Monotonicity and the Precision of Program Analysis

MARCO CAMPION, Inria - ENS - Université PSL, France
MILA DALLA PREDA, University of Verona, Italy
ROBERTO GIACOBAZZI, University of Arizona, USA
CATERINA URBAN, Inria - ENS - Université PSL, France

It is widely known that the precision of a program analyzer is closely related to intensional program properties, namely, properties concerning how the program is written. This explains, for instance, the interest in code obfuscation techniques, namely, tools explicitly designed to degrade the results of program analysis by operating syntactic program transformations. Less is known about a possible relation between what the program extensionally computes, namely, its input-output relation, and the precision of a program analyzer. In this paper we explore this potential connection in an effort to isolate program fragments that can be precisely analyzed by abstract interpretation, namely, programs for which there exists a complete abstract interpretation. In the field of static inference of numeric invariants, this happens for programs, or parts of programs, that manifest a monotone (either non-decreasing or non-increasing) behavior. We first formalize the notion of program monotonicity with respect to a given input and a set of numerical variables of interest. A sound proof system is then introduced with judgments specifying whether a program is monotone relatively to a set of variables and a set of inputs. The interest in monotonicity is justified because we prove that the family of monotone programs admits a complete abstract interpretation over a specific class of non-trivial numerical abstractions and inputs. This class includes all non-relational abstract domains that refine interval analysis (i.e., at least as precise as the intervals abstraction) and that satisfy a topological convexity hypothesis.

CCS Concepts: • **Theory of computation → Program analysis**; **Abstraction**; *Program verification*; Program reasoning.

Additional Key Words and Phrases: Abstract Interpretation, Program Analysis, Complete-Analyzability, Completeness, Program Monotonicity

ACM Reference Format:
Marco Campion, Mila Dalla Preda, Roberto Giacobazzi, and Caterina Urban. 2024. Monotonicity and the Precision of Program Analysis. *Proc. ACM Program. Lang.* 8, POPL, Article 55 (January 2024), 34 pages. https://doi.org/10.1145/3632897

1 INTRODUCTION

Static program analysis has been widely investigated and used to help programmers and software engineers in producing reliable code [Distefano et al. 2019; O'Hearn 2018; Rival and Yi 2020; Sadowski et al. 2018]. Static analysis relies on symbolic reasoning and over-approximation to reason on program behaviors and to verify correctness specifications, also known as safety properties, without actually executing the programs. For instance, common safety specifications are: "Variable x is not negative" or "Variable y ranges in the interval $[a, b]$". Given a program P, a correctness

Authors' addresses: Marco Campion, Inria - ENS - Université PSL, Paris, France, marco.campion@inria.fr; Mila Dalla Preda, University of Verona, , Italy, mila.dallapreda@univr.it; Roberto Giacobazzi, University of Arizona, Tucson, USA, giacobazzi@arizona.edu; Caterina Urban, Inria - ENS - Université PSL, Paris, France, caterina.urban@inria.fr.

ACM 2475-1421/2024/1-ART55
https://doi.org/10.1145/3632897

specification *Spec* and a set of inputs S, a static analyzer either proves that the execution of P on S satisfies *Spec*, also written $[\![P]\!]S \subseteq Spec$, or it raises some alarms.

Abstract interpretation [Cousot and Cousot 1977, 1979, 1992, 2014] generalizes most existing static analysis methods into a unique sound-by-construction framework based on a simple but striking idea that extracting properties of programs' execution means over-approximating their semantics. Given an abstract domain $\mathcal{A}$ representing the properties of interest ordered by a partial order $\leq_{\mathcal{A}}$, we denote with $\alpha_{\mathcal{A}}$ and $\gamma_{\mathcal{A}}$ respectively the abstraction and concretization maps associated with $\mathcal{A}$, and with $[\![\cdot]\!]_{\mathcal{A}}$ an abstract interpreter defined on $\mathcal{A}$ and computing the abstract semantics of a program. Let us assume that *Spec* is expressible in $\mathcal{A}$, namely $Spec = \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(Spec))$. In this case, the abstract interpreter is sound when $[\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)$ implies $[\![P]\!]S \subseteq Spec$. However, due to the spurious elements introduced by the abstract interpreter, it may happen that $[\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) \not\leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)$ even if $[\![P]\!]S \subseteq Spec$. In this case the elements in $\gamma_{\mathcal{A}}([\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S)) \setminus Spec$ are called false-alarms. We have *completeness* when no false-alarms are raised when verifying *Spec*: in this optimal case, proving $[\![P]\!]S \subseteq Spec$ by executing the program is the same as checking whether $[\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)$ holds, namely, $[\![P]\!]S \subseteq Spec \Leftrightarrow [\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)$. Completeness represents an ideal and rare situation where there is no loss of precision between the abstract and concrete interpretation up to the abstraction chosen, and therefore the analysis is *precise*. Previous works have investigated the features of abstract domains and of programs under analysis that make an abstract interpreter complete.

It has been proved that completeness is possible only if the *Best Correct Approximation* (BCA) $[\![\cdot]\!]^{\alpha}_{\mathcal{A}} \stackrel{def}{=} \alpha_{\mathcal{A}} \circ [\![P]\!] \circ \gamma_{\mathcal{A}}$ of the concrete semantics of P on $\mathcal{A}$ is complete [Cousot and Cousot 1977; Giacobazzi et al. 2000]. This means that, given a program P, completeness is a domain property and domain refinements have been proposed in order to minimally transform abstraction $\mathcal{A}$ to gain completeness with respect to P [Bruni et al. 2022; Giacobazzi et al. 2000]. Observe that the BCA relies on the concrete program semantics and, in general, it may not be directly used to implement an abstract interpreter, therefore further abstractions are needed. This means that the BCA $[\![\cdot]\!]^{\alpha}_{\mathcal{A}}$ is more precise than any other abstract interpreter $[\![\cdot]\!]_{\mathcal{A}}$ on $\mathcal{A}$. Thus, the BCA represents the mathematical limit on the best precision that we can reach in abstract interpretation.

Furthermore, it has been shown that the completeness of an abstract interpreter with respect to a program P is strictly influenced by the way P is written, namely, precision in abstract interpretation is an *intensional* program property [Bruni et al. 2020, 2021; Giacobazzi et al. 2015]. This is not hard to observe, in fact it is well known that code obfuscation [Collberg and Nagra 2009] refers to syntactic program transformations explicitly designed to degrade the results of program analysis, namely to induce imprecision, and therefore incompleteness [Dalla Preda and Giacobazzi 2005; Dalla Preda et al. 2006; Giacobazzi 2008; Giacobazzi and Mastroeni 2012; Giacobazzi et al. 2017].

In this paper we investigate the *class of programs* that *can be precisely analyzed*, namely, admitting a complete abstract interpretation, over a specific family of non-trivial abstract domains and inputs. We focus our study on the analysis of numeric properties of program variables, thus considering numerical abstract domains. Exploring the possibility for programs to be precisely analyzed means that we need to refer directly to the BCA of $\mathcal{A}$ and not to a generic (less precise) abstract interpreter. This turns the focus on the concrete semantics of the program, from which the BCA derives. Note that the program equivalence induced by the BCA on an abstract domain $\mathcal{A}$, namely P is equivalent to Q if and only if $[\![P]\!]^{\alpha}_{\mathcal{A}} = [\![Q]\!]^{\alpha}_{\mathcal{A}}$, is an *extensional* equivalence, namely a property related to what a program computes and not to how it is written.

