'+1$:

$$\begin{aligned}
\Delta_G^{n'+1}(\bot^{q_1}_{\vec{x}:0}) &= \Delta_G(\Delta_G^{n'}(\bot^{q_1}_{\vec{x}:0})) \\
&\approx \Delta_G(\bot^{t+1}_{\vec{x}':0}) && \text{[as } \Delta_G^{n'}(\bot^{q_1}_{\vec{x}:0}) \approx \bot^{t+1}_{\vec{x}':0}\text{, by Lemma 5.10]} \\
&= \lambda v.\varnothing. && \text{[by def. of } G\text{]}
\end{aligned}$$

- $i > n'+1$:

$$\begin{aligned}
\Delta_G^{i}(\bot^{q_1}_{\vec{x}:0}) &= \Delta_G(\Delta_G^{i-1}(\bot^{q_1}_{\vec{x}:0})) \\
&\approx \Delta_G(\lambda v.\varnothing) && \text{[by ind. hyp. and Lemma 5.10]} \\
&= \lambda v.\varnothing
\end{aligned}$$

Thus, for all $i \neq n'$, we have that $\Delta_G^i(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] = \varnothing$. Therefore:

$$\begin{aligned}
[\![G]\!]_{\vec{x}:0}[q_{t+1}] \restriction_{k_P} &= \cup_{i\in\mathbb{N}} F^i(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] \restriction_{k_P} && \text{[by Kleene's fixpoint theorem]} \\
&= \cup_{i\in\mathbb{N}} \Delta_G^i(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] \restriction_{k_P} && \text{[by Lemma 5.7]} \\
&= \Delta_G^{n'}(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] \restriction_{k_P} && \text{[as } \forall i \neq n'.\ \Delta_G^i(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] = \varnothing\text{]} \\
&= \{\vec{x}'\}. && \text{[as } \Delta_G^{n'}(\bot^{q_1}_{\vec{x}:0}) \approx \bot^{t+1}_{\vec{x}':0}\text{]}
\end{aligned}$$

This therefore closes the proof. □

Proposition 5.15. *Let $P = (I_1, \ldots, I_t)$ be a given URM program and $G = (N, E, q_1, q_{t+1}) \in \tau(P)$. Then, for all $\vec{x} \in \mathbb{N}^{k_P}$:*

if for all $\vec{x}' \in \mathbb{N}^{k_P}, n \in \mathbb{N}$, $\langle \vec{x}, 1\rangle \not\Rightarrow^n \langle \vec{x}', t+1\rangle$ then $[\![G]\!]_{\vec{x}:0}[q_{t+1}] = \varnothing$.

Proof. For all $n' \in \mathbb{N}$, by Proposition 5.13, we have that $\Delta_G^{n'}(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] = \varnothing$ holds. As a consequence:

$$\begin{aligned}
[\![G]\!]_{\vec{x}:0}[q_{t+1}] &= \cup_{i\in\mathbb{N}} F^i(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] && \text{[by Kleene's fixpoint theorem]} \\
&= \cup_{i\in\mathbb{N}} \Delta_G^i(\bot^{q_1}_{\vec{x}:0})[q_{t+1}] && \text{[by Lemma 5.7]} \\
&= \varnothing. && \square
\end{aligned}$$

We are now in position to prove the main result of this section.

Theorem 5.3 *(Turing completeness of BACFGs).* BACFGs *are a Turing complete computational model.*

Proof. This follows from Propositions 5.14 and 5.15 and Turing completeness of URMs [12, Theorem 4.7]. □

5.2. Concrete and abstract semantics

A key insight is that our concrete semantics is given by URM programs that satisfy the Assumption 2.1 of Turing completeness, while BACFGs provide the abstract semantics. Let us consider two Gödel numberings for URMs and BACFGs, so that for an index $a \in \mathbb{N}$, RM_a and G_a denote, resp., the a-th URM and BACFG programs. The concrete semantics $\langle \phi, = \rangle$ for URMs is defined as follows: for any index $a \in \mathbb{N}$, arity $n \in \mathbb{N}$, and input $\vec{x} \in \mathbb{N}^n$,

$$\phi_a^{(n)}(\vec{x}) \triangleq \begin{cases} y & \text{if } RM_a \text{ on input } \vec{x} \text{ halts with } y \text{ stored on its first register} \\ \uparrow & \text{otherwise.} \end{cases} \tag{3}$$

The abstract semantics of BACFGs is defined as follows.

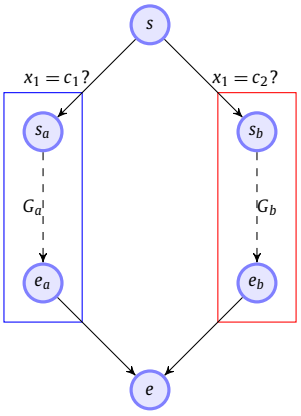

Fig. 4. The BACFG $G_{r(a,b,c_1,c_2)}$, output of the function r.

Definition 5.16 *(State semantics of BACFGs).* Let $Q \subseteq \wp(\mathbb{N}^t)$ be a predicate on sets of states with $t \in \mathbb{N}$ variables. The *state semantics* $\langle Q, = \rangle$ of BACFGs, for any index $a \in \mathbb{N}$ and arity $n \in \mathbb{N}$, is given by the function $Q_a^{(n)} : \mathbb{N}^n \to \{0, 1\}$ defined as follows: for all input $\vec{x} \in \mathbb{N}^n$,

$$Q_a^{(n)}(\vec{x}) \triangleq \begin{cases} 1 & \text{if } [\![G_a]\!]_{\vec{x}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{\vec{x}}[e_a] \restriction_t \in Q \\ 0 & \text{if } [\![G_a]\!]_{\vec{x}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{\vec{x}}[e_a] \restriction_t \notin Q \\ \uparrow & \text{if } [\![G_a]\!]_{\vec{x}}[e_a] = \varnothing, \end{cases}$$

where e_a is the end node of the BACFG G_a. □

Predicates of type $Q \subseteq \wp(\mathbb{N}^t)$ are also known as hyperproperties [8] in program security and the state semantics of Definition 5.16 models the validity of a given predicate Q at the end node of a BACFG. Note that, from a computability perspective, it is not restrictive to focus on the end node, since this can be arbitrarily chosen in a BACFG.

