Abstract Extensionality: On the Properties of Incomplete Abstract Interpretations

ROBERTO BRUNI, University of Pisa, Italy
ROBERTO GIACOBAZZI, University of Verona, Italy and IMDEA Software Institute, Spain
ROBERTA GORI, University of Pisa, Italy
ISABEL GARCIA-CONTRERAS, IMDEA Software Institute and Univ. Politécnica de Madrid, Spain
DUSKO PAVLOVIC, University of Hawaii, USA

In this paper we generalise the notion of extensional (functional) equivalence of programs to *abstract equivalences* induced by *abstract interpretations*. The standard notion of extensional equivalence is recovered as the special case, induced by the concrete interpretation. Some properties of the extensional equivalence, such as the one spelled out in Rice's theorem, lift to the abstract equivalences in suitably generalised forms. On the other hand, the generalised framework gives rise to interesting and important new properties, and allows refined, non-extensional analyses. In particular, since programs turn out to be extensionally equivalent if and *only if* they are equivalent just for the concrete interpretation, it follows that any non-trivial abstract interpretation uncovers some intensional aspect of programs. This striking result is also effective, in the sense that it allows constructing, for any non-trivial abstraction, a pair of programs that are extensionally equivalent, but have different abstract semantics. The construction is based on the fact that abstract interpretations are always sound, but that they can be made incomplete through suitable code transformations. To construct these transformations, we introduce a novel technique for building *incompleteness cliques* of extensionally equivalent yet abstractly distinguishable programs: They are built together with abstract interpretations that produce false alarms. While programs are forced into incompleteness cliques using both control-flow and data-flow transformations, the main result follows from limitations of data-flow transformations with respect to control-flow ones. A further consequence is that the class of incomplete programs for a non-trivial abstraction is Turing complete. The obtained results also shed a new light on the relation between the techniques of code obfuscation and the precision in program analysis.

CCS Concepts: • **Theory of computation → Program analysis**; **Recursive functions**; **Abstraction**; **Invariants**; *Denotational semantics*; Semantics and reasoning.

Additional Key Words and Phrases: Abstract Interpretation, Extensionality, Intensionality, Obfuscation.

ACM Reference Format:
Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. 2020. Abstract Extensionality: On the Properties of Incomplete Abstract Interpretations. *Proc. ACM Program. Lang.* 4, POPL, Article 28 (January 2020), 28 pages. https://doi.org/10.1145/3371096

Authors' addresses: Roberto Bruni, University of Pisa, Italy, bruni@di.unipi.it; Roberto Giacobazzi, University of Verona, Italy , IMDEA Software Institute, Spain, roberto.giacobazzi@univr.it; Roberta Gori, University of Pisa, Italy, gori@di.unipi.it; Isabel Garcia-Contreras, IMDEA Software Institute, isabel.garcia@imdea.org, Univ. Politécnica de Madrid, Spain; Dusko Pavlovic, University of Hawaii, USA, dusko@hawaii.edu.

2475-1421/2020/1-ART28
https://doi.org/10.1145/3371096

1 INTRODUCTION

Motivation. The main tradition of theory of computation [Rogers 1992] has been concerned with extensional aspects of computation, i.e., with properties of programs reduced to the functions that they implement: Two programs (often represented as indices in an enumeration of all programs) are extensionally equivalent if they produce the same outputs on the same inputs. Theory of computation thus studies extensional properties of programs, which cannot tell apart any pair of extensionally equivalent programs. In other words, it studies properties of computable functions. Much less is known about the intensional properties of programs, which distinguish different algorithms for computing the same function, different descriptions in programming languages, or different executions [Abramsky 2014]. The intensional side of computation includes everything that happens after the input data are read, and before the output data are written: All states and state changes, how many steps are made, or how much memory is used. But besides the objective properties such as program complexity, the intensional properties include relations of programs with programmers, such as understandability and quality; or with other programs, such as optimising compilers, static analysers, software debuggers; or with abstract interpreters. In any case, intensional properties are the central concern of software design and maintenance, and they lie at the heart of the process of software development in general.

The problem. Program analysis is concerned with intensional properties not just because semantically equivalent programs may exhibit different properties, but also because semantically different programs may appear identical when analysed. This familiar phenomenon [Cousot and Cousot 2014; Laviron and Logozzo 2009] can be overcome by increasing the precision of program analysis; but the incurred costs make it into one of the main challenges in program analysis, and into one of the main obstacles to general-purpose program analysis.

Abstract interpretation [Cousot and Cousot 1977, 1979] has been developed as an effective method for constructing sound-by-construction program analysis tools. The underlying idea of program analysis by abstract interpretation is strikingly simple: *Extracting a program property means approximating its semantics.* This idea is implemented by interpreting program structures in an abstract domain. Such abstract interpretations usually approximate some *extensional* properties of the computations. Static program analysis even imposes termination through its fixpoint extrapolation techniques. In this way, abstract interpretation thus reduces reasoning about programs to total computable functions, in contrast with the traditional, concrete interpretation, which associates with each program the *partial* function that it computes, and reason about such functions[1].

We illustrate the sensitivity of program analysis to code structure with an example. Consider the abstract domain of intervals Int. Each element in the domain corresponds to an interval $[a, b]$, where $a \in \mathbb{Z} \cup \{-\infty\}$ and $b \in \mathbb{Z} \cup \{+\infty\}$ and $a \leq b$. This domain is an abstraction of properties of integer valued variables; i.e., Int abstracts the set $\wp(\mathbb{Z})$ of sets of integers. Consider the programs P and Q in Figure 1, where the symbol @ is used to mark some program points that interest us. Suppose that we want to prove the Hoare triples $\{x \in \mathbb{Z}\}P\{x = 0\}$ and $\{x \in \mathbb{Z}\}Q\{x = 0\}$. It is easy to see that the programs are extensionally equivalent, and that they always output $x = 0$. However, interpreted in Int, the programs P and Q exhibit different abstract semantics. The reason is that abstract interpreters approximate the sets of values that program variables may take during the concrete executions. Abstract interpretations in the Int domain approximate these sets of values with intervals. At the point @ of the program Q, the values will thus be approximated by the increasing sequence of intervals: $[10, 10] \subset [8, 10] \subset [6, 10] \subset [4, 10] \subset [2, 10] \subset [0, 10]$. When the

[1]Here the tacit assumption is that the program is deterministic and effect-free. Otherwise, the reasoning is reduced to more general families of functions, often captured by computational monads, which are in any case extensional objects.

```
P :                          Q :                          R :
x := 10;                     x := 10;                     x := −9;
while @ (x > 0)              while @ (x > 1)              while @ (x < 0)
  do x := x − 1;               do x := x − 2;               do x := x + 2;
```

Fig. 1. Some simple programs.

loop condition becomes false, and the execution exits the loop, the postcondition $x \leq 1 \wedge x \in [0, 10]$ is reached, which is true if and only if $x \in [0, 1]$. Intepreting Q in Int thus only allows proving $\{x \in \mathbb{Z}\}Q\{x \in [0, 1]\}$, whereas it is easy to see that interpreting P in Int we can prove the stronger postcondition $\{x \in \mathbb{Z}\}P\{x = 0\}$. This shows that the integer interval domain Int is incomplete for abstract interpretation of the program Q, while it remains complete for the program P.

It is well known that the abstract semantics of *complete* abstract interpretations produces the same property approximations as the direct abstraction of the concrete semantics [Cousot and Cousot 1979; Giacobazzi et al. 2000]. On the other hand, *incomplete* abstract interpretations give rise to weaker properties than those encountered under direct inspections of effective computations of the program, as in the case of program Q above. This means that, for instance in debugging, abstract interpretations yield false alarms. And this is not a rare phenomenon, since incompleteness is in program analysis more common than completeness [Giacobazzi et al. 2015].

For a different example, consider the programs Q and R in Figure 1. These two satisfy the Hoare triples $\{x \in \mathbb{Z}\}Q\{x = 0\}$ and $\{x \in \mathbb{Z}\}R\{x = 1\}$, so their extensional semantics is different; yet their abstract semantics in Int are here the same: $\{x \in \mathbb{Z}\}R\{x \in [0, 1]\}$.

This striking disparity between abstract and concrete semantics gives rise to a basic question: *Can the extensional program equivalence be extended based on abstract semantics?* If this is possible, then it may be of great interest to explore the properties of this new notion of program equivalence.

Contribution. In this paper we explore abstract interpretation from the perspective of computable functions. We introduce a new family $\Pi^{\mathcal{A}}$ of index sets for partial recursive functions which are parameterised by a given abstraction $\mathcal{A}$. The indices represent programs, assumed to be enumerated in some way, as it is usually done in theory of computation. The family of index sets $\Pi^{\mathcal{A}}$ is obtained by replacing the usual concrete program equivalence with an equivalence based on the abstract semantics induced by $\mathcal{A}$. The results are properties of programs captured by what an abstract interpreter computes rather than what the actual programs compute. We call these properties *abstract program properties*. The paper has three main contributions.

(1) We prove that abstract program properties are the union of equivalence classes, where two programs are considered equivalent if they have the same abstract semantics. These classes are not in general extensional, i.e., they are not closed w.r.t. standard semantic equivalence of programs. Indeed, we prove that these equivalence classes are extensional if and only if the abstraction is trivial (see Theorem 28), i.e., it is the identity abstraction (no approximation) making the abstract semantics coincide with the concrete one, or it is the abstraction that is unable to distinguish any pair of programs (the greatest possible approximation).

(2) We introduce two classes of programs called respectively completeness and incompleteness cliques. The completeness (resp., incompleteness) clique $\mathbb{C}(P, \mathcal{A})$ (resp., $\overline{\mathbb{C}}(P, \mathcal{A})$) of a program P and an abstraction $\mathcal{A}$ is the set of all programs that are semantically equivalent to P and for which $\mathcal{A}$ is complete (resp., incomplete). These two classes have interesting properties: $\mathbb{C}(P, \mathcal{A})$ represents the class of all variants of P for which the analysis based on $\mathcal{A}$ is precise (no false alarms)

while $\overline{\mathbb{C}}(P, \mathcal{A})$ is instead the class of all variants of P for which $\mathcal{A}$ is imprecise. The connections between $\mathbb{C}(P, \mathcal{A})$ and $\overline{\mathbb{C}}(P, \mathcal{A})$ are the focus of the rest of the paper. In particular:

(2a) We prove that there is an infinity of non-trivial abstractions for which the systematic computable removal of false alarms for all programs is impossible, namely under mild assumptions on the abstraction $\mathcal{A}$ there is no many-to-one reducibility of $\overline{\mathbb{C}}(P, \mathcal{A})$ to $\mathbb{C}(P, \mathcal{A})$ (see Theorem 16).

(2b) For a wide class of abstract domains, called variable finite (see Definition 9), we provide a systematic reduction of $\mathbb{C}(P, \mathcal{A})$ into $\overline{\mathbb{C}}(P, \mathcal{A})$, namely an effective transformation that maps any program P into another program $\tau(P)$ which is equivalent to P with respect to the concrete semantics but that is distinguished from P by the abstraction $\mathcal{A}$ (see Corollary 24). In this case, although τ preserves the concrete semantics, the abstract semantics distinguishes any complete program P from $\tau(P)$, because $\tau(P)$ produces false alarms.

(2c) We observe that the above transformation can always be implemented by a control-flow transformation, i.e, a semantically equivalent morph of the control-flow graph of the program, while it cannot be in general obtained by a pure homomorphic data-flow transformation (see Theorem 33), i.e., by a pure change in data structures manipulated by expressions. These results shed a new light in the relation between the precision of abstract interpretation and code obfuscation.

(3) As a consequence of our construction we prove that the class of all programs that are incomplete for a given non-trivial variable finite abstraction $\mathcal{A}$ is Turing complete.

Our results give new insights about the impossibility to automatically remove false alarms from program analyses. In particular we expose the typical structure of incomplete programs. These include predicates, i.e., Boolean-valued functions, that the abstract interpreter fails to evaluate in a precise enough way, producing an undetermined result. The structure of these predicates turns out to arise from the structure of the abstract domain from which the abstract interpreter is designed. This result, together with the proof system in [Giacobazzi et al. 2015], may suggest a practical path towards specific strategies for code refactoring with the goal to improve the precision of program analysis. Moreover, the Turing completeness of the class of all programs that are incomplete for a given non-trivial variable finite abstraction $\mathcal{A}$, suggests specific code protecting transformations against reverse engineering. Given any (non-trivial) abstract domain $\mathcal{A}$, every computable function can be implemented by a program that is incomplete on $\mathcal{A}$, i.e., every computable function can be intentionally obfuscated against a given non-trivial program analysis. This establishes a *possibility* result concerning code obfuscation when the attack model is any non-trivial abstract interpreter. Several constructive obfuscation strategies are proposed in this paper.

Structure of the paper. In Section 2 we discuss related works and in Section 3 we introduce the basic set-theoretic notation for ordered structures, functions, and programs. In Section 4 we recall the basic concepts of Galois insertion-based abstract interpretations with particular emphasis on the notion of complete and incomplete abstract semantics. In Section 5 we introduce the notion of abstract extensional property of programs for a given abstract interpretation and prove the relation between abstract extensionality and standard (later called Rice) extensionality. The notions of completeness and incompleteness cliques are introduced in Section 6, while in Section 7 we define a general control-flow transformation, applicable to a large class of abstract domains, called variable finite, that allows us to transform any program in a completeness clique into a semantically equivalent one in its incompleteness clique. In Section 8 we show that for non-trivial abstract domains we can always build two programs that are semantically equivalent but different with respect to their abstract semantics, therefore proving that any abstract semantics is Rice extensional if and only if the abstraction is trivial. Section 9 defines a general data-flow transformation framework for introducing incompleteness and shows the limitations of data-flow techniques with respect to

control-flow transformations in mapping programs in a completeness cliques into an equivalent one in its incompleteness cliques. Some final remarks are in Section 10.

2 RELATED WORK

Interesting results concerning the possibility of widening standard recursion theory towards intensional aspects of code have been produced in the last decade. These include, among others, the results in computational complexity via programming languages in [Ben-Amram and Jones 2000], the intentional contents of Rice theorem in the case of Blum's complexity in [Asperti 2008], the semantics of intensionality [Kavvos 2017], the field of implicit computational complexity (see [Dal Lago 2011] for a short survey), until the results in *Analyzing Program Analyses* in [Giacobazzi et al. 2015] and the comparison of the hardness of program analysis with respect to program verification in [Cousot et al. 2018].

The very first recursive theoretic account of complete abstract interpretations is in [Giacobazzi et al. 2015], where the notion of completeness class for an abstract interpretation was introduced. Given an abstract interpretation $\mathcal{A}$, the completeness class of $\mathcal{A}$, denoted $\mathbb{C}(\mathcal{A})$, is the set of all programs for which $\mathcal{A}$ is complete, i.e., for which $\mathcal{A}$ produces no false alarms. The complement set $\overline{\mathbb{C}(\mathcal{A})}$ is instead the set of programs that are incomplete for $\mathcal{A}$. These two classes have many interesting properties but they fail to capture the extensional behaviour of programs, namely, within $\mathbb{C}(\mathcal{A})$ and $\overline{\mathbb{C}(\mathcal{A})}$ there may coexist programs that are semantically different, just because they have complete or incomplete abstract semantics respectively.

We consider here the notion of completeness clique which is a transposition of the complexity cliques introduced in [Asperti 2008] for Blum's complexity to abstract interpretations. We prove some recursion-theoretic properties of completeness and, in particular, incompleteness cliques. For the latter case we provide effective transformations that allow to transform any program for which a given abstract interpretation is complete (it belongs to a completeness clique) into an extensionally equivalent one (i.e., preserving its concrete semantics) for which the same abstract interpreter produces false alarms, i.e., it is in the corresponding incompleteness clique. As a consequence this allows us to prove properties and limitations of control and data-flow semantics-preserving obfuscating transformations [Collberg and Nagra 2009]. In particular we can go beyond [Drape et al. 2007; Majumdar et al. 2006] and prove that semantics-preserving transformations cannot be extended to all abstract domains when applied for data-flow obfuscations, hence formally justifying the need of data-type complication in data-flow obfuscation of programs, as postulated in [Drape et al. 2007], while they can always be implemented when dealing with control-flow obfuscations.