The Intuition. There are programs whose extensional behavior, namely their input-output relation, guarantees the existence of a complete abstract interpretation on a given abstract domain, whereas others deny this possibility as witnessed by the following examples. Consider the simple

rectifier program

$$\text{ReLU} \stackrel{def}{=} \textbf{if } x \leq 0 \textbf{ then } x := 0 \textbf{ else } x := x$$

also known as ReLU in artificial neural networks [Nair and Hinton 2010], that filters the input below 0. Consider the abstract domain of intervals Int, where sets of integers $S \subseteq \mathbb{Z}$ are abstracted by their bounds, i.e., the least interval $\alpha_{\mathsf{Int}}(S) = [l, u]$ such that $S \subseteq [l, u]$, where $l \in \mathbb{Z} \cup \{-\infty\}$, $u \in \mathbb{Z} \cup \{+\infty\}$, and $l \leq u$. ReLU can be soundly analyzed by the abstract interpreter $[\![\text{ReLU}]\!]_{\mathsf{Int}}([l, u]) = [l', u']$ where $l' = 0$ if $l \leq 0$ and $l' = l$ otherwise, and $u' = 0$ if $u \leq 0$ and $u' = u$ otherwise. No matter what set of numbers $S \subseteq \mathbb{Z}$ is given in input, $[\![\text{ReLU}]\!]_{\mathsf{Int}}$ is complete: $\alpha_{\mathsf{Int}}([\![\text{ReLU}]\!]S) = [\![\text{ReLU}]\!]_{\mathsf{Int}}\alpha_{\mathsf{Int}}(S)$. This means that the bounds computed by ReLU on S are not altered if we run $[\![\text{ReLU}]\!]_{\mathsf{Int}}$ on intervals. Thus, if ReLU is used as divisor in an expression $e = f(x)/\text{ReLU}(x)$, then checking division by 0 is possible without false-alarms: $0 \in \alpha_{\mathsf{Int}}([\![\text{ReLU}]\!]S) \Leftrightarrow 0 \in [\![\text{ReLU}]\!]_{\mathsf{Int}}\alpha_{\mathsf{Int}}(S)$. In this case $[\![\text{ReLU}]\!]_{\mathsf{Int}}$ corresponds to the BCA $[\![\text{ReLU}]\!]^{\alpha}_{\mathsf{Int}}$.

This is not the case for the program

$$\text{ABS} \stackrel{def}{=} \textbf{if } x \geq 0 \textbf{ then } x := x \textbf{ else } x := -x$$

computing the absolute value of x. In this case, even the BCA on Int may report a false-alarm for $e = f(x)/\text{ABS}(x)$. For instance, with $S = \{-7, 7\}$, we have: $\alpha_{\mathsf{Int}}([\![\text{ABS}]\!]\{-7, 7\}) = [7, 7]$, while $[\![\text{ABS}]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(\{-7, 7\}) = [0, 7]$ and $[7, 7] <_{\mathsf{Int}} [0, 7]$. Hence, even if $0 \notin \alpha_{\mathsf{Int}}([\![\text{ABS}]\!]S)$, interval analysis may return a potential false-alarm for e, namely $0 \in [\![\text{ABS}]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(S)$. Consequently, any sound approximation $[\![\text{ABS}]\!]_{\mathsf{Int}}$ of $[\![\text{ABS}]\!]^{\alpha}_{\mathsf{Int}}$ produces intervals larger than $[0, 7]$. However, when all the inputs in S have the same sign, no matter if positive or negative, completeness of $[\![\text{ABS}]\!]^{\alpha}_{\mathsf{Int}}$ holds. For example, with $S_1 = \{1, 3, 6\}$ and $S_2 = \{-2, -5\}$ we get

$$\alpha_{\mathsf{Int}}([\![\text{ABS}]\!]S_1) = [1, 6] = [\![\text{ABS}]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(S_1)$$
$$\alpha_{\mathsf{Int}}([\![\text{ABS}]\!]S_2) = [2, 5] = [\![\text{ABS}]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(S_2)$$

Observe that in these latter cases ABS is *monotone* on the considered input, decreasing if negative (e.g., on S_2) or increasing if positive (e.g., on S_1), while ReLU is always monotone (specifically non-decreasing) on all inputs.

Few questions naturally arise from these two examples: *Is there a relation between the monotonicity of programs and the completeness of the analysis on an abstract domain and input? If yes, is there a way to locate program fragments that behave monotonically for a given set of inputs?*

Main Contribution. In this paper we formalize and study two central notions: the notion of *monotone program*, adapting the standard mathematical notion of monotonicity to programs, and the notion of *complete-analyzability*, identifying all programs admitting a complete abstract interpretation on a given abstract domain and set of inputs. Our contribution is twofold: (1) we define a proof system able to soundly verify whether a set of program variables behaves monotonically, namely either non-decreasing or non-increasing, in the portion of program under inspection and on the given set of inputs; (2) we establish a relation between program monotonicity and the complete-analyzability property: the monotonicity of a program is a sufficient condition that guarantees the possibility of designing a precise abstract interpretation for it, over a specific collection of numerical abstractions and inputs.

After introducing some basic mathematical notions, a simple imperative language and some background on abstract interpretation (Section 2), we start with Section 3 by providing a formal definition of program monotonicity (either non-increasing or non-decreasing) with respect to a set of numerical variables and inputs of interest. Then, we present a proof system designed to verify whether a program is non-decreasing with regard to a specific set of variables and a set

of inputs. The proof judgments have the form $Mon^{\nearrow}(P, S, V)$ meaning that program P is non-decreasing on the set of inputs S with respect to variables in V (the judgments $Mon^{\searrow}(P, S, V)$ for the non-increasing case follow by duality). Our proof system is sound, meaning that all derived monotonicity judgments correspond to effectively monotone code. Similar to the analysis of program continuity [Chaudhuri et al. 2010] and differentiability [Beck and Fischer 1994], the major challenge in proving monotonicity arises from branches, namely conditional statements **if** b **then** P_1 **else** P_2 and loops **while** b **do** P. Our idea is to restrict the analysis to the Boolean guards b made by only predicates of the form $e \leq 0$ where expression e is positive linear, i.e., e has the form $\sum_{i=1}^{|Var(e)|} v_i x_i + k$ where x_i is a program variable, $v \in \mathbb{R}_{\geq 0}$, $k \in \mathbb{R}$ and $|Var(e)|$ represents the number of variables occurring in e. In this scenario, for the if-statements, whenever the two branches P_1 and P_2 are proved non-decreasing, it is sufficient to check the non-decreasing property on *boundary states*, i.e., those states that satisfy the equation $e = 0$ for the predicates $e \leq 0$ occurring in b. A similar reasoning is applied when Boolean guards b are composed by only predicates in the form $e \geq 0$. For example, ReLU is non-decreasing on variable x at any input as the branches $x := 0$ and $x := x$ are non-decreasing, and the order is preserved on the boundary state $x = 0$ after the execution of both branches: $[\![x := 0]\!](0) \leq [\![x := x]\!](0)$. This idea resembles the continuity check on if-statements presented in [Chaudhuri et al. 2010] where boundary states are checked to preserve the same values after the execution of each branch whereas, for the non-decreasing case, we check that the order is preserved. A similar approach is employed for loops, where the notion of boundary states is refined giving rise to the notion of *limit states*.

In Section 4 we investigate the relation between monotonicity and complete-analyzability. In particular, when a program P is monotone for a set of variables V, it is possible to characterize a class of non-trivial numerical abstract domains and sets of inputs where the BCA of P is complete on them when analyzing variables in V. These abstract domains are non-relational abstractions that refine interval analysis (i.e., at least as precise as intervals) and satisfy a topological convexity condition. This result explains why we have no false-alarms when using $[\![\mathrm{ReLU}]\!]_{\mathsf{Int}}$ for checking numerical properties of ReLU expressible in the interval abstraction. Moreover, as composing monotonically non-decreasing (resp. non-increasing) programs preserves the non-decreasing (resp. non-increasing) property, the complete-analyzability property also holds on the composition. For example, the program ReLU; Bin made by composing ReLU and the non-decreasing binary step program Bin $\stackrel{def}{=}$ **if** $x < 0$ **then** $x := 0$ **else** $x := 1$ preserves the complete-analyzability property.