Theorem 5.17. *The state semantics of BACFGs in Definition 5.16 is ssmn, branching and discharging.*

We split the proof of Theorem 5.17 into three separate results that are given below. In BACFGs, we write the command $[x_a, x_{a+i}] := [x_b, x_{b+i}]$, for some indices a, b and $i \geq 0$, to denote a sequence of adjacent edges with commands $x_a := x_b$, $x_{a+1} := x_{b+1}$, ..., $x_{a+i} := x_{b+i}$. Likewise, $[x_a, x_{a+i}] := 0$ denotes a sequence of adjacent edges labelled with $x_a := 0$, $x_{a+1} := 0$, ..., $x_{a+i} := 0$.

Proposition 5.18. *The state semantics of BACFGs in Definition 5.16 is branching.*

Proof. Let $\langle Q, = \rangle$ be the state semantics of Definition 5.16 for a given predicate $Q \subseteq \wp(\mathbb{N}^t)$ on sets of states with $t \in \mathbb{N}$ variables. We define a total computable function $r : \mathbb{N}^4 \to \mathbb{N}$ as follows: given two indices a, b of BACFGs, say $G_a = (N_a, E_a, s_a, e_a)$ and $G_b = (N_b, E_b, s_b, e_b)$, and two values $c_1, c_2 \in \mathbb{N}$, the function r suitably renames the nodes of G_a and G_b to avoid clashes, and adds two fresh nodes s (for start) and e (for end) whose in/outgoing edges are depicted by the BACFG in Fig. 4, thus denoted by $G_{r(a,b,c_1,c_2)}$.

Observe that in the BACFG $G_{r(a,b,c_1,c_2)}$ with start and end nodes, resp., s and e, with inputs ranging in $\mathbb{N}^n$, for some $n \in \mathbb{N}$, the maximum variable index is $k = \max(k_a, k_b, n)$, where k_a, k_b are, resp., the maximum variable indices in G_a and G_b. Moreover, for all inputs $\vec{y} = (y_1, y_2, \ldots, y_n) \in \mathbb{N}^n$ and $c_1 \neq c_2$, it turns out that:

- if $y_1 = c_1$ then $[\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_a] = [\![G_a]\!]_{\vec{y}}[e_a] \restriction_k$ and $[\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_b] = \varnothing$;
- if $y_1 = c_2$ then $[\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_b] = [\![G_b]\!]_{\vec{y}}[e_b] \restriction_k$ and $[\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_a] = \varnothing$;
- otherwise, i.e. when $y_1 \notin \{c_1, c_2\}$, we have that $[\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_a] = [\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_b] = \varnothing$.

Consequently:

$$[\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e] = [\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_a] \cup [\![G_{r(a,b,c_1,c_2)}]\!]_{\vec{y}}[e_b] = \begin{cases} [\![G_a]\!]_{\vec{y}}[e_a] \restriction_k & \text{if } y_1 = c_1 \\ [\![G_b]\!]_{\vec{y}}[e_b] \restriction_k & \text{if } y_1 = c_2 \\ \varnothing & \text{otherwise.} \end{cases}$$

Hence, r is a total computable function such that for all $a, b, c_1, c_2, x \in \mathbb{N}$ with $c_1 \neq c_2$:

$$
\begin{aligned}
&\lambda\vec{y}.Q^{(n)}_{r(a,b,c_1,c_2)}(x,\vec{y})\\
&= \lambda\vec{y}.\begin{cases} 1 & \text{if } [\![G_{r(a,b,c_1,c_2)}]\!]_{x:\vec{y}}[e] \neq \varnothing \wedge [\![G_{r(a,b,c_1,c_2)}]\!]_{x:\vec{y}}[e] \restriction_t \in Q \\ 0 & \text{if } [\![G_{r(a,b,c_1,c_2)}]\!]_{x:\vec{y}}[e] \neq \varnothing \wedge [\![G_{r(a,b,c_1,c_2)}]\!]_{x:\vec{y}}[e] \restriction_t \notin Q \\ \uparrow & \text{if } [\![G_{r(a,b,c_1,c_2)}]\!]_{x:\vec{y}}[e] = \varnothing \end{cases}\\
&= \lambda\vec{y}.\begin{cases} 1 & \text{if } [\![G_a]\!]_{x:\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{x:\vec{y}}[e_a] \restriction_k \restriction_t \in Q \wedge x = c_1 \\ 0 & \text{if } [\![G_a]\!]_{x:\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{x:\vec{y}}[e_a] \restriction_k \restriction_t \notin Q \wedge x = c_1 \\ \uparrow & \text{if } [\![G_a]\!]_{x:\vec{y}}[e_a] = \varnothing \wedge x = c_1 \\ 1 & \text{if } [\![G_b]\!]_{x:\vec{y}}[e_b] \neq \varnothing \wedge [\![G_b]\!]_{x:\vec{y}}[e_b] \restriction_k \restriction_t \in Q \wedge x = c_2 \\ 0 & \text{if } [\![G_b]\!]_{x:\vec{y}}[e_b] \neq \varnothing \wedge [\![G_b]\!]_{x:\vec{y}}[e_b] \restriction_k \restriction_t \notin Q \wedge x = c_2 \\ \uparrow & \text{if } [\![G_b]\!]_{x:\vec{y}}[e_b] = \varnothing \wedge x = c_2 \\ \uparrow & \text{otherwise} \end{cases}\\
&= \lambda\vec{y}.\begin{cases} 1 & \text{if } [\![G_a]\!]_{x:\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{x:\vec{y}}[e_a] \restriction_t \in Q \wedge x = c_1 \\ 0 & \text{if } [\![G_a]\!]_{x:\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{x:\vec{y}}[e_a] \restriction_t \notin Q \wedge x = c_1 \\ \uparrow & \text{if } [\![G_a]\!]_{x:\vec{y}}[e_a] = \varnothing \wedge x = c_1 \\ 1 & \text{if } [\![G_b]\!]_{x:\vec{y}}[e_b] \neq \varnothing \wedge [\![G_b]\!]_{x:\vec{y}}[e_b] \restriction_t \in Q \wedge x = c_2 \\ 0 & \text{if } [\![G_b]\!]_{x:\vec{y}}[e_b] \neq \varnothing \wedge [\![G_b]\!]_{x:\vec{y}}[e_b] \restriction_t \notin Q \wedge x = c_2 \\ \uparrow & \text{if } [\![G_b]\!]_{x:\vec{y}}[e_b] = \varnothing \wedge x = c_2 \\ \uparrow & \text{otherwise} \end{cases}\\
&= \lambda\vec{y}.\begin{cases} Q^{(n)}_a(x,\vec{y}) & \text{if } x = c_1 \\ Q^{(n)}_b(x,\vec{y}) & \text{if } x = c_2 \\ \uparrow & \text{otherwise} \end{cases}\\
&= \begin{cases} \lambda\vec{y}.Q^{(n)}_a(x,\vec{y}) & \text{if } x = c_1 \\ \lambda\vec{y}.Q^{(n)}_b(x,\vec{y}) & \text{if } x = c_2 \\ \lambda\vec{y}.\uparrow & \text{otherwise} \end{cases}
\end{aligned}
$$