The systematic construction of generic abstraction-agnostic semantics-preserving obfuscators for a given abstract interpreter or model checker was also considered in [Dalla Preda and Giacobazzi 2009; Giacobazzi 2008; Giacobazzi et al. 2012] and more recently in [Bruni et al. 2018]. Here we generalise those approaches and consider the more general problem of correlating the completeness and incompleteness cliques, therefore providing a more general setting to reason about the precision of an abstract interpretation. In [Giacobazzi and Mastroeni 2012] and [Giacobazzi and Mastroeni 2016] the authors introduced model deformations making a semantics respectively complete and incomplete, with the aim of giving a measure of the strength of obfuscation. None of these approaches however considered extensional equivalence as requirement, i.e., the transformed code may not have the same concrete semantics as the source.

The possibility of establishing a sound representation of the extensional equivalence of two programs by abstract interpretation was studied in [Partush and Yahav 2013]. The authors introduced the notion of *correlating program*, i.e., a new program $P \bowtie Q$ obtained as the combination of P and Q, together with a *correlating abstract domain* such that the analysis of $P \bowtie Q$ in this abstract

domain gives information about the extensional equivalence of P and Q. In the light of our results, it is therefore always possible to morph $P \bowtie Q$ in order to break the completeness of the proof of equivalence, and hence to foil the equivalence analysis. We believe that this has reflections on the structure of both P and Q, i.e., when P and Q have some specific shape, and include the predicates that expose the incompleteness of the analysis, the differencing analysis is imprecise. The early detection of these predicates may help in driving the use of the correlating program method in [Partush and Yahav 2013] to achieve a more precise analysis.

3 BASIC NOTATION

Sets and order. Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \setminus T$ the set-difference between S and T, with $S \subset T$ strict inclusion, with $S \subseteq T$ inclusion, with $\overline{S}$ set complementation, and with $|S|$ the cardinality of S. A set S is finite if $|S| < \omega$. A set L with ordering relation $\leq$ is a poset and it is denoted as $\langle L, \leq \rangle$. A poset $\langle L, \leq \rangle$ is a lattice, denoted $\langle L, \leq, \vee, \wedge, \top, \bot \rangle$, if for all $x, y \in L$ we have that the least upper bound (lub) $x \vee y$, the greatest lower bound (glb) $x \wedge y$, the greatest element (top) $\top$, and the least element (bottom) $\bot$ belong to L. It is complete when for every $X \subseteq L$ we have that $\bigvee X, \bigwedge X \in L$. We sometimes use subscripts, like in $\bot_L$, $\top_L$, or $\bigvee_L X$ to disambiguate the underlying lattice when it is not evident from the context.

Functions. Given $f : S \to T$ and $g : T \to Q$ we denote with $g \circ f : S \to Q$ their composition, i.e., $(g \circ f)x = g(f(x))$. Function application $f(x)$ is sometimes denoted fx. $\mathtt{id}_S : S \to S$ is the identity function over S, and we omit the subscript S when clear from the context. $f : L \to D$ on complete lattices is *additive (resp. co-additive)* if for any $Y \subseteq L$, $f(\bigvee_L Y) = \bigvee_D f(Y)$ (resp. $f(\bigwedge_L Y) = \bigwedge_D f(Y)$). Continuity holds when f preserves *lubs* of chains. If $f : L \to D$ we overload the notation by writing $f : \wp(L) \to \wp(D)$ for the additive extension of f to sets of values (i.e., for any $S \in \wp(L)$ we have $f(S) = \{f(v) \mid v \in S\}$). For a continuous function f: $lfp(f) = \bigwedge \left\{ x \middle| \; x = f(x) \; \right\} = \bigvee_{n \in \mathbb{N}} f^n(\bot)$ where $f^0(\bot) = \bot$ and $f^{n+1}(\bot) = f(f^n(\bot))$. Note that additive functions are continuous.

Programs. We will consider a basic deterministic while-language Imp with arithmetic and Boolean expressions, as defined, e.g., in [Winskel 1993], whose syntax is as follows:

$$\begin{aligned} \mathtt{AExp} \ni a &::= v \in \mathbb{Z} \mid x \in \mathit{Var} \mid f(\widetilde{a}) \\ \mathtt{BExp} \ni b &::= \mathbf{tt} \mid \mathbf{ff} \mid a = a \mid a > a \mid b \wedge b \mid \neg b \\ \mathtt{Imp} \ni P &::= \mathbf{skip} \mid x := a \mid P; P \mid \mathbf{if}\ b\ \mathbf{then}\ P\ \mathbf{else}\ P \mid \mathbf{while}\ b\ \mathbf{do}\ P \end{aligned}$$

where $\mathbb{Z}$ denotes the set of integers, *Var* is a denumerable set of program variables, f ranges over total recursive functions, and $\widetilde{a}$ denotes a list of arithmetic expressions. As usual we denote by $var(P)$ the finite set of variables that occur in the program P.

4 ABSTRACT INTERPRETATION

In the following we consider program analysers as specified by Galois insertion-based abstract interpreters (see [Cousot and Cousot 1977, 1979] for details). Our results are based on some hypotheses that are often left implicit in the context of program analysis. Here, in order to emphasise their role, we prefer to name them explicitly as [H1], [H2] and [H3] in the text that follows.

4.1 Abstraction

Abstract domains. Let C and A be complete lattices, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ forms an *adjunction* or a *Galois connection* between C and A if for any $c \in C$ and $a \in A$ we have $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. The function α (resp. γ) is the *left-adjoint* (resp. *right-adjoint*) to

γ (resp. α) and it is additive (resp. co-additive). A Galois connection such that $\alpha \circ \gamma = \mathtt{id}_A$ is called a *Galois insertion*.

Given a Galois insertion $\alpha : C \longrightarrow A$ and $\gamma : A \longrightarrow C$, we call $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$ an *abstract domain*, with join operator $\sqcup$. We say that $\mathcal{A}$ is *strict* if $\gamma(\bot_A) = \bot_C$. We say that an element $c \in C$ is *exactly represented* in $\mathcal{A}$ if $(\gamma \circ \alpha)c = c$. $\mathcal{A}$ satisfies the *ascending chain condition* (*ACC*) if $\langle A, \leq \rangle$ has no infinite ascending chains. In such cases the fixpoint of any monotone function can be effectively computed in a finite number of steps. We say that an abstract domain is *trivial* if it is isomorphic to the concrete domain, i.e., $\gamma \circ \alpha = \mathtt{id}_C$, or if it consists of only one element $A = \{\top_A\}$, i.e., for all $c \in C$, $(\gamma \circ \alpha)c = \top_C$. The former is called identity abstraction and the latter top abstraction, in which case we denote $\mathcal{A} = \Top$ and $\gamma \circ \alpha = \Top \stackrel{\text{def}}{=} \lambda x.\top_C$.

> *In the rest of the paper we will only consider Galois insertions over strict or trivial abstract domains.* [H1]

As a consequence of [H1], any non-trivial abstract domain is assumed strict (e.g., $\Top$ is not strict).

Sound and complete abstractions. Given an abstract domain $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$ and a monotone operator $f : C \longrightarrow C$, we say that a function $f^\sharp : A \longrightarrow A$ is a *correct abstract interpretation* of f if $\alpha(f(c)) \leq f^\sharp(\alpha(c))$ for any $c \in C$. Note that if $f^\sharp$ is a correct abstract interpretation of f then we have also fixpoint correctness, i.e., $\alpha(lfp(f)) \leq lfp(f^\sharp)$. An abstract function f^α is the *best correct approximation* of f in $\mathcal{A}$ iff $f^\alpha \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma : A \longrightarrow A$, because for any correct abstract interpretation $f^\sharp$ it holds that $f^\alpha(a) \leq f^\sharp(a)$ for any $a \in A$.

We say that $f^\sharp$ is *complete* if $\alpha \circ f = f^\sharp \circ \alpha$. We say that $\mathcal{A}$ is a complete abstraction for f if there exists a complete abstract interpretation $f^\sharp$ of f. Completeness of $f^\sharp$ intuitively encodes the greatest achievable precision when abstracting the concrete behaviour of f on $\mathcal{A}$. In complete abstractions the only loss of precision is due to the abstract domain and not to the abstract functions $f^\sharp$. If $f^\sharp$ is complete for f then we have fixpoint completeness (also called fixpoint transfer): $\alpha(lfp(f)) = lfp(f^\sharp)$, which follows from $\alpha \circ f = f^\sharp \circ \alpha$. It turns out that completeness $\alpha \circ f = f^\sharp \circ \alpha$ holds iff $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$. Thus, the possibility of defining a complete approximation $f^\sharp$ of f on some abstract domain $\mathcal{A}$ only depends upon the best correct approximation of f in $\mathcal{A}$, i.e., completeness is a property of the concrete semantics and of the abstract domain only [Giacobazzi et al. 2000]. We will say both "$\mathcal{A}$ is complete for f" and "f is complete on $\mathcal{A}$" to refer to completeness. Note that the identity and the top abstractions, $\mathtt{id}$ and $\Top$, are both complete for any f.

It is also worth noting that by replacing the abstract join operator $\sqcup$, defined in the abstract domain $\mathcal{A}$, with a widening operator $\nabla : A \times A \longrightarrow A$, such that for any $a, b \in A$: $a \sqcup b \leq a \nabla b$ and ∇ extrapolates the possibly infinite chain of the iterates of $f^\sharp$ to a finite chain, see [Cousot and Cousot 1977], we always reduce the precision of the fixpoint computed in $\mathcal{A}$, i.e., $lfp(f^\sharp) \leq \nabla_{n<\omega} f^{\sharp n}(\bot_A)$. This means that if a widening-based program analysis on the abstract domain $\mathcal{A}$ is fipoint complete for $f^\sharp$, i.e., $\alpha(lfp(f)) = \nabla_{n<\omega} f^{\sharp n}(\bot_A)$, then $\mathcal{A}$ is obviously fixpoint complete for f. Because we are interested in the recursive properties of the class of programs for which an abstract interpreter defined on an abstract domain $\mathcal{A}$ is incomplete, in the following we do not consider widening-based fixpoint extrapolating operators to enforce termination in program analysis, and consider the tighter condition of incompleteness caused only by the Galois insertion specifying $\mathcal{A}$. This will produce the smallest class of programs for which a given abstract domain produces false alarms.

4.2 Program Semantics

Our denotations are *stores*, i.e., partial functions $\mathbb{m}$ in $\mathbb{S} \stackrel{\text{def}}{=} Var \longrightarrow \mathbb{Z}$ that assign values only to a finite set of variables. We will often represent a store $\mathbb{m} \in \mathbb{S}$ as a tuple $\langle x_1/v_1, \ldots, x_n/v_n \rangle$ of its defined values, i.e., such that $\mathbb{m}(y) = \$$ if $y \notin \{x_1, \ldots, x_n\}$ and $\mathbb{m}(x_i) = v_i$ for all $i \in [1, n]$. As usual

we let $var(\mathbb{m}) = \{\, x \in Var \mid \mathbb{m}(x) \neq \$ \,\}$. Without loss of generality, in the following instead of dealing with values in the set $\mathbb{Z} \cup \{\$\}$ we assume that the default value \$ is just a chosen element of $\mathbb{Z}$. In this sense a store $\mathbb{m}$ is seen as a total function where \$ is the default value for unused variables. For S a set of stores, we denote by $var(S)$ the set $\bigcup_{\mathbb{m}\in S} var(\mathbb{m}) = \{\, x \in Var \mid \exists \mathbb{m} \in S \wedge \mathbb{m}(x) \neq \$ \,\}$.

Programs represent partial recursive functions from input stores to output stores. Store update is written $\mathbb{m}[x \mapsto v]$ with

$$\mathbb{m}[x \mapsto v](y) = \begin{cases} v & \text{if } y = x \\ \mathbb{m}(y) & \text{otherwise} \end{cases}$$

As a matter of notation, for a set of values V, we write $\mathbb{m}[x \mapsto V]$ for the set of stores where x is updated with values in V: $\mathbb{m}[x \mapsto V] = \{\mathbb{m}[x \mapsto v] \mid v \in V\}$. Similarly, for a set of stores S we let $S[x \mapsto V] = \{\mathbb{m}[x \mapsto v] \mid \mathbb{m} \in S \wedge v \in V\}$. The semantics of arithmetic and boolean expressions are the functions $(\!|a|\!) : \mathbb{S} \to \mathbb{Z}$ and $(\!|b|\!) : \mathbb{S} \to \{\mathbf{tt}, \mathbf{ff}\}$ defined as usual.

Abstract interpretations consider *collecting semantics* of programs, which are the additive extension of the standard semantics, defined as partial recursive functions on stores, to functions on properties of stores. The collecting semantics of arithmetic expressions $a \in \mathtt{AExp}$, $[\![a]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{Z})$, is defined as $[\![a]\!]S \stackrel{\text{def}}{=} \{(\!|a|\!)\mathbb{m} \mid \mathbb{m} \in S\}$ for any set of stores S. Similarly, for boolean expressions $b \in \mathtt{BExp}$, $[\![b]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ is defined as $[\![b]\!]S \stackrel{\text{def}}{=} \{\mathbb{m} \in S \mid (\!|b|\!)\mathbb{m} = \mathbf{tt}\}$, i.e., $[\![b]\!]S$ filters the stores of S which make b true. The collecting semantics of a program P is the function $[\![P]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$, defined inductively as follows:

$$\begin{aligned}
[\![\mathbf{skip}]\!]S &\stackrel{\text{def}}{=} S \\
[\![x := a]\!]S &\stackrel{\text{def}}{=} \{\, \mathbb{m}[x \mapsto (\!|a|\!)\mathbb{m}] \mid \mathbb{m} \in S \,\} \\
[\![P_1; P_2]\!]S &\stackrel{\text{def}}{=} [\![P_2]\!]([\![P_1]\!]S) \\
[\![\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]S &\stackrel{\text{def}}{=} [\![P_1]\!]([\![b]\!]S) \cup [\![P_2]\!]([\![\neg b]\!]S) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ P]\!]S &\stackrel{\text{def}}{=} [\![\neg b]\!]\big(lfp(\lambda T.\, S \cup [\![P]\!]([\![b]\!]T))\big).
\end{aligned}$$

In this case $\lambda\mathbb{m} \in \mathbb{S}\,.\,[\![P]\!]\{\mathbb{m}\}$ is the partial recursive function computed by P, where $[\![P]\!]\{\mathbb{m}\} = \varnothing$ means non-termination of program P when evaluated in the store $\mathbb{m}$. Conversely, when a program terminates on a store $\mathbb{m}$ we have $[\![P]\!]\{\mathbb{m}\} = \{\mathbb{m}'\}$ for a suitable store $\mathbb{m}'$. Note that, when P does not contain any assignment to a variable x, if $[\![P]\!]\{\mathbb{m}\} = \{\mathbb{m}'\}$ then $\mathbb{m}'(x) = \mathbb{m}(x)$.

Despite the definition of collecting semantics applies to any subset of $\mathbb{S}$, in the following we consider only sets of stores that predicate over a finite set of variables, as is always the case in abstract interpretation. This still allows the same variable x to be assigned infinitely many different values by the stores in a set $S \subseteq \mathbb{S}$.

Definition 1 (Variable finite sets of stores). *We say a set of stores $S \subseteq \mathbb{S}$ is* variable finite *if $|var(S)| < \omega$. We denote by $\widehat{\wp}(\mathbb{S})$ the powerset of variable finite sets of stores, together with the top element $\mathbb{S}$, i.e.,*

$$\widehat{\wp}(\mathbb{S}) \stackrel{def}{=} \{\, S \subseteq \mathbb{S} \mid |var(S)| < \omega \vee S = \mathbb{S} \,\}.$$

Lemma 2. *$\langle \widehat{\wp}(\mathbb{S}), \subseteq, \cup, \cap, \mathbb{S}, \varnothing \rangle$ is a complete lattice.*

Proof. Obviously the bottom is $\varnothing$ and the top is $\mathbb{S}$. The fact that $\widehat{\wp}(\mathbb{S})$ is closed under intersection follows from the fact that countable intersection of variable finite sets is also variable finite. For countable union, if the ordinary union is not variable finite, then we take $\mathbb{S}$ as its lub. *q.e.d.*

Because programs always manipulate a finite set of variables, the concrete collecting semantics $[\![\cdot]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ defined above can be seen as restricted to $\widehat{\wp}(\mathbb{S})$, that is $[\![\cdot]\!] : \widehat{\wp}(\mathbb{S}) \to \widehat{\wp}(\mathbb{S})$[2]. We will often abuse notation and represent with $[\![P]\!]$ both the above mentioned collecting semantics (i.e., a total function from set of stores to set of stores) and the ordinary denotational semantics of P (i.e., a partial function $\lambda x.\, [\![P]\!]\{x\}$ from stores to stores).