To the best of our knowledge, the results presented in this paper establish for the first time a relation between an extensional program property (monotonicity) and the possibility of designing a precise abstract interpretation, confirming the intuition that precision in abstract interpretation, although intensional, has also an extensional aspect, namely it is influenced not only by how programs are written but also by what they compute. For instance, changing the implementation of ReLU into the semantically equivalent (on $\mathbb{Z}$) program $\mathrm{ReLU}^w \stackrel{def}{=}$ **while** $x < 0$ **do** $x := x + 1$ does not change the complete-analyzability of the program: ReLU^w is still monotonically non-decreasing on its variable x therefore it can be precisely analyzed on the interval abstract domain. This aspect sheds new lights on the existence of provably complete abstract interpreters over a family of (non-trivial) abstractions for a restricted (non-trivial) class of programs, namely, the monotone programs. The complete-analyzability property could play an important role in verifying safety-critical properties of (parts of) programs, such as runtime errors in avionics software [Bertrane et al. 2011, 2015], where even false-alarms are not admissible. Our proof system may help in factorizing programs in sub-components that behave monotonically. On these sub-components, precise program analyses can be obtained by using computationally less expensive non-relational abstract domains, such as the intervals abstraction.

2 PRELIMINARIES

After introducing some preliminaries on sets and order theory, in Section 2.1 we define a simple untyped deterministic while-language and its collecting denotational semantics, while in Section 2.2 we provide a recap of the necessary background on the abstract interpretation framework.

2.1 Programs and Semantics

Order Theory. Given two sets S and T, $\wp(S)$ denotes the powerset of S, the symbol $\varnothing$ corresponds to the empty set, $S \setminus T$ denotes the set-difference, $|S|$ denotes the cardinality of S, $S \subseteq T$ denotes sets inclusion while $S \subset T$ denotes strict sets inclusion. We denote with $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ the sets of all, respectively, natural, integer and real numbers, and with $\mathbb{I} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ one of the three mentioned sets. A set $S \subseteq \mathbb{R}^n$ is *convex* when, for all $x, y \in S$ and for any scalar $t \in [0, 1]$, the vector $(1-t)x+ty$ is also in S. Although the notion of convexity does not directly apply to sets of integers, we misuse the term convex to indicate also when a set of (vectors of) integers $S \subseteq \mathbb{I}^n$ exhibits a form of convexity. In this case, we refer to the definition of integrally convex set [Yang 2009]. Specifically, a set of vectors of integers $S \subseteq \mathbb{Z}^n$ is convex if any point y in the convex hull (which is a subset of $\mathbb{R}^n$) of S can be expressed as a convex combination of the points of S that are "near" y, where "near" means that the (Euclidean) distance between each two coordinates is less than 1. So for instance, the set $\{0, 1, 2\}$ is convex because it represents a consecutive sequence of integers without "holes", which is not the case for $\{0, 1, 2, 5\}$ as the integer numbers 3 and 4 are missing. When a binary relation $\sim \subseteq S \times S$ is defined over a set which differs from $\mathbb{N}, \mathbb{Z}$ and $\mathbb{R}$, we will use the subscript $\sim_S$ except for the straightforward equivalence relation $=$. A set L endowed with a partial order relation $\leq_L$ is called a partially ordered set, or briefly poset, and it is denoted by $\langle L, \leq_L \rangle$. Its strict version is denoted by the symbol $<_L$ such that for all $x, y \in L$, $x <_L y$ if and only if $x \leq_L y$ and $x \neq y$. We will consider posets L for which all subsets $X \subseteq L$ have a unique join, also called least upper bound (lub), denoted $\bigvee_L X$, and a unique meet, also called greatest lower bound (glb), denoted $\bigwedge_L X$. The tuple $\langle L, \leq_L, \vee_L, \wedge_L, \top_L, \bot_L \rangle$, where $\top_L$ and $\bot_L$ are, respectively, the greatest (top) and least (bottom) elements in L, while $\vee_L$ and $\wedge_L$ are, respectively, the lub and glb binary operators, is called a complete lattice.

Monotonicity plays a central role in our work. The following represents the canonical definition of monotonicity of functions over posets [Scott and Strachey 1971] (to simplify the presentation, we consider unary functions):

*Definition 2.1 (**Monotone mappings**).* A function $f : L \longrightarrow L$ over a poset $\langle L, \leq_L \rangle$ is *non-decreasing* (resp. *non-increasing*) if and only if for all $x, y \in L$ such that $x \leq_L y$, f *preserves* (resp. *reverses*) the order, i.e., $f(x) \leq_L f(y)$ (resp. $f(x) \geq_L f(y)$).

f is *monotone* if it is either non-decreasing or non-increasing. ■

The composition of two functions $f_1 : L_1 \longrightarrow L_2$, $f_2 : L_2 \longrightarrow L_3$ is denoted by $f_2 \circ f_1 : L_1 \longrightarrow L_3$. A function $f : L_1 \longrightarrow L_2$ between complete lattices is additive (resp. co-additive) if for all $Y \subseteq L_1$, $f(\vee_{L_1} Y) = \vee_{L_2} f(Y)$ (resp. $f(\wedge_{L_1} Y) = \wedge_{L_2} f(Y)$). The Knaster-Tarski theorem guarantees that if L is a complete lattice and $f : L \longrightarrow L$ a monotone function, then the set of fixpoints of f in L is also a complete lattice. As a consequence, since complete lattices cannot be empty (they must contain the supremum of empty set), the theorem guarantees the existence of at least one fixpoint of f, and even the existence of a least (or greatest) fixpoint, denoted $\mathsf{lfp}(f)$ (resp. $\mathsf{gfp}(f)$). Moreover, if $f : L \longrightarrow L$ is (Scott) continuous, i.e., f preserves lubs of chains in L, then $\mathsf{lfp}(f) = \bigvee_{L\, n \in \mathbb{N}} f^n(\bot_L)$, where, for all $n \in \mathbb{N}$ and $x \in L$, f^n is inductively defined by: $f^0(x) \stackrel{def}{=} x$ and $f^{n+1}(x) \stackrel{def}{=} f(f^n(x))$.

Syntax and Semantics. For our purposes we consider a standard untyped deterministic while-language Prog with no runtime errors, as e.g. the one defined in [Winskel 1993], with the syntax

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

Fig. 1. Collecting denotational semantics of Prog.

defined as follows:

$$\begin{aligned}
\mathrm{AExp} \ni \mathrm{e} &::= v \in \mathbb{I} \mid x \in \mathit{Var} \mid \mathrm{e} + \mathrm{e} \mid \mathrm{e} - \mathrm{e} \mid \mathrm{e} * \mathrm{e}\\
\mathrm{BExp} \ni \mathrm{b} &::= \mathit{true} \mid \mathit{false} \mid \mathrm{e} \lessdot 0 \mid \mathrm{e} \gtrdot 0 \mid \mathrm{b} \wedge \mathrm{b} \mid \mathrm{b} \vee \mathrm{b}\\
\mathrm{Prog} \ni P &::= \textbf{skip} \mid x := \mathrm{e} \mid P_1; P_2 \mid\\
&\qquad \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \mid \textbf{while } b \textbf{ do } P
\end{aligned}$$