Therefore, r satisfies the branching property of Definition 4.6. □

Proposition 5.19. *The state semantics of BACFGs in Definition 5.16 is discharging.*

Proof. Let $\langle Q, = \rangle$ be a state semantics for a predicate $Q \subseteq \wp(\mathbb{N}^t)$ on sets of states with $t \in \mathbb{N}$ variables. Similarly to the proof of Proposition 5.18, let us define a total computable function $r : \mathbb{N} \to \mathbb{N}$ as follows: given an index a of a BACFG $G_a = (N_a, E_a, s_a, e_a)$, where k_a is the maximum variable index occurring in G_a, the function r suitably renames the nodes of G_a to avoid clashes, and adds two fresh nodes s and e whose in/outgoing edges are depicted by the BACFG in Fig. 5.

Notice that in the BACFG $G_{r(a)}$ with start and end nodes, resp., s and e_a, given $n \geq 1$, for all input $\vec{y} = (y_1, y_2, \ldots, y_n) \in \mathbb{N}^n$ and $x \in \mathbb{N}$ we have that $[\![G_{r(a)}]\!]_{x:\vec{y}}[e_a] \restriction_t = [\![G_a]\!]_{\vec{y}}[e_a] \restriction_t$: this happens because the command $[x_1, x_n] := [x_2, x_{n+1}]$ left shifts the variables and the assignment $x_{n+1} := x_{\max(n+1,k_a+1,t)+1}$ guarantees that x_{n+1} is undefined. Hence, r is a total computable function such that for all $a, x \in \mathbb{N}$:

$$
\begin{aligned}
&\lambda\vec{y}.Q^{(n+1)}_{r(a)}(x,\vec{y})\\
&= \lambda\vec{y}.\begin{cases} 1 & \text{if } [\![G_{r(a)}]\!]_{x:\vec{y}}[e_a] \neq \varnothing \wedge [\![G_{r(a)}]\!]_{x:\vec{y}}[e_a] \restriction_t \in Q \\ 0 & \text{if } [\![G_{r(a)}]\!]_{x:\vec{y}}[e_a] \neq \varnothing \wedge [\![G_{r(a)}]\!]_{x:\vec{y}}[e_a] \restriction_t \notin Q \\ \uparrow & \text{if } [\![G_{r(a)}]\!]_{x:\vec{y}}[e_a] = \varnothing \end{cases}\\
&= \lambda\vec{y}.\begin{cases} 1 & \text{if } [\![G_a]\!]_{\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{\vec{y}}[e_a] \restriction_t \in Q \\ 0 & \text{if } [\![G_a]\!]_{\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{\vec{y}}[e_a] \restriction_t \notin Q \\ \uparrow & \text{if } [\![G_a]\!]_{\vec{y}}[e_a] = \varnothing \end{cases}\\
&= \lambda\vec{y}.Q^{(n)}_a(\vec{y}).
\end{aligned}
$$

Thus, r is a function satisfying the discharging property of Definition 4.6. □

Proposition 5.20. *The state semantics of BACFGs in Definition 5.16 is ssmn.*

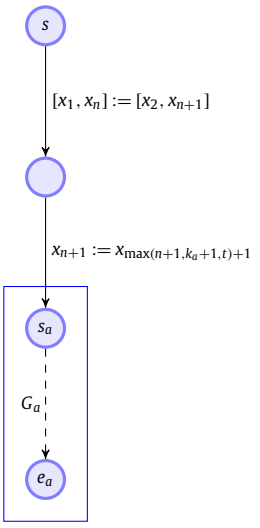

Fig. 5. The BACFG $G_{r(a)}$, output of the function r, where irrelevant node names are omitted.

Proof. Let $m, n \geq 1$ and $\langle Q, = \rangle$ be a state semantics for a given predicate $Q \subseteq \wp(\mathbb{N}^p)$ on sets of states with $p \in \mathbb{N}$ variables. We define a total computable function $s : \mathbb{N}^{m+2} \to \mathbb{N}$ which takes as input two indices a, b and a m-dimensional vector $\vec{z} \in \mathbb{N}^m$. Intuitively, to satisfy the ssmn property of Definition 3.1, the output of $s(a, b, \vec{z})$ should be a BACFG that simulates the computation of the concrete semantics $\phi_b^{(m)}$ as defined in (3). Since this latter concrete semantics is defined on URMs, it is enough to simulate the program $RM_b = (I_1, \ldots, I_t)$. To this aim, recall that the total computable function τ of Definition 5.8 transforms URMs into BACFGs having equivalent semantics. Roughly, the BACFG $G_{s(a,b,\vec{z})}$ on input $\vec{y} \in \mathbb{N}^n$ first simulates $G_{b'} = (N_{b'}, E_{b'}, q_1, q_{t+1}) \in \tau(RM_b)$ on input $\vec{z}$, and, then, simulates $G_a = (N_a, E_a, s_a, e_a)$ on input $\phi_b^{(m)}(\vec{z}) : \vec{y}$. Before going into the details, recall that, in general, URMs set unused registers to 0, so that, by a slight abuse of notation, we define the vector projection $\vec{z}\restriction_k \in \mathbb{N}^k$, for all $\vec{z} = (z_1, \ldots, z_{k'}) \in \mathbb{N}^{k'}$, as follows:

$$\vec{z}\restriction_k \triangleq \begin{cases} (z_1, \ldots, z_{k'}, 0)\restriction_k & \text{if } 0 \leq k' < k \\ (z_1, \ldots, z_k) & \text{if } k \leq k' \end{cases}$$