Some points of the concrete domain are particularly important because their membership can be effectively tested with computable predicates. We call such sets of stores recursive.

Definition 3 (Recursive set of stores). *Let $\mathbb{m}_{|V} : Var \to \mathbb{Z}$ be the restriction of $\mathbb{m}$ to the set $V \subseteq Var$ defined as*

$$\mathbb{m}_{|V}(x) \stackrel{def}{=} \begin{cases} \mathbb{m}(x) & \text{if } x \in V \\ \$ & \text{otherwise} \end{cases}$$

and similarly let $S_{|V} \stackrel{def}{=} \{\mathbb{m}_{|V} \mid \mathbb{m} \in S\}$. We say that $S \in \widehat{\wp}(\mathbb{S})$ is recursive *iff there exists a total recursive function f_S that decides membership in $\left\{ \mathbb{m} \in \mathbb{S} \,\middle|\, \mathbb{m}_{|var(S)} \in S_{|var(S)} \right\}$.*

Note that, if V is finite the set of stores $\mathbb{S}_{|V}$ belongs to $\widehat{\wp}(\mathbb{S})$ and for any $S \in \widehat{\wp}(\mathbb{S})$: $S \subseteq \mathbb{S}_{|var(S)}$.

4.3 Abstract Semantics

We now define the *abstract semantics* for a generic abstract domain $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$. As usual in abstract interpretation, we are interested in abstract domains whose elements represent recursive properties (sets) of stores. This is because membership, i.e, whether any concrete computed store satisfies the property detected by the analyser at a given program point, must be decidable in order for the analysis to be useful.

Definition 4 (Recursive abstract domain). *Given an abstract domain $\mathcal{A}$, an abstract store $S^\sharp \in \mathcal{A}$ is* recursive *if $\gamma(S^\sharp)$ is recursive (according to Definition 3). $\mathcal{A}$ is* recursive *if all its elements are recursive.*

In the following we only consider recursive or trivial abstractions of $\widehat{\wp}(\mathbb{S})$. [H2]

Note that $\top\!\!\top$ is recursive while `id` is not.

It is well known that abstract interpretation is not compositional, namely the composition of two best correct abstract semantics may not be the best correct abstraction of the semantics of the two components. This is indeed the main source of imprecision in program analysis. In the following, for the sake of simplicity, we will focus only on loss of precision induced by the inductive composition of commands. Therefore, for arithmetic or boolean expressions $e \in \mathtt{AExp} \cup \mathtt{BExp}$, we consider as abstract semantics their best correct approximating semantics in $\mathcal{A}$, i.e., we assume that the function $[\![e]\!]^{\mathcal{A}} \stackrel{\text{def}}{=} \alpha \circ [\![e]\!] \circ \gamma$ is computable in our language. This way the abstract semantics of expressions is computed independently from their syntax, allowing us to focus on the composition of commands as the main source of imprecision in our presentation. This assumption can be of course weakened by composing the abstract semantics of sub-expressions, at the price of further loss of precision, which is superfluous for our purposes.

The abstract semantics $[\![P]\!]^{\mathcal{A}} : A \to A$ of a program $P \in \mathtt{Imp}$ is therefore inductively defined on the syntax of commands as the composition of the the abstract semantics of their components.

[2]Our focus is on recursive-theoretic properties of false alarm removal and injection, therefore we consider here the simplest possible Turing complete language `Imp` where programs manipulate a finite set of variables. Richer languages manipulating an unbound number of variables, e.g., by recursion, can be considered at the price of complicating the model and replacing variable finiteness with abstract domains defined as functions from natural numbers $n \in \mathbb{N}$ to Galois connections on a concrete domain with n variables (e.g., see [Venet 1996]).

As above, we assume that the best correct approximation of assignments is computable in our language. In particular, in any Galois insertion the abstract join $\sqcup$ is by definition the best correct approximation of the concrete join $\cup$. If $S^\sharp \in A$ is an abstract store:

$$\begin{aligned}
[\![\mathbf{skip}]\!]^{\mathcal{A}} S^\sharp &\stackrel{\text{def}}{=} S^\sharp \\
[\![x := a]\!]^{\mathcal{A}} S^\sharp &\stackrel{\text{def}}{=} \alpha(\{\, \mathbb{m}[x \mapsto (\!|a|\!)\mathbb{m}] \mid \mathbb{m} \in \gamma(S^\sharp) \,\}) \\
[\![P_1; P_2]\!]^{\mathcal{A}} S^\sharp &\stackrel{\text{def}}{=} [\![P_2]\!]^{\mathcal{A}}([\![P_1]\!]^{\mathcal{A}} S^\sharp) \\
[\![\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2]\!]^{\mathcal{A}} S^\sharp &\stackrel{\text{def}}{=} [\![P_1]\!]^{\mathcal{A}}([\![b]\!]^{\mathcal{A}} S^\sharp) \sqcup [\![P_2]\!]^{\mathcal{A}}([\![\neg b]\!]^{\mathcal{A}} S^\sharp) \\
[\![\mathbf{while}\ b\ \mathbf{do}\ P]\!]^{\mathcal{A}} S^\sharp &\stackrel{\text{def}}{=} [\![\neg b]\!]^{\mathcal{A}}\big(\mathit{lfp}(\lambda T^\sharp.\, S^\sharp \sqcup [\![P]\!]^{\mathcal{A}}[\![b]\!]^{\mathcal{A}} T^\sharp)\big)
\end{aligned}$$

Note that the best correct approximation of an assignment $[\![x := a]\!]^{\mathcal{A}}$ does not rely on the best correct abstract semantics $[\![a]\!]^{\mathcal{A}}$ of the arithmetic expression a. Moreover in the least fixpoint definition of $[\![\mathbf{while}\ b\ \mathbf{do}\ P]\!]^{\mathcal{A}}$, the abstract function $\lambda T^\sharp.\, S^\sharp \sqcup [\![P]\!]^{\mathcal{A}}([\![b]\!]^{\mathcal{A}} T^\sharp)$ turns out to be the best correct approximation on $\mathcal{A}$ of the concrete function $\lambda T.\, S \cup [\![P]\!][\![b]\!]T$. Because the abstract semantics $[\![\cdot]\!]^{\mathcal{A}}$ above is fully determined by the abstract domain $\mathcal{A}$, we often abuse notation and indicate $\mathcal{A}$ as abstract semantics or abstract interpretation.

> *In the following we assume that the best correct approximation in $\mathcal{A}$ of expressions and assignments are computable in our language.* [H3]

Example 5. *As a running example of an abstract interpretation, consider the abstract domain of intervals* Int *on the integers $\mathbb{Z}$, already mentioned in the introduction. Elements of* Int *are finite intervals $[a, b]$ with $a \leq b$, or infinite intervals of the form $[-\infty, b]$ or $[a, \infty]$, together with the empty interval $\varnothing$ (the bottom element). The top element is $[-\infty, \infty]$. Intervals are ordered by inclusion. The concretisation function γ is defined as expected, while the abstraction function α maps a set of integers to the smallest interval that contains it (see [Cousot and Cousot 1977]).*

In this example, let the default value \$ be 0, so that in the abstract store the default value is the interval $[0, 0]$. Moreover, since the abstraction is non-relational, an abstract store $S^\sharp$ maps a finite number of variables to (non-default) intervals of the form $[a, b]$ with

$$\gamma(S^\sharp) = \{\, \mathbb{m} \mid \forall y, a, b.\, S^\sharp(y) = [a, b] \Rightarrow a \leq \mathbb{m}(y) \leq b \,\}$$

and for any set of stores S we let

$$\alpha(S)(y) = [\min_{\mathbb{m} \in S} \mathbb{m}(y), \max_{\mathbb{m} \in S} \mathbb{m}(y)]$$

Let us consider the program

$$P \stackrel{\text{def}}{=} \begin{array}{l} \mathbf{if}\ x = 1\ \mathbf{then}\ y := 1 \\ \mathbf{else}\ y := 3 \end{array}$$

and the concrete set of stores $S = \{\langle x/1\rangle, \langle x/2\rangle\}$. In the collecting semantics we have of course $[\![P]\!]S = \{\langle x/1, y/1\rangle, \langle x/2, y/3\rangle\}$. Let $S^\sharp = \alpha(S)$. Clearly $S^\sharp(x) = [1, 2]$. Then, we have

$$[\![P]\!]^{\mathcal{A}} S^\sharp \quad = \quad [\![y := 1]\!]^{\mathcal{A}}([\![x = 1]\!]^{\mathcal{A}} S^\sharp) \sqcup [\![y := 3]\!]^{\mathcal{A}}([\![x \neq 1]\!]^{\mathcal{A}} S^\sharp)$$

Now let $[\![x = 1]\!]^{\mathcal{A}} S^\sharp = S^\sharp_1$ and $[\![x \neq 1]\!]^{\mathcal{A}} S^\sharp = S^\sharp_2$, then $S^\sharp_1(x) = [1, 1]$ and $S^\sharp_2(x) = [2, 2]$. Therefore

$$[\![P]\!]^{\mathcal{A}} S^\sharp = ([\![y := 1]\!]^{\mathcal{A}} S^\sharp_1) \sqcup ([\![y := 3]\!]^{\mathcal{A}} S^\sharp_2) = T^\sharp$$

with $T^\sharp(x) = [1, 2]$ and $T^\sharp(y) = [1, 3]$. Thus, in this particular case we have $\alpha([\![P]\!]S) = [\![P]\!]^{\mathcal{A}}\alpha(S)$.

Below we list some technical lemmas about the abstract semantics of programs that will be used in the proofs of our main results. The proofs of these results are quite standard since they are derived directly from the definitions.

Lemma 6. *For any arithmetic expression $a \in \mathtt{AExp}$, boolean expression $b \in \mathtt{BExp}$, and program $P \in \mathtt{Imp}$, we have that $[\![a]\!]^{\mathcal{A}}$, $[\![b]\!]^{\mathcal{A}}$ and $[\![P]\!]^{\mathcal{A}}$ are monotone.*

Lemma 7. *For any boolean expression $b \in \mathtt{BExp}$ and any abstract store $S^\sharp \in A$,*

$$[\![b]\!]^{\mathcal{A}}([\![b]\!]^{\mathcal{A}} S^\sharp) = [\![b]\!]^{\mathcal{A}} S^\sharp \,.$$

Lemma 8. *For any program $P \in \mathtt{Imp}$ and any $S^\sharp, T^\sharp \in A$, if $S^\sharp \leq T^\sharp$ we have*

$$[\![P]\!]^{\mathcal{A}} S^\sharp \sqcup [\![P]\!]^{\mathcal{A}} T^\sharp = [\![P]\!]^{\mathcal{A}} T^\sharp \,.$$

Lemma 8 follows clearly by monotonicity of $[\![P]\!]^{\mathcal{A}}$ (see Lemma 6).

Some abstract domains preserve, in the abstraction, the variable finiteness condition of the concrete domain. We will exploit this condition to prove some important properties of the abstraction.

Definition 9 (Variable finite abstract domains). *We say that an abstract domain $\mathcal{A}$ is* variable finite *if for any $S \in \widehat{\wp}(\mathbb{S})$ we have $var((\gamma \circ \alpha)S) = var(S)$.*

The condition of variable finiteness amounts to require that the abstraction does not introduce information about unused variables and therefore preserves the variable finiteness of any concrete set $S \neq \mathbb{S}$. Note that, except for the top abstraction, most standard abstract domains in abstract interpretation are variable finite. This is because programs manipulate a finite set of variables and it is always possible to increase the number of interesting variables making a new program. Abstract domains are therefore usually defined having variables as parameter (e.g., see [Venet 1996]).

A simple consequence of Definition 9 is that for any variable finite abstract domain $\mathcal{A}$ we have that $\alpha(S) = \alpha(\mathbb{S})$ implies $S = \mathbb{S}$. This is because for any $S \neq \mathbb{S}$ it holds $S \subseteq \mathbb{S}_{|var(S)}$ and therefore we have $\alpha(S) \leq \alpha(\mathbb{S}_{|var(S)}) < \alpha(\mathbb{S})$. The following lemma says that when $\mathcal{A}$ is variable finite, then $[\![P]\!]^{\mathcal{A}}$ cannot map to top an abstract element which is different from top.

Lemma 10. *If $\mathcal{A}$ is variable finite, for any $P \in \mathtt{Imp}$ and $S^\sharp \neq \alpha(\mathbb{S})$, we have that $[\![P]\!]^{\mathcal{A}} S^\sharp \neq \alpha(\mathbb{S})$.*

The proof of Lemma 10 is by structural induction on the program P, exploiting the variable finiteness condition on the abstract domain and the fact that P can only manipulate a finite number of variables.

It is worth noting that, as a consequence of [H2], if a domain is variable finite and recursive then it is not trivial. In fact the identity abstraction is not recursive and the top abstraction is not variable finite.

Completeness classes. The notion of completeness class of programs has been introduced in [Giacobazzi et al. 2015] as the set of all programs for which an abstract interpretation is complete:

$$\mathbb{C}(\mathcal{A}) \stackrel{\text{def}}{=} \left\{ P \in \mathtt{Imp} \,\middle|\, \alpha \circ [\![P]\!] = [\![P]\!]^{\mathcal{A}} \circ \alpha \right\}.$$

Roughly, the completeness class $\mathbb{C}(\mathcal{A})$ is defined to be the set of all programs whose static analysis on a given abstraction $\mathcal{A}$ will never produce false alarms. By complement notation, we denote by $\overline{\mathbb{C}(\mathcal{A})}$ the set of all programs whose abstract analysis can produce false alarms. This is a property of programs with respect to a fixed abstraction. It is worth noting that the number of programs that meet this property is always infinite. For any abstract domain $\mathcal{A}$ whose abstraction function is computable we have that $|\mathbb{C}(\mathcal{A})| = \omega$. This is shown by a straightforward padding argument by observing that **skip** $\in \mathbb{C}(\mathcal{A})$ for any $\mathcal{A}$ and because the composition of two complete functions

is complete, therefore the sequential composition of complete commands is still complete, i.e., if $P \in \mathbb{C}(\mathcal{A})$ then **skip**; $P \in \mathbb{C}(\mathcal{A})$. In [Giacobazzi et al. 2015] the authors proved that $\mathbb{C}(\mathcal{A}) = \mathtt{Imp}$ if and only if $\mathcal{A}$ is trivial, moreover $\mathbb{C}(\mathcal{A})$ is a non recursive enumerable set for non-trivial abstractions. In the following of this paper we will focus our attention on the structure of $\overline{\mathbb{C}(\mathcal{A})}$.

5 ABSTRACT EXTENSIONALITY

A set (i.e., a property) of programs $\Pi \subseteq \mathtt{Imp}$, or an index set for partial recursive functions, is Rice-extensional when

$$P \in \Pi \ \wedge \ [\![P]\!] = [\![Q]\!] \implies Q \in \Pi \tag{1}$$

In this case $P \in \Pi$ is the index of the partial recursive function $[\![P]\!]$. We recall that an index set Π is recursive if and only if it is trivial, i.e., $\Pi = \varnothing$ or $\Pi = \mathtt{Imp}$. This is the well known Rice theorem [Rice 1953]. In the following we generalise this notion by replacing the concrete semantics $[\![\cdot]\!]$ with a generic abstract semantics $[\![\cdot]\!]^{\mathcal{A}}$ for an abstract domain $\mathcal{A}$. This allows us to introduce a parametric notion of extensionality, called *abstract extensionality* which depends upon the abstraction $\mathcal{A}$.