where $\lessdot \in \{<, \leq\}$, $\gtrdot \in \{>, \geq\}$ and, by abusing notation, *Var* is both used to denote a denumerable set of variables and, when applied to a program $P \in \mathrm{Prog}$, denotes the (finite) set of variables in the text of P, namely, $\mathit{Var} : \mathrm{Prog} \to \wp(\mathit{Var})$. Similarly, when applied to arithmetic and Boolean expressions, $\mathit{Var}(e)$ and $\mathit{Var}(b)$ denote the variables appearing in those expressions. We sometimes abbreviate operations like $2 * x$ and $x * x$ to, respectively, $2x$ and x^2. From now on and in the rest of the paper, whenever we talk about a program $P \in \mathrm{Prog}$, we assume $|\mathit{Var}(P)| = n$, unless otherwise specified. A *store* σ_P for P is a total function from variables in the text of P to their values, namely, $\sigma_P : \mathit{Var}(P) \to \mathbb{I}$. A store σ_P can be equivalently specified as a n-tuple $(v_1, \ldots, v_n) \in \mathbb{I}^n$ where for all $i \in [1, n]$ and $\mathit{Var}(P) = \{x_1, \ldots, x_n\}$, $\sigma(x_i) = v_i$, therefore $\mathbb{I}^{|\mathit{Var}(P)|}$ is the set of all possible stores for P. Most of the examples shown in the paper consider programs with $\mathit{Var}(P) = \{x, y, z\}$, so that, a tuple like $(10, 2, 5) \in \mathbb{I}^3$ corresponds to the store σ_P such that $\sigma_P(x) = 10$, $\sigma_P(y) = 2$ and $\sigma_P(z) = 5$. A single store update is written $\sigma_P[x \mapsto v]$. We will omit the subscript P to σ when it is clear from context. The semantics of arithmetic and Boolean expressions of P is defined by the functions, respectively, $(\!|\mathrm{e}|\!) : \mathbb{I}^n \to \mathbb{I}$ and $(\!|\mathrm{b}|\!) : \mathbb{I}^n \to \{\mathit{true}, \mathit{false}\}$ whose definitions are straightforward and therefore omitted. The collecting semantics of arithmetic and Boolean expressions is respectively defined by the functions $[\![\mathrm{e}]\!] : \wp(\mathbb{I}^n) \to \wp(\mathbb{I})$ and $[\![\mathrm{b}]\!] : \wp(\mathbb{I}^n) \to \wp(\mathbb{I}^n)$ defined as: $[\![\mathrm{e}]\!]S \stackrel{def}{=} \{(\!|\mathrm{e}|\!)\sigma \mid \sigma \in S\}$ and $[\![\mathrm{b}]\!]S \stackrel{def}{=} \{\sigma \in S \mid (\!|\mathrm{b}|\!)\sigma = \mathit{true}\}$ so that $[\![\mathrm{b}]\!]S \subseteq S$ filters the stores of S making b true. The *collecting denotational program semantics* is $[\![P]\!] : \wp(\mathbb{I}^n) \to \wp(\mathbb{I}^n)$ and it is defined in Fig. 1, where the operator $\neg\,\mathrm{b}$ transforms the Boolean expression b into its negate. It is the standard predicate transformer semantics (also called strongest postcondition semantics) since $[\![P]\!]S \in \wp(\mathbb{I}^n)$ turns out to be the strongest store predicate for the store precondition $S \in \wp(\mathbb{I}^n)$. The terminology "collecting semantics" comes from the fact that for all $P \in \mathrm{Prog}$, $[\![P]\!] : \wp(\mathbb{I}^{|\mathit{Var}(P)|}) \to \wp(\mathbb{I}^{|\mathit{Var}(P)|})$ is an additive function on the complete lattice $\langle \wp(\mathbb{I}^{|\mathit{Var}(P)|}), \subseteq, \cup, \cap, \mathbb{I}^{|\mathit{Var}(P)|}, \varnothing \rangle$, so that $[\![P]\!]S = \cup_{\sigma \in S}\ [\![P]\!]\{\sigma\}$ holds. When $[\![P]\!]$ is applied to a singleton $\{\sigma\}$, we use the simpler notation $[\![P]\!]\sigma$ in place of $[\![P]\!]\{\sigma\}$.

2.2 Abstract Interpretation

We recall some background on abstract interpretation as defined by Cousot [2021]; Cousot and Cousot [1977, 1979, 1992] and based on the correspondence between a domain of concrete or exact properties and a domain of abstract or approximate properties.

Abstract Domains. In the following we consider abstract domains (also called abstractions) as specified by Galois connections/insertions (GCs/GIs for short). Concrete and abstract domains are assumed to be complete lattices, respectively $\langle C, \leq_C \rangle$ and $\langle \mathcal{A}, \leq_{\mathcal{A}} \rangle$, which are related by abstraction and concretization maps, $\alpha_{\mathcal{A}} : C \to \mathcal{A}$ and $\gamma_{\mathcal{A}} : \mathcal{A} \to C$, that give rise to a GC $(\alpha_{\mathcal{A}}, C, \mathcal{A}, \gamma_{\mathcal{A}})$, that is, for all $a \in \mathcal{A}$ and $c \in C$: $\alpha_{\mathcal{A}}(c) \leq_{\mathcal{A}} a \;\Leftrightarrow\; c \leq_C \gamma_{\mathcal{A}}(a)$, where we use the subscript to functions $\alpha_{\mathcal{A}}$ and $\gamma_{\mathcal{A}}$ in order to emphasize the abstract domain $\mathcal{A}$ considered. A GC is a GI when $\alpha_{\mathcal{A}} \circ \gamma_{\mathcal{A}} = \lambda x.x$. Let us recall some basic properties of a GC $(\alpha_{\mathcal{A}}, C, \mathcal{A}, \gamma_{\mathcal{A}})$: (1) $\alpha_{\mathcal{A}}$ is additive and $\gamma_{\mathcal{A}}$ is co-additive; (2) $\gamma_{\mathcal{A}} \circ \alpha_{\mathcal{A}} : C \to C$ is a closure operator, namely, it is a monotone, idempotent and increasing function; (3) if $\rho : C \to C$ is a closure operator then $(\rho, C, \rho(C), \lambda x.x)$ is a GI. For our purposes, we will deal only with GI which are standard in abstract interpretation (e.g., Sign, Intervals, Zones, etc.) ensuring the existence of abstraction functions. We use $\mathrm{Abs}(C)$ to denote all the possible abstractions of a concrete domain C, where $\mathcal{A} \in \mathrm{Abs}(C)$ means that $\mathcal{A}$ is an abstract domain of C defined by some GI which is left unspecified. We say that a concrete element $c \in C$ is *representable* (or expressible) in $\mathcal{A}$ whenever $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(c)) = c$. If we consider two abstract domains $\mathcal{A}_1, \mathcal{A}_2 \in \mathrm{Abs}(C)$ then $\mathcal{A}_1$ is a *more precise* abstraction than $\mathcal{A}_2$, or, equivalently, $\mathcal{A}_2$ abstracts $\mathcal{A}_1$, if and only if for all $c \in C$, $\gamma_{\mathcal{A}_1}(\alpha_{\mathcal{A}_1}(c)) \leq_C \gamma_{\mathcal{A}_2}(\alpha_{\mathcal{A}_2}(c))$, and it is denoted by $\mathcal{A}_1 \leq_{\mathrm{Abs}(C)} \mathcal{A}_2$. An abstract domain $\mathcal{A} \in \mathrm{Abs}(C)$ is said to be *trivial* when $\mathcal{A} = C$, namely, it is isomorphic to the concrete domain C (i.e., $\gamma_{\mathcal{A}} \circ \alpha_{\mathcal{A}}$ is the identity function).