Let k_a and k_b be the maximum variable (or register) index occurring, resp., in G_a and RM_b. Recall the operational semantics $\Rightarrow$ for URMs of Definition 5.4 and notice that:

$$\phi_b^{(m)}(\vec{z}) = \begin{cases} z'_1 & \text{if } \exists \vec{z}' \in \mathbb{N}^{k_b}. \langle \vec{z}\restriction_{k_b}, 1\rangle \Rightarrow^* \langle \vec{z}', t+1\rangle \\ \uparrow & \text{otherwise.} \end{cases}$$

Therefore, by Propositions 5.14 and 5.15, in order to simulate $\phi_b^{(m)}(\vec{z})$ it is enough to execute $G_{b'}$ on input $\vec{z}\restriction_{k_b} : 0$. More in detail, the transform $s(a, b, \vec{z})$ will add the following commands:

1. $[x_{k_b+2}, x_{k_b+n+1}] := [x_1, x_n]$, to safely store the original input $\vec{y} \in \mathbb{N}^n$; in fact, the execution of $[\![G_{b'}]\!]_{\vec{z}\restriction_{k_b}:0}$ will use the first $k_b + 1$ variables only;
2. $[x_1, x_{\min(m,k_b)}] := [z_1, z_{\min(m,k_b)}]$, so that the first $\min(m, k_b)$ variables contain $\vec{z}\restriction_{k_b}$ except for the 0-padding;
3. $[x_{\min(m,k_b)+1}, x_{k_b+1}] := 0$, to (possibly) add the missing 0-padding;

This allows us to execute $G_{b'}$ on input $(\vec{z}\restriction_{k_b} : 0) : \vec{y}$. The next step is to execute G_a on input $\phi_b^{(m)}(\vec{z}) : \vec{y}$. Therefore, we add the following commands:

4. $[x_2, x_{n+1}] := [x_{k_b+2}, x_{k_b+n+1}]$, to restore the original input $(\vec{y})$ on the variables starting from x_2;
5. $[x_{n+2}, x_{\max(k_a,p)}] := [x_{k_b+n+2}, x_{k_b+\max(k_a,p)}]$, to ensure that all the remaining variables up to $x_{\max(k_a,p)}$ are left undefined.

Finally, the BACFG G_a is executed. The resulting BACFG $G_{s(a,b,\vec{z})}$, with start and end nodes s and e_a, resp., is described by the graph in Fig. 6. Observe that, by definition:

- if $\phi_b^{(m)}(\vec{z}) \uparrow$ then, by Proposition 5.15, $[\![G_{s(a,b,\vec{z})}]\!]_{\vec{y}}[e_a] = [\![G_{b'}]\!]_{\vec{z}}[q_{t+1}] = \varnothing$;
- otherwise, by Proposition 5.14, $[\![G_{s(a,b,\vec{z})}]\!]_{\vec{y}}[e_a]\restriction_p = [\![G_a]\!]_{\phi_b^{(m)}(\vec{z}):\vec{y}}[e_a]\restriction_p$.

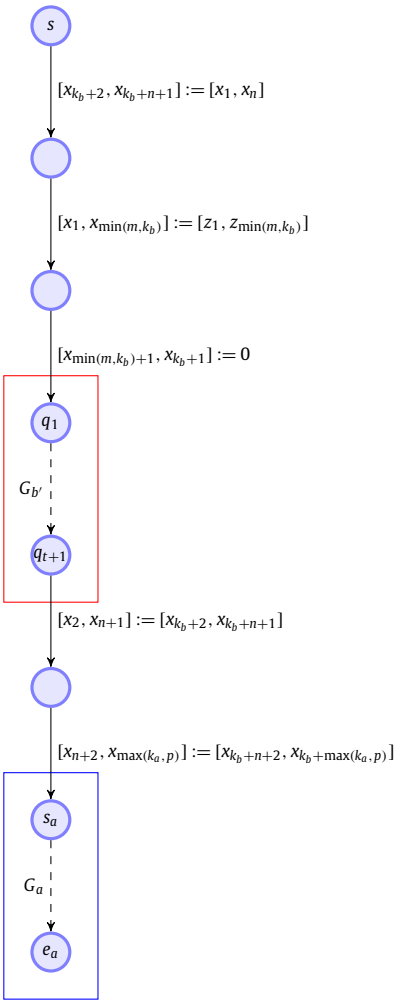

Fig. 6. The BACFG $G_{s(a,b,\vec{z})}$, output of the function s, where irrelevant node names are omitted.

Hence, we defined a total computable function s such that for all $a, b \in \mathbb{N}$ and $\vec{z} \in \mathbb{N}^m$:

$$
\begin{aligned}
&\lambda\vec{y}.Q^{(n)}_{s(a,b,\vec{z})}(\vec{y})\\
&= \lambda\vec{y}.\begin{cases} 1 & \text{if } [\![G_{s(a,b,\vec{z})}]\!]_{\vec{y}}[e_a] \neq \varnothing \wedge [\![G_{s(a,b,\vec{z})}]\!]_{\vec{y}}[e_a]\restriction_p \in Q \\ 0 & \text{if } [\![G_{s(a,b,\vec{z})}]\!]_{\vec{y}}[e_a] \neq \varnothing \wedge [\![G_{s(a,b,\vec{z})}]\!]_{\vec{y}}[e_a]\restriction_p \notin Q \\ \uparrow & \text{if } [\![G_{s(a,b,\vec{z})}]\!]_{\vec{y}}[e_a] = \varnothing \end{cases}\\
&= \lambda\vec{y}.\begin{cases} 1 & \text{if } \phi_b^{(m)}(\vec{z})\downarrow \wedge [\![G_a]\!]_{\phi_b^{(m)}(\vec{z}):\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{\phi_b^{(m)}(\vec{z}):\vec{y}}[e_a]\restriction_p \in Q \\ 0 & \text{if } \phi_b^{(m)}(\vec{z})\downarrow \wedge [\![G_a]\!]_{\phi_b^{(m)}(\vec{z}):\vec{y}}[e_a] \neq \varnothing \wedge [\![G_a]\!]_{\phi_b^{(m)}(\vec{z}):\vec{y}}[e_a]\restriction_p \notin Q \\ \uparrow & \text{otherwise} \end{cases}\\
&= \lambda\vec{y}.Q_a^{(n+1)}(\phi_b^{(m)}(\vec{z}), \vec{y})
\end{aligned}
$$

