Program Analysis is Harder than Verification: A Computability Perspective

Patrick Cousot[1], Roberto Giacobazzi[2], and Francesco Ranzato[3]

[1] New York University, USA
[2] University of Verona, Italy and IMDEA Software Institute, Spain
[3] University of Padova, Italy

Abstract. We study from a computability perspective static program analysis, namely detecting sound program assertions, and verification, namely sound checking of program assertions. We first design a general computability model for domains of program assertions and corresponding program analysers and verifiers. Next, we formalize and prove an instantiation of Rice's theorem for static program analysis and verification. Then, within this general model, we provide and show a precise statement of the popular belief that program analysis is a harder problem than program verification: we prove that for finite domains of program assertions, program analysis and verification are equivalent problems, while for infinite domains, program analysis is strictly harder than verification.

1 Introduction

It is common to assume that program analysis is harder than program verification (e.g. [1,17,22]). The intuition is that this happens because in program analysis we need to synthesize a correct program invariant while in program verification we have *just* to check whether a given program invariant is correct. The distinction between checking a proof and computing a witness for that proof can be traced back to Leibniz [18] in his *ars iudicandi* and *ars inveniendi*, respectively representing the analytic and synthetic method. In Leibniz's *ars combinatoria*, the ars inveniendi is defined as the art of discovering "correct" questions while ars iudicandi is defined as the art of discovering "correct" answers. These foundational aspects of mathematical reasoning have a peculiar meaning when dealing with questions and answers concerning the behaviour of computer programs as objects of our investigation.

Our main goal is to define a general and precise model for reasoning on the computability aspects of the notions of (sound or complete) static analyser and verifier for generic programs (viz. Turing machines). Both static analysers and verifiers assume a given domain A of abstract program assertions, that may range from synctatic program properties (e.g., program sizes or LOCs) to complexity properties (e.g., number of execution steps in some abstract machine) and all the semantic properties of the program behaviour (e.g., value range of program variables or shape of program memories). A program analyser is defined to be any total computable (i.e., total recursive) function that for any program P returns an assertion a_P in A, which is sound when the concrete meaning of the assertion a_P includes P. Instead, a program verifier is a (total) decision

procedure which is capable of checking whether a given program P satisfies a given assertion a ranging in A, answering "true" or "don't know", which is sound when a positive check of a for P means that the concrete meaning of the assertion a includes P. Completeness, which coupled with soundness is here called precision, for a program analyser holds when, for any program P, it returns the strongest assertion in A for P, while a program verifier is called precise if it is able to prove any true assertion in A for a program P. This general and minimal model allows us to extend to static program analysis and verification some standard results and methods of computability theory. We provide an instance of the well-known Rice's Theorem [29] for generic analysers and verifiers, by proving that sound and precise analysers (resp. verifiers) exist only for trivial domains of assertions. This allows us to generalise known results about undecidability of program analysis, such as the undecidability of the meet over all paths (MOP) solution for monotone dataflow analysis frameworks [15], making them independent from the structure of the domain of assertions. Then, we define a model for comparing the relative "verification power" of program analysers and verifiers. In this model, a verifier $\mathcal{V}$ on a domain A of assertions is more precise than an analyser $\mathcal{A}$ on the same domain A when any assertion a in A which can be proved by $\mathcal{A}$ for a program P — this means that the output of the analyser $\mathcal{A}(P)$ is stronger than the assertion a — can be also proved by $\mathcal{V}$. Conversely, $\mathcal{A}$ is more precise than $\mathcal{V}$ when any assertion a proved by $\mathcal{V}$ can be also proved by $\mathcal{A}$. We prove that while it is always possible to constructively transform a program analyser into an equivalent verifier (i.e., with the same verification power), the converse does not hold in general. In fact, we first show that for *finite* domains of assertions, any "reasonable" verifier can be constructively transformed into an equivalent analyser, where reasonable means that the verifier $\mathcal{V}$ is: (i) nontrivial: for any program, $\mathcal{V}$ is capable to prove some assertion, possibly a trivially true assertion; (ii) monotone: if $\mathcal{V}$ proves an assertion a and a is stronger than a' then $\mathcal{V}$ is also capable of proving a'; (iii) logically meet-closed: if $\mathcal{V}$ proves both a_1 and a_2 and the logical conjunction $a_1 \wedge a_2$ is a representable assertion then $\mathcal{V}$ is also capable of proving it. Next, we prove the following impossibility result: for any *infinite* abstract domain of assertions A, no constructive reduction from reasonable verifiers on A to equivalent analysers on A is possible. This provides, to the best of our knowledge, the first formalization of the common folklore that program analysis is harder than program verification.

2 Background

We follow the standard terminology and notation for sets and computable functions in recursion theory (e.g., [12,26,30]). If X and Y are sets then $X \to Y$ and $X \pfun Y$ denote, respectively, the set of all total and partial functions from X to Y. If $f : X \pfun Y$ then $f(x)\!\downarrow$ and $f(x)\!\uparrow$ mean that f is defined/undefined on $x \in X$. Hence $\mathrm{dom}(f) = \{x \in X \mid f(x)\!\downarrow\}$. If $S \subseteq Y$ then $f(x) \in S$ denotes the implification $f(x)\!\downarrow \Rightarrow f(x) \in S$. If $f, g : X \pfun Y$ then $f = g$ means that $\mathrm{dom}(f) = \mathrm{dom}(g)$ and for any $x \in \mathrm{dom}(f) = \mathrm{dom}(g)$, $f(x) = g(x)$. The set of all partial (total) recursive functions on natural numbers is denoted by $\mathbb{N} \stackrel{\mathrm{r}}{\pfun} \mathbb{N}$ ($\mathbb{N} \stackrel{\mathrm{r}}{\to} \mathbb{N}$). Recall that $A \subseteq \mathbb{N}$ is a recursively enumerable (r.e., or semidecidable) set if $A = \mathrm{dom}(f)$ for some $f \in \mathbb{N} \stackrel{\mathrm{r}}{\pfun} \mathbb{N}$, while $A \subseteq \mathbb{N}$ is a recursive (or decidable) set if both A and its complement $\bar{A} = \mathbb{N} \setminus A$

are recursively enumerable, and this happens when there exists $f \in \mathbb{N} \xrightarrow{r} \mathbb{N}$ such that $f = \lambda n.\, n \in A \,\mathbf{?}\, 1 \,\mathbf{:}\, 0$.

Let Prog denote some deterministic programming language which is Turing complete. More precisely, this means that for any partial recursive function $f : \mathbb{N} \xrightarrow{r} \mathbb{N}$ there exists a program $P \in \mathrm{Prog}$ such that $[\![P]\!] \cong f$, where $[\![P]\!] : D \nrightarrow D$ is a denotational input/output semantics of P on a domain D of input/output values for Prog, where: undefinedness encodes nontermination and $\cong$ means equality up to some recursive encoding $\mathrm{enc} : D \xrightarrow{r} \mathbb{N}$ and decoding $\mathrm{dec} : \mathbb{N} \xrightarrow{r} D$ functions, i.e., $f = \mathrm{enc} \circ [\![P]\!] \circ \mathrm{dec}$. We also assume a small-step transition relation $\Rightarrow \subseteq (\mathrm{Prog} \times D) \times ((\mathrm{Prog} \times D) \cup D)$ for Prog defining an operational semantics which is functionally equivalent to the denotational semantics: $\langle P, i\rangle \Rightarrow^* o$ iff $[\![P]\!]i = o$. By an abuse of notation, we will identify the input/output semantics of a program P with the partial recursive function computed by P, i.e., we will consider programs $P \in \mathrm{Prog}$ whose input/output semantics is a partial recursive function $[\![P]\!] : \mathbb{N} \xrightarrow{r} \mathbb{N}$, so that, by Turing completeness, $\{[\![P]\!] : \mathbb{N} \xrightarrow{r} \mathbb{N} \mid P \in \mathrm{Prog}\} = \mathbb{N} \xrightarrow{r} \mathbb{N}$.

3 Abstract Domains

Static program analysis and verification are always defined with respect to a given (denumerable) domain of program assertions, that we call here *abstract domain* [7], where the meaning of assertions is formalized by a function which induces a logical implication relation between assertions.

Definition 3.1 (Abstract Domain). An *abstract domain* is a tuple $\langle A, \gamma, \leq_\gamma\rangle$ such that:

(1) A is any denumerable set;
(2) $\gamma : A \to \wp(\mathrm{Prog})$ is any function;
(3) $\leq_\gamma \triangleq \{(a_1, a_2) \in A \times A \mid \gamma(a_1) \subseteq \gamma(a_2)\}$ is a decidable relation.

An abstract element $a \in A$ such that $\gamma(a) = \mathrm{Prog}$ is called an *abstract top*, while a is called an *abstract bottom* when $\gamma(a) = \varnothing$. □

The elements of A are called assertions or abstract values, γ is called concretization function (this may also be a nonrecursive function, which is typical of abstract domains representing semantic program properties), and $\leq_\gamma$ is called the implication or approximation relation of A. Thus, in this general model, a program assertion $a \in A$ plays the role of some abstract representation of any program property $\gamma(a) \in \wp(\mathrm{Prog})$, while the comparison relation $a_1 \leq_\gamma a_2$ holds when a_1 is a stronger (or more precise) property than a_2. Let us also observe that, as a limit case, Definition 3.1 allows an abstract domain to be empty, that is, the tuple $\langle\varnothing, \varnothing, \varnothing\rangle$ satisfies the definition of abstract domain, where $\varnothing$ denotes both the empty set, the empty function (i.e., the unique subset of $\varnothing \times \varnothing$) and the empty relation.

Example 3.2 Let us give some simple examples of abstract domains.

(1) Consider $A = \mathbb{N}$ with $\gamma(n) \triangleq \{P \in \mathrm{Prog} \mid \mathrm{size}(P) \leq n\}$, where $\mathrm{size} : \mathrm{Prog} \to \mathbb{N}$ is some computable program size function. Here, $\leq_\gamma$ is clearly decidable and coincides with the partial order $\leq_{\mathbb{N}}$ on numbers.