Non-relational Abstractions. Given a program P with $|Var(P)| = n$ variables, an abstract domain in $\mathrm{Abs}(\wp(\mathbb{I}^n))$ is said to be *non-relational* when it does not take into account any relationship between different variables. Let $\mathcal{A} \in \mathrm{Abs}(\wp(\mathbb{I}))$ be any abstract domain abstracting $\wp(\mathbb{I})$, then $\mathcal{A}^n \in \mathrm{Abs}(\wp(\mathbb{I}^n))$ is its non-relational extension to n variables with abstraction and concretization maps, respectively, $\alpha_{\mathcal{A}^n} : \wp(\mathbb{I}^n) \to \mathcal{A}^n$ and $\gamma_{\mathcal{A}^n} : \mathcal{A}^n \to \wp(\mathbb{I}^n)$, defined as follows. $\mathcal{A}^n$ is the domain of abstract tuples $(a_1, \ldots, a_n) \in \mathcal{A}^n$ with $a_i \in \mathcal{A}$, representing the abstract stores, equivalently denoted by the function $\sigma^\sharp : Var(P) \to \mathcal{A}$, where $\sigma^\sharp(x_i) = a_i$ returns the abstract value assumed by variable x_i. We denote with $\bot_{\mathcal{A}^n} \stackrel{def}{=} (\bot_{\mathcal{A}}, \ldots, \bot_{\mathcal{A}})$ the bottom element, with $\top_{\mathcal{A}^n} \stackrel{def}{=} (\top_{\mathcal{A}}, \ldots, \top_{\mathcal{A}})$ the top element, and with $\leq_{\mathcal{A}^n}$ the order on abstract stores: $\sigma_1^\sharp \leq_{\mathcal{A}^n} \sigma_2^\sharp$ if and only if $\sigma_1^\sharp = \bot_{\mathcal{A}^n}$, or $\sigma_1^\sharp, \sigma_2^\sharp \neq \bot_{\mathcal{A}^n}$ and $\forall i \in [1, n] : \sigma_1^\sharp(x_i) \leq_{\mathcal{A}} \sigma_2^\sharp(x_i)$. Given $S \in \wp(\mathbb{I}^n)$ and $\sigma^\sharp \in \mathcal{A}^n$, then $\alpha_{\mathcal{A}^n}(S)$ and $\gamma_{\mathcal{A}^n}(\sigma^\sharp)$ can be defined as, respectively:

$$\alpha_{\mathcal{A}^n}(S) \stackrel{def}{=} \begin{cases} \bot_{\mathcal{A}^n} & \text{if } S = \varnothing, \\ (a_1, \ldots, a_n) \in \mathcal{A}^n \text{ where } a_i \stackrel{def}{=} \alpha_{\mathcal{A}}(\{\sigma(x_i) \mid \sigma \in S\}) & \text{otherwise} \end{cases}$$

$$\gamma_{\mathcal{A}^n}(\sigma^\sharp) \stackrel{def}{=} \begin{cases} \varnothing & \text{if } \sigma^\sharp = \bot_{\mathcal{A}^n}, \\ \{(v_1, \ldots, v_n) \in \mathbb{I}^n \mid v_i \in \gamma_{\mathcal{A}}(\sigma^\sharp(x_i))\} & \text{otherwise.} \end{cases}$$

In what follows, we abuse notation and drop the superscript n in $\mathcal{A}^n$, $\alpha_{\mathcal{A}^n}$, $\gamma_{\mathcal{A}^n}$, $\bot_{\mathcal{A}^n}$, $\top_{\mathcal{A}^n}$ and $\leq_{\mathcal{A}^n}$ as it will be clear from the context. The following are examples of non-relational abstractions in $\mathrm{Abs}(\wp(\mathbb{I}))$[1].

Example 2.2. The classical pedagogical examples include the abstract domains $\mathrm{Sign} \stackrel{def}{=} \{\mathbb{I}, -, 0, +, \varnothing\}$ and $\mathrm{Parity} \stackrel{def}{=} \{\mathbb{Z}, \mathit{even}, \mathit{odd}, \varnothing\}$ for, respectively, *sign* and *parity* analysis of numerical variables. These are straightforward non-relational abstractions of $\langle \wp(\mathbb{I}), \subseteq \rangle$ [Cousot and Cousot 1976], namely, $\mathrm{Sign} \in \mathrm{Abs}(\wp(\mathbb{I}))$ and $\mathrm{Parity} \in \mathrm{Abs}(\wp(\mathbb{Z}))$, where the order relation $\leq_{\mathrm{Sign}}$ is defined as $\varnothing <_{\mathrm{Sign}} 0 <_{\mathrm{Sign}} - <_{\mathrm{Sign}} \mathbb{I}$ and $\varnothing <_{\mathrm{Sign}} 0 <_{\mathrm{Sign}} + <_{\mathrm{Sign}} \mathbb{I}$, while $\varnothing <_{\mathrm{Parity}} \mathit{even} <_{\mathrm{Parity}} \mathbb{Z}$ and

[1]For simplicity, we assume that we use some perfect mathematical version of numeric sets and not machine-integers nor floating-point numbers used actually in most computer languages.

$\varnothing <_{\mathsf{Parity}} \mathit{odd} <_{\mathsf{Parity}} \mathbb{Z}$. The abstraction maps $\alpha_{\mathsf{Sign}} : \wp(\mathbb{I}) \to \mathsf{Sign}$ and $\alpha_{\mathsf{Parity}} : \wp(\mathbb{Z}) \to \mathsf{Parity}$ are defined as follows:

$$\alpha_{\mathsf{Sign}}(X) \stackrel{def}{=} \begin{cases} \varnothing & \text{if } X = \varnothing, \\ 0 & \text{if } X = \{0\}, \\ + & \text{if } \forall x \in X.\ x \geq 0, \\ - & \text{if } \forall x \in X.\ x \leq 0, \\ \mathbb{I} & \text{otherwise} \end{cases} \qquad \alpha_{\mathsf{Parity}}(X) \stackrel{def}{=} \begin{cases} \varnothing & \text{if } X = \varnothing, \\ \mathit{even} & \text{if } \forall x \in X.\ x \bmod 2 = 0, \\ \mathit{odd} & \text{if } \forall x \in X.\ x \bmod 2 \neq 0, \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

where mod is the integer modulo operation. ♦

Example 2.3. The *interval* abstraction Int [Cousot and Cousot 1976] is an efficient and useful non-relational abstract domain for deriving bounds to numerical variables, e.g., the absence of arithmetic overflows or out-of-bounds array accesses. Let $\mathbb{I}^* \stackrel{def}{=} \mathbb{I} \cup \{-\infty, +\infty\}$ and assume that the standard ordering $\leq$ on $\mathbb{I}$ is extended to $\mathbb{I}^*$ in the usual way. Hence $\mathsf{Int} \stackrel{def}{=} \{[a, b] \mid a, b \in \mathbb{I}^*,\ a \leq b\} \cup \{\bot_{\mathsf{Int}}\}$ endowed with the standard ordering $\leq_{\mathsf{Int}}$ induced by the interval containment gives rise to a complete lattice, where $\bot_{\mathsf{Int}}$ is the bottom element and $\top_{\mathsf{Int}} \stackrel{def}{=} [-\infty, +\infty]$ is the top element. We have that $\mathsf{Int} \in \mathsf{Abs}(\wp(\mathbb{I}))$. Consider the function $\mathit{min} : \wp(\mathbb{I}) \to \mathbb{I}^*$ defined as $\mathit{min}(S) \stackrel{def}{=} x$ if there exists $x \in S$ such that for all $y \in S$ $x \leq y$, while $\mathit{min}(S) \stackrel{def}{=} -\infty$ otherwise, and the function $\mathit{max} : \wp(\mathbb{I}) \to \mathbb{I}^*$ dually defined. The abstraction map $\alpha_{\mathsf{Int}} : \wp(\mathbb{I}) \to \mathsf{Int}$ is defined by:

$$\alpha_{\mathsf{Int}}(X) \stackrel{def}{=} \begin{cases} \bot_{\mathsf{Int}} & \text{if } X = \varnothing, \\ [\mathit{min}(X), \mathit{max}(X)] & \text{otherwise.} \end{cases}$$

Note that α_{Int} preserves arbitrary unions in $\wp(\mathbb{I})$ and therefore gives rise to a GI. ♦

Abstract Interpretation. Let $f : C \to C$ be a concrete monotone (transfer) function (to keep notation simple we consider unary functions) and let $f^\sharp : \mathcal{A} \to \mathcal{A}$ be a corresponding abstract (transfer) function defined on some abstraction $\mathcal{A} \in \mathsf{Abs}(C)$. Then, $f^\sharp$ is a *correct* (or *sound*) approximation of f on $\mathcal{A}$ when $\alpha_{\mathcal{A}} \circ f \leq_{\mathcal{A}} f^\sharp \circ \alpha_{\mathcal{A}}$ holds. If $f^\sharp$ is correct for f then least fixpoint correctness holds, that is, $\alpha_{\mathcal{A}}(\mathsf{lfp}(f)) \leq_{\mathcal{A}} \mathsf{lfp}(f^\sharp)$ holds. When dealing with GIs between all abstract transfer functions that approximate a concrete one, we can define the most precise one.