Definition 11. *Let $\mathcal{A}$ be an abstract domain. An (abstract) $\mathcal{A}$-index set for partial recursive functions or abstract program property is any $\Pi^{\mathcal{A}} \subseteq \mathtt{Imp}$ such that*

$$P \in \Pi^{\mathcal{A}} \ \wedge \ [\![P]\!]^{\mathcal{A}} = [\![Q]\!]^{\mathcal{A}} \implies Q \in \Pi^{\mathcal{A}}$$

Abstract program properties are properties of programs that are closed by abstract semantics. All properties in program analysis as formalised by abstract interpretation are abstract semantic properties, or equivalently induce an abstract index set for partial recursive functions. This models precisely program analysis as an equivalence relation on programs.

Theorem 12. *If $\mathcal{A}$ is trivial then $\Pi^{\mathcal{A}}$ is Rice-extensional.*

Proof. If $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$ is trivial then $\gamma \circ \alpha = \mathtt{id}$ or $\gamma \circ \alpha = \top\!\!\top$. In the first case any abstract program property $\Pi^{\mathtt{id}}$ satisfies the notion of Rice-extensionality, as $[\![P]\!]^{\mathtt{id}} = [\![Q]\!]^{\mathtt{id}}$ iff $[\![P]\!] = [\![Q]\!]$. In the second case, if $\Pi^{\top\!\!\top} = \varnothing$ then $\Pi^{\top\!\!\top}$ is vacuously Rice-extensional. If instead $\Pi^{\top\!\!\top} \neq \varnothing$ then $\Pi^{\top\!\!\top} = \mathtt{Imp}$, because for any $P, Q \in \mathtt{Imp}$: $[\![P]\!]^{\top\!\!\top} = [\![Q]\!]^{\top\!\!\top}$, and therefore $\Pi^{\top\!\!\top}$ is Rice-extensional. *q.e.d.*

We know that, in general, an abstract program property $\Pi^{\mathcal{A}}$ can be non Rice-extensional and that a Rice-extensional property may not be an abstract program property, so that the two notions of Rice-extensional property of programs and abstract program property are in general incomparable (e.g., see the simple examples P, Q, and R in the introduction).

Given an abstraction $\mathcal{A}$, we consider the following similarity relation on programs based on the equivalence of the analysis performed by the abstract interpreter induced by $\mathcal{A}$:

$$P \approx^{\mathcal{A}} Q \ \text{ iff } \ [\![P]\!]^{\mathcal{A}} = [\![Q]\!]^{\mathcal{A}}$$

In the same way that Rice-extensional properties of programs are the union of equivalence classes of programs, where two programs are equivalent if they represent the same partial recursive function, abstract program properties are union of similar classes, where two programs are considered equivalent if they have the same abstract semantics, i.e., the same analysis. The following proposition comes straight from the definitions.

Proposition 13. *For any abstraction $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$:*

- *$\approx^{\mathcal{A}}$ is an equivalence relation.*
- *For any $\Pi^{\mathcal{A}}$ there exists $F \subseteq A \to A$ such that $\Pi^{\mathcal{A}} = \bigcup_{f \in F} \left\{ P \in \mathtt{Imp} \,\middle|\, [\![P]\!]^{\mathcal{A}} = f \right\}$.*
- *$\Pi^{\mathcal{A}} = \bigcup_{P \in \Pi^{\mathcal{A}}} [P]_{\approx^{\mathcal{A}}}$.*

In order to understand the structure of abstract program properties as sets of indices of partial recursive functions we need to study the structure of the equivalence classes of programs as induced by an abstract semantics $[\![\cdot]\!]^{\mathcal{A}}$. In particular we are interested in determining whether such equivalence classes are Rice-extensional, namely, if they are closed under the Rice-extensional equivalence. We will prove that a partition of programs into Rice-extensional equivalence classes induced by an abstract semantics is possible if and only if the abstraction is trivial, i.e., *all the equivalence classes of programs induced by meaningful abstract interpretations are not Rice-extensional.* This formalises the widely accepted folklore for which in program analysis it is always possible to transform a program into a semantically equivalent one for which the abstract semantics (i.e., the analysis) of the source is different from the one of the transformed program [Giacobazzi 2008; Laviron and Logozzo 2009]. In order to prove this result we need to find a program Q such that for a given $P \in \Pi^{\mathcal{A}}$: $[\![P]\!] = [\![Q]\!]$ but $[\![Q]\!]^{\mathcal{A}} \neq [\![P]\!]^{\mathcal{A}}$.

In the following we prove that for any non-trivial (recursive) abstraction there exist such P, Q. Moreover, for a large class of abstractions, those based on variable finite abstract domains (see Definition 9), we prove the stronger result that *for any complete program P* we can always find a program Q and an abstract store $S^\sharp$ such that $[\![P]\!] = [\![Q]\!]$ and $[\![P]\!]^{\mathcal{A}}S^\sharp < [\![Q]\!]^{\mathcal{A}}S^\sharp$ (Theorem 23). This can be formalised in terms of incompleteness of Q, i.e., Q is an incomplete version of P with respect to a non-trivial program analysis $\mathcal{A}$ (Corollary 24). In the case of non variable finite domains we just provide two sample programs P and Q such that $P \in \Pi^{\mathcal{A}}$: $[\![P]\!] = [\![Q]\!]$ but $[\![Q]\!]^{\mathcal{A}} < [\![P]\!]^{\mathcal{A}}$ (see Theorem 27).

6 COMPLETENESS AND INCOMPLETENESS CLIQUES

Given an abstract interpretation $\mathcal{A}$, the incompleteness (resp. completeness) clique of a program $P \in \texttt{Imp}$ is the set of all programs that compute the same partial function as P and whose abstract semantics is incomplete (resp. complete). Hence in a clique every two distinct programs are semantically equivalent on the concrete domain of stores. In an incompleteness clique these programs produce, when analysed, false alarms, while in a completeness clique they produce no false alarms.

Definition 14 (Completeness clique). *The completeness clique of $P \in \texttt{Imp}$ w.r.t. $\mathcal{A}$ is*

$$\mathbb{C}(P, \mathcal{A}) \stackrel{def}{=} \{\, Q \mid [\![P]\!] = [\![Q]\!] \,\} \cap \mathbb{C}(\mathcal{A}).$$

Definition 15 (Incompleteness clique). *The incompleteness clique of $P \in \texttt{Imp}$ w.r.t. $\mathcal{A}$ is*

$$\overline{\mathbb{C}}(P, \mathcal{A}) \stackrel{def}{=} \{\, Q \mid [\![P]\!] = [\![Q]\!] \,\} \cap \overline{\mathbb{C}(\mathcal{A})}.$$

Completeness (incompleteness) cliques model precisely the notion of semantically equivalent programs for which completeness (incompleteness) holds with respect to a given abstract semantics $\mathcal{A}$. Note that $\overline{\mathbb{C}}(P, \mathcal{A}) \neq \overline{\mathbb{C}(P, \mathcal{A})}$, because $\overline{\mathbb{C}}(P, \mathcal{A})$ only contains programs that compute the same partial function as P, while this is not the case for $\overline{\mathbb{C}(P, \mathcal{A})}$. The cliques $\mathbb{C}(P, \mathcal{A})$ and $\overline{\mathbb{C}}(P, \mathcal{A})$ form a partition of the set $[P]_{\approx_{\texttt{id}}}$ of extensively equivalent programs to P, therefore $\mathbb{C}(P, \mathcal{A})$ and $\overline{\mathbb{C}}(P, \mathcal{A})$ cannot be both empty. Whenever $\mathcal{A}$ is trivial we have that $\mathbb{C}(\mathcal{A}) = \texttt{Imp}$ and thus $\overline{\mathbb{C}}(P, \mathcal{A}) = \varnothing$ and $\mathbb{C}(P, \mathcal{A}) = [P]_{\approx_{\texttt{id}}}$. Also note that it may well happen that $P \notin \mathbb{C}(P, \mathcal{A})$ or that $P \notin \overline{\mathbb{C}}(P, \mathcal{A})$, because $\mathcal{A}$ can be either complete or incomplete for P. Moreover for any $P \in \texttt{Imp}$: (1) if $P \in \mathbb{C}(\mathcal{A})$ then $|\mathbb{C}(P, \mathcal{A})| = \omega$ and (2) if $P \in \overline{\mathbb{C}(\mathcal{A})}$ then $|\overline{\mathbb{C}}(P, \mathcal{A})| = \omega$. This is indeed obvious in both cases by padding P with **skip**.

As well as complexity cliques [Asperti 2008], completeness and incompleteness cliques are not in general extensional and enjoy interesting recursive properties that show the nature of false alarms in program analysis and the limit of their systematic removal. In particular, there are infinitely

many abstract domains for which it is impossible to systematically remove false alarms by effective code transformations. This is proved by the following theorem, where we denote by $X \preceq_m Y$ the *many-to-one reducibility* [Rogers 1992], i.e., the existence of a total recursive function f such that $x \in X \Leftrightarrow f(x) \in Y$.

Theorem 16. *For any strict ACC abstract domain $\mathcal{A}$, there exists $P \in \mathtt{Imp}$ such that if $\overline{\mathbb{C}}(P, \mathcal{A}) \neq \varnothing$ then $\overline{\mathbb{C}}(P, \mathcal{A}) \not\preceq_m \mathbb{C}(P, \mathcal{A})$.*

Proof. Note that any strict ACC abstract domain $\mathcal{A}$ is non-trivial: The identity abstraction is not ACC and the topmost abstraction is such that $\gamma(\bot_A) = \mathbb{S}$ contradicting the strictness condition $\gamma(\bot_A) = \varnothing$. Assume by contradiction that for all $P \in \mathtt{Imp}$: $\overline{\mathbb{C}}(P, \mathcal{A}) \neq \varnothing$ and $\overline{\mathbb{C}}(P, \mathcal{A}) \preceq_m \mathbb{C}(P, \mathcal{A})$. $\overline{\mathbb{C}}(P, \mathcal{A}) \preceq_m \mathbb{C}(P, \mathcal{A})$ implies that if $\mathbb{C}(P, \mathcal{A}) = \varnothing$ then also $\overline{\mathbb{C}}(P, \mathcal{A}) = \varnothing$, which contradicts the hypothesis. Then we can conclude that $\mathbb{C}(P, \mathcal{A}) \neq \varnothing$. By the above assumption, for any $P \in \mathtt{Imp}$ there exists $f_P : \mathtt{Imp} \longrightarrow \mathtt{Imp}$ which is total recursive and $Q \in \overline{\mathbb{C}}(P, \mathcal{A}) \Leftrightarrow f_P(Q) \in \mathbb{C}(P, \mathcal{A})$. Hence for any $P \in \mathtt{Imp}$, any program $Q \in \overline{\mathbb{C}}(P, \mathcal{A})$ is such that $[\![P]\!] = [\![f_P(Q)]\!]$ and $f_P(Q) \in \mathbb{C}(\mathcal{A})$. Because $\mathcal{A}$ is strict then $\alpha(S) = \bot_A$ iff $S = \varnothing$, therefore because $f_P(Q) \in \mathbb{C}(\mathcal{A})$, for any S:

$$[\![P]\!]S = [\![f_P(Q)]\!]S = \varnothing \;\Leftrightarrow\; \alpha([\![f_P(Q)]\!]S) = \bot_A \;\Leftrightarrow\; [\![f_P(Q)]\!]^{\mathcal{A}}\alpha(S) = \bot_A.$$

We know that $\mathcal{A}$ is ACC, therefore it is decidable whether $[\![f_P(Q)]\!]^{\mathcal{A}}\alpha(S) = \bot_A$, namely whether $[\![P]\!]S = \varnothing$, for any $S \in \widehat{\wp}(\mathbb{S})$. Therefore it would be possible to effectively transform any program P into an equivalent one $f_P(Q)$ for which termination is decidable, which is impossible. *q.e.d.*

Our goal now is to show that, when $\mathcal{A}$ is variable finite and not trivial, for any program $P \in \mathtt{Imp}$ we can always effectively generate a program $\tau(P) \in \overline{\mathbb{C}}(P, \mathcal{A})$, therefore for any $P \in \mathtt{Imp}$: $\overline{\mathbb{C}}(P, \mathcal{A}) \neq \varnothing$. In the light of the previous theorem, this result would have relevant consequences on the structure of the class of incomplete programs $\overline{\mathbb{C}(\mathcal{A})}$. In the above case, $\overline{\mathbb{C}(\mathcal{A})}$ includes at least a program for all computable functions, i.e., it is a Turing complete language.

7 REDUCING COMPLETENESS TO INCOMPLETENESS

In this section we design the code transformation aiming at making the analysis of a program incomplete for a given non-trivial abstract domain, still preserving the concrete semantics. As observed above, trivial abstract domains $\mathcal{A}$, being complete for any program, cause $\overline{\mathbb{C}}(P, \mathcal{A})$ to be empty, making the definition of a transformation in these cases unfeasible. For non-trivial abstract domains, our goal is to define a transformation function $\tau : \mathtt{Imp} \longrightarrow \mathtt{Imp}$ that must satisfy the following conditions:

(1) *Semantics preservation*: For each $P \in \mathtt{Imp}$: $[\![P]\!] = [\![\tau(P)]\!]$.
(2) *Incompleteness*: There exists $S \in \widehat{\wp}(\mathbb{S})$: $\alpha([\![\tau(P)]\!]S) < [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S)$.

The first condition states that transformed programs must have the same functional behaviour on every variable. The second condition requires that the abstract analysis is incomplete for at least one set of stores. All this means that $\tau : \mathbb{C}(P, \mathcal{A}) \longrightarrow \overline{\mathbb{C}}(P, \mathcal{A})$.

We introduce a simple control-flow transformation that injects in the source program some dead code that cannot be recognised by the abstract semantics. The main idea is to exploit the dead code assigning fixed values to unused variables of the program so that the abstract semantics of the transformed program would report that change. We use a conditional statement with an opaque predicate [Collberg and Nagra 2009] that depends upon the abstract domain. The opacity of this predicate enforces the abstract interpreter to consider both branches into account in the control-flow, even if only one of them is actually taken in any concrete execution.

In order to define our control-flow transformation we just require that the abstract domain is *variable finite* (see Definition 9) and *recursive* (see Definition 4). Equivalently, these are the variable finite abstract domains that are not trivial because of our hypothesis [H2].

The next lemma guarantees that recursive abstract domains cannot be the identical abstraction on recursive sets, i.e., there exists a concrete recursive set S that is not exactly represented in the abstract domain. The main results in this section, namely, Theorems 22–23 and Corollary 24, additionally require that the abstract element $\alpha(S)$ is different than $\mathbb{S}$, which is a consequence of the variable finiteness condition (see Corollary 18). We will exploit this S to inject incompleteness in arbitrary programs in `Imp`.