(2) Consider $A = \mathbb{N}$ with $\gamma(n) \triangleq \{P \in \mathrm{Prog} \mid \forall i.\exists o, k.(\langle P, i\rangle \Rightarrow^k o)\ \&\ k \leq n\}$, i.e., n represents all the programs which, given any input, terminate in at most n steps. Here again, $n \leq_\gamma m$ iff $n \leq_{\mathbb{N}} m$, so that $\leq_\gamma$ is decidable.
(3) Consider $A = \mathbb{N}$ with $\gamma(n) \triangleq \{P \in \mathrm{Prog} \mid \forall i \in [0, n].\exists o.\ \langle P, i\rangle \Rightarrow^* o\}$, that is, n represents all the programs which terminate for any input $i \leq n$. Once again, $n \leq_\gamma m$ iff $n \leq_{\mathbb{N}} m$.
(4) Consider $A = \mathbb{N}$ with $\gamma(n) \triangleq \{P \in \mathrm{Prog} \mid \forall i \in \mathbb{N}.\, [\![P]\!](i) = o \Rightarrow o \leq n\}$, that is, n represents those programs which, in case of termination, give an output o bounded by n. Again, $n \leq_\gamma m$ iff $n \leq_{\mathbb{N}} m$.
(5) Consider $A = \mathbb{N} \stackrel{\mathrm{r}}{\to} \mathbb{N}$ with $\gamma(g) \triangleq \{P \in \mathrm{Prog} \mid \forall i.\ \big(g(i)\!\downarrow \Rightarrow (\exists o, k.\langle P, i\rangle \Rightarrow^k o,\ k \leq g(i))\big) \wedge \big((\exists o, k.\langle P, i\rangle \Rightarrow^k o) \Rightarrow g(i)\!\downarrow,\ k \leq g(i)\big)\}$, that is, g represents those programs whose time complexity is bounded by the function g. Here, $g \leq_\gamma g'$ iff $\forall i.\ g(i)\!\downarrow \Rightarrow (g'(i)\!\downarrow\ \wedge\ g(i) \leq g'(i))$. □

Definition 3.1 does not require injectivity of the concretization function γ, thus multiple assertions could have the same meaning. Two abstract values $a_1, a_2 \in A$ are called equivalent when $\gamma(a_1) = \gamma(a_2)$. Let us observe that since $\leq_\gamma$ is required to be decidable, the equivalence $\gamma(a_1) = \gamma(a_2)$ is decidable as well. For example, for the well-known numerical abstract domain of convex polyhedra [11] represented through linear constraints between program variables, we may well have multiple representations P_1 and P_2 for the same polyhedron, e.g., $P_1 = \{x = z, z \leq y\}$ and $P_2 = \{x = z, x \leq y\}$ both represent the same polyhedron. Thus, in general, an abstract domain A is not required to be partially ordered by $\leq_\gamma$. On the other hand, the relation $\leq_\gamma$ is clearly a preorder on A. The only basic requirement is that for any pair of abstract values $a_1, a_2 \in A$, one can decide if a_1 is a more precise program assertion than a_2, i.e., if $\gamma(a_1) \subseteq \gamma(a_2)$ holds. In this sense we do not require that a partial order $\leq$ is defined a priori on A and that γ is monotone w.r.t. $\leq$, since for our purposes it is enough to consider the preorder $\leq_\gamma$ induced by γ. If instead A is endowed with a partial order $\leq_A$ and A is defined in abstract interpretation [7,8] through a Galois insertion based on the concretization map γ, then it turns out that $\gamma(a_1) \subseteq \gamma(a_2) \Leftrightarrow a_1 \leq_A a_2$ holds, so that the decidability of the relation $\leq_\gamma = \{(a_1, a_2) \in A \times A \mid \gamma(a_1) \subseteq \gamma(a_2)\}$ boils down to the decidability of the partial order relation $\leq_A$. As an example, it is well known that the abstract domain of polyhedra does not admit a Galois insertion [11], nevertheless its induced preorder relation $\leq_\gamma$ is decidable: for example, for polyhedra represented by linear constraints, there exist algorithms for deciding if $\gamma(P_1) \subseteq \gamma(P_2)$ for any pair of convex polyhedra representations P_1 and P_2 (see e.g. [23, Section 5.3]).

3.1 Abstract Domains in Abstract Interpretation

An abstract domain in standard abstract interpretation [7,8,9] is usually defined by a poset $\langle A, \leq_A\rangle$ containing a top element $\top \in A$ and a concretization map $\gamma_A : A \to \wp(\mathrm{Dom})$, where Dom denotes some concrete semantic domain (e.g., program stores or program traces), such that: (a) A is machine representable, namely the abstract elements of A are encoded by some data structures (e.g., tuples, vectors, lists, matrices, etc.), and some algorithms are available for deciding if $a_1 \leq_A a_2$ holds; (b) $a_1 \leq_A a_2 \Leftrightarrow \gamma_A(a_1) \subseteq \gamma_A(a_2)$ holds (this equivalence always holds for Galois

insertions); (c) $\gamma_A(\top) = \mathrm{Dom}$. Let us point out that Definition 3.1 is very general since the concretization of an abstract value can be any program property, possibly a purely syntactic property or some space or time complexity property, as in the simple cases of Example 3.2 (1)-(2)-(5).

Let $\gamma_A : A \to \wp(\mathrm{Dom})$ and assume that Dom is defined by program stores, namely $\mathrm{Dom} \triangleq \mathrm{Var} \to \mathrm{Val}$, where Var is a finite set of program variables and Val is a corresponding denumerable set of values. Since $\mathrm{Var} \to \mathrm{Val}$ has a finite domain and a denumerable range, we can assume a recursive encoding of finite tuples of values into natural numbers $\mathbb{N}$, i.e. $\mathrm{Var} \to \mathrm{Val} \cong \mathbb{N}$, and define $\gamma_A : A \to \wp(\mathbb{N})$. This is equivalent assuming that programs have one single variable, say x, which may assume tuples of values in Val. A set of numbers $\gamma_A(a) \in \wp(\mathbb{N})$ is meant to represent a property of the values stored in the program variable x at the end of the program execution, that is, if the program terminates its execution then the variable x stores a value in $\gamma_A(a)$. Hence, as usual, the property $\varnothing \in \wp(\mathbb{N})$ means that the program does not correctly terminate its execution either by true nontermination or by some run-time error, namely, that the exit program point is not reachable. For simplicity, we do not consider intermediate program points and assertions in our semantics.

For an abstract domain $\langle A, \gamma_A, \leq_A \rangle$ in standard abstract interpretation, the corresponding concretization function $\gamma : A \to \wp(\mathrm{Prog})$ of Definition 3.1 is defined as:

$$\gamma(a) \triangleq \{P \in \mathrm{Prog} \mid \forall i \in \mathbb{N}.\, [\![P]\!](i) \in \gamma_A(a)\}$$

where we recall that $[\![P]\!](i) \in \gamma_A(a)$ means $[\![P]\!](i) = o \Rightarrow o \in \gamma_A(a)$. Hence, if A contains top $\top_A$ and bottom $\bot_A$ such that $\gamma_A(\top_A) = \mathbb{N}$ and $\gamma_A(\bot_A) = \varnothing$ then $\gamma(\top_A) = \mathrm{Prog}$ and $\gamma(\bot_A) = \{P \in \mathrm{Prog} \mid P \text{ never terminates}\}$. Moreover, since γ_A is monotonic, we have that γ is monotonic as well. The fact that all the elements in A are machine representable boils down to the requirement that A is a recursive set, while the binary preorder relation $\leq_\gamma$ is decidable because $a_1 \leq_A a_2 \Leftrightarrow \gamma(a_1) \subseteq \gamma(a_2)$ holds and $\leq_A$ is decidable. This therefore defines an abstract domain according to Definition 3.1.

In this simple view of the abstract domain A, there is no input property for the variable x, meaning that at the beginning x may store any value. It is easy to generalize the above definition by requiring an input abstract property in A for x, so that the abstract domain is a Cartesian product $A \times A$ together with a concretization $\gamma^{i/o} : A \times A \to \wp(\mathrm{Prog})$ defined as follows:

$$\gamma^{i/o}(\langle a_i, a_o \rangle) \triangleq \{P \in \mathrm{Prog} \mid \forall i \in \mathbb{N}.\, i \in \gamma_A(a_i) \Rightarrow [\![P]\!](i) \in \gamma_A(a_o)\}.$$

This is a generalization since, for any $a \in A$, we have that $\gamma(a) = \gamma^{i/o}(\langle \top_A, a \rangle)$.

Example 3.3 (Interval Abstract Domain) Let Int be the standard interval domain [7] restricted to natural numbers in $\mathbb{N}$, endowed with the standard subset ordering:

$$\mathrm{Int} \triangleq \{[a, b] \mid a, b \in \mathbb{N},\ a \leq b\} \cup \{\bot_{\mathrm{Int}}\} \cup \{[a, +\infty) \mid a \in \mathbb{N}\}$$

with concretization $\gamma_{\mathrm{Int}} : \mathrm{Int} \to \wp(\mathbb{N})$, where $\gamma_{\mathrm{Int}}(\bot_{\mathrm{Int}}) = \varnothing$, $\gamma_{\mathrm{Int}}([a, b]) = [a, b]$ and $\gamma_{\mathrm{Int}}([0, +\infty)) = \mathbb{N}$, so that $[0, +\infty)$ is also denoted by $\top_{\mathrm{Int}}$. Thus, here, for the

concretization function $\gamma : \mathrm{Int} \to \wp(\mathrm{Prog})$ we have that: $\gamma(\top_{\mathrm{Int}}) = \mathrm{Prog}$, $\gamma(\bot_{\mathrm{Int}}) = \{P \in \mathrm{Prog} \mid \forall i.\, [\![P]\!](i)\uparrow\}$, $\gamma([a, +\infty)) = \{P \in \mathrm{Prog} \mid \forall i \in \mathbb{N}.\, [\![P]\!](i)\downarrow \Rightarrow [\![P]\!](i) \geq a\}$. We also have the input/output concretization $\gamma^{i/o} : \mathrm{Int} \times \mathrm{Int} \to \wp(\mathrm{Prog})$, where

$$\gamma^{i/o}(\langle I, J\rangle) \triangleq \{P \in \mathrm{Prog} \mid \forall i \in \mathbb{N}.\, i \in \gamma_{\mathrm{Int}}(I) \Rightarrow [\![P]\!](i) \in \gamma_{\mathrm{Int}}(J)\}. \qquad \square$$

4 Program Analysers and Verifiers

In our model, the notions of program analyser and verifier are as general as possible.