*Definition 2.4 (**Best correct approximation**).* The abstract function $f^\alpha : \mathcal{A} \to \mathcal{A}$ defined as $f^\alpha \stackrel{def}{=} \alpha_{\mathcal{A}} \circ f \circ \gamma_{\mathcal{A}}$ is called the *best correct approximation* (BCA for short) of f on $\mathcal{A}$. ■

It turns out that any abstract function $f^\sharp$ is a correct approximation of f if and only if $f^\alpha \leq_{\mathcal{A}} f^\sharp$ [Cousot and Cousot 1977]. An abstract function $f^\sharp$ is precise when it is complete.

*Definition 2.5 (**Complete approximations over an input**).* Given an input $c \in C$, an abstract function $f^\sharp : \mathcal{A} \to \mathcal{A}$ is said to be a *complete approximation* of $f : C \to C$ on $\mathcal{A}$ *at the input* c, when $\alpha_{\mathcal{A}}(f(c)) = f^\sharp(\alpha_{\mathcal{A}}(c))$ holds. ■

This definition of completeness is taken from the *local* completeness notion introduced by Bruni et al. [2021, 2023], which is a weakening of the standard notion of completeness requiring Definition 2.5 to hold over all possible inputs $c \in C$ [Cousot 2021; Giacobazzi et al. 2000]. Since we deal with local properties only, namely, properties requiring to specify an input (e.g., completeness, monotonicity, convexity, etc.), we will simply omit the word "local" as the input will be always specified. Conversely, when we do not specify the input, we implicitly assume that the property holds for all possible inputs. Intuitively, when $f^\sharp$ is an abstract transfer function on $\mathcal{A}$ used in some static program analysis algorithm, completeness encodes an optimal precision for $f^\sharp$ at input c, meaning that the

abstract behavior of $f^\sharp$ on $\mathcal{A}$ *exactly* matches the abstraction in $\mathcal{A}$ of the concrete behavior of f. It turns out that the possibility of defining a complete approximation $f^\sharp$ of f on some $\mathcal{A} \in \mathrm{Abs}(C)$ and $c \in C$ only depends upon the concrete function f, the abstraction $\mathcal{A}$ and the input c, that is, $f^\alpha(c)$ is *the only possible option as complete approximation* of $f(c)$, as stated by the following theorem [Bruni et al. 2021; Giacobazzi et al. 2000].

Theorem 2.6. *Completeness of a sound abstract function $f^\sharp$ over an input $c \in C$ holds if and only if*

$$\alpha_{\mathcal{A}}(f(c)) = \alpha_{\mathcal{A}}(f(\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(c)))) = f^\alpha(\alpha_{\mathcal{A}}(c)) = f^\sharp(\alpha_{\mathcal{A}}(c)) \qquad \square$$

Completeness and Static Verification. The abstract interpreter applied to a program $P \in \mathrm{Prog}$ is specified by the function $[\![P]\!]_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$, where $\mathcal{A} \in \mathrm{Abs}(\wp(\mathbb{I}^{|Var(P)|}))$ is the abstraction of properties of interest and $[\![P]\!]_{\mathcal{A}}$ soundly approximates the concrete semantics $[\![P]\!]$ on the abstract domain $\mathcal{A}$. We denote with $[\![P]\!]^\alpha_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ the abstract interpreter given by the BCA and defined as $[\![P]\!]^\alpha_{\mathcal{A}} \stackrel{def}{=} \alpha_A \circ [\![P]\!] \circ \gamma_A$. We are not going to further specify how the abstract semantics $[\![P]\!]_{\mathcal{A}}$ is defined, since, thanks to Theorem 2.6, in order to conclude that $[\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S)$ is complete over the input $S \in \wp(\mathbb{I}^{|Var(P)|})$, it is sufficient to show that $[\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) = [\![P]\!]^\alpha_{\mathcal{A}}\alpha_{\mathcal{A}}(S) = \alpha_{\mathcal{A}}([\![P]\!]S)$. We will use the symbol $[\![\cdot]\!]_{\mathcal{A}}$ for referring to a generic abstract interpreter without specifying the program.

The goal of a static analysis is to soundly answer some questions on the dynamic (concrete) execution of programs. More specifically, given a program $P \in \mathrm{Prog}$, an input $S \subseteq \mathbb{I}^n$ and a *safety* property (also called *correctness* property) $Spec \subseteq \mathbb{I}^n$ representable in our chosen abstract domain $\mathcal{A}$, the aim of a static verification $[\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S)$ is either to prove $[\![P]\!]S \subseteq Spec$, namely that the behavior of P on input S satisfies $Spec$, or to raise some alerts that point out which circumstances may cause a violation of $Spec$. The presence of false alarms is in this case unavoidable due to the need of program verifiers $[\![P]\!]_{\mathcal{A}}$ to over-approximate the program behaviour $[\![P]\!]$: this is an unavoidable consequence of the will to solve an otherwise undecidable analysis problem. However, when the abstract interpreter is proved to be complete on P with input S, namely when $\alpha_{\mathcal{A}}([\![P]\!]S) = [\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S)$, then proving $[\![P]\!]S \subseteq Spec$ by executing the program is the same as checking whether $[\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)$ holds, i.e. no false alarms can arise from checking the specification through the abstract interpreter: all the raised alarms are surely real. This is summarized by the following theorem:

Theorem 2.7. *If $[\![P]\!]_{\mathcal{A}}$ is complete at the set of inputs $S \subseteq \mathbb{I}^n$ and $Spec \subseteq \mathbb{I}^n$ is representable in $\mathcal{A}$, then the following holds: $[\![P]\!]S \subseteq Spec \Leftrightarrow [\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)$.*

Proof. Let $\mathcal{A} \in \mathrm{Abs}(\wp(\mathbb{I}^n))$, $[\![P]\!]_{\mathcal{A}}$ be complete at input S, and $Spec = \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(Spec))$.

($\Leftarrow$) This implication is a direct consequence of the soundness assumption of $[\![P]\!]_{\mathcal{A}}$ and it does not make use of the completeness hypothesis:

$$\begin{aligned}
[\![P]\!]_{\mathcal{A}}\alpha(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec) &\Rightarrow [\text{by Soundness of } [\![\cdot]\!]_{\mathcal{A}}]\\
\alpha_{\mathcal{A}}([\![P]\!]S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec) &\Rightarrow [\text{by } \gamma_{\mathcal{A}} \text{ monotone}]\\
\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S)) \subseteq \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(Spec)) &\Rightarrow [\text{by } Spec = \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(Spec))]\\
\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S)) \subseteq Spec &\Rightarrow [\text{by } \gamma_{\mathcal{A}} \circ \alpha_{\mathcal{A}} \text{ closure operator}]\\
[\![P]\!]S \subseteq Spec
\end{aligned}$$

($\Rightarrow$) This implication is a direct consequence of the completeness assumption of $[\![P]\!]_{\mathcal{A}}$ at S:

$$\begin{aligned}
[\![P]\!]S \subseteq Spec &\Rightarrow [\text{by } \alpha_{\mathcal{A}} \text{ monotone}]\\
\alpha_{\mathcal{A}}([\![P]\!]S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec) &\Rightarrow [\text{by } [\![P]\!]_{\mathcal{A}} \text{ complete at } S]\\
[\![P]\!]_{\mathcal{A}}\alpha(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)
\end{aligned}$$

$\square$

3 VERIFYING MONOTONICITY

In this section we: (1) *adapt* the notion of monotonicity to programs, and (2) *verify* whether a given set of program variables manifests a monotone behavior in the considered program on a specified set of inputs. This last point requires to understand how monotonicity is propagated during computation.