Lemma 17. *For any recursive abstract domain $\mathcal{A}$ there exists a recursive set $S \in \widehat{\wp}(\mathbb{S})$ that is not exactly represented in $\mathcal{A}$.*

Proof. Towards a contradiction, assume that all recursive sets in $\widehat{\wp}(\mathbb{S})$ are exactly represented in $\mathcal{A}$, i.e., $(\gamma \circ \alpha)S = S$ for any recursive set $S \in \widehat{\wp}(\mathbb{S})$. Let U_k be the set of indices of Turing machines that after k steps on input 0 do not terminate. For any $k \in \mathbb{N}$: U_k is a recursive set. With each U_k we can associate a variable finite set of stores $S_k = \left\{ \langle x/v \rangle \,\middle|\, v \in U_k \right\}$. Clearly the set $U = \bigcap_k U_k$ is the set of indices of Turing machines that on input 0 do not terminate, which is not recursive. Let $S = \left\{ \langle x/v \rangle \,\middle|\, v \in U \right\}$. Because the pair $\langle \alpha, \gamma \rangle$ forms a Galois insertion: $\bigwedge_k \alpha(S_k) \in \mathcal{A}$. Being $\mathcal{A}$ recursive, the set $T = \gamma(\bigwedge_k \alpha(S_k))$ is also recursive. Since γ is co-additive in any Galois insertion, we have $T = \gamma(\bigwedge_k \alpha(S_k)) = \bigcap_k \gamma(\alpha(S_k))$. Since we have assumed that all recursive sets are exactly represented, we have $T = \bigcap_k \gamma(\alpha(S_k)) = \bigcap_k S_k = S$, which yields a contradiction, because T is recursive while S is not. *q.e.d.*

Corollary 18. *If $\mathcal{A}$ is variable finite and recursive, then there exists a recursive set $S \in \widehat{\wp}(\mathbb{S})$ such that $S \subset (\gamma \circ \alpha)S \subset \mathbb{S}$.*

Proof. The existence of a recursive set S such that $S \subset (\gamma \circ \alpha)S$ is guaranteed by Lemma 17. Since $\mathcal{A}$ is variable finite it follows that for any $V \subset Var$:

$$|V| < \omega \implies \alpha(\mathbb{S}_{|V}) < \alpha(\mathbb{S}).$$

This means that, for any finite set of variables V, the abstract domain has to represent the set $\mathbb{S}_{|V}$ of all stores defined at most on V, and this abstract object $\alpha(\mathbb{S}_{|V})$ cannot represent the set of all stores. Clearly, $S \subseteq \mathbb{S}_{|var(S)}$, and thus $(\gamma \circ \alpha)S \leq (\gamma \circ \alpha)(\mathbb{S}_{|var(S)}) < \mathbb{S}$, by the properties of Galois insertions and because the abstract domain is variable finite. *q.e.d.*

Let S be a set satisfying the condition in Corollary 18. Let $V = var((\gamma \circ \alpha)S) = var(S)$, by the variable finiteness condition. Since S is recursive, there exists a total recursive function f such that

$$f(\mathbb{m}) = \begin{cases} 1 & \text{if } \mathbb{m} \in \left\{ \mathbb{m} \,\middle|\, \mathbb{m}_{|V} \in S_{|V} \right\} \\ 0 & \text{otherwise} \end{cases}$$

Since f is at most concerned with only a finite list of variables $\widetilde{x} \subseteq V$ it can be expressed as an arithmetic expression $f(\widetilde{x})$ such that $(\!|f(\widetilde{x})|\!)\mathbb{m} = 1$ if $\mathbb{m} \in \left\{ \mathbb{m} \,\middle|\, \mathbb{m}_{|V} \in S_{|V} \right\}$ and $(\!|f(\widetilde{x})|\!)\mathbb{m} = 0$ otherwise. We let the predicate $\mathtt{In}(S)$ be defined as $f(\widetilde{x}) = 1$. We prove that $\mathtt{In}(S)$ is an opaque predicate for the abstract interpreter, i.e., the abstract interpreter cannot decide whether $\mathtt{In}(S)$ is true or false on some input property of memories. In this way $\mathtt{In}(S)$ may drive the injection in the abstract semantics of values that are not computed in the concrete semantics. We design this by an assignment that will produce some store outside the scope of P. Since $(\gamma \circ \alpha)S \subset \mathbb{S}$, then there exists at least one store $\mathbb{m}' \notin (\gamma \circ \alpha)S$. Moreover, for any program P, we have $[\![P]\!]^{\mathcal{A}}\alpha(S) < \alpha(\mathbb{S})$ by Lemma 10, because $(\gamma \circ \alpha)S \subset \mathbb{S}$. Therefore, without loss of generality, we can find a store

$\mathbb{m}' \notin (\gamma \circ \alpha)S$ such that $var(\mathbb{m}')$ is disjoint from $var((\gamma \circ [\![P]\!]^{\mathcal{A}} \circ \alpha)S)$, because the (non-top) points of our concrete domain $\widehat{\wp}(\mathbb{S})$ predicate over finite set of variables. Suppose $\mathbb{m}' = \langle x_1/v_1, \ldots, x_n/v_n \rangle$ with $var((\gamma \circ \alpha)S) \setminus var(\mathbb{m}') = \{x_{n+1}, ..., x_{n+k}\}$ we let $\mathtt{Set}(\mathbb{m}', S)$ be the program

$$x_1 := v_1; \ldots; x_n := v_n; x_{n+1} := \$; \ldots; x_{n+k} := \$$$

We transform P into:

$$\begin{array}{rcl} \tau_{\mathbb{m}',S}(P) & \stackrel{\text{def}}{=} & \textbf{if } \mathtt{In}(S) \textbf{ then} \\ & & \quad \textbf{if } \neg\mathtt{In}(S) \textbf{ then } \mathtt{Set}(\mathbb{m}', S) \\ & & \quad \textbf{else } P \\ & & \textbf{else } P \end{array}$$

The intuition of the transformation is the following: Since $S \subset (\gamma \circ \alpha)S$ there will be at least one set of stores that can go through both branches in the abstract semantics. As a consequence $\mathtt{Set}(\mathbb{m}', S)$ is dead code for the concrete semantics but it is not dead code for the abstract one, marking $\tau_{\mathbb{m}',S}(P)$ incomplete for $\mathcal{A}$.

Note that in $\tau_{\mathbb{m}',S}(P)$, the choice of S may be independent of P, while that of $\mathbb{m}'$ is not, i.e., the opaque predicate depends on the abstract domain while the dead code depends on the source code. For brevity we denote by $\tau(P)$ the code injection transformation $\tau_{\mathbb{m}',S}(P)$ for suitable $\mathbb{m}'$ and S.

Proposition 19. *Let be τ defined as above. For any $P \in \mathtt{Imp}$ we have $[\![\tau(P)]\!] = [\![P]\!]$.*

Proof. In the following we let

$$\begin{array}{rcl} Q & \stackrel{\text{def}}{=} & \textbf{if } \neg\mathtt{In}(S) \textbf{ then } \mathtt{Set}(\mathbb{m}', S) \\ & & \textbf{else } P \end{array}$$

such that $\tau(P) = \textbf{if } \mathtt{In}(S) \textbf{ then } Q \textbf{ else } P$. Then, for any $\mathbb{m} \in \mathbb{S}$ we have:

$$\begin{array}{rcl} [\![\tau(P)]\!]\mathbb{m} & = & [\![Q]\!]([\![\mathtt{In}(S)]\!]\mathbb{m}) \cup [\![P]\!]([\![\neg\mathtt{In}(S)]\!]\mathbb{m}) \\ & = & [\![\mathtt{Set}(\mathbb{m}', S)]\!]([\![\neg\mathtt{In}(S)]\!]([\![\mathtt{In}(S)]\!]\mathbb{m})) \\ & & \cup\, [\![P]\!]([\![\mathtt{In}(S)]\!]([\![\mathtt{In}(S)]\!]\mathbb{m})) \cup\ [\![P]\!]([\![\neg\mathtt{In}(S)]\!]\mathbb{m}) \\ & = & [\![P]\!]([\![\mathtt{In}(S)]\!]\mathbb{m}) \cup [\![P]\!]([\![\neg\mathtt{In}(S)]\!]\mathbb{m}) \\ & = & [\![P]\!]\mathbb{m} \end{array}$$

since $[\![\neg\mathtt{In}(S)]\!]([\![\mathtt{In}(S)]\!]\mathbb{m}) = \varnothing$ and either $[\![\mathtt{In}(S)]\!]\mathbb{m} = \{\mathbb{m}\}$ and $[\![\neg\mathtt{In}(S)]\!]\mathbb{m} = \varnothing$ or $[\![\mathtt{In}(S)]\!]\mathbb{m} = \varnothing$ and $[\![\neg\mathtt{In}(S)]\!]\mathbb{m} = \{\mathbb{m}\}$. *q.e.d.*

We show in Theorem 22 that the above control-flow transformation makes any program incomplete. In the proof we exploit the following two technical lemmas.

Lemma 20. *Let $\mathcal{A}$ be variable finite and let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set such that $S \subset (\gamma \circ \alpha)S \subset \mathbb{S}$. Then $[\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S) = \alpha(S)$.*

Proof. By definition, $[\![\mathtt{In}(S)]\!]^{\mathcal{A}} = \alpha \circ [\![\mathtt{In}(S)]\!] \circ \gamma$. Thus $[\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S) = (\alpha \circ [\![\mathtt{In}(S)]\!] \circ \gamma \circ \alpha)S$ $= \alpha(\{(\!|\mathtt{In}(S)|\!)\mathbb{m} \mid \mathbb{m} \in (\gamma \circ \alpha)S\})$. Let $V = var((\gamma \circ \alpha)S) = var(S)$. We recall that $(\!|\mathtt{In}(S)|\!)\mathbb{m} = \mathbf{tt}$ iff $\mathbb{m} \in \{\mathbb{m} \mid \mathbb{m}_{|V} \in S_{|V}\}$. Now, for $\mathbb{m} \in (\gamma \circ \alpha)S$, we have $\mathbb{m}_{|V} \in S_{|V}$ iff $\mathbb{m} \in S$. Therefore we have that $\{(\!|\mathtt{In}(S)|\!)\mathbb{m} \mid \mathbb{m} \in (\gamma \circ \alpha)S\} = S$ and $[\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S) = \alpha(S)$. *q.e.d.*

Lemma 21. *Let $\mathcal{A}$ be variable finite and let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set such that $S \subset (\gamma \circ \alpha)S \subset \mathbb{S}$. Then $\bot < [\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S) \leq \alpha(S)$.*

Proof. By definition, $[\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}} = \alpha \circ [\![\neg\mathtt{In}(S)]\!] \circ \gamma$. Thus

$$\begin{array}{rcl} [\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S) & = & (\alpha \circ [\![\neg\mathtt{In}(S)]\!] \circ \gamma \circ \alpha)S \\ & = & \alpha(\{(\!|\neg\mathtt{In}(S)|\!)\mathbb{m} \mid \mathbb{m} \in (\gamma \circ \alpha)S\}) \end{array}$$

Let $S' = \{(\!|\neg\mathtt{In}(S)|\!)\mathbb{m} \mid \mathbb{m} \in (\gamma \circ \alpha)S\} \subseteq (\gamma \circ \alpha)S$. By monotonicity $\alpha(S') \leq (\alpha \circ \gamma \circ \alpha)S = \alpha(S)$ (because the abstract domain defines a Galois insertion). Let $V = var((\gamma \circ \alpha)S) = var(S)$. We recall that $(\!|\neg\mathtt{In}(S)|\!)\mathbb{m} = \mathbf{tt}$ iff $\mathbb{m} \notin \{\mathbb{m} \mid \mathbb{m}_{|V} \in S_{|V}\}$. Since $(\gamma\circ\alpha)S \supset S$, there exists some $\mathbb{m}'' \in (\gamma\circ\alpha)S$ and $\mathbb{m}'' \notin \{\mathbb{m} \mid \mathbb{m}_{|V} \in S_{|V}\}$. It follows that $\mathbb{m}'' \in S' \neq \varnothing$. Since $\mathcal{A}$ is strict we have $\alpha(S') > \alpha(\varnothing) = \bot$. *q.e.d.*

Theorem 22. *Let $\mathcal{A}$ be variable finite and recursive and let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set such that $S \subset (\gamma \circ \alpha)S \subset \mathbb{S}$. If τ is defined as above then for any $P \in \mathtt{Imp}$: $\alpha([\![\tau(P)]\!]S) < [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S)$.*

Proof. We have:

$$\begin{array}{rcl} [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S) & = & [\![\mathtt{Set}(\mathbb{m}', S)]\!]^{\mathcal{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}([\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S))) \\ & & \sqcup\, [\![P]\!]^{\mathcal{A}}([\![\mathtt{In}(S)]\!]^{\mathcal{A}}([\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S))) \\ & & \sqcup\, [\![P]\!]^{\mathcal{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) \end{array}$$

By Lemma 7 and 20: $[\![\mathtt{In}(S)]\!]^{\mathcal{A}}([\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) = [\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S) = \alpha(S)$. Moreover, since

$$[\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) \leq \alpha(S),$$

by Lemma 8: $[\![P]\!]^{\mathcal{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) \sqcup [\![P]\!]^{\mathcal{A}}([\![\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) = [\![P]\!]^{\mathcal{A}}\alpha(S)$. Thus

$$\begin{array}{rcl} [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S) & = & [\![\mathtt{Set}(\mathbb{m}', S)]\!]^{\mathcal{A}}([\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) \\ & & \sqcup\, [\![P]\!]^{\mathcal{A}}\alpha(S) \end{array}$$

By Lemma 21 there exists S' with $\bot < \alpha(S') \leq \alpha(S)$ such that $[\![\neg\mathtt{In}(S)]\!]^{\mathcal{A}}\alpha(S) = \alpha(S')$ and thus $[\![\mathtt{Set}(\mathbb{m}', S)]\!]^{\mathcal{A}}\alpha(S') = \alpha(\{\mathbb{m}'\})$, because $\mathtt{Set}(\mathbb{m}', S)$ sets to \$ all variables in $var((\gamma \circ \alpha)S) \setminus var(\mathbb{m}')$ and $\alpha(S') \leq \alpha(S)$. Hence $[\![\tau(P)]\!]^{\mathcal{A}}\alpha(S) = \alpha(\{\mathbb{m}'\}) \sqcup [\![P]\!]^{\mathcal{A}}\alpha(S)$. By hypothesis $var(\mathbb{m}')$ is disjoint from the set of variables manipulated by $[\![P]\!]^{\mathcal{A}}\alpha(S)$, therefore we have:

$$[\![\tau(P)]\!]^{\mathcal{A}}\alpha(S) = \alpha(\{\mathbb{m}'\}) \sqcup [\![P]\!]^{\mathcal{A}}\alpha(S) > [\![P]\!]^{\mathcal{A}}\alpha(S) \geq \alpha([\![P]\!]S) = \alpha([\![\tau(P)]\!]S)$$

where the last equality follows from Proposition 19. *q.e.d.*

Theorem 23. *Let $\mathcal{A}$ be variable finite and recursive and let $S \in \widehat{\wp}(\mathbb{S})$ be a recursive set such that $S \subset \gamma \circ \alpha(S) \subset \mathbb{S}$. If τ is defined as above then for any $P \in \mathbb{C}(P, \mathcal{A})$: $[\![P]\!]^{\mathcal{A}}\alpha(S) < [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S)$.*

Proof. We have

$$\begin{array}{rcll} [\![P]\!]^{\mathcal{A}}\alpha(S) & = & \alpha([\![P]\!]S) & \text{(because } P \in \mathbb{C}(P, \mathcal{A})) \\ & = & \alpha([\![\tau(P)]\!])S & \text{(by Proposition 19)} \\ & < & [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S) & \text{(by Theorem 22)} \end{array}$$

q.e.d.

As a straight consequence, the semantics-preserving transformation described above can be used to define an effective transformation $\tau : \mathbb{C}(P, \mathcal{A}) \to \overline{\mathbb{C}}(P, \mathcal{A})$.

Corollary 24. *If $\mathcal{A}$ is variable finite and recursive and $\mathbb{C}(P, \mathcal{A}) \neq \varnothing$ then there exists a computable code transformation $\tau : \mathbb{C}(P, \mathcal{A}) \to \overline{\mathbb{C}}(P, \mathcal{A})$.*

It follows immediately that if $\mathcal{A}$ is variable finite and recursive then for any program $P \in \texttt{Imp}$: $|\overline{\mathbb{C}}(P, \mathcal{A})| = \omega$. This is because the control-flow transformation introduced above is semantics-preserving. Therefore for any $P \in \texttt{Imp}$, $\tau(P) \in \overline{\mathbb{C}}(P, \mathcal{A})$. Then by padding with **skip**, or by applying any number of further transformation steps, we obtain infinitely many programs that are equivalent to P but incomplete for $\mathcal{A}$.