Definition 4.1 (Program Analyser). Given an abstract domain $\langle A, \gamma, \leq_\gamma\rangle$, a *program analyser* on A is any total recursive function $\mathcal{A} : \mathrm{Prog} \to A$.
The set of analysers on a given abstract domain A will be denoted by $\mathbb{A}_A$.
An analyser $\mathcal{A} \in \mathbb{A}_A$ is *sound* if for any $P \in \mathrm{Prog}$ and $a \in A$,

$$\mathcal{A}(P) \leq_\gamma a \;\Rightarrow\; P \in \gamma(a)$$

while $\mathcal{A}$ is *precise* if it is also complete, i.e., if the reverse implication also holds:

$$P \in \gamma(a) \;\Rightarrow\; \mathcal{A}(P) \leq_\gamma a. \qquad \square$$

Notice that this definition of soundness is equivalent to the standard notion of sound static analysis, namely, for any program P, $\mathcal{A}(P)$ always outputs a program assertion which is satisfied by P, i.e., $P \in \gamma(\mathcal{A}(P))$. Let us also note that on the empty abstract domain $\varnothing$, no analyser can be defined simply because there exists no function in $\mathrm{Prog} \to \varnothing$. Instead, for a singleton abstract domain $A_\bullet \triangleq \{\bullet\}$, if $\mathcal{A} \in \mathbb{A}_{A_\bullet}$ is sound then $\gamma(\bullet) = \mathrm{Prog}$, so that $\bullet$ is necessarily an abstract top. Also, if the abstract domain A contains a top abstract value $\top_A \in A$ then, as expected, $\lambda P.\top_A$ is a trivially sound analyser on A. Finally, we observe that if $\mathcal{A}_1$ and $\mathcal{A}_2$ are both precise on the same abstract domain then we have $\mathcal{A}_1 =_\gamma \mathcal{A}_2$, meaning that $\mathcal{A}_1$ and $\mathcal{A}_2$ coincide up to equivalent abstract values, i.e., $\gamma \circ \mathcal{A}_1 = \gamma \circ \mathcal{A}_2$. In fact, for any $P \in \mathrm{Prog}$, we have that $P \in \gamma(\mathcal{A}_2(P))$ implies $\gamma(\mathcal{A}_1(P)) \subseteq \gamma(\mathcal{A}_2(P))$ and $P \in \gamma(\mathcal{A}_1(P))$ implies $\gamma(\mathcal{A}_2(P)) \subseteq \gamma(\mathcal{A}_1(P))$, so that $\mathcal{A}_1 =_\gamma \mathcal{A}_2$.

Example 4.2 Software metrics static analysers [35] deal with nonsemantic program properties, such as the domain in Example 3.2 (1). Bounded model checking [4,34] handles program properties such as those encoded by the domains of Example 3.2 (2)-(3). Complexity bound analysers such as [32,36] cope with domains of properties such as those in Example 3.2 (4)-(5). Numerical abstract domains used in program analysis (see [23]) include the interval abstraction described in Example 3.3. $\square$

Definition 4.3 (Program Verifier). Given an abstract domain $\langle A, \gamma, \leq_\gamma\rangle$, a *program verifier* on A is any total recursive function $\mathcal{V} : \mathrm{Prog} \times A \to \{\mathbf{t}, \mathbf{?}\}$.
The set of verifiers on a given abstract domain A will be denoted by $\mathbb{V}_A$.
A verifier $\mathcal{V} \in \mathbb{V}_A$ is *sound* if for any $P \in \mathrm{Prog}$ and $a \in A$,

$$\mathcal{V}(P, a) = \mathbf{t} \;\Rightarrow\; P \in \gamma(a)$$

while $\mathcal{V}$ is *precise* if it is also complete, i.e., if the reverse implication also holds:

$$P \in \gamma(a) \;\Rightarrow\; \mathcal{V}(P, a) = \mathbf{t}.$$

A verifier $\mathcal{V} \in \mathbb{V}_A$ is *nontrivial* if for any program there exists at least one assertion which $\mathcal{V}$ is able to prove, i.e., for any $P \in \mathrm{Prog}$ there exists some $a \in A$ such that $\mathcal{V}(P, a) = \mathbf{t}$. Also, a verifier is defined to be *trivial* when it is not nontrivial.
A verifier $\mathcal{V} \in \mathbb{V}_A$ is *monotone* when the verification algorithm is monotone w.r.t. $\leq_\gamma$, i.e., $(\mathcal{V}(P, a) = \mathbf{t} \;\wedge\; a \leq_\gamma a') \;\Rightarrow\; \mathcal{V}(P, a') = \mathbf{t}$. □

Remark 4.4 Let us observe some straight consequences of Definition 4.3.
(1) Notice that for all nonempty abstract domains A, $\lambda(P, a).\,\mathbf{?}$ is a legal and vacuously sound verifier. Also, if $A = \varnothing$ is the empty abstract domain then the empty verifier $\mathcal{V} : \mathrm{Prog} \times \varnothing \to \{\mathbf{t}, \mathbf{?}\}$ (namely, the function with empty graph) is trivially precise.
(2) Let us observe that if $\mathcal{V}$ is nontrivial and monotone then $\mathcal{V}$ is able to prove any abstract top: in fact, if $\top \in A$ and $\gamma(\top) = \mathrm{Prog}$ then, for any $P \in \mathrm{Prog}$, since there exists some $a \in A$ such that $\mathcal{V}(P, a) = \mathbf{t}$ and $a \leq_\gamma \top$, then, by monotonicity, $\mathcal{V}(P, \top) = \mathbf{t}$.
(3) Note that if a verifier $\mathcal{V}$ is precise then $\mathcal{V}(P, a) = \mathbf{?} \Leftrightarrow P \notin \gamma(a)$, so that in this case an output $\mathcal{V}(P, a) = \mathbf{?}$ always means that P does not satisfy the property a.
(4) Finally, if $\mathcal{V}_1$ and $\mathcal{V}_2$ are precise on the same abstract domain then $\mathcal{V}_1(P, a) = \mathbf{t} \Leftrightarrow P \in \gamma(a) \Leftrightarrow \mathcal{V}_2(P, a) = \mathbf{t}$, so that $\mathcal{V}_1 = \mathcal{V}_2$. □

Example 4.5 Program verifiers abund in literature, e.g., [3,21,27]. For example, [13] aims at complexity verification on domains like that in Example 3.2 (5) while reachability verifiers like [33] can check numerical properties of program variables such as those of Example 3.3. □

5 Rice's Theorem for Static Program Analysis and Verification

Classical Rice's Theorem in computability theory [26,29,30] states that an extensional property $\mathit{\Pi} \subseteq \mathbb{N}$ of an effective numbering $\{\varphi_n \mid n \in \mathbb{N}\} = \mathbb{N} \stackrel{\cdot}{\to} \mathbb{N}$ of partial recursive functions is a recursive set if and only if $\mathit{\Pi} = \varnothing$ or $\mathit{\Pi} = \mathbb{N}$, i.e., $\mathit{\Pi}$ is trivial. Let us recall that $\mathit{\Pi} \subseteq \mathbb{N}$ is extensional when $\varphi_n = \varphi_m$ implies $n \in \mathit{\Pi} \;\Leftrightarrow\; m \in \mathit{\Pi}$. When dealing with program properties rather than indices of partial recursive functions, i.e., when $\mathit{\Pi} \subseteq \mathrm{Prog}$, Rice's Theorem states that any nontrivial semantic program property is undecidable (see [28] for a statement of Rice's Theorem tailored for program properties). It is worth recalling that Rice's Theorem has been extended by Asperti [2] through an interesting generalization to so-called "complexity cliques", namely nonextensional program properties which may take into account the space or time complexity of programs: for example, the abstract domain of Example 3.2 (5) is not extensional but when logically "intersected" with an extensional domain (i.e., it is a product domain $A_1 \times A_2$ where the concretization function is the set intersection $\lambda\langle a_1, a_2\rangle.\gamma_1(a_1) \cap \gamma_2(a_2)$) falls into this generalized version of Rice's Theorem.

In the following, we provide an instantiation of Rice's Theorem to sound static program analysis and verification by introducing a notion of extensionality for abstract

domains. Abstract domains commonly used in abstract interpretation turn out to be extensional, when they are used for approximating the input/output behaviour of programs. For example, if a sound abstract interpretation of a program P in the interval abstract domain computes as abstract output a program assertion such as $x \in [1, 5]$ and $y \in [2, +\infty)$ then this assertion is a sound abstract output for any other program Q having the same input/output behaviour of P.

Definition 5.1 (Extensional Abstract Domain). An abstract domain $\langle A, \gamma, \leq_\gamma \rangle$ is *extensional* when for any $a \in A$, $\gamma(a) \subseteq \mathrm{Prog}$ is an extensional program property, namely, if $[\![P]\!] = [\![Q]\!]$ then $P \in \gamma(a) \Leftrightarrow Q \in \gamma(a)$. □

As usual, the intuition is that an extensional program property depends exclusively on the input/output program semantics $[\![\cdot]\!]$. As a simple example, the domains of Example 3.2 (3)-(4) are extensional while the domains of Example 3.2 (1)-(2)-(5) are not.

Definition 5.2 (Trivial Abstract Domain). An abstract domain $\langle A, \gamma, \leq_\gamma \rangle$ is *trivial* when A contains abstract bottom or top elements only, i.e., for any $a \in A$, $\gamma(a) \in \{\varnothing, \mathrm{Prog}\}$. □

Definition 5.2 allows 4 possible types for a trivial abstract domain A: (1) $A = \varnothing$; (2) A is nonempty and consists of bottom elements only, i.e., $A \neq \varnothing$ and for all $a \in A$, $\gamma(a) = \varnothing$; (3) A is nonempty and consists of top elements only, i.e., $A \neq \varnothing$ and for all $a \in A$, $\gamma(a) = \mathrm{Prog}$; (4) A satisfies (2) and (3), i.e., A contains both bottom and top elements.