Let us start by defining when a program can be considered monotone. Definition 2.1 already provides us the standard notion of monotonicity for functions over posets. Given a program $P \in \mathsf{Prog}$, its denotational semantics $[\![P]\!]$ over singletons $\sigma \in \mathbb{I}^{|Var(P)|}$ can be considered as a function from $\langle \mathbb{I}^{|Var(P)|}, \leq \rangle$ to $\langle \mathbb{I}^{|Var(P)|}, \leq \rangle$ where here $\leq$ is the componentwise inequality between tuples (stores). This means that Definition 2.1 can also be adopted for defining when a program preserves or reverses the order of its inputs. However, since programs may manipulate temporary variables or variables associated to computations that are not the target of our study, we may be interested in a notion of monotonicity that considers only a subset of program variables. For this reason we introduce the notion of *V-monotonicity* that is parametric w.r.t. a finite set $V \subseteq Var$ of program variables. Let $P \in \mathsf{Prog}$ be a program with $|Var(P)| = n$ variables, and $\leq^V$ be the elementwise inequality only for variables in V and that also appear in $Var(P)$, formally, $\forall \sigma_1, \sigma_2 \in \mathbb{I}^n$:

$$\sigma_1 \leq^V \sigma_2 \overset{def}{\Leftrightarrow} \forall x \in V \cap Var(P).\ \sigma_1(x) \leq \sigma_2(x)$$

and, similarly, $=^V$ be the elementwise equality for V. Further, let $\mathcal{D}_P \subseteq \mathbb{I}^n$ be the domain of P, namely, the set of input stores over which P terminates: $\mathcal{D}_P \overset{def}{=} \{\sigma \in \mathbb{I}^n \mid [\![P]\!]\sigma \neq \varnothing\}$.

*Definition 3.1 (**V-Monotone program**).* The program $P \in \mathsf{Prog}$ is said to be *V-non-decreasing* (resp. *V-non-increasing*) at inputs $S \subseteq \mathbb{I}^n$ for the variables in $V \subseteq Var$, if and only if for all $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_P$ the following condition holds:

$$\sigma_1 \leq^V \sigma_2 \Rightarrow [\![P]\!]\sigma_1 \leq^V [\![P]\!]\sigma_2 \quad (\text{resp. } [\![P]\!]\sigma_1 \geq^V [\![P]\!]\sigma_2)$$

P is called *V-monotone* at $S \subseteq \mathbb{I}^n$ if it is either V-non-decreasing or V-non-increasing at S. ■

Intuitively, a program P is V-non-decreasing (resp. V-non-increasing) for a set of variables $V \subseteq Var$ whenever for all comparable input stores w.r.t. V and for which P terminates, the execution of P *preserves* (resp. *reverses*) the relative order of stores on variables in V. There are no constraints on the result of computation from input states that are not comparable or on the values of variable not in V: they can behave in a non-monotone way.

Example 3.2. Consider the sequential composition of the following assignments:

$$P \ : \ x := 2x; \ \ y := y + 1; \ \ z := x + y$$

Variables x and y increase monotonically their respective inputs, and z is the sum of those two variables. This implies that each time we take two states $\sigma_1, \sigma_2 \in \mathbb{R}^3$ such that $\sigma_1 \leq^{\{x,y,z\}} \sigma_2$, after executing P on both input states, the order is preserved: $[\![P]\!]\sigma_1 \leq^{\{x,y,z\}} [\![P]\!]\sigma_2$. Therefore, we can conclude that this program is $\{x, y, z\}$-non-decreasing over $\mathbb{R}^3$. However, note that P is not $\{x, z\}$-non-decreasing: in this case we can consider states having the y component change arbitrarily. Consider $(1, 0, 0) \leq^{\{x,z\}} (2, -10, 0)$, then $[\![P]\!](1, 0, 0) = (2, 1, 3) \not\leq^{\{x,z\}} (4, -9, -5) = [\![P]\!](2, -10, 0)$. ♦

When we say that P is V-monotone (or V-non-decreasing or V-non-increasing) without specifying the set of input stores, we implicitly assume that P is V-monotone (or V-non-decreasing or V-non-increasing) over all possible inputs $\mathbb{I}^{|Var(P)|}$. Similarly, if we do not specify the set V of variables, we implicitly assume for all variables used in the considered program. This last assumption also applies to $\leq$ and $=$ when they are used for comparing stores.

3.1 Monotonicity Judgments

Given a portion of code, can we automate the process of proving whether a set of variables behave monotonically w.r.t. Definition 3.1? The general answer is no because our core language Prog is Turing-complete and monotonicity is an extensional program property, therefore undecidable [Rice 1953]. For this reason, we propose a sound approximation by defining a proof system able to infer, inductively from the program syntax, the monotonicity judgment $Mon(P, S, V)$ meaning that program P is V-monotone w.r.t. the set of variables $V \subseteq Var$ and over a set of inputs S[2]. We write $\vdash Mon(P, S, V)$ when the proof system allows us to formalize a derivation for the monotonicity judgement $Mon(P, S, V)$. We will show that the proposed proof system is *sound*, namely, $\vdash Mon(P, S, V)$ implies that $Mon(P, S, V)$ holds. Our proof system computes an *underapproximation* of the sets of monotone variables:

$$\{V \subseteq Var \mid \vdash Mon(P, S, V)\} \subseteq \{R \subseteq Var \mid Mon(P, S, R)\}.$$

Since the analysis targets a Turing-complete language, the proof system is *incomplete*, i.e., it may happen that a program is V-monotone but the verifier is not able to find a derivation for proving it.

From the next section, we will focus our attention on the analysis of the V-non-decreasing program property, also denoted by the predicate $Mon^{\nearrow}(P, S, V)$ (the non-increasing case $Mon^{\searrow}(P, S, V)$ follows by duality). We start by providing the rules for expressions, assignments and command compositions (Fig. 2), then we proceed by treating programs with if-branches (Fig. 3) and while-loops (Fig. 5).

3.2 Expressions, Assignments and Sequential Compositions

To infer that $x := e$ is V-non-decreasing at S, we need to verify that for all $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_{x:=e}$ it holds that $\sigma_1 \leq^V \sigma_2 \implies [\![x := e]\!]\sigma_1 \leq^V [\![x := e]\!]\sigma_2$. For $x \in V$, we can soundly derive monotonicity whenever the following three conditions are satisfied: (1) the expression e uses only variables in V, i.e., $Var(e) \subseteq V$, (2) the set of inputs S is convex, and (3) the gradient of the function $(\!|e|\!) : \mathbb{I}^{|Var(e)|} \to \mathbb{I}$, i.e., the column vector of all partial derivatives of $(\!|e|\!)$ denoted

$$\nabla(\!|e|\!) \stackrel{def}{=} \left(\frac{\partial(\!|e|\!)}{\partial x_1}, \ldots, \frac{\partial(\!|e|\!)}{\partial x_{|Var(e)|}}\right)^T$$

is always non-negative at all inputs in the convex space S (for the mathematical notion of differentiation of multivariate functions see, e.g., [Trench 2013]). Since our language Prog admits only polynomial expressions, all functions $(\!|e|\!)$ are differentiable over $\mathbb{I}^{|Var(e)|}$, namely, the gradient $\nabla(\!|e|\!)$ is always defined and can be obtained by symbolic differentiation. By requiring $\nabla(\!|e|\!) \geq \mathbf{0}_{|Var(e)|\times 1}$, where $\mathbf{0}_{|Var(e)|\times 1}$ is the column vector of all 0s having number of rows equal to $|Var(e)|$, the result is a system of constraints on $Var(e)$, limiting the input states to only those that make the gradient $\nabla(\!|e|\!)$ non-negative [Trench 2013].