Example 25. *Consider the abstract domain of intervals* Int *on integer numbers* $\mathbb{Z}$ *already discussed in the Example 5. Clearly* Int *is strict, recursive, variable finite and of course non-trivial. A simple* Imp *program belonging to* $\mathbb{C}(\mathsf{Int})$ *is for example* $x := x + n$ *for any* $n \in \mathbb{Z}$. *Indeed for any store* $\mathbb{m}$ *we have* $[\![x := x + n]\!]\mathbb{m} = \mathbb{m}[x \mapsto \mathbb{m}(x) + n]$ *and for any set of stores* S

$$\alpha([\![x := x + n]\!]S)(y) \quad = \quad \begin{cases} [\min_{\mathbb{m}\in S} \mathbb{m}(y), \max_{\mathbb{m}\in S} \mathbb{m}(y)] & \text{if } y \neq x \\ [\min_{\mathbb{m}\in S}(\mathbb{m}(x) + n), \max_{\mathbb{m}\in S}(\mathbb{m}(x) + n)] & \text{if } y = x \end{cases}$$

while

$$\begin{aligned} [\![x := x + n]\!]^{\mathcal{A}}\alpha(S) &= \alpha(\{\ \mathbb{m}[x \mapsto \mathbb{m}(x) + n] \mid \mathbb{m} \in \gamma(\alpha(S))\ \}) \\ &= \alpha(\{\ \mathbb{m}[x \mapsto \mathbb{m}(x) + n] \mid \forall y.\ \mathbb{m}(y) \in \gamma([\min_{\mathbb{m}'\in S} \mathbb{m}'(y), \max_{\mathbb{m}'\in S} \mathbb{m}'(y)])\ \}) \\ &= \alpha(\{\ \mathbb{m}[x \mapsto \mathbb{m}(x) + n] \mid \forall y.\ \min_{\mathbb{m}'\in S} \mathbb{m}'(y) \leq \mathbb{m}(y) \leq \max_{\mathbb{m}'\in S} \mathbb{m}'(y)\ \}). \end{aligned}$$

Thus: $([\![x := x + n]\!]^{\mathcal{A}}\alpha(S))(y) \quad = \quad \begin{cases} [\min_{\mathbb{m}\in S} \mathbb{m}(y), \max_{\mathbb{m}\in S} \mathbb{m}(y)] & \text{if } y \neq x \\ [\min_{\mathbb{m}\in S}(\mathbb{m}(x) + n), \max_{\mathbb{m}\in S}(\mathbb{m}(x) + n)] & \text{if } y = x \end{cases}$

Therefore $x := x+n \in \mathbb{C}(\mathsf{Int})$ *and therefore* $\mathbb{C}(x := x+n, \mathsf{Int})$ *is not empty. Let* $P = x := x+1 \in \mathbb{C}(\mathsf{Int})$. *We plan to apply our control-flow transformation to derive a program* $\tau(P)$ *that is equivalent to* P *but such that* $\tau(P) \in \overline{\mathbb{C}(\mathsf{Int})}$ *or, in other terms,* $\tau(P) \in \overline{\mathbb{C}}(x := x + 1, \mathsf{Int})$. *To this aim, let us take the set of stores* $S = \{\langle x/1\rangle, \langle x/3\rangle\}$ *and* $\mathbb{m}' = \langle y/2\rangle$. *We have in this case* $\texttt{In}(S) = (x = 1) \vee (x = 3)$. P *is then transformed into, where @ is an annotation for program points introduced to ease the presentation:*

$$\begin{aligned} \tau(P) \quad \stackrel{\text{def}}{=} \quad & \textbf{if } \texttt{In}(S) \textbf{ then} \\ & \quad \textbf{if } \neg\texttt{In}(S) \textbf{ then } (y := 2; x := 0) \\ & \quad \textbf{else } x := x + 1 \ \ (@1) \\ & \textbf{else } x := x + 1 \ (@2) \end{aligned}$$

We show that program $\tau(P)$ *is incomplete for the abstract domain* Int. *We have* $\alpha([\![\tau(P)]\!]S) = \alpha([\![x := x + 1]\!]S) = \alpha(S[x \mapsto \{2, 4\}]) = \alpha(S)[x \mapsto [2, 4]]$. *Moreover because* $\alpha(S)(x) = [1, 3]$ *and for each* $z \neq x$: $\alpha(S)(z) = [0, 0]$, *we have:*

$$\begin{aligned} [\![\texttt{In}(S)]\!]^{\mathcal{A}}\alpha(S) &= \alpha(S) \\ [\![\neg\texttt{In}(S)]\!]^{\mathcal{A}}\alpha(S) &= \alpha(S)[x \mapsto [2, 2]] \\ [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S) &= [\![y := 2; x := 0]\!]^{\mathcal{A}}([\![\neg\texttt{In}(S)]\!]^{\mathcal{A}}([\![\texttt{In}(S)]\!]^{\mathcal{A}}\alpha(S))) \\ &\quad \sqcup [\![x := x + 1\ (@1)]\!]^{\mathcal{A}}([\![\texttt{In}(S)]\!]^{\mathcal{A}}([\![\texttt{In}(S)]\!]^{\mathcal{A}}\alpha(S))) \\ &\quad \sqcup [\![x := x + 1\ (@2)]\!]^{\mathcal{A}}([\![\neg\texttt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) \\ &= [\![y := 2; x := 0]\!]^{\mathcal{A}}([\![\neg\texttt{In}(S)]\!]^{\mathcal{A}}\alpha(S)) \\ &\quad \sqcup [\![x := x + 1\ (@1)]\!]^{\mathcal{A}}\alpha(S) \\ &\quad \sqcup [\![x := x + 1\ (@2)]\!]^{\mathcal{A}}(\alpha(S)[x \mapsto [2, 2]]) \\ &= [\![y := 2; x := 0]\!]^{\mathcal{A}}(\alpha(S)[x \mapsto [2, 2]]) \sqcup\ \alpha(S)[x \mapsto [2, 4]] \sqcup \alpha(S)[x \mapsto [3, 3]] \\ &= \alpha(S)[y \mapsto [2, 2], x \mapsto [0, 0]] \sqcup \alpha(S)[x \mapsto [2, 4]] = \alpha(S)[x \mapsto [0, 4], y \mapsto [0, 2]]) \end{aligned}$$

This shows that the interval abstraction Int *is incomplete for the program* $\tau(P)$*, because e.g.,*

$$\alpha(\llbracket\tau(P)\rrbracket S)(y) = (\alpha(S)[x \mapsto [2,4]])(y) = [0,0] < [0,2] = (\llbracket \mathtt{In}(S)\rrbracket^{\mathcal{A}}\alpha(S))(y).$$

Example 26. *As a second example, we consider a simple relational domain formed by two-variables inequalities:* $\mathcal{A} = \{\bot, x-y<0, x-y>0, x-y=0, x-y\leqslant 0\}$*, where* $\bot$ *is the bottom element,* $x-y\leqslant 0$ *is the top element, and the other three elements are pairwise incomparable. The concrete denotation of an abstract element* $S^\sharp$ *is the set of stores satisfying all the relationships in* $S^\sharp$*. For example, given the singleton* $S = \{\langle x/1, y/2\rangle\}$ *we have:*

$$\alpha(S) = \{\forall z \neq y.\ y - z > 0, \quad \forall z \neq x, y.\ x - z > 0, \quad \forall z_1, z_2 \neq x, y.\ z_1 - z_2 = 0\ \}\ .$$

Then we take $\mathbb{m}' = \langle w/3\rangle$ *and consider the (complete) program* $P \stackrel{\text{def}}{=} u := x$*. We have:*

$$\begin{array}{rcl}\tau(P) & \stackrel{\text{def}}{=} & \textbf{if } (x = 1 \wedge y = 2) \textbf{ then} \\ & & \quad \textbf{if } (x \neq 1 \vee y \neq 2) \textbf{ then } (w := 3; x := 0; y := 0) \\ & & \quad \textbf{else } P \\ & & \textbf{else } P\end{array}$$

By some simple calculation we get

$$\begin{array}{rcl}\llbracket P\rrbracket & = & \llbracket\tau(P)\rrbracket \\ \llbracket x = 1 \wedge y = 2\rrbracket^{\mathcal{A}}\alpha(S) & = & \alpha(S) \\ \llbracket x \neq 1 \vee y \neq 2\rrbracket^{\mathcal{A}}\alpha(S) & = & \alpha(S) \\ \alpha(\llbracket P\rrbracket S) = \llbracket P\rrbracket^{\mathcal{A}}\alpha(S) & = & \{\ \forall z \neq y.\ y - z > 0, \quad \forall z \neq x, y, u.\ x - z > 0, \\ & & \quad \forall z \neq x, y, u.\ u - z > 0, \quad \forall z_1, z_2 \neq y.\ z_1 - z_2 = 0\ \} \\ \llbracket\tau(P)\rrbracket^{\mathcal{A}}\alpha(S) & = & \llbracket w := 3; x := 0; y := 0\rrbracket^{\mathcal{A}}(\llbracket x \neq 1 \vee y \neq 2\rrbracket^{\mathcal{A}}(\llbracket x = 1 \wedge y = 2\rrbracket^{\mathcal{A}}\alpha(S))) \\ & & \sqcup\ \llbracket P\rrbracket^{\mathcal{A}}(\llbracket x = 1 \wedge y = 2\rrbracket^{\mathcal{A}}(\llbracket x = 1 \wedge y = 2\rrbracket^{\mathcal{A}}\alpha(S))) \\ & & \sqcup\ \llbracket P\rrbracket^{\mathcal{A}}(\llbracket x \neq 1 \vee y \neq 2\rrbracket^{\mathcal{A}}\alpha(S)) \\ & = & \llbracket w := 3; x := 0; y := 0\rrbracket^{\mathcal{A}}\alpha(S) \\ & & \sqcup\ \llbracket P\rrbracket^{\mathcal{A}}\alpha(S) \\ & = & \{\ \forall z \neq w.\ w - z > 0, \quad \forall z_1, z_2 \neq w.\ z_1 - z_2 = 0\ \} \sqcup\ \llbracket P\rrbracket^{\mathcal{A}}\alpha(S) \\ & = & \{\ \forall z \neq y, w.\ y - z \leqslant 0, \quad \forall z \neq x, y, u, w.\ x - z \leqslant 0, \\ & & \quad \forall z \neq x, y, u, w.\ u - z \leqslant 0, \quad \forall z \neq w.\ w - z \leqslant 0, \\ & & \quad \forall z_1, z_2 \neq y, w.\ z_1 - z_2 = 0\ \}\end{array}$$

This shows that $\mathcal{A}$ *is incomplete for* $\tau(P)$ *because* $\llbracket\tau(P)\rrbracket^{\mathcal{A}}\alpha(S) \neq \alpha(\llbracket P\rrbracket S) = \alpha(\llbracket\tau(P)\rrbracket S)$ *.*

8 RICE EXTENSIONALITY OF THE ABSTRACT SEMANTICS

The top trivial abstraction $\top$ is an example of non variable finite abstraction. The next theorem proves that whenever $\mathcal{A}$ is neither variable finite nor trivial, there exists a pair of programs P and Q such that they have the same concrete semantics $\llbracket P\rrbracket = \llbracket Q\rrbracket$, but different abstract semantics $\llbracket P\rrbracket^{\mathcal{A}} < \llbracket Q\rrbracket^{\mathcal{A}}$.

Theorem 27. *If $\mathcal{A}$ is neither variable finite nor trivial then there exist $P, Q \in$ Imp such that $[\![P]\!] = [\![Q]\!]$, and $[\![P]\!]^{\mathcal{A}} < [\![Q]\!]^{\mathcal{A}}$.*

Proof. Since $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma\rangle$ is not trivial, it must be strict, and since $\mathcal{A}$ is not variable finite then there exists a set of stores $S \in \widehat{\wp}(\mathbb{S})$ such that $var(S) \subset var((\gamma \circ \alpha)S)$. Thus, there exists a store $\mathtt{m} \in (\gamma \circ \alpha)S$ and a variable $x \in var((\gamma \circ \alpha)S) \setminus var(S)$ with $\mathtt{m}(x) = v \neq \$$. Moreover, we have that $\gamma(\alpha(\varnothing)) = \varnothing$, because $\mathcal{A}$ is strict. We can now build P and Q such that $[\![P]\!] = [\![Q]\!]$, and $[\![P]\!]^{\mathcal{A}} < [\![Q]\!]^{\mathcal{A}}$ in the following way:

$$P \stackrel{\text{def}}{=} \textbf{while tt do skip}$$

$$\begin{array}{rcl} Q & \stackrel{\text{def}}{=} & \textbf{if } x \neq v \textbf{ then} \\ & & \quad \textbf{if } x = v \textbf{ then skip} \\ & & \quad \textbf{else } P \\ & & \textbf{else } P \end{array}$$

Clearly $[\![P]\!] = [\![Q]\!]$. Next we prove that $[\![P]\!]^{\mathcal{A}}\alpha(S) < [\![Q]\!]^{\mathcal{A}}\alpha(S)$. In fact, for any abstract store $S^\sharp$ (including $\alpha(S)$): $[\![P]\!]^{\mathcal{A}}S^\sharp = \bot \leq [\![Q]\!]^{\mathcal{A}}S^\sharp$. Moreover, as far as $\alpha(S)$ is concerned we have:

$$\begin{array}{rcl} [\![Q]\!]^{\mathcal{A}}\alpha(S) & = & [\![\textbf{skip}]\!]^{\mathcal{A}}([\![x = v]\!]^{\mathcal{A}}([\![x \neq v]\!]^{\mathcal{A}}\alpha(S))) \\ & & \sqcup\, [\![P]\!]^{\mathcal{A}}([\![x \neq v]\!]^{\mathcal{A}}([\![x \neq v]\!]^{\mathcal{A}}\alpha(S))) \\ & & \sqcup\, [\![P]\!]^{\mathcal{A}}([\![x = v]\!]^{\mathcal{A}}\alpha(S)) \\ & = & [\![\textbf{skip}]\!]^{\mathcal{A}}([\![x = v]\!]^{\mathcal{A}}([\![x \neq v]\!]^{\mathcal{A}}\alpha(S))) \\ & & \sqcup\, \bot \\ & = & [\![\textbf{skip}]\!]^{\mathcal{A}}([\![x = v]\!]^{\mathcal{A}}([\![x \neq v]\!]^{\mathcal{A}}\alpha(S))) \end{array}$$

Note that all the memories in S are such that $\mathtt{m}(x) \neq v$, thus $[\![x \neq v]\!]^{\mathcal{A}}\alpha(S) = \alpha(S)$. Thus

$$\begin{array}{rcl} [\![Q]\!]^{\mathcal{A}}\alpha(S) & = & [\![\textbf{skip}]\!]^{\mathcal{A}}([\![x = v]\!]^{\mathcal{A}}\alpha(S)) \\ & = & [\![\textbf{skip}]\!]^{\mathcal{A}}\alpha(S') \\ & = & \alpha(S') \end{array}$$

for some $S' \supset \{\mathtt{m}\} \supset \varnothing$. Since $\mathcal{A}$ is strict, $\alpha(S') > \bot$. Hence, we have that $[\![P]\!]^{\mathcal{A}}\alpha(S) < [\![Q]\!]^{\mathcal{A}}\alpha(S)$ and therefore $[\![P]\!]^{\mathcal{A}} < [\![Q]\!]^{\mathcal{A}}$. *q.e.d.*

We are now in the position of proving that the equivalence classes $[P]_{\approx^{\mathcal{A}}}$, e.g., those forming a non-trivial abstract program property $\Pi^{\mathcal{A}}$, are Rice-extensional if and only if $\mathcal{A}$ is trivial.

Theorem 28. *An abstract domain $\mathcal{A}$ is trivial iff $[P]_{\approx^{\mathcal{A}}}$ is Rice-extensional, for any $P \in$ Imp.*

Proof. Let $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma\rangle$. If $\mathcal{A}$ is trivial then by Theorem 12 for any $P \in$ Imp: $[P]_{\approx^{\mathcal{A}}}$ is Rice-extensional. For the converse implication, assume $\mathcal{A}$ is non-trivial. If $\mathcal{A}$ is variable finite then consider $P \in [P]_{\approx^{\mathcal{A}}}$. By Proposition 19: $[\![P]\!] = [\![\tau(P)]\!]$, and by Theorem 22: $[\![P]\!]^{\mathcal{A}} \neq [\![\tau(P)]\!]^{\mathcal{A}}$, which proves that $[P]_{\approx^{\mathcal{A}}}$ is not Rice-extensional. If instead $\mathcal{A}$ is not variable finite then by Theorem 27 we have that there exist $P, Q \in$ Imp such that $P \in [P]_{\approx^{\mathcal{A}}}$, $[\![P]\!] = [\![Q]\!]$ and $[\![P]\!]^{\mathcal{A}} \neq [\![Q]\!]^{\mathcal{A}}$, which implies that $[P]_{\approx^{\mathcal{A}}}$ is not Rice extensional. *q.e.d.*

9 MAKING ABSTRACTIONS INCOMPLETE BY DATA-FLOW TRANSFORMATIONS

The control-flow transformation used in the previous sections to make a program incomplete for an abstraction is an instance of a common transformation widely used to protect code against reverse engineering [Collberg et al. 1998]. It is well known that simple control-flow transformations are not resilient against dynamic attacks. While a static analyser may not be able to detect the presence of dead code, repeated concrete executions of the program can suggest that some instructions will be never executed, i.e., they are dead code. This is a well known phenomenon in code-protecting transformations (see for instance [Collberg and Nagra 2009] for a survey) where dynamic attacks are often sufficient to break *pure control-flow* transformations like opaque predicates and code flattening. By pure control-flow transformation we indicate any code transformation that alters the topology of the control flow graph of the program yet keeping the standard semantics unchanged [Majumdar et al. 2006]. All the constructions in the previous sections are based on pure control-flow transformations, keeping the concrete semantics unchanged and transforming complete programs into incomplete ones for a fixed non-trivial abstraction.