Theorem 5.3 (Rice's Theorem for Program Analysis). *Let $\langle A, \gamma, \leq_\gamma \rangle$ be an extensional abstract domain and let $\mathcal{A} \in \mathbb{A}_A$ be a sound analyser. Then, $\mathcal{A}$ is precise iff A is trivial.*

Proof. Since we assume the existence of a sound analyser $\mathcal{A} \in \mathbb{A}_A$ on the extensional abstract domain A, observe that necessarily $A \neq \varnothing$.

Assume that A is trivial. We have to show that for any $a \in A$ and $P \in \mathrm{Prog}$, $\mathcal{A}(P) \leq_\gamma a \Leftrightarrow P \in \gamma(a)$. Assume that $P \in \gamma(a)$ for some $a \in A$. Then, we have that $\gamma(a) \neq \varnothing$, so that, since A is trivial, it must necessarily be that $\gamma(a) = \mathrm{Prog}$. By soundness of $\mathcal{A}$, $P \in \gamma(\mathcal{A}(P))$, so that, since A is trivial, $\gamma(\mathcal{A}(P)) = \mathrm{Prog}$. Hence, we have that $\gamma(\mathcal{A}(P)) = \gamma(a)$, thus implying $\mathcal{A}(P) \leq_\gamma a$. On the other hand, if $\mathcal{A}(P) \leq_\gamma a$ then $\gamma(\mathcal{A}(P)) \subseteq \gamma(a)$, so that, since, by soundness of $\mathcal{A}$, $P \in \gamma(\mathcal{A}(P))$, we also have that $P \in \gamma(a)$.

Conversely, assume now that $\mathcal{A}$ is precise, namely, $P \in \gamma(a)$ iff $\mathcal{A}(P) \leq_\gamma a$. Thus, since $\mathcal{A}$ is a total recursive function and $\leq_\gamma$ is decidable, we have that, for any $a \in A$, $P \in^? \gamma(a)$ is decidable. Since $\gamma(a)$ is an extensional program property, by Rice's Theorem, $\gamma(a)$ must necessarily be trivial, i.e., $\gamma(a) \in \{\varnothing, \mathrm{Prog}\}$. This means that the abstract domain A is trivial. □

Rice's Theorem for program analysis can be applied to several abstract domains. Due to lack of space, we just mention that the well-known undecidability of computing the meet over all paths (MOP) solution for a monotone dataflow analysis problem,

proved by Kam and Ullman [15, Section 6] by resorting to undecidability of Post's Correspondence Problem, can be derived as a simple consequence of Theorem 5.3.

Along the same lines of Theorem 5.3, Rice's Theorem can be instantiated to program verification as follows.

Theorem 5.4 (Rice's Theorem for Program Verification). *Let $\langle A, \gamma, \leq_\gamma \rangle$ be an extensional abstract domain and let $\mathcal{V} \in \mathbb{V}_A$ be a sound, nontrivial and monotone verifier. Then, $\mathcal{V}$ is precise iff A is trivial.*

Proof. Let A be an extensional abstract domain and $\mathcal{V} \in \mathbb{V}_A$ be sound and nontrivial. If $A = \varnothing$ then A is trivial while the only possible verifier $\mathcal{V} : \mathrm{Prog} \times \varnothing \to \{\mathbf{t}, \mathbf{?}\}$ is the empty verifier, which is vacuously precise but it is not nontrivial. Thus, $A \neq \varnothing$ holds.

Assume that $\mathcal{V}$ is precise, that is, $P \in \gamma(a)$ iff $\mathcal{V}(P, a) = \mathbf{t}$. Hence, since $\mathcal{V}$ is a total recursive function, $\mathcal{V}(P, a) =^? \mathbf{t}$ is decidable, so that $P \in^? \gamma(a)$ is decidable as well. As in the proof of Theorem 5.3, since $\gamma(a)$ is an extensional program property, by Rice's Theorem, $\gamma(a) \in \{\varnothing, \mathrm{Prog}\}$. Thus, the abstract domain A is trivial.

Conversely, let $A \neq \varnothing$ be a trivial abstract domain. We have to prove that for any $a \in A$ and $P \in \mathrm{Prog}$, $\mathcal{V}(P, a) = \mathbf{t} \Leftrightarrow P \in \gamma(a)$. Consider any $a \in A$. Since A is trivial, $\gamma(a) \in \{\varnothing, \mathrm{Prog}\}$. If $\gamma(a) = \varnothing$ then, by soundness of $\mathcal{V}$, for any $P \in \mathrm{Prog}$, $\mathcal{V}(P, a) = \mathbf{?}$, so that $\mathcal{V}(P, a) = \mathbf{t} \Leftrightarrow P \in \gamma(a)$ holds. If, instead, $\gamma(a) = \mathrm{Prog}$, i.e. a is an abstract top, then, since $\mathcal{V}$ is assumed to be nontrivial and monotone, by Remark 4.4 (2), $\mathcal{V}$ is able to prove the abstract top a for any program, namely, for any $P \in \mathrm{Prog}$, $\mathcal{V}(P, a) = \mathbf{t}$, so that $\mathcal{V}(P, a) = \mathbf{t} \Leftrightarrow P \in \gamma(a)$ holds. □

Let us remark a noteworthy difference of Theorem 5.4 w.r.t. Rice's theorem for static analysis. Let us consider a trivial abstract domain $A \triangleq \{\top\}$ with $\gamma(\top) = \mathrm{Prog}$. Here, the trivially sound analyser $\lambda P.\top$ is also precise, in accordance with Theorem 5.3. Instead, the trivially sound verifier $\mathcal{V}_\mathbf{?} \triangleq \lambda(P, a).\mathbf{?}$ is not precise, because $P \in \gamma(\top) \Leftrightarrow \mathcal{V}_\mathbf{?}(P, \top) = \mathbf{t}$ does not hold. The point here is that $\mathcal{V}_\mathbf{?}$ lacks the property of being nontrivial, and therefore Theorem 5.4 cannot be applied. On the other hand, $\mathcal{V}_\mathbf{t} \triangleq \lambda(P, a).\mathbf{t}$ is nontrivial and precise, because, in this case, $P \in \gamma(\top) \Leftrightarrow \mathcal{V}_\mathbf{t}(P, \top) = \mathbf{t}$ holds. Similarly, if we consider the trivial abstract domain $A' \triangleq \{\top, \top'\}$, with $\gamma(\top) = \mathrm{Prog} = \gamma(\top')$, then the verifier

$$\mathcal{V}'(P, a) \triangleq \begin{cases} \mathbf{t} & \text{if } a = \top \\ \mathbf{?} & \text{if } a = \top' \end{cases}$$

is sound and nontrivial, but still $\mathcal{V}'$ is not precise, because $P \in \gamma(\top') \Leftrightarrow \mathcal{V}'(P, \top') = \mathbf{t}$ does not hold. The point here is that $\mathcal{V}'$ is not monotone, because $\mathcal{V}'(P, \top) = \mathbf{t}$ and $\top \leq_\gamma \top'$ but $\mathcal{V}'(P, \top') \neq \mathbf{t}$, so that Theorem 5.4 cannot be applied.

6 Comparing Analysers and Verifiers

Let us now focus on a model for comparing the relative precision of program analysers and verifiers w.r.t. a common abstract domain $\langle A, \gamma, \leq_\gamma \rangle$.

Definition 6.1 (Comparison Relations). *Let $\mathcal{V}, \mathcal{V}' \in \mathbb{V}_A$, $\mathcal{A}, \mathcal{A}' \in \mathbb{A}_A$, and $\mathcal{X}, \mathcal{Y} \in \mathbb{V}_A \cup \mathbb{A}_A$.*

(1) $\mathcal{V} \sqsubseteq \mathcal{V}'$ *iff* $\forall P \in \mathrm{Prog}.\forall a \in A.\ \mathcal{V}'(P,a) = \mathbf{t} \Rightarrow \mathcal{V}(P,a) = \mathbf{t}$
(2) $\mathcal{A} \sqsubseteq \mathcal{A}'$ *iff* $\forall P \in \mathrm{Prog}.\ \mathcal{A}(P) \leq_\gamma \mathcal{A}'(P)$
(3) $\mathcal{V} \sqsubseteq \mathcal{A}$ *iff* $\forall P \in \mathrm{Prog}.\forall a \in A.\ \mathcal{A}(P) \leq_\gamma a \Rightarrow \mathcal{V}(P,a) = \mathbf{t}$
(4) $\mathcal{A} \sqsubseteq \mathcal{V}$ *iff* $\forall P \in \mathrm{Prog}.\forall a \in A.\ \mathcal{V}(P,a) = \mathbf{t} \Rightarrow \mathcal{A}(P) \leq_\gamma a$
(5) $\mathcal{X} \cong \mathcal{Y}$ *when* $\mathcal{X} \sqsubseteq \mathcal{Y}$ *and* $\mathcal{Y} \sqsubseteq \mathcal{X}$ □

Let us comment on the previous definitions, which intuitively take into account the relative "verification powers" of verifiers and analysers. The relation $\mathcal{V} \sqsubseteq \mathcal{V}'$ holds when every assertion proved by $\mathcal{V}'$ can be also proved by $\mathcal{V}$, while $\mathcal{A} \sqsubseteq \mathcal{A}'$ means that the output assertion provided by $\mathcal{A}$ is more precise than that produced by $\mathcal{A}'$. Also, a verifier $\mathcal{V}$ is more precise than an analyser $\mathcal{A}$ when the verification power of $\mathcal{V}$ is not less than the verification power of $\mathcal{A}$, namely, any assertion a which can be proved by $\mathcal{A}$ for a program P, i.e. $\mathcal{A}(P) \leq_\gamma a$ holds, can be also proved by $\mathcal{V}$. Likewise, $\mathcal{A}$ is more precise than $\mathcal{V}$ when any assertion a proved by $\mathcal{V}$ can be also proved by $\mathcal{A}$, i.e., $\mathcal{V}(P,a) = \mathbf{t}$ implies $\mathcal{A}(P) \leq_\gamma a$.