Therefore, s is a function satisfying the ssmn property of Definition 3.1. □

5.3. An application to affine program invariants

Consider a state semantics $\langle Q, = \rangle$ for some predicate $Q \subseteq \wp(\mathbb{N}^t)$. For all $n \geq 1$, let us define two sets $A^{\forall Q}$ and $A^{\exists Q}$, by distinguishing two cases depending on whether Q includes the empty set, that models nontermination, or not:

(1) If $\varnothing \notin Q$ then:
$A^{\forall Q} \triangleq \{a \in \mathbb{N} \mid \forall \vec{y}.\ Q_a^{(n)}(\vec{y}) = 1\}$;
$A^{\exists Q} \triangleq \{a \in \mathbb{N} \mid \exists \vec{y}.\ Q_a^{(n)}(\vec{y}) = 1\}$.
(2) If $\varnothing \in Q$ then:
$A^{\forall Q} \triangleq \{a \in \mathbb{N} \mid \forall \vec{y}.\ Q_a^{(n)}(\vec{y}) \in \{1, \uparrow\}\}$;
$A^{\exists Q} \triangleq \{a \in \mathbb{N} \mid \exists \vec{y}.\ Q_a^{(n)}(\vec{y}) \in \{1, \uparrow\}\}$.

Hence, $A^{\forall Q}$ ($A^{\exists Q}$) is the set of BACFGs such that Q holds at e_a for any (some) input state. It turns out that if the property Q is nontrivial then neither $A^{\forall Q}$ nor $A^{\exists Q}$ can be recursive.

Corollary 5.21. *If Q is not trivial then $A^{\forall Q}$ and $A^{\exists Q}$ are not recursive.*

Proof. Observe that $A^{\forall Q}$ is $\sim_Q$-extensional. Thus, Theorem 5.17 enables applying our intensional Theorem 4.10 to the state semantics $\langle Q, = \rangle$ to derive that $A^{\forall Q}$ is not recursive. The same argument applies to the existential version $A^{\exists Q}$. □

Thus, Corollary 5.21 means that we cannot decide if a nontrivial predicate Q holds at a given program point of a BACFG for all input states, neither whether there exists an input state that will make Q true. Let us remark that the predicates Q are arbitrary and include, but are not limited to, relational predicates between program variables such as affine equalities of Karr's abstract domain. Let us define some noteworthy examples of predicates:

(i) Given a set of affine equalities $aff = \{\vec{a}_j \cdot \vec{x} = b_j\}_{j=1}^m$, with $\vec{a}_j \in \mathbb{Z}^t$ and $b_j \in \mathbb{Z}$, $Q_{aff} \triangleq \{S \in \wp(\mathbb{N}^t) \mid \forall \vec{v} \in S. \forall j \in [1, m].\, \vec{a}_j \cdot \vec{v} = b_j\}$;
(ii) Given $i \in [1, t]$ and $c \in \mathbb{N}$, $Q_{=c} \triangleq \{S \in \wp(\mathbb{N}^t) \mid \exists \vec{v} \in S.\ v_i = c\}$;
(iii) Given a size $k \in \mathbb{N}$, $Q_{\mathrm{fin}_k} \triangleq \{S \in \wp(\mathbb{N}^t) \mid |S| = k\}$ and $Q_{\mathrm{fin}} \triangleq \cup_{k \in \mathbb{N}} Q_{\mathrm{fin}_k}$.

Therefore, it turns out that Corollary 5.21 for $A^{\forall Q_{aff}}$ entails the undecidability result of Müller-Olm and Seidl [26, Section 7] discussed at the beginning of Section 5. The predicate $Q_{=c}$ can be used to derive the undecidability of checking if some variable x_i may store a given constant c for affine programs with positive affine guards, e.g., for $c = 0$ this amounts to the undecidability of detecting division-by-zero bugs. Finally, with Q_{fin_0} we obtain the undecidability of dead code elimination, Q_{fin_1} entails the well-known undecidability of constant detection [13,31], while the existential predicate Q_{fin} encodes whether some program point may only have finitely many different states.

6. Discussion of related work

In this section we discuss more in detail the relation with some of Asperti's results [1] and with Rogers' systems of indices [35,36].

6.1. Relation with Asperti's approach

We show that our ssmn property in Definition 3.1 is a generalisation of the smn property in Asperti's approach [1], in a way that Kleene's second recursion theorem and Rice's theorem for complexity cliques in [1] arise as instances of the corresponding results in our approach. Let us first recall and elaborate on the axioms for the complexity of function composition studied by Lischke [21–23] and assumed in [1, Section 4].

Definition 6.1 *(Linear time and space complexity composition).* Consider a given concrete semantics ϕ and a Blum complexity Φ. The pair $\langle \phi, \Phi \rangle$ has the *linear time composition* property if there exists a total computable function $h : \mathbb{N}^2 \to \mathbb{N}$ such that for all $i, j \in \mathbb{N}$:

(1) $\phi_{h(i,j)} = \phi_i \circ \phi_j$,
(2) $\Phi_{h(i,j)} \in \Theta(\Phi_i \circ \phi_j + \Phi_j)$.