Nevertheless, other forms of obfuscations are known in code-protecting methods. *Data-flow* transformations, for instance, focus on data manipulation rather than altering the control structure of the program. By a *pure data-flow* transformation of programs we mean any transformation that modifies the values of program variables in an homomorphic way, i.e., keeping the concrete semantics unchanged. Of course data-flow and control-flow may be combined in very effective transformations, but the analysis of their combination is outside the scope of this paper.

The relevance of data-flow transformations was first witnessed in [Drape et al. 2007], where the authors characterise the class of data refinements by means of injective functions from an original data-type D to an obfuscated data-type O. Injectivity guarantees the existence of a left-inverse function, which is needed for semantic preservation on the original program variables.

In this section we explore the scope of pure data-flow transformations in making programs incomplete for an abstraction. We introduce in Theorem 31 a general data-flow transformation framework for injecting incompleteness and we establish in Theorem 33 the limit of applicability of pure data-flow transformations. This is achieved by finding the exact conditions that the abstract domains have to meet in order to have incompleteness by data-flow transformations.

For simplicity, in the following we assume that the abstraction on stores is induced by variables-wise abstraction, i.e., the store abstraction we consider is the one obtained by applying the same abstraction on all variable values separately. This implies that the abstract domain is variable finite. Given an abstract domain $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$ for the concrete domain of values $\wp(\mathbb{Z})$, we overload the symbol $\mathcal{A}$ to denote also the abstract domain of stores induced by the abstraction on values. For example, the interval abstraction considered in Example 25 falls in this category: An abstract store maps each variable to an interval of integers. Abstractions of this kind are called non-relational, as they are not expressive enough to catch mutual dependencies between variable. As any (finite) set of variables can always be encoded by a single one, the restriction to non-relational abstractions does not seem a real limitation, at least theoretically, to the applicability of data-flow transformations as presented here.

The idea of data-flow transformations is to define a pair of computable functions $g, h : \mathbb{Z} \to \mathbb{Z}$ such that $h \circ g = \mathtt{id}$ but, when their code is interpreted in $\mathcal{A}$, they fail to be one the inverse of the other, namely the abstract semantics $g^\#$ of g and $h^\#$ of h are such that $g^\# \circ h^\# \neq \mathtt{id}$. In the context of program analysis, injecting $h \circ g$ into the program by implementing g and h makes the program analysis incomplete for the abstract domain of stores induced by $\mathcal{A}$.

Because of the assumption [H3], all expressions are computed as their best correct approximation semantics in $\mathcal{A}$. Therefore, in order to inject incompleteness we need to split the computation of

$h \circ g$ into two separated commands in Imp. To explain how g and h can be injected in the program, let us consider any program $P_{x,f}$ that computes function f for a given variable x, i.e., such that for any $\mathbb{m} \in \mathbb{S}$: $[\![P_{x,f}]\!]\mathbb{m} = \mathbb{m}[x \mapsto f(\mathbb{m}(x))]$. An example of such a program is, of course, $x := f(x)$, but any equivalent program will do fine. Since $h \circ g = \mathtt{id}$ we can inject in any program point and for any variable x the sequence of instructions $P_{x,g}; P_{x,h}$ without changing the concrete semantics of the program.

Proposition 29. *Given a program P and two functions g and h as above, let $\tau(P)$ be the transformation of P obtained by inserting the command $P_{x,g}; P_{x,h}$ in the program P at any point, any number of times and for any program variable $x \in var(P)$. Then, for all $\mathbb{m} \in \mathbb{S}$: $[\![P]\!]\mathbb{m} = [\![\tau(P)]\!]\mathbb{m}$ holds.*

Proof. For all $\mathbb{m} \in \mathbb{S}$, we have that

$$\begin{aligned}
[\![P_{x,g}; P_{x,h}]\!]\mathbb{m} &= [\![P_{x,h}]\!]([\![P_{x,g}]\!]\mathbb{m}) \\
&= [\![P_{x,h}]\!](\mathbb{m}[x \mapsto g(\mathbb{m}(x))]) \\
&= \mathbb{m}[x \mapsto h(g(\mathbb{m}(x)))] \\
&= \mathbb{m}[x \mapsto \mathbb{m}(x)] \\
&= \mathbb{m}
\end{aligned}$$

Therefore for all $\mathbb{m} \in \mathbb{S}$, $[\![P_{x,g}; P_{x,h}]\!]\mathbb{m} = [\![\mathsf{skip}]\!]\mathbb{m}$. Then it is immediate to check that for any program P one has $[\![P]\!]\mathbb{m} = [\![\mathsf{skip}; P]\!]\mathbb{m} = [\![P_{x,g}; P_{x,h}; P]\!]\mathbb{m}$ and since the denotational semantics is compositional we get the thesis. *q.e.d.*

We design the functions g and h in such a way that we have

$$\alpha \circ [\![\mathsf{skip}]\!] = \alpha \circ [\![P_{x,g}; P_{x,h}]\!] < [\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}} \circ \alpha.$$

This allows us to transform any program P into the program $\tau(P) = P_{x,g}; P_{x,h}; P$ with

$$[\![\tau(P)]\!] = [\![P]\!] \text{ but } \alpha \circ [\![\tau(P)]\!] < [\![\tau(P)]\!]^{\mathcal{A}} \circ \alpha.$$

To prove that our transformation introduces incompleteness, as required by the condition $\alpha \circ [\![\tau(P)]\!] < [\![\tau(P)]\!]^{\mathcal{A}} \circ \alpha$, the functions g and h need to be defined in such a way that

$$\alpha([\![P_{x,g}; P_{x,h}]\!]S) < [\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S)$$

for some exactly represented element S (i.e., $S = (\gamma \circ \alpha)S$). The variable x must be chosen in such a way that $[\![P]\!]^{\mathcal{A}}$ is strictly monotone on x. Formally, let $S(x) = \{\mathbb{m}(x) \mid \mathbb{m} \in S\}$ denote the set of values that can be assigned to x by stores in S. We say that $[\![P]\!]^{\mathcal{A}}$ is strictly monotone on x for $S^\sharp \in \mathcal{A}$ if, taken any abstract store $T^\sharp \in \mathcal{A}$ such that $S^\sharp < T^\sharp$ and $\gamma(S^\sharp)(x) \subset \gamma(T^\sharp)(x)$, we have $[\![P]\!]^{\mathcal{A}}S^\sharp < [\![P]\!]^{\mathcal{A}}T^\sharp$. Note that this is always the case for a variable x not occurring in P when $[\![P]\!]^{\mathcal{A}}S^\sharp$ converges, since $[\![P]\!]^{\mathcal{A}}$ acts as the identity function on x because we assume that $\mathcal{A}$ is non-relational, and therefore variable finite. The next theorem shows that this is a sufficient condition to derive the incompleteness of $\tau(P)$.

Theorem 30. *Let g and h be such that $[\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S) > \alpha([\![P_{x,g}; P_{x,h}]\!]S)$ holds for some exactly represented element S. If $[\![P]\!]^{\mathcal{A}}$ is strictly monotone on variable x for $\alpha(S)$ then*

$$[\![P_{x,g}; P_{x,h}; P]\!]^{\mathcal{A}}\alpha(S) > \alpha([\![P_{x,g}; P_{x,h}; P]\!]S).$$

Proof. First notice that by Proposition 29 we have $\alpha([\![P_{x,g}; P_{x,h}]\!]S) = \alpha(S)$. Thus by hypothesis

$$[\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S) > \alpha([\![P_{x,g}; P_{x,h}]\!]S) = \alpha(S)$$

and thus

$$(\gamma \circ [\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}} \circ \alpha)S \supset (\gamma \circ \alpha)S.$$

Since $P_{x,g}$ and $P_{x,h}$ leaves the variables different from x unchanged, it must be the case that

$$((\gamma \circ [\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}} \circ \alpha)S)(x) \supset ((\gamma \circ \alpha)S)(x).$$

Since $[\![P]\!]^{\mathcal{A}}$ is strictly monotone on x for $\alpha(S)$, we have $[\![P]\!]^{\mathcal{A}}([\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S)) > [\![P]\!]^{\mathcal{A}}\alpha(S)$. By definition of the best correct abstraction, $[\![P]\!]^{\mathcal{A}}\alpha(S) \geq (\alpha \circ [\![P]\!] \circ \gamma \circ \alpha)S$. Since S is exactly represented we have $(\alpha \circ [\![P]\!] \circ \gamma \circ \alpha)S = \alpha([\![P]\!]S)$. By Proposition 29 we get $\alpha([\![P]\!]S) = \alpha([\![P_{x,g}; P_{x,h}; P]\!]S)$. Thus by the previous inequalities we obtain the result $[\![P_{x,g}; P_{x,h}; P]\!]^{\mathcal{A}}\alpha(S) > \alpha([\![P_{x,g}; P_{x,h}; P]\!]S)$. *q.e.d.*

There are many kinds of data-flow transformations applicable under different assumptions on the abstract domain. Here we present one transformation that is applicable in the most general case. It requires one recursive set $W' \in \wp(\mathbb{Z})$ such that W' is not exactly represented but there exists an exactly represented set W such that $W \simeq W'$ (i.e., W and W' are isomorphic) and $0 < |\mathbb{Z} \setminus W| \leq |\mathbb{Z} \setminus W'|$.

Given W, W' as above, we show how to define g and h. Assume that g_1 is any (computable) bijective correspondence from W into W' and g_2 is any (computable) injective correspondence from $\mathbb{Z} \setminus W$ into $\mathbb{Z} \setminus W'$ such that also $g_2(\mathbb{Z} \setminus W)$ is recursive. Then g can be defined as follows

$$g(x) \stackrel{\text{def}}{=} \begin{cases} g_1(x) & \text{if } x \in W \\ g_2(x) & \text{otherwise } (x \notin W) \end{cases}$$

The function h must behave as the left-inverse to g:

$$h(x) \stackrel{\text{def}}{=} \begin{cases} g_1^{-1}(x) & \text{if } x \in W' \\ g_2^{-1}(x) & \text{if } x \in g_2(\mathbb{Z} \setminus W) \\ v & \text{otherwise} \end{cases}$$

where v can be any element in $\mathbb{Z} \setminus W$. The following theorem shows that g and h defined as above satisfy the requirements for the application of Theorem 30.

Theorem 31. *Let $W' \in \wp(\mathbb{Z})$ be a recursive set that is not exactly represented in $\mathcal{A}$ and let W be an exactly represented set isomorphic to W' and such that $0 < |\mathbb{Z} \setminus W| \leq |\mathbb{Z} \setminus W'|$. Let $S = \mathbb{m}[x \mapsto W]$ for some store $\mathbb{m}$ and g and h be defined as above. Then: $[\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S) > \alpha([\![P_{x,g}; P_{x,h}]\!]S)$.*

Proof. By definition of the best correct abstraction we have

$$\begin{aligned} [\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S) &= ([\![P_{x,h}]\!]^{\mathcal{A}} \circ [\![P_{x,g}]\!]^{\mathcal{A}})\alpha(S) \\ &\geq ([\![P_{x,h}]\!]^{\mathcal{A}} \circ \alpha \circ [\![P_{x,g}]\!] \circ \gamma \circ \alpha)S. \end{aligned}$$

Since W is exactly represented and letting $S_1 = (\gamma \circ \alpha)\mathbb{m}$

$$\begin{aligned} ([\![P_{x,h}]\!]^{\mathcal{A}} \circ \alpha \circ [\![P_{x,g}]\!] \circ \gamma \circ \alpha)S &= ([\![P_{x,h}]\!]^{\mathcal{A}} \circ \alpha \circ [\![P_{x,g}]\!])(S_1[x \mapsto W]) \\ &= [\![P_{x,h}]\!]^{\mathcal{A}}\alpha(S_1[x \mapsto g(W)]) \\ &= [\![P_{x,h}]\!]^{\mathcal{A}}\alpha(S_1[x \mapsto W']) \\ &\geq (\alpha \circ [\![P_{x,h}]\!] \circ \gamma \circ \alpha)(S_1[x \mapsto W']). \end{aligned}$$

Since W' is not exactly represented there exists $v' \in \mathbb{Z} \setminus W'$ such that $(\gamma \circ \alpha)W' \supseteq W' \cup \{v'\}$. Thus

$$\begin{aligned} (\alpha \circ [\![P_{x,h}]\!] \circ \gamma \circ \alpha)(S_1[x \mapsto W']) &\geq \alpha([\![P_{x,h}]\!]S_1[x \mapsto W' \cup \{v'\}]) \\ &= \alpha(S_1[x \mapsto h(W') \cup \{h(v')\}]) \\ &= \alpha(S_1[x \mapsto W \cup \{h(v')\}]) \\ &> \alpha(S) \end{aligned}$$

because $h(v') \in \mathbb{Z} \setminus W$, the set W is exactly represented and $\alpha(S_1) = (\alpha \circ \gamma \circ \alpha)\mathbb{m} = \alpha(\mathbb{m})$. *q.e.d.*

Example 32. *Consider again the non-relational abstract domain of intervals* Int *on the integers* $\mathbb{Z}$ *from Example 25. The abstraction is induced by abstract values on the interval* Int. *Let* $\mathbb{N}^+ = \{n \mid n > 1\}$. *We apply our data-flow transformation considering the set* $W' = \mathbb{N}^+ \setminus \{2\}$ *which is not exactly represented in* Int. *Indeed, on this abstract domain* $W' \subset \gamma(\alpha(W')) = \mathbb{N}^+$. *We also fix* $W = \mathbb{N}^+$ *that is exactly represented using the interval* $[1, \infty]$. *The function* g *can be defined as*

$$g(k) \stackrel{def}{=} \begin{cases} k & if\, k < 2, \\ k+1 & if\, k \geq 2\,. \end{cases}$$

Note that the function g *defines an isomorphism from* W *to* W' *and an injection from* $\mathbb{Z} \setminus W$ *to* $\mathbb{Z} \setminus W'$ *(the value* 2 *is not in the range of* g*). The function* h *can be defined as*

$$h(k) \stackrel{def}{=} \begin{cases} k & if\, k \leq 1, \\ -10 & if\, k = 2, \\ k-1 & if\, k > 2. \end{cases}$$

Here the image of 2 *can be set to any number in* $\mathbb{Z} \setminus W$, *we choose* -10.

Consider again the program $P = x := x + 1 \in \mathbb{C}(\mathsf{Int})$ *that we have already shown to be complete in Example 25. Let* $\tau(P)$ *be the program obtained by injecting* $P_{x,g}; P_{x,h}$ *before the code of* P, *where:*

$$\begin{array}{ll} P_{x,g} = & \textbf{if } (x \geq 2) \textbf{ then } x := x + 1 \\ & \textbf{else skip} \\ P_{x,h} = & \textbf{if } (x = 2) \textbf{ then } x := -10 \\ & \textbf{else if } (x > 2) \textbf{ then } x := x - 1 \\ & \quad \textbf{else skip} \end{array}$$

Let us now prove that program $\tau(P)$ *is incomplete for the abstract domain of stores induced by the abstraction* Int *on variables values. Consider* $S = \mathbb{m}[x \mapsto \mathbb{N}^+]$ *for some store* $\mathbb{m}$, *we have*

$$\begin{array}{rcl} \alpha([\![\tau(P)]\!]S) & = & \alpha([\![x := x+1]\!]([\![P_{x,h}]\!]([\![P_{x,g}]\!]S))) \\ & = & \alpha([\![x := x+1]\!]([\![P_{x,h}]\!]\mathbb{m}[x \mapsto \{g(i) \mid i \in \mathbb{N}\}])) \\ & = & \alpha([\![x := x+1]\!]S) \\ & = & \alpha(\mathbb{m})[x \mapsto [2, \infty]]. \end{array}$$

On the other hand, we have $\alpha(S) = \alpha(\mathbb{m})[x \mapsto [1, \infty]]$ *and*

$$\begin{array}{rcl} [\![\tau(P)]\!]^{\mathcal{A}}\alpha(S) & = & [\![x := x+1]\!]^{\mathcal{A}}([\![P_{x,h}]\!]^{\mathcal{A}}([\![P_{x,g}]\!]^{\mathcal{A}}\alpha(S))) \\ & = & [\![x := x+1]\!]^{\mathcal{A}}([\![P_{x,h}]\!]^{\mathcal{A}}([\![P_{x,g}]\!]^{\mathcal{A}}\alpha(\mathbb{m})[x \mapsto [1, \infty]])) \\ & = & [\![x := x+1]\!]^{\mathcal{A}}([\![P_{x,h}]\!]^{\mathcal{A}}\alpha(\mathbb{m})[x \mapsto [1, \infty]]) \\ & = & [\![x := x+1]\!]^{\mathcal{A}}\alpha(\mathbb{m})[x \mapsto [-10, \infty]] \\ & = & \alpha(\mathbb{m})[x \mapsto [-9, \infty]]. \end{array}$$

As a final result, we show the impossibility of obtaining an incomplete program with respect to a non-relational non-trivial abstract domain $\mathcal{A}$ by applying pure data-flow transformations whenever the conditions of the above theorems are not satisfied. This proves that the conditions of Theorem 31 are tight for pure data-flow transformations.