Let us observe that $\langle \mathbb{V}_A, \sqsubseteq \rangle$ turns out to be a poset, while $\langle \mathbb{A}_A, \sqsubseteq \rangle$ is just a pre-ordered set (cf. the lattice of abstract interpretations in [8]). We have that $\langle \mathbb{V}_A, \sqsubseteq \rangle$ has a greatest element $\mathcal{V}_{\mathbf{?}} \triangleq \lambda(P,a).\mathbf{?}$, which, in particular, is always sound although it is trivial. On the other hand, if A includes a top element $\top$ then $\mathcal{A}_\top \triangleq \lambda P.\top$ is a sound analyser which is a maximal element in $\langle \mathbb{A}_A, \sqsubseteq \rangle$. Also, $\mathcal{V} \cong \mathcal{V}'$ means that $\mathcal{V} = \mathcal{V}'$ as total functions, while $\mathcal{A} \cong \mathcal{A}'$ means that $\gamma \circ \mathcal{A} = \gamma \circ \mathcal{A}'$. Moreover, the comparison relation $\sqsubseteq$ is transitive even when considering analysers and verifiers together: if $\mathcal{V} \sqsubseteq \mathcal{A}$ and $\mathcal{A} \sqsubseteq \mathcal{V}'$ then $\mathcal{V} \sqsubseteq \mathcal{V}'$, and if $\mathcal{A} \sqsubseteq \mathcal{V}$ and $\mathcal{V} \sqsubseteq \mathcal{A}'$ then $\mathcal{A} \sqsubseteq \mathcal{A}'$. Also, the relation $\sqsubseteq$ shifts soundness from verifiers to analysers, and from analysers to verifiers as follows (due to lack of space the proof is omitted).

Lemma 6.2. *Let $\mathcal{V} \in \mathbb{V}_A$ and $\mathcal{A} \in \mathbb{A}_A$. If $\mathcal{V}$ is sound and $\mathcal{V} \sqsubseteq \mathcal{A}$ then $\mathcal{A}$ is sound; if $\mathcal{A}$ is sound and $\mathcal{A} \sqsubseteq \mathcal{V}$ then $\mathcal{V}$ is sound.*

As expected, any sound analyser can be used to refine a given sound verifier (cf. [19,20,24,25]) and this can be formalized and proved in our framework as follows.

Lemma 6.3. *Given $\mathcal{A} \in \mathbb{A}_A$ and $\mathcal{V} \in \mathbb{V}_A$ which are both sound, let*

$$\tau_{\mathcal{A}}(\mathcal{V})(P,a) \triangleq \begin{cases} \mathbf{t} & \text{if } \mathcal{A}(P) \leq_\gamma a \\ \mathcal{V}(P,a) & \text{if } \mathcal{A}(P) \not\leq_\gamma a \end{cases}$$

Then, $\tau_{\mathcal{A}}(\mathcal{V}) \in \mathbb{V}_A$ is sound, $\tau_{\mathcal{A}}(\mathcal{V}) \sqsubseteq \mathcal{V}$ and $\tau_{\mathcal{A}}(\mathcal{V}) = \mathcal{V} \Leftrightarrow \mathcal{V} \sqsubseteq \mathcal{A}$.

Proof. $\tau_{\mathcal{A}}(\mathcal{V}) \in \mathbb{V}_A$ is sound because both $\mathcal{A}$ and $\mathcal{V}$ are sound. If $\mathcal{V}(P,a) = \mathbf{t}$ then $\tau_{\mathcal{A}}(\mathcal{V})(P,a) = \mathbf{t}$, i.e., $\tau_{\mathcal{A}}(\mathcal{V}) \sqsubseteq \mathcal{V}$. Moreover, $\tau_{\mathcal{A}}(\mathcal{V}) = \mathcal{V}$ iff $\mathcal{A}(P) \leq_\gamma a \Rightarrow \mathcal{V}(P,a) = \mathbf{t}$ iff $\mathcal{V} \sqsubseteq \mathcal{A}$. □

6.1 Optimal and Best Analysers and Verifiers

It makes sense to define optimality by restricting to sound analysers and verifiers only. Optimality is defined as minimality w.r.t. the precision relation $\sqsubseteq$, while being the best analyser/verifier means to be the most precise.

Definition 6.4 (Optimal and Best Analysers and Verifiers) A sound analyser $\mathcal{A} \in \mathbb{A}_A$ is *optimal* if for any sound $\mathcal{A}' \in \mathbb{A}_A$, $\mathcal{A}' \sqsubseteq \mathcal{A} \Rightarrow \mathcal{A}' \cong \mathcal{A}$, while $\mathcal{A}$ is a *best* analyser if for any sound $\mathcal{A}' \in \mathbb{A}_A$, $\mathcal{A} \sqsubseteq \mathcal{A}'$.
A sound verifier $\mathcal{V} \in \mathbb{V}_A$ is *optimal* if for any $\mathcal{V}' \in \mathbb{V}_A$, $\mathcal{V}' \sqsubseteq \mathcal{V} \Rightarrow \mathcal{V}' \cong \mathcal{V}$, while $\mathcal{V}$ is the *best* verifier if for any $\mathcal{V}' \in \mathbb{V}_A$, $\mathcal{V} \sqsubseteq \mathcal{V}'$. □

Let us first observe that if a best analyser or verifier exists then this is unique, while for analysers if $\mathcal{A}_1$ and $\mathcal{A}_2$ are two best analysers on A then $\mathcal{A}_1 \cong \mathcal{A}_2$ holds. Of course, the possibility of defining an optimal/best analyser or verifier depends on the abstract domain A. For example, for a variable sign domain such as $\{\mathbb{Z}_{\leq 0}, \mathbb{Z}_{\geq 0}, \mathbb{Z}\}$ just optimal analysers and verifiers could be defined, because for approximating the set $\{0\}$ two optimal sound abstract values are available rather than a best sound abstract value. Here, the expected but interesting property to remark is that the notion of precise (i.e., sound and complete) analyser turns out to coincide with the notion of being the best analyser.

Lemma 6.5. *Let $\mathcal{A} \in \mathbb{A}_A$ be sound. Then, $\mathcal{A}$ is precise iff $\mathcal{A}$ is a best analyser.*

Proof. ($\Rightarrow$) Consider any sound $\mathcal{A}' \in \mathbb{A}_A$. Assume, by contradiction, that $\mathcal{A} \not\sqsubseteq \mathcal{A}'$, namely, there exists some $P \in \mathrm{Prog}$ such that $\gamma(\mathcal{A}(P)) \not\subseteq \gamma(\mathcal{A}'(P))$. By soundness of $\mathcal{A}'$, $[\![P]\!] \in \gamma(\mathcal{A}'(P))$, so that, by precision of $\mathcal{A}$, $\gamma(\mathcal{A}(P)) \subseteq \gamma(\mathcal{A}'(P))$, which is a contradiction. Thus, $\mathcal{A} \sqsubseteq \mathcal{A}'$ holds. This means that $\mathcal{A}$ is a best analyser on A.
($\Leftarrow$) We have to prove that for any $P \in \mathrm{Prog}$ and $a \in A$, $[\![P]\!] \in \gamma(a) \Rightarrow \gamma(\mathcal{A}(P)) \subseteq \gamma(a)$. Assume, by contradiction, that there exist $Q \in \mathrm{Prog}$ and $b \in A$ such that $[\![Q]\!] \in \gamma(b)$ and $\gamma(\mathcal{A}(Q)) \not\subseteq \gamma(b)$. Then, we define $\mathcal{A}' : \mathrm{Prog} \to A$ as follows:

$$\mathcal{A}'(P) \triangleq \begin{cases} \mathcal{A}(P) & \text{if } P \not\equiv Q \\ b & \text{if } P \equiv Q \end{cases}$$

It turns out that $\mathcal{A}'$ is a total recursive function because $P \equiv Q$ is decidable. Moreover, $\mathcal{A}'$ is sound: assume that $\gamma(\mathcal{A}'(P)) \subseteq \gamma(a)$; if $P \not\equiv Q$ then $\mathcal{A}'(P) = \mathcal{A}(P)$ so that $\gamma(\mathcal{A}(P)) \subseteq \gamma(a)$, and, by soundness of $\mathcal{A}$, $[\![P]\!] \in \gamma(a)$; if $P \equiv Q$ then $\mathcal{A}'(Q) = b$ so that $\gamma(b) = \gamma(\mathcal{A}'(Q)) = \gamma(\mathcal{A}'(P)) \subseteq \gamma(a)$, hence, $[\![Q]\!] \in \gamma(b)$ implies $[\![Q]\!] \in \gamma(a)$. Since $\mathcal{A}$ is a best analyser on A, we have that $\mathcal{A} \sqsubseteq \mathcal{A}'$, so that $\gamma(\mathcal{A}(Q)) \subseteq \gamma(\mathcal{A}'(Q)) = \gamma(b)$, which is a contradiction. □

We therefore derive the following consequence of Rice's Theorem 5.3 for static analysis: the best analyser on an extensional abstract domain A exists if and only if A is trivial. This fact formalizes in our model the common intuition that, given any abstract domain, the best static analyser (where best means for any input program) cannot be defined due to Rice's Theorem. An analogous result can be given for verifiers.