If (2) is replaced by

(2′) $\Phi_{h(i,j)} \in \Theta(\max\{\Phi_i \circ \phi_j, \Phi_j\})$

then $\langle \phi, \Phi \rangle$ is said to have the *linear space composition* property. □

Roughly speaking, the linear time composition property states that there exists a program $h(i, j)$ which computes the composition $\phi_i(\phi_j(x))$ in an amount of time which is asymptotically equivalent to the sum of the time needed for computing

P_j on input x, eventually producing some output $\phi_j(x)$, and the time form computing P_i on such value. On the other hand, the linear space composition property aims at modelling the needed space, so that rather than adding the complexities of P_i and P_j, their maximum is considered, since this intuitively is the maximum amount of space needed for computing a composition of programs.

By observing that $\Theta(\max\{\Phi_i \circ \phi_j, \Phi_j\}) = \Theta(\Phi_i \circ \phi_j + \Phi_j)$ we can merge the linear time and space properties of Definition 6.1 and extend them for n-ary compositions as follows.

Definition 6.2 *(Linear complexity composition).* Given a concrete semantics ϕ and a Blum complexity Φ, the pair $\langle \phi, \Phi \rangle$ has the *linear complexity composition* property if, given $n, m \geq 1$, there exists a total computable function $h : \mathbb{N}^2 \to \mathbb{N}$ such that for all $i, j \in \mathbb{N}$:

$$\phi^{(m+n)}_{h(i,j)} = \lambda\vec{x}\lambda\vec{y}.\, \phi^{(n+1)}_i(\phi^{(m)}_j(\vec{x}), \vec{y}),$$
$$\Phi^{(m+n)}_{h(i,j)} \in \Theta(\lambda\vec{x}\lambda\vec{y}.\, (\Phi^{(n+1)}_i(\phi^{(m)}_j(\vec{x}), \vec{y})) + \Phi^{(m)}_j(\vec{x}))).$$ □

We can now recall the smn property as defined in [1, Definition 11].

Definition 6.3 *(Asperti's smn property).* Given a concrete semantics ϕ, a Blum complexity Φ and $m, n \geq 1$, the pair $\langle \phi, \Phi \rangle$ has the *Asperti's smn* property if there exists a total computable function $s : \mathbb{N}^{m+1} \to \mathbb{N}$ such that $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m$:

$$\lambda\vec{y}.\phi^{(m+n)}_e(\vec{x}, \vec{y}) = \phi^{(n)}_{s(e,\vec{x})},$$
$$\lambda\vec{y}.\Phi^{(m+n)}_e(\vec{x}, \vec{y}) \in \Theta(\lambda\vec{y}.\Phi^{(n)}_{s(e,\vec{x})}(\vec{y})).$$ □

Informally, the smn property of Definition 6.3 states that the operation of fixing parameters preserves both the concrete semantics and the asymptotic complexity. Under these assumptions, we can show that Asperti's complexity clique semantics satisfies our ssmn property. The proof is a simple adaptation of the one used in Section 3 to argue that the concrete semantics of Example 2.3 is ssmn.

Lemma 6.4. *Let $\langle \pi, \equiv_\pi \rangle$ be the complexity clique semantics of Example 2.6. If $\langle \pi, \equiv_\pi \rangle$ satisfies Asperti's smn and linear complexity composition properties then $\langle \pi, \equiv_\pi \rangle$ is ssmn.*

Proof. We have to show that given $m, n \geq 1$, there exists a total computable function $s : \mathbb{N}^{m+2} \to \mathbb{N}$ such that for all $a, b \in \mathbb{N}$, $\vec{x} \in \mathbb{N}^m$:

$$\lambda\vec{y}.\pi^{(n+1)}_a(\phi^{(m)}_b(\vec{x}), \vec{y}) \equiv_\pi \pi^{(n)}_{s(a,b,\vec{x})}.$$

We have that

$$\begin{aligned}
&\lambda\vec{y}.\pi^{(n+1)}_a(\phi^{(m)}_b(\vec{x}), \vec{y}) = \\
&\quad = \lambda\vec{y}.\langle\langle \phi^{(n+1)}_a(\phi^{(m)}_b(\vec{x}), \vec{y}), \Phi^{(n+1)}_a(\phi^{(m)}_b(\vec{x}), \vec{y})\rangle\rangle \\
&\quad [\text{by definition of } \pi_a] \\
&\quad \equiv_\pi \lambda\vec{y}.\langle\langle \phi^{(m+n)}_{h(a,b)}(\vec{x}, \vec{y}), \Phi^{(m+n)}_{h(a,b)}(\vec{x}, \vec{y})\rangle\rangle \\
&\quad [\text{with } h : \mathbb{N}^2 \to \mathbb{N} \text{ total computable, by linear complexity composition}] \\
&\quad \equiv_\pi \lambda\vec{y}.\langle\langle \phi^{(n)}_{s'(h(a,b),\vec{x})}(\vec{y}), \Phi^{(n)}_{s'(h(a,b),\vec{x})}(\vec{y})\rangle\rangle \\
&\quad [\text{with } s' : \mathbb{N}^{m+1} \to \mathbb{N} \text{ total computable, by Asperti's smn property}] \\
&\quad = \langle\langle \phi^{(n)}_{s'(h(a,b),\vec{x})}, \Phi^{(n)}_{s'(h(a,b),\vec{x})}\rangle\rangle = \pi^{(n)}_{s'(h(a,b),\vec{x})}
\end{aligned}$$

The desired function $s : \mathbb{N}^{m+2} \to \mathbb{N}$ can therefore be defined as $s(a, b, \vec{x}) \triangleq s'(h(a, b), \vec{x})$. Note that s is total computable since h and s' are so. □

This result, together with the observation that the notion of fairness in Definition 3.2 instantiated to the complexity clique semantics is exactly that of [1, Definition 26], allows us to retrieve Kleene's second recursion theorem and Rice's theorem for complexity cliques in [1] as instances of our corresponding results in Section 4.1.

6.2. Relation with systems of indices

As mentioned in Section 2, our definition of abstract semantics resembles the acceptable systems of indices [28, Definition II.5.1] or numberings [36, Exercise 2-10], firstly studied by Rogers [35]. In this section we discuss how such notions compare.

Definition 6.5 *(System of indices [28, Definition II.5.1]).* A *system of indices* is a family of functions $\{\psi^n\}_{n\in\mathbb{N}}$ such that each $\psi^n : \mathbb{N} \to \mathcal{C}_n$ is a surjective map that associates program indices to n-ary partial recursive functions.