Theorem 33. *It is not possible to define* $P_{x,g}$ *and* $P_{x,h}$ *such that, for every* P,

(1) $[\![P_{x,g}; P_{x,h}; P]\!] = [\![P]\!]$, *and*

(2) $[\![P_{x,g}; P_{x,h}; P]\!]^{\mathcal{A}}\alpha(S) > \alpha([\![P_{x,g}; P_{x,h}; P]\!]S)$ *for some* S,

if there does not exist an exactly represented set W and a non exactly represented element W' with $|W| = |W'|$ and $|\mathbb{Z} \setminus W| < |\mathbb{Z} \setminus W'|$.

Proof. Since the conditions 1–2 must hold for every program P, in particular they must hold for $P =$ **skip**. Therefore the conditions subsume:

(1) $[\![P_{x,g}; P_{x,h}]\!]\mathbb{m} = \mathbb{m}$ for every $\mathbb{m}$, and
(2) $[\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S) > \alpha(S)$ for some S.

From (1), it follows that g must be total and injective and that h must be injective over the range of g. Since (2) must hold, $\alpha(S)$ cannot be $\mathbb{S}$. Moreover, since the condition (2) is about the abstraction $\alpha(S)$, and $\gamma(\alpha(S))$ is always exactly represented, without loss of generality we can consider S to be exactly represented, i.e., $S = \gamma(\alpha(S))$. Let us consider the best correct abstraction

$$\begin{aligned}[\![P_{x,g}; P_{x,h}]\!]^{\mathcal{A}}\alpha(S) &\geq (\alpha \circ [\![P_{x,h}]\!] \circ \gamma \circ \alpha \circ [\![P_{x,g}]\!] \circ \gamma)\alpha(S) \\ &= (\alpha \circ [\![P_{x,h}]\!] \circ \gamma \circ \alpha \circ [\![P_{x,g}]\!])S.\end{aligned}$$

Let $W = S(x)$, $S' = [\![P_{x,g}]\!]S$ and $W' = g(W)$, so that $S' = S[x \mapsto W']$. Remember that g is injective. If W' was exactly represented then $(\gamma \circ \alpha)W' = W'$, $(\gamma \circ \alpha)S' = S'$ and $[\![P_{x,h}]\!]S' = S$ because h is injective on the range of g and left-inverse to it. Thus W' cannot be exactly represented and is isomorphic to W. Moreover, as g is total and injective, the image of $\mathbb{Z} \setminus W$ must be in $\mathbb{Z} \setminus W'$, meaning that $|\mathbb{Z} \setminus W| < |\mathbb{Z} \setminus W'|$. But this contradicts the hypothesis that such W and W' do not exist. *q.e.d.*

10 CONCLUSION

In this paper we proved that the equivalence induced by the abstract semantics on programs is an index set of partial recursive functions if and only if the abstraction is trivial. We considered the strongest possible scenario in order to establish when incompleteness can be injected. In particular the assumptions [H1] and [H3] ensure the existence of the best correct approximation for expressions and assignements, so that making this approximation incomplete would make incomplete any other weaker approximations, i.e., we proved that incompleteness can be injected in every program also when the abstraction is designed to be the most precise one. Note that we propose effective (control-flow and sometimes data-flow) program transformations to inject incompleteness and that these transformations are all based on the structure of the abstract domain. This result has important consequences in program analysis and abstract interpretation:

(1) It shows that any non-trivial abstraction of extensional (functional) properties of programs is susceptible to their intensional structure. This means that any non-trivial abstract interpretation always unveils implicitly also properties concerning the way programs are written. While true alarms only concern the extensional (functional) behaviour of the program, false alarms always concern their intensional structure. Stated in a different way: We can look at the log of alarms generated by an abstract interpreter to classify programs according to extensional—what they compute, and intensional—how they are implemented, similarity. This log is a footprint of the code analysed which, to the best of our knowledge, has never been used for program analysis, e.g., in the context of program similarity by encompassing both semantic and implementation similarity.

(2) Program analysis behaves precisely as other well known intensional properties of programs, like computational complexity [Asperti 2008]. This relates program analysis with computational complexity in an unexpected and remarkable way. The question whether these two fields can be unified under a unique formal setting and what properties and structures are shared by both is still an open question. Our paper is in this sense another step towards this ambitious goal: *What is left of the standard model of recursive functions when intensional aspects of computation are considered?*

(3) We concentrated our attention on the class $\overline{\mathbb{C}}(P, \mathcal{A})$. This is the space of action of any code protecting transformations whose aim is to foil program analysis and therefore foil any tool supporting reverse engineering. We proved that the set of all programs that are incomplete for any non-trivial abstraction $\mathcal{A}$, i.e., the set $\overline{\mathbb{C}(\mathcal{A})}$, is a very rich structure: A Turing complete language! This means that it is possible to build a compiler that compiles any program P into an equivalent program in $\overline{\mathbb{C}}(P, \mathcal{A})$, therefore justifying code transformations that protect code against program analysis. On the other side, the expressivity of the class $\mathbb{C}(\mathcal{A})$ of all programs that are complete for a non-trivial abstraction $\mathcal{A}$ is still obscure. We know by Theorem 16 that for terminating non-trivial program analyses we cannot find a many-to-one reduction of $\overline{\mathbb{C}}(P, \mathcal{A})$ into $\mathbb{C}(P, \mathcal{A})$. This implies that $\mathbb{C}(\mathcal{A})$ cannot be always Turing complete, otherwise by the first Futamura projection (e.g., see [Futamura 1999; Jones 2004]), we could build inside $\mathbb{C}(\mathcal{A})$ a compiler τ mapping any program in $\overline{\mathbb{C}}(P, \mathcal{A})$ into $\mathbb{C}(P, \mathcal{A})$, and conversely any program outside $\overline{\mathbb{C}}(P, \mathcal{A})$ into a program outside $\mathbb{C}(P, \mathcal{A})$. The question: *Given a non-trivial abstraction $\mathcal{A}$, what are the functions that we can program in $\mathbb{C}(\mathcal{A})$?* is still open. This question may have relevant applications in automating systematic false alarm removal by refactoring code snippets.

(4) The proofs of our results show that effective program transformations can be derived under a very weak hypothesis, what we called variable finiteness of an abstraction. The connection with code obfuscation is particularly interesting here. We gave the minimal conditions for the existence of universal (i.e., working for any program) control-flow transformations and proved that homomorphic data-flow obfuscations require strictly stronger hypothesis even for non-relational abstract domains. This shows an asymmetry in the fundamental code obfuscation strategies that deeply relies upon the foundations of abstract interpretation and computability. Abstract interpretation is not compositional, i.e., we typically lose precision when composing the abstract semantics of programs. For instance the composition of the best correct approximations of the semantics two programs is not the best correct approximation of the composition of their semantics. This is not the case when the abstract semantics is complete. Injecting incompleteness therefore exploits this fundamental aspect of abstract interpretation. Moreover pure control-flow transformations exploit dead code injection, i.e., the undecidability of termination, making any non-trivial abstraction to fail in detecting dead code. Pure data-flow transformations instead simply act on data homomorphically, without necessarily encroaching on undecidability. Of course hybrid control/data-flow transformations are used in practice in order to strengthen protection and avoid easy removal of dead code and opaque predicates. This is nothing else than finding a suitable program in $\overline{\mathbb{C}}(P, \mathcal{A})$ that maximises some metrics. Being $\overline{\mathbb{C}}(P, \mathcal{A})$ Turing complete, we believe that code obfuscation, which is nowadays mostly considered a cryptographic concept [Barak et al. 2012], can be fully reconciled with recursion theory and programming languages.

ACKNOWLEDGMENTS

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is supported by the Ministero dell'Università e della Ricerca Scientifica of Italy under Grant No. 201784YSZ5, PRIN2017 – ASPRA, and by the Fondazione Cariverona, Bando Ricerca 2017, under Grant: ATEN. The work of I. Garcia-Contreras is partially supported by MINECO TIN2015-67522-C3-1-R TRACES project, FPU grant 16/04811, and the Madrid P2018/TCS-4339 BLOQUES-CM program. The work of D. Pavlovic is partially supported by NSF and AFOSR.

REFERENCES

S. Abramsky. 2014. Intensionality, Definability and Computation. In *Johan van Benthem on Logic and Information Dynamics*, A. Baltag and S. Smets (Eds.). Springer, 121–142. https://doi.org/10.1007/978-3-319-06025-5_5

A. Asperti. 2008. The intensional content of Rice's theorem. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*, G. C. Necula and P. Wadler (Eds.). ACM, 113–119. https://doi.org/10.1145/1328438.1328455

B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.P. Vadhan, and K. Yang. 2012. On the (im)possibility of obfuscating programs. *Journal of the ACM* 59, 2 (2012), 6. https://doi.org/10.1145/2160158.2160159

A. M. Ben-Amram and N. D. Jones. 2000. Computational complexity via programming languages: constant factors do matter. *Acta Inf.* 37, 2 (2000), 83–120. https://doi.org/10.1007/s002360000038

R. Bruni, R. Giacobazzi, and R. Gori. 2018. Code obfuscation against abstraction refinement attacks. *Formal Asp. Comput.* 30, 6 (2018), 685–711. https://doi.org/10.1007/s00165-018-0462-6

C. Collberg and J. Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional.

C. Collberg, C. D. Thomborson, and D. Low. 1998. Manufactoring Cheap, Resilient, and Stealthy Opaque Constructs. In *Proc. of Conf. Record of the 25st ACM Symp. on Principles of Programming Languages* (*POPL '98*). ACM Press, 184–196. https://doi.org/10.1145/268946.268962

P. Cousot and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL '77)*. ACM Press, 238–252. https://doi.org/10.1145/512950.512973

P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (*POPL '79*). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

P. Cousot and R. Cousot. 2014. Abstract interpretation: past, present and future. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, 2014*, T. A. Henzinger and D. Miller (Eds.). ACM, 2:1–2:10. https://doi.org/10.1145/2603088.2603165

P. Cousot, R. Giacobazzi, and F. Ranzato. 2018. Program Analysis Is Harder Than Verification: A Computability Perspective. In *Computer Aided Verification - 30th International Conference, CAV 2018, Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, H. Chockler and G. Weissenbacher (Eds.), Vol. 10982. Springer, 75–95. https://doi.org/10.1007/978-3-319-96142-2_8

U. Dal Lago. 2011. A Short Introduction to Implicit Computational Complexity. In *Lectures on Logic and Computation - ESSLLI 2010 and ESSLLI 2011, Selected Lecture Notes (Lecture Notes in Computer Science)*, N. Bezhanishvili and V. Goranko (Eds.), Vol. 7388. Springer, 89–109. https://doi.org/10.1007/978-3-642-31485-8_3

M. Dalla Preda and R. Giacobazzi. 2009. Semantics-based code obfuscation by abstract interpretation. *Journal of Computer Security* 17, 6 (2009), 855–908. https://doi.org/10.1007/11523468_107

S. Drape, C. Thomborson, and A. Majumdar. 2007. Specifying Imperative Data Obfuscations. In *ISC - Information Security (Lecture Notes in Computer Science)*, J. A. Garay, et al. (Eds.), Vol. 4779. Springer Verlag, 299 – 314. https://doi.org/10.1007/978-3-540-75496-1_20

Y. Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

R. Giacobazzi. 2008. Hiding Information in Completeness Holes - New perspectives in code obfuscation and watermarking. In *Proc. of the 6th IEEE Int. Conferences on Software Engineering and Formal Methods (SEFM '08)*. IEEE Press, 7–20. https://doi.org/10.1109/SEFM.2008.41

R. Giacobazzi, N. D. Jones, and I. Mastroeni. 2012. Obfuscation by Partial Evaluation of Distorted Interpreters. In *Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12)*. ACM Press, 63–72. https://doi.org/10.1145/2103746.2103761

R. Giacobazzi, F. Logozzo, and F. Ranzato. 2015. Analyzing Program Analyses. In *Proc. of the 42nd ACM Symp. on Principles of Programming Languages (POPL '15)*. ACM Press, 261–273. https://doi.org/10.1145/2676726.2676987

R. Giacobazzi and I. Mastroeni. 2012. Making Abstract Interpretation Incomplete: Modeling the Potency of Obfuscation. In *Static Analysis - 19th International Symposium, SAS, 2012. Proc. (Lecture Notes in Computer Science)*, A. Miné and D. Schmidt (Eds.), Vol. 7460. Springer, 129–145. https://doi.org/10.1007/978-3-642-33125-1_11

R. Giacobazzi and I. Mastroeni. 2016. Making abstract models complete. *Mathematical Structures in Computer Science* 26, 4 (2016), 658–701. https://doi.org/10.1017/S0960129514000358

R. Giacobazzi, F. Ranzato, and F. Scozzari. 2000. Making Abstract Interpretations Complete. *Journal of the ACM* 47, 2 (March 2000), 361–416. https://doi.org/10.1145/333979.333989

N. D. Jones. 2004. Transformation by interpreter specialisation. *Science of Computer Programming* 52, 17(1) (2004), 307–339. https://doi.org/10.1016/j.scico.2004.03.010

G. A. Kavvos. 2017. On the Semantics of Intensionality. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Proceedings (Lecture Notes in Computer Science)*, J. Esparza and A. S. Murawski

(Eds.), Vol. 10203. 550–566. https://doi.org/10.1007/978-3-662-54458-7_32

V. Laviron and F. Logozzo. 2009. Refining Abstract Interpretation-Based Static Analyses with Hints. In *Proc. of APLAS'09 (Lecture Notes in Computer Science)*, Vol. 5904. Springer-Verlag, 343–358. https://doi.org/10.1007/978-3-642-10672-9_24

A. Majumdar, C. D. Thomborson, and S. Drape. 2006. A Survey of Control-Flow Obfuscations. In *Information Systems Security, Second International Conference, ICISS 2006, Kolkata, India, December 19-21, 2006, Proceedings (Lecture Notes in Computer Science)*, A. Bagchi and V. Atluri (Eds.), Vol. 4332. Springer, 353–356. https://doi.org/10.1007/11961635_26

N. Partush and E. Yahav. 2013. Abstract Semantic Differencing for Numerical Programs. In *Static Analysis - 20th International Symposium, SAS 2013. Proceedings (Lecture Notes in Computer Science)*, F. Logozzo and M. Fähndrich (Eds.), Vol. 7935. Springer, 238–258. https://doi.org/10.1007/978-3-642-38856-9_14

H.G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366. https://doi.org/10.1090/S0002-9947-1953-0053041-6

H. Rogers. 1992. *Theory of recursive functions and effective computability.* The MIT press.

A. Venet. 1996. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings (Lecture Notes in Computer Science)*, R. Cousot and D. A. Schmidt (Eds.), Vol. 1145. Springer, 366–382. https://doi.org/10.1007/3-540-61739-6_53

G. Winskel. 1993. *The formal semantics of programming languages: an introduction.* MIT press.