Lemma 6.6. *Let $\mathcal{V} \in \mathbb{V}_A$ be sound. Then $\mathcal{V}$ is precise iff $\mathcal{V}$ is the best verifier on A.*

Proof. Assume that $\mathcal{V}$ is precise and $\mathcal{V}' \in \mathbb{V}_A$ be sound. If $\mathcal{V}'(P, a) = \mathbf{t}$ then, by soundness of $\mathcal{V}'$, $[\![P]\!] \in \gamma(a)$, and in turn, by completeness of $\mathcal{V}$, $\mathcal{V}(P, a) = \mathbf{t}$, thus proving that $\mathcal{V} \sqsubseteq \mathcal{V}'$. On the other hand, assume that $\mathcal{V}$ is the best verifier on A. Assume, by contradiction, that $\mathcal{V}$ is not complete, namely that there exist some $Q \in \mathrm{Prog}$ and

$b \in A$ such that $[\![Q]\!] \in \gamma(b)$ and $\mathcal{V}(Q,b) = \mathbf{?}$. We then define $\mathcal{V}' : \mathrm{Prog} \times A \to \{\mathbf{t}, \mathbf{?}\}$ as follows:

$$\mathcal{V}'(P,a) \triangleq \begin{cases} \mathbf{t} & \text{if } P \equiv Q \ \wedge \ a = b \\ \mathcal{V}(P,a) & \text{otherwise} \end{cases}$$

Then, $\mathcal{V}'$ is a total recursive function because $P \equiv Q$ and $a = b$ are decidable. Also, $\mathcal{V}'$ is sound because $[\![Q]\!] \in \gamma(b)$ and $\mathcal{V}$ is sound. Since $\mathcal{V}$ is the best verifier, we have that $\mathcal{V} \sqsubseteq \mathcal{V}'$, so that $\mathcal{V}'(Q,b) = \mathbf{t}$ implies $\mathcal{V}(Q,b) = \mathbf{t}$, which is a contradiction. □

Thus, similarly to static analysis, as a consequence of Rice's Theorem 5.4 for verification, the best nontrivial and monotone verifier on an extensional abstract domain A exists if and only if A is trivial, which is a common belief in program verification. Let us also remark that best abstract program semantics, rather than program analysers, do exist for nontrivial domains (see e.g. [6]). Clearly, this is not in contradiction with Theorem 5.3 since these abstract program semantics are not total recursive functions, i.e., they are not program analysers.

7 Reducing Verification to Analysis and Back

As usual in computability and complexity, our comparison between verification and analysis is made through a many-one reduction, namely by reducing a verification problem into an analysis problem and vice versa. The minimal requirement is that these reduction functions are total recursive. Moreover, we require that the reduction function does not depend upon a fixed abstract domain. This allows us to be problem agnostic and to prove a reduction for all possible verifiers and analysers. Program verification and analysis are therefore equivalent problems whenever we can reduce one to the other. In the following, we prove that while it is always possible to transform a program analyser into an equivalent program verifier, the converse does not hold in general, but it can always be done for finite abstract domains.

7.1 Reducing Verification to Analysis

Theorem 7.1. *Let $\langle A, \gamma, \leq_\gamma \rangle$ be any given abstract domain. There exists a transform $\sigma : \mathbb{A}_A \to \mathbb{V}_A$ such that:*

(1) *σ is a total recursive function such that for all $\mathcal{A} \in \mathbb{A}_A$, $\sigma(\mathcal{A}) \cong \mathcal{A}$;*
(2) *if $\mathcal{A} \in \mathbb{A}_A$ is sound then $\sigma(\mathcal{A})$ is sound;*
(3) *σ is monotonic;*
(4) *$\sigma(\mathcal{A}) \cong \sigma(\mathcal{A}') \Rightarrow \mathcal{A} \cong \mathcal{A}'$.*

Proof. Given $\mathcal{A} \in \mathbb{A}_A$, we define $\sigma(\mathcal{A}) : \mathrm{Prog} \times A \to \{\mathbf{t}, \mathbf{?}\}$ as follows:

$$\sigma(\mathcal{A})(P,a) \triangleq \begin{cases} \mathbf{t} & \text{if } \mathcal{A}(P) \leq_\gamma a \\ \mathbf{?} & \text{if } \mathcal{A}(P) \not\leq_\gamma a \end{cases}$$

(1) Since $\mathcal{A}$ is a total recursive function and $\leq_\gamma$ is decidable, we have that $\sigma(\mathcal{A})$ is a total recursive function, namely $\sigma(\mathcal{A}) \in \mathbb{V}_A$, and σ is a total recursive function as well.

Since, by definition, $\sigma(\mathcal{A})(P,a) = \mathbf{t} \Leftrightarrow \mathcal{A}(P) \leq_\gamma a$, we have that $\sigma(\mathcal{A}) \cong \mathcal{A}$. (2) By Lemma 6.2, if $\mathcal{A}$ is sound then the equivalent verifier $\sigma(\mathcal{A})$ is sound as well. (3) It turns out that σ is monotonic: if $\mathcal{A} \sqsubseteq \mathcal{A}'$ then $\sigma(\mathcal{A}')(P,a) = \mathbf{t} \Leftrightarrow \mathcal{A}'(P) \leq_\gamma a \Rightarrow \mathcal{A}(P) \leq_\gamma \mathcal{A}'(P) \leq_\gamma a \Leftrightarrow \sigma(\mathcal{A})(P,a) = \mathbf{t}$, so that $\sigma(\mathcal{A}) \sqsubseteq \sigma(\mathcal{A}')$ holds. (4) Assume that $\sigma(\mathcal{A}) \cong \sigma(\mathcal{A}')$, hence, for any $P \in \mathrm{Prog}$, $\sigma(\mathcal{A})(P, \mathcal{A}(P)) = \sigma(\mathcal{A}')(P, \mathcal{A}(P))$, namely, $\mathcal{A}(P) \leq_\gamma \mathcal{A}(P) \Leftrightarrow \mathcal{A}'(P) \leq_\gamma \mathcal{A}(P)$, so that $\mathcal{A}'(P) \leq_\gamma \mathcal{A}(P)$ holds. On the other hand, $\mathcal{A}(P) \leq_\gamma \mathcal{A}'(P)$ can be dually obtained, therefore $\gamma(\mathcal{A}(P)) = \gamma(\mathcal{A}'(P))$ holds, namely $\mathcal{A} \cong \mathcal{A}'$. □

Intuitively, Theorem 7.1 shows that program verification on a given abstract domain A can always and unconditionally be reduced to program analysis on A. This means that a solution to the program analysis problem on A, i.e. the definition of an analyser $\mathcal{A}$, can constructively be transformed into a solution to the program verification problem on the same domain A, i.e. the design of a verifier $\sigma(\mathcal{A})$ which is equivalent to $\mathcal{A}$. The proof of Theorem 7.1 provides this constructive transform σ, which is defined as expected: an analyser $\mathcal{A}$ on any (possibly infinite) abstract domain A can be used as a verifier for any assertion $a \in A$ simply by checking whether $\mathcal{A}(P) \leq_\gamma a$ holds or not.

7.2 Reducing Analysis to Verification

It turns out that the converse of Theorem 7.1 does not hold, namely a program analysis problem in general cannot be reduced to a verification problem. Instead, this reduction can be always done for finite abstract domains. Given a verifier $\mathcal{V} \in \mathbb{V}_A$, for any program $P \in \mathrm{Prog}$, let us define $\mathcal{V}_{\mathbf{t}}(P) \triangleq \{a \in A \mid \mathcal{V}(P,a) = \mathbf{t}\}$, namely, $\mathcal{V}_{\mathbf{t}}(P)$ is the set of assertions proved by $\mathcal{V}$ for P. Also, given an assertion $a \in A$, we define $\uparrow a \triangleq \{a' \in A \mid a \leq_\gamma a'\}$ as the set of assertions weaker than a. The following result provides a useful characterization of the equivalence between verifiers and analysers.

Lemma 7.2. *Let $\langle A, \gamma, \leq_\gamma \rangle$ be an abstract domain, $\mathcal{A} \in \mathbb{A}_A$ and $\mathcal{V} \in \mathbb{V}_A$. Then, $\mathcal{A} \cong \mathcal{V}$ if and only if for any $P \in \mathrm{Prog}$, $\mathcal{V}_{\mathbf{t}}(P) = \uparrow\mathcal{A}(P)$.*

Proof. By Definition 6.1, it turns out that $\mathcal{A} \sqsubseteq \mathcal{V}$ iff for any P, $\mathcal{V}_{\mathbf{t}}(P) \subseteq \uparrow\mathcal{A}(P)$, while we have that $\mathcal{V} \sqsubseteq \mathcal{A}$ iff for any P, $\uparrow\mathcal{A}(P) \subseteq \mathcal{V}_{\mathbf{t}}(P)$. Thus, $\mathcal{A} \cong \mathcal{V}$ if and only if for any $P \in \mathrm{Prog}$, $\mathcal{V}_{\mathbf{t}}(P) = \uparrow\mathcal{A}(P)$. □

A consequence of Lemma 7.2 is that, given $\mathcal{V} \in \mathbb{V}_A$, $\mathcal{V}$ can be transformed into an equivalent analyser $\tau(\mathcal{V}) \in \mathbb{A}_A$ if and only if for any program P, an assertion $a_P \in A$ exists such that $\mathcal{V}_{\mathbf{t}}(P) = \uparrow a_P$. In this case, one can then define $\tau(\mathcal{V})(P) \triangleq a_P$.

Lemma 7.3. *Let $\langle A, \gamma, \leq_\gamma \rangle$ be an abstract domain and $\mathcal{V} \in \mathbb{V}_A$. If $\mathcal{A} \in \mathbb{A}_A$ is such that $\mathcal{A} \cong \mathcal{V}$ then:* (1) *$A \neq \varnothing$;* (2) *$\mathcal{V}$ is not trivial;* (3) *$\mathcal{V}$ is monotone.*

Proof. (1) We observed just after Definition 4.1 that no analyser can be defined on the empty abstract domain. (2) If $\mathcal{V}$ is trivial then there exists a program $Q \in \mathrm{Prog}$ such that for any $a \in A$, $\mathcal{V}(Q,a) = \mathbf{?}$, so that if $\mathcal{V} \cong \mathcal{A}$ for some $\mathcal{A} \in \mathbb{A}_A$ then, from $\mathcal{V} \sqsubseteq \mathcal{A}$ we would derive $\mathcal{V}(Q, \mathcal{A}(Q)) = \mathbf{t}$, which is a contradiction. (3) Assume that $\mathcal{V}$ is not monotone. Then, there exist $Q \in \mathrm{Prog}$ and $a, a' \in A$ such that $a \in \mathcal{V}_{\mathbf{t}}(Q)$, $a \leq_\gamma a'$ but $a' \notin \mathcal{V}_{\mathbf{t}}(Q)$. If $\mathcal{V} \cong \mathcal{A}$, for some $\mathcal{A} \in \mathbb{A}_A$, then, by Lemma 7.2, $\mathcal{V}_{\mathbf{t}}(Q) = \uparrow\mathcal{A}(Q)$, so that we would have that $a \in \uparrow\mathcal{A}(Q)$ but $a' \notin \uparrow\mathcal{A}(Q)$, which is a contradiction. □

We also observe that even for a nontrivial and monotone verifier $\mathcal{V} \in \mathbb{V}_A$ on a finite abstract domain A, it is not guaranteed that an equivalent analyser exists. In fact, if an equivalent analyser $\mathcal{A}$ exists then, by Lemma 7.2, for any program P, $\mathcal{V}_{\mathbf{t}}(P)$ must contain the least element, namely for any program P it must be the case that there exists a strongest assertion proved by $\mathcal{V}$ for P.