- $\{\psi^n\}_{n\in\mathbb{N}}$ has the *parametrization* (or smn) property if for every $m, n \in \mathbb{N}$ there is a total computable function $s : \mathbb{N}^{m+1} \to \mathbb{N}$ such that $\forall e \in \mathbb{N}, \vec{x} \in \mathbb{N}^m$:

$$\lambda\vec{y}.\psi_e^{m+n}(\vec{x}, \vec{y}) = \psi^n_{s(e,\vec{x})}.$$

- $\{\psi^n\}_{n\in\mathbb{N}}$ has the *enumeration* property if for every $n \in \mathbb{N}$ there exists $u \in \mathbb{N}$ such that for all and $e \in \mathbb{N}$ and $\vec{y} \in \mathbb{N}^n$:

$$\psi_e^n = \lambda\vec{y}.\psi_u^{n+1}(e, \vec{y}).$$ □

Any standard Gödel numbering associating a program with the function it computes is a system of indices with the parametrization and enumeration properties. Moreover, exactly as we did in Example 2.3, any system of indices $\{\psi^n\}_{n\in\mathbb{N}}$ can be viewed as an abstract semantics $\langle \pi, = \rangle$ with $\pi_n^a \triangleq \psi_a^n$. In this context, the enumeration and parametrization properties correspond to our fairness and ssmn conditions: fairness is exactly enumeration while ssmn follows from parametrization and enumeration, as discussed in Section 3 for the concrete semantics (cf. Example 2.3).

A system of indices is defined to be *acceptable* if it allows to get back and forth with a given system of indices satisfying the parametrization and enumeration properties through a pair of total computable functions.

Definition 6.6 *(Acceptable system of indices [35, Definition 4]).* Let $\{\varphi^n\}_{n\in\mathbb{N}}$ be a given system of indices with the parametrization and enumeration properties. A system of indices $\{\psi^n\}_{n\in\mathbb{N}}$ is *acceptable* if there exist two total computable functions $f, g : \mathbb{N} \to \mathbb{N}$ such that for all $a, n \in \mathbb{N}$:

$$\psi_a^n = \varphi^n_{f(a)} \quad \text{and} \quad \varphi_a^n = \psi^n_{g(a)}.$$ □

As shown in [28, Proposition II.5.3], it turns out that a system of indices is acceptable if and only if it satisfies both enumeration and parametrization (a proof of this characterization was first given by Rogers [35, Section 2]). Consequently, an acceptable system of indices $\{\psi^n\}_{n\in\mathbb{N}}$ can be viewed as an abstract semantic $\langle \pi, = \rangle$, where $\pi_a^n = \psi_a^n$, which, by this characterization of acceptability, is ssmn and fair, and therefore, by Theorem 4.1 it enjoys Kleene's second recursion theorem, as already known from [28, Corollary II.5.4].

Under this perspective, a generic abstract semantics according to Definition 2.2 can be viewed as a proper generalisation of the notion of acceptable system of indices, in the sense that the latter merely encodes a change of program numbering and does not allow to take into account an actual abstraction of the concrete input/output behaviour of programs.

7. Conclusion and future work

This work generalises some traditional extensional results of computability theory, notably Kleene's second recursion theorem and Rice's theorem, to intensional abstract program semantics that include the complexity cliques investigated by Asperti [1]. Our approach was also inspired by Moyen and Simonsen [25] and relies on weakening the classical definition of extensional program property to a notion of partial extensionality w.r.t. abstract program semantics that satisfy some structural conditions. As an application, after showing the Turing completeness of the class of affine control flow graphs with positive affine guards (BACFGs), we strengthened and generalised a result by Müller-Olm and Seidl [26] proving that for affine programs with positive affine guards it is undecidable whether an affine relation holds at a given program point. Our results also shed further light on the claim that these undecidability results hinge on the Turing completeness of the underlying computational model, as argued in [25].

It is worth observing that our approach, similarly to those of [1,25], relies on the possibility of constructing or transforming programs while preserving the abstract semantics as required, e.g., by the fairness property (cf. Definition 3.2) or the branching property (cf. Definition 4.6). These requirements can be hard to meet when the semantics of interest is too concrete. For instance, properties of programs related to the exact complexity (e.g., the set of programs p terminating in exactly $\Phi_p(n) = n^2 + 2$ steps), can hardly be cast into our framework, even though we do not provide a formal impossibility result.

As future work, an interesting direction concerns the chance of leveraging our framework to achieve new (or already known) undecidability results of static program analyses. Some viable candidates appear to be the undecidability results already mentioned in the introduction, such as the undecidability of flow-sensitive alias analysis [6,20], points-to analysis [6], and context-sensitive data-dependence analysis [32].

A further stimulating task would be to investigate intensional generalisations of Rice-Shapiro's theorem that fit our framework based on abstract semantics. This appears to be a nontrivial challenge. Generalisations of Rice-Shapiro's theorem have been given in [1, Section 5] and [25, Section 5.1]. A generalisation in the vein of the approach in [1] seems to be viable, but would require structural assumptions on abstract program semantics that, while natural in [1] whose focus is on complexity properties, would be artificial for abstract program semantics and would limit a general applicability. A further stimulating research topic is to apply our approach to abstract semantics as defined by abstract interpretation of programs [9], in particular for investigating the relationship with the notion of abstract extensionality studied by Bruni et al. [5]. Finally, while our framework relies on the assumption of an underlying Turing complete computational model, in a different direction, one could try to consider intensional properties for classes of programs indexing subrecursive functions (e.g., primitive recursive functions), whose extensional properties have been already studied (see, e.g., [14,19]). Despite the fact that we suppose that our approach will fall short on these program classes, as one cannot expect to have a universal program inside the class itself or the validity of Kleene's second recursion theorem, we think that this represents an intriguing research challenge.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We are grateful to Roberto Giacobazzi for thorough discussions and comments.