Example 7.4 Consider a sign domain such as $S \triangleq \{\mathbb{Z}_{\leq 0}, \mathbb{Z}_{\geq 0}, \mathbb{Z}\}$ where $\mathbb{Z}_{\leq 0} \leq_\gamma \mathbb{Z}$ and $\mathbb{Z}_{\geq 0} \leq_\gamma \mathbb{Z}$. For a program such as $Q \equiv x := 0$, a sound verifier $\mathcal{V} \in \mathbb{V}_S$ could be able to prove all the assertions in S, namely $\mathcal{V}_{\mathbf{t}}(Q) = S$. However, there exists no assertion $a_Q \in S$ such that $\mathcal{V}_{\mathbf{t}}(Q) = \uparrow a_Q$. Hence, by Lemma 7.2, there exists no analyser in $\mathbb{A}_S$ which is equivalent to $\mathcal{V}$. Also, if $S' \triangleq \{\mathbb{Z}_{=0}, \mathbb{Z}_{\leq 0}, \mathbb{Z}_{\geq 0}, \mathbb{Z}\}$, so that S' is a meet-semilattice, and $\mathcal{V}' \in \mathbb{V}_{S'}$ is a sound verifier such that $\mathcal{V}'_{\mathbf{t}}(Q) = S' \smallsetminus \{\mathbb{Z}_{=0}\}$, still, by Lemma 7.2, there exists no analyser in $\mathbb{A}_{S'}$ which is equivalent to $\mathcal{V}'$. □

Definition 7.5 A verifier $\mathcal{V} \in \mathbb{V}_A$ is *finitely meet-closed* when for any $P \in \mathrm{Prog}$ and $a, a_1, a_2 \in A$, if $\mathcal{V}(P, a_1) = \mathbf{t} = \mathcal{V}(P, a_2)$ and $\gamma(a) = \gamma(a_1) \cap \gamma(a_2)$ then $\mathcal{V}(P, a) = \mathbf{t}$. The following notation will be used: for any domain A,

$$\mathbb{V}_A^+ \triangleq \{\mathcal{V} \in \mathbb{V}_A \mid \mathcal{V} \text{ is nontrivial, monotone and finitely meet-closed}\}. \qquad \square$$

Thus, finitely meet-closed verifiers can prove logical conjunctions of provable assertions.

Theorem 7.6 (Reduction for Finite Domains). *Let $\langle A, \gamma, \leq_\gamma \rangle$ be a nonempty finite abstract domain. There exists a transform $\tau : \mathbb{V}_A^+ \to \mathbb{A}_A$ such that:*

(1) *τ is a total recursive function such that for all $\mathcal{V} \in \mathbb{V}_A^+$, $\tau(\mathcal{V}) \cong \mathcal{V}$;*
(2) *if $\mathcal{V} \in \mathbb{V}_A^+$ is sound then $\tau(\mathcal{V})$ is sound;*
(3) *τ is monotonic;*
(4) $\tau(\mathcal{V}) \cong \tau(\mathcal{V}') \Rightarrow \mathcal{V} \cong \mathcal{V}'$.

Proof. (1) Let $A = \{a_1, ..., a_n\}$ be any enumeration of A, with $n \geq 1$. Given $\mathcal{V} \in \mathbb{V}_A^+$, we define $\tau(\mathcal{V}) : \mathrm{Prog} \to A$ as follows:

$$\tau(\mathcal{V})(P) \triangleq \begin{cases} r := \text{undef}; \\ \textbf{forall } i \in 1..n \textbf{ do} \\ \qquad \textbf{if } \big(a_i \in \mathcal{V}_{\mathbf{t}}(P) \ \wedge \ (r = \text{undef} \ \vee \ a_i \leq_\gamma r)\big) \textbf{ then } r := a_i; \\ \textbf{output } r \end{cases}$$

Then, it turns out that τ is a total recursive function. Since $\mathcal{V}$ is a total recursive function, A is finite and $\leq_\gamma$ is decidable, we have that $\tau(\mathcal{V})$ is a total recursive function, so that $\tau(\mathcal{V}) \in \mathbb{A}_A$. Since $\mathcal{V}$ is not trivial, for any $P \in \mathrm{Prog}$, $\mathcal{V}_{\mathbf{t}}(P) \neq \varnothing$. Also, since A is finite and $\mathcal{V}$ is finitely meet-closed there exists some $a_k \in \mathcal{V}_{\mathbf{t}}(P)$ such that $\mathcal{V}_{\mathbf{t}}(P) \subseteq \uparrow a_k$, so that $\tau(\mathcal{V})(P)$ outputs some value in A. Moreover, since $\mathcal{V}$ is monotone, $\uparrow a_k \subseteq \mathcal{V}_{\mathbf{t}}(P)$, so that $\uparrow a_k = \mathcal{V}_{\mathbf{t}}(P)$. Thus, the above procedure defining $\tau(\mathcal{V})(P)$ finds and outputs a_k. Hence, for any $P \in \mathrm{Prog}$ and $a \in A$, $\mathcal{V}(P, a) = \mathbf{t} \Leftrightarrow a \in \mathcal{V}_{\mathbf{t}}(P) \Leftrightarrow a \in \uparrow a_k \Leftrightarrow a_k \leq_\gamma a \Leftrightarrow \tau(\mathcal{V})(P) \leq_\gamma a$, that is, $\tau(\mathcal{V}) \cong \mathcal{V}$ holds.
(2) By Lemma 6.2, if $\mathcal{V}$ is sound then the equivalent analyser $\tau(\mathcal{V})$ is sound as well.

(3) It turns out that τ is monotonic: if $\mathcal{V} \sqsubseteq \mathcal{V}'$ then, by definition, $\mathcal{V}'_{\mathbf{t}}(P) \subseteq \mathcal{V}_{\mathbf{t}}(P)$, so that, since $\mathcal{V}_{\mathbf{t}}(P) = \uparrow\tau(\mathcal{V})(P)$ and $\mathcal{V}'_{\mathbf{t}}(P) = \uparrow\tau(\mathcal{V}')(P)$, we obtain $\tau(\mathcal{V})(P) \leq_\gamma \tau(\mathcal{V}')(P)$, namely $\tau(\mathcal{V}) \sqsubseteq \tau(\mathcal{V}')$ holds.
(4) Assume that $\tau(\mathcal{V}) \cong \tau(\mathcal{V}')$. Hence, for any $P \in \mathrm{Prog}$, $\gamma(\tau(\mathcal{V})(P)) = \gamma(\tau(\mathcal{V}')(P))$, so that, since $\mathcal{V}_{\mathbf{t}}(P) = \uparrow\tau(\mathcal{V})(P)$ and $\mathcal{V}'_{\mathbf{t}}(P) = \uparrow\tau(\mathcal{V}')(P)$, we obtain $\mathcal{V}_{\mathbf{t}}(P) = \mathcal{V}'_{\mathbf{t}}(P)$, namely $\mathcal{V} = \mathcal{V}'$. □

An example of this reduction of verification to static analysis for finite domains is dataflow analysis as model checking shown in [31] (excluding Kildall's constant propagation domain [16]). Let us now focus on infinite domains of assertions.

Lemma 7.7. *There exists a denumerable infinite abstract domain $\langle A, \gamma, \leq_\gamma\rangle$ and a verifier $\mathcal{V} \in \mathbb{V}^+_A$ such that for any analyser $\mathcal{A} \in \mathbb{A}_A$, $\mathcal{A} \ncong \mathcal{V}$.*

Proof. Let us consider the infinite domain $\mathrm{T} \triangleq \mathbb{N} \cup \{\top\}$ together with the following concretization function: $\gamma(\top) \triangleq \mathrm{Prog}$ and, for any $n \in \mathbb{N}$,

$$\gamma(n) \triangleq \{P \in \mathrm{Prog} \mid P \text{ on input } 0 \text{ converges in } n \text{ or fewer steps}\}$$

where the number of steps is determined by a small-step operational semantics $\Rightarrow$, as recalled in Section 2. Thus, we have that if $n, m \in \mathbb{N}$ then $n \leq_\gamma m$ iff $n \leq_{\mathbb{N}} m$, while $n \leq_\gamma \top$. We define a function $\mathcal{V} : \mathrm{Prog} \times \mathrm{T} \to \{\mathbf{t}, \mathbf{?}\}$ as follows:

$$\mathcal{V}(P, a) \triangleq \begin{cases} \mathbf{t} & \text{if } a = \top \\ \mathbf{t} & \text{if } a = n \text{ and } P \text{ on input } 0 \text{ converges in } n \text{ or fewer steps} \\ \mathbf{?} & \text{if } a = n \text{ and } P \text{ on input } 0 \text{ does not converge in } n \text{ or fewer steps} \end{cases}$$

Clearly, for any number $n \in \mathbb{N}$, the predicate "P on input 0 converges in n or fewer steps" is decidable, where the input 0 could be replaced by any other (finite set of) input value(s). Hence, $\mathcal{V}$ turns out to be a total recursive function, that is, a verifier on the abstract domain T. In particular, let us remark that $\mathcal{V}$ is a sound verifier. Moreover, $\mathcal{V}$ is nontrivial, since, for any $P \in \mathrm{Prog}$, $\mathcal{V}(P, \top) = \mathbf{t}$, and monotone because if $\mathcal{V}(P, n) = \mathbf{t}$ and $n \leq_\gamma a$ then either $a = \top$ and $\mathcal{V}(P, \top) = \mathbf{t}$ or $a = m$, so that $n \leq_{\mathbb{N}} m$ and therefore $\mathcal{V}(P, m) = \mathbf{t}$. Clearly, $\mathcal{V}$ is also finitely meet-closed, because if $\mathcal{V}(P, a_1) = \mathbf{t} = \mathcal{V}(P, a_2)$ and $\gamma(a) = \gamma(a_1) \cap \gamma(a_2)$ then either $a = a_1$ or $a = a_2$, so that $\mathcal{V}(P, a) = \mathbf{t}$. Summing up, it turns out that $\mathcal{V} \in \mathbb{V}^+_{\mathrm{T}}$. Assume now, by contradiction, that there exists an analyser $\mathcal{A} \in \mathbb{A}_{\mathrm{T}}$ such that $\mathcal{A} \cong \mathcal{V}$. By Lemma 7.2, for any $P \in \mathrm{Prog}$, we have that $\mathcal{V}_{\mathbf{t}}(P) = \uparrow\mathcal{A}(P)$. Hence, if P on input 0 diverges then $\mathcal{V}_{\mathbf{t}}(P) = \{\top\}$ so that $\mathcal{A}(P) = \top$, while if P on input 0 converges in exactly n steps then $\mathcal{V}_{\mathbf{t}}(P) = \{m \in \mathbb{N} \mid m \geq n\} \cup \{\top\}$, so $\mathcal{A}(P) = n$, namely $\mathcal{A}$ goes as follows:

$$\mathcal{A}(P) = \begin{cases} \top & \text{if } P \text{ on input } 0 \text{ diverges} \\ n & \text{if } P \text{ on input } 0 \text{ converges in exactly } n \text{ steps} \end{cases}$$

Since $\mathcal{A}$ is a total recursive function, we would have defined an algorithm $\mathcal{A}$ for deciding if a program $P \in \mathrm{Prog}$ on input 0 terminates or not. Since Prog is assumed to be Turing complete with respect to the operational semantics $\Rightarrow$, this leads to a contradiction. □

As a straight consequence of Lemma 7.7, the following theorem proves that for any infinite abstract domain A, no reduction from verifiers in $\mathbb{V}^+_A$ to equivalent analysers in $\mathbb{A}_A$ is possible.

Theorem 7.8 (Impossibility of the Reduction for Infinite Domains). *For any denumerable infinite abstract domain $\langle A, \gamma, \leq_\gamma \rangle$, there exists no function $\tau : \mathbb{V}_A^+ \to \mathbb{A}_A$ such that τ is a total recursive function and for all $\mathcal{V} \in \mathbb{V}_A^+$, $\tau(\mathcal{V}) \cong \mathcal{V}$.*

Proof. Assume, by contradiction, that $\tau : \mathbb{V}_A^+ \to \mathbb{A}_A$ is a total recursive function such that for all $\mathcal{V} \in \mathbb{V}_A^+$, $\tau(\mathcal{V}) \in \mathbb{A}_A$ and $\tau(\mathcal{V}) \cong \mathcal{V}$. Then, for the infinite domain A and verifier $\mathcal{V} \in \mathbb{V}_A^+$ provided by Lemma 7.7, we would be able to construct an analyser $\tau(\mathcal{V}) \in \mathbb{A}_A$ such that $\tau(\mathcal{V}) \cong \mathcal{V}$, which would be in contradiction with Lemma 7.7. □

Intuitively, this result states that given any infinite abstract domain A, no general algorithm exists for constructively designing out of a reasonable (i.e., nontrivial, monotone and finitely meet-closed) verifier $\mathcal{V}$ on A an equivalent analyser on the same domain A. This can be read as a precise statement proving the folklore belief that "program analysis is harder than verification", at least for infinite domains of program assertions. It is important to remark that the verifier $\mathcal{V} \in \mathbb{V}_A^+$ on the infinite domain A defined by the proof of Lemma 7.7 is sound. Thus, even if we restrict the reduction transform $\tau : \mathbb{V}_A^{+,\text{sound}} \to \mathbb{A}_A^{\text{sound}}$ of Theorem 7.8 to be applied to sound verifiers — so that by Lemma 6.2 the range would be the sound analysers in $\mathbb{A}_A$ — the same proof of Lemma 7.7 could still be used for proving that such transform τ cannot exist.

A further consequence of Theorem 7.8 is the fact proved in [10] that abstract interpretation-based program analysis with infinite domains and widening/narrowing operators is strictly more powerful than with finite domains.

8 Conclusion and Future Work

We put forward a general model for studying static program analysers and verifiers from a computability perspective. This allowed us to state and prove, with simple arguments borrowed from standard computability theory, that for infinite abstract domains of program assertions, program analysis is a harder problem than program verification. This is, to the best of our knowledge, the first formalization and proof of this popular belief, which also includes the relationship between type inference and type checking. We think that this foundational model can be extended to study further properties of program analysers and verifiers. In particular, this opens interesting perspectives in reasoning about program analysis and verification in a more abstract way towards a theory of computation that may include approximate methods, such as program analysers and verifiers, as objects of investigation, as suggested in [5,14]. For instance, the precision of program analysis and program verification, as well as their computational complexity, are intensional program properties. Intensionally different but extensionally equivalent programs may exhibit completely different behaviours when analysed or verified. In this perspective, new intensional versions of Rice's Theorem can be stated for program analysis, similarly to what is known for Blum's complexity in [2]. Also, new models for reasoning about the space and time complexities of program analysis and verification algorithms can be studied, especially for defining a notion of complexity class of program analysers and verifiers.

References

1. J. Alglave, A. F. Donaldson, D. Kroening, and M. Tautschnig. Making software verification tools really work. In *Proc. Int. Symp. on Automated Technology for Verification and Analysis (ATVA'11)*, LNCS vol. 6996, pages 28–42. Springer, 2011.
2. A. Asperti. The intensional content of Rice's theorem. In *Proc. 35th ACM Symposium on Principles of Programming Languages (POPL'08)*, pages 113–119, ACM, 2008.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. Int. Symp. on Formal Methods for Components and Objects (FMCO'05)*, LNCS vol. 4111, pages 364–387. Springer, 2006.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, LNCS vol. 1579, pages 193–207. Springer, 1999.
5. C. Cadar and A. F. Donaldson. Analysing the program analyser. In *Proc. 38th Int. Conf. on Software Engineering (ICSE'16)*, pages 765–768. ACM, 2016.
6. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages* (*POPL'77*), pages 238–252. ACM Press, 1977.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM Symposium on Principles of Programming Languages* (*POPL'79*), pages 269–282. ACM Press, 1979.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, 1992.
10. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation (Invited Paper). In *Proc. 4th Int. Symp. on Programming Language Implementation and Logic Programming* (*PLILP'92*), LNCS vol. 631, pages 269–295. Springer, 1992.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96, ACM Press, 1978.
12. N. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
13. P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-upsilon-omega: An assistant algorithms analyzer. In *Proc. Int. Conf. on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, LNCS vol. 357, pages 201–212, 1988.
14. R. Giacobazzi, F. Logozzo, and F. Ranzato. Analyzing program analyses. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, pages 261–273. ACM Press, 2015.
15. J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
16. G. A. Kildall. A unified approach to global program optimization. In *Proc. 1st ACM Symp. on Principles of Programm. Lang. (POPL'73)*, pp. 194-206, 1973.
17. J. Laski and W. Stanley. *Software Verification and Analysis: an integrated, hands-on approach*. Springer, 2009.
18. G. Leibniz. *Dissertatio de arte combinatoria*, Habilitation Thesis in Philosophy at Leipzig University. https://en.wikipedia.org/wiki/De_Arte_Combinatoria, 1666.
19. K. Leino and F. Logozzo. Loop invariants on demand. In *Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, LNCS vol. 3780, pages 119–134, 2005.

20. K. Leino and F. Logozzo. Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain. In *Proc. International Workshop on Invariant Generation (WING'07)*, 2007.
21. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10)*, LNCS vol. 6355, pages 348–370. Springer, 2010.
22. F. Merz, C. Sinz, and S. Falke. Challenges in comparing software verification tools for C. In *Proc. 1st Int. Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE'12)*, Manchester, UK, pages 60–65, 2012.
23. A. Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017.
24. Y. Moy. Sufficient preconditions for modular assertion checking. In *Proc. Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'08)*, LNCS vol. 4905, pages 188–202. Springer, 2008.
25. Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *J. Symb. Comput.*, 45(11):1184–1211, 2010.
26. P. Odifreddi. *Classical Recursion Theory*. Studies in logic and the foundations of mathematics. Elsevier, 1999.
27. C. O'Halloran. Where is the value in a program verifier? In *Verified Software: Theories, Tools, Experiments (VSTTE'08)*, LNCS vol. 5295, pages 255–262. Springer, 2008.
28. B. Reus. *Limits of Computation from a Programming Perspective*. Springer, 2016.
29. H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74(2):358–366, 1953.
30. H. Rogers. *Theory of recursive functions and effective computability*. The MIT press, 1992.
31. D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM Symp. on Principles of Programm. Lang. (POPL'98)*, pages 38–48. ACM, 1998.
32. M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *J. Autom. Reasoning*, 59(1):3–45, 2017.
33. A. Stefanescu, D. Park, S. Yuwen, Y. Li, and G. Rosu. Semantics-based program verifiers for all languages. In *Proc. ACM Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91, ACM, 2016.
34. O. Strichman. Tuning SAT checkers for bounded model checking. In *Proc. 12th Int. Conf. on Computer Aided Verification (CAV'00)*, LNCS vol. 1855, pages 480–494, 2000.
35. A. Vogelsang, A. Fehnker, R. Huuck, and W. Reif. Software metrics in static program analysis. In *Proc. Int. Conf. on Formal Engineering Methods (ICFEM'10), Formal Methods and Software Engineering*, LNCS vol. 6447, pages 485–500. Springer, 2010.
36. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.