Paolo Baldan and Francesco Ranzato have been partially funded by *University of Padova*, under the SID 2018 project "Analysis of STatic Analyses (ASTA)", and by *Italian Ministry of University and Research*, under the PRIN 2017 project no. 201784YSZ5 "AnalysiS of PRogram Analyses (ASPRA)". Francesco Ranzato has been partially funded by *Facebook Research*, under a "Probability and Programming Research Award", and by an *Amazon Research Award* for "AWS Automated Reasoning".

References

[1] Andrea Asperti, The intensional content of Rice's theorem, in: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, ACM, New York, NY, USA, 2008, pp. 113–119.
[2] Paolo Baldan, Francesco Ranzato, Linpeng Zhang, A Rice's theorem for abstract semantics, in: Nikhil Bansal, Emanuela Merelli, James Worrell (Eds.), Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, in: LIPIcs, vol. 198, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 117.
[3] Manuel Blum, A machine-independent theory of the complexity of recursive functions, J. ACM 14 (2) (April 1967) 322–336.
[4] Manuel Blum, On effective procedures for speeding up algorithms, J. ACM 18 (2) (April 1971) 290–305.
[5] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, Dusko Pavlovic, Abstract extensionality: on the properties of incomplete abstract interpretations, in: Proceedings of the ACM on Programming Languages (POPL 2020), 2020, 4.
[6] Venkatesan T. Chakaravarthy, New results on the computability and complexity of points-to analysis, in: Proceedings of the ACM on Programming Languages (POPL 2003), ACM, 2003, pp. 115–125.
[7] Arthur Charlesworth, The undecidability of associativity and commutativity analysis, ACM Trans. Program. Lang. Syst. 24 (5) (2002) 554–565.
[8] Michael R. Clarkson, Fred B. Schneider, Hyperproperties, J. Comput. Secur. 18 (6) (2010) 1157–1210.
[9] Patrick Cousot, Principles of Abstract Interpretation, MIT Press, 2021.
[10] Patrick Cousot, Radhia Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. 4th ACM Symp. on Principles of Programming Languages (POPL 1977), ACM, 1977.
[11] Patrick Cousot, Roberto Giacobazzi, Francesco Ranzato, Program analysis is harder than verification: a computability perspective, in: Proc. Int. Conf. on Computer Aided Verification (CAV 2018), Springer, 2018, pp. 75–95.
[12] Nigel Cutland, Computability: An Introduction to Recursive Function Theory, Cambridge University Press, 1980.
[13] Matthew S. Hecht, Flow Analysis of Computer Programs, Elsevier, 1977.
[14] Mathieu Hoyrup, The decidable properties of subrecursive functions, in: Proc. Int. Coll. on Automata, Languages and Programming (ICALP 2016), in: LIPIcs, vol. 55, 2016, 108.
[15] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, James Worrell, Polynomial invariants for affine programs, in: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2018), ACM, New York, NY, USA, 2018, pp. 530–539.
[16] John B. Kam, Jeffrey D. Ullman, Monotone data flow analysis frameworks, Acta Inform. 7 (1977) 305–317.
[17] Michael Karr, Affine relationships among variables of a program, Acta Inform. 6 (1976) 133–151.
[18] Stephen C. Kleene, On notation for ordinal numbers, J. Symb. Log. 3 (4) (1938) 150–155.
[19] Dexter Kozen, Indexings of subrecursive classes, Theor. Comput. Sci. 11 (1980) 277–301.
[20] William Landi, Undecidability of static analysis, ACM Lett. Program. Lang. Syst. 1 (4) (dec 1992) 323–337.
[21] Gerhard Lischke, Über die Erfüllung gewisser Erhaltungssätze durch Kompliziertheitsmasse, Math. Log. Q. 21 (1) (1975) 159–166.
[22] Gerhard Lischke, Natürliche Kompliziertheitsmasse und Erhaltungssätze I, Math. Log. Q. 22 (1) (1976) 413–418.
[23] Gerhard Lischke, Natürliche Kompliziertheitsmasse und Erhaltungssätze II, Math. Log. Q. 23 (13–15) (1977) 193–200.
[24] Antoine Miné, Tutorial on static inference of numeric invariants by abstract interpretation, Found. Trends Program. Lang. 4 (3–4) (2017) 120–372.
[25] Jean-Yves Moyen, Jakob Grue Simonsen, More intensional versions of Rice's theorem, in: Proc. Computability in Europe (CIE 2019), Computing with Foresight and Industry, Springer, 2019, pp. 217–229.
[26] Markus Müller-Olm, Helmut Seidl, A note on Karr's algorithm, in: Proc. Int. Coll. on Automata, Languages and Programming (ICALP 2004), Springer, 2004, pp. 1016–1028.
[27] Markus Müller-Olm, Helmut Seidl, Precise interprocedural analysis through linear algebra, in: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004), ACM, New York, NY, USA, 2004, pp. 330–341.

[28] Piergiorgio Odifreddi, Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers, sole distributors for the USA and Canada, Elsevier Science Pub. Co., 1989.
[29] Christos H. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, MA, 1994.
[30] Francesco Ranzato, Decidability and synthesis of abstract inductive invariants, in: Proc. 31st International Conference on Concurrency Theory (CONCUR 2020), in: LIPIcs, vol. 171, 2020, 30.
[31] John H. Reif, Harry R. Lewis, Symbolic evaluation and the global value graph, in: Proc. 4th ACM Symp. on Principles of Programming Languages (POPL 1977), ACM, 1977, pp. 104–118.
[32] Thomas W. Reps, Undecidability of context-sensitive data-independence analysis, ACM Trans. Program. Lang. Syst. 22 (1) (2000) 162–186.
[33] G. Rice Henry, Classes of recursively enumerable sets and their decision problems, Trans. Am. Math. Soc. 74 (1953) 358–366.
[34] Xavier Rival, Kwangkeun Yi, Introduction to Static Analysis – An Abstract Interpretation Perspective, MIT Press, 2020.
[35] Hartley Rogers, Gödel numberings of partial recursive functions, J. Symb. Log. 23 (3) (1958) 331–341.
[36] Hartley Rogers, Theory of Recursive Functions and Effective Computability, Higher Mathematics Series, McGraw-Hill, 1967.
[37] Raymond M. Smullyan, Undecidability and recursive inseparability, Math. Log. Q. 4 (7–11) (1958) 143–147.