Example 3.3. Consider the expression $x^2 + y^2$ over $\mathbb{R}^2$. By calculating the (symbolic) gradient of the function $(\!|x^2 + y^2|\!)$, we get $\nabla(\!|x^2 + y^2|\!) = (2x, 2y)^T$. By setting $\nabla(\!|x^2 + y^2|\!) \geq \mathbf{0}_{2\times 1}$, namely, $2x \geq 0 \wedge 2y \geq 0$, we can conclude that the gradient of the function $(\!|x^2 + y^2|\!)$ is non-negative for all points $(x, y) \in \mathbb{R}^2$ such that $x \geq 0$ and $y \geq 0$. As the region $R = \{(x, y, z) \mid x \geq 0 \wedge y \geq 0\}$ is convex, the function $(\!|x^2 + y^2|\!)$ is non-decreasing at any set $S \subseteq R$. ♦

Note that the space region satisfying the condition $\nabla(\!|e|\!) \geq \mathbf{0}_{|Var(e)|\times 1}$ might not be convex (this is the case, e.g., for the function $(\!|x^3 + x^2|\!)$). In order to be sure that each pair of states σ_1, σ_2 such that $\sigma_1 \leq^V \sigma_2$, are taken in the same convex region where the gradient $\nabla(\!|e|\!)$ is non-negative, we require

[2]We will consider input sets S by either stating their values or by using their characterization as first-order predicates. For instance, the set $\{0, 1, 2\} \in \wp(\mathbb{Z})$ may also be represented by the predicate $0 \leq x \leq 2$, and the empty set $\varnothing$ by $false$.

$$\frac{}{Mon^{\nearrow}(\textbf{skip}, S, V)}\ (\textbf{skip}) \qquad \frac{}{Mon^{\nearrow}(P, false, V)}\ (\textbf{empty}_{in})$$

$$\frac{Assign(P) \cap V = \varnothing}{Mon^{\nearrow}(P, S, V)}\ (\textbf{empty}_{var}) \qquad \frac{Var(e) \subseteq V \quad Conv(S) \quad S \Rightarrow \nabla(\!|e|\!) \geq 0}{Mon^{\nearrow}(x := e, S, V)}\ (\textbf{assign})$$

$$\frac{Mon^{\nearrow}(P, S, V) \quad S' \Rightarrow S}{Mon^{\nearrow}(P, S', V)}\ (\textbf{weaken}) \qquad \frac{Mon^{\nearrow}(P_1, S_1, V) \quad Mon^{\nearrow}(P_2, S_2, V) \quad \{S_1\}P_1\{S_2\}}{Mon^{\nearrow}(P_1; P_2, S_1, V)}\ (\textbf{seq})$$

Fig. 2. Non-decreasing analysis of base commands and sequential composition.

that both S forms a convex set, denoted by the predicate $Conv(S)$, and all the states in S satisfies $\nabla(\!|e|\!) \geq \mathbf{0}_{|Var(e)|\times 1}$ (written more concisely $\nabla(\!|e|\!) \geq 0$). Whenever these conditions are satisfied and the expression e in the assignment $x := e$ uses only variables in V, then we can safely conclude that $x := e$ is V-non-decreasing, as formalized in rule (**assign**) of Fig. 2.

We give now an intuition of the other rules in Fig. 2. Rule (**skip**) does not modify the conclusion since no operations are involved. In addition, monotonicity trivially holds on any variable when either there is no input to consider (rule ($\textbf{empty}_{in}$)), or the program P does not modify any variables of V. This last condition is stated by rule ($\textbf{empty}_{var}$) where $Assign(P)$ represents the set of all variables that appear on the left side of an assignment in P.

(**weaken**) observes that the non-decreasing program property of P can be soundly weakened by restricting the set of input states at which monotonicity is asserted. Note that the weakening rule is only possible on the input states and not on the set of variables V otherwise it may lead to unsound derivations. For instance, the program P in Example 3.2 is $\{x, y, z\}$-non-decreasing but not $\{x, z\}$-non-decreasing. Furthermore, observe that (**weaken**) does not require any convexity assumption on the weaker set S': whenever we can prove monotonicity on S, then we are sure that any pair of states in S satisfies Definition 3.1, and, therefore, any subset S' of S. This fact turns out useful, e.g., when we want to prove non-decreasing an assignment over a non-convex set S': we may deduce first that the assignment is non-decreasing over a convex overapproximation S, i.e., such that $S' \subseteq S \wedge Conv(S)$, through the rule (**assign**), and then apply (**weaken**) to come back to the non-convex set S'.

The rule (**seq**) addresses sequential composition of programs. In domain theory (see, e.g., [Scott and Strachey 1971]), it is well known that the composition of two monotonically non-decreasing mappings $f : \langle L_1, \leq_{L_1}\rangle \to \langle L_2, \leq_{L_2}\rangle$ and $g : \langle L_2, \leq_{L_2}\rangle \to \langle L_3, \leq_{L_3}\rangle$, gives as result a non-decreasing function $g \circ f : \langle L_1, \leq_{L_1}\rangle \to \langle L_3, \leq_{L_3}\rangle$. Here the result is similar, the only condition to verify is that all the output states of P_1 on input S_1 satisfy S_2, namely $[\![P_1]\!]S_1 \subseteq S_2$. This condition is formalized in the premise of (**seq**) as the *Hoare triple* [Hoare 1969] $\{S_1\}P_1\{S_2\}$. If these premises are true, then we can safely conclude that the composition $P_1; P_2$ is V-non-decreasing at all input states satisfying S_1.

Example 3.4. Consider the program $P = P_1; P_2; P_3$ over $\mathbb{R}^3$ made by composing $P_1 : x := x^2 + y^2$, $P_2 : y := 2y$ and $P_3 : z := x + y$. We want to prove that P is $\{x, y, z\}$-non-decreasing over all states $S = \{(x, y, z) \mid ((0 \leq x \leq 1) \wedge (0 \leq y \leq 1)) \vee ((1 \leq x \leq 2) \wedge (1 \leq y \leq 2))\}$ by deriving the judgment $Mon^{\nearrow}(P, S, \{x, y, z\})$. The set S, when represented into two dimensions $\mathbb{R}^2$ (x and y), depicts two squares intersecting at the point $(1, 1)$. Clearly, S is not convex. Let us consider the set $\hat{S} = \{(x, y, z) \mid (0 \leq x \leq 2) \wedge (0 \leq y \leq 2)\}$ which in two dimensions represents a square containing the two squares of S, thus $S \subseteq \hat{S}$ and $\hat{S}$ is convex.

Let us start by analyzing P_1. We have seen in Example 3.3 that the function $(\!|x^2 + y^2|\!)$ is non-decreasing on sets satisfying $x \geq 0 \wedge y \geq 0$. Since all the states in the convex set $\hat{S}$ satisfy also

$$\frac{S \Rightarrow b \quad Mon^{\nearrow}(P_1, S, V)}{Mon^{\nearrow}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, S, V)}\ (\textbf{if}_{true}) \qquad \frac{S \Rightarrow \neg b \quad Mon^{\nearrow}(P_2, S, V)}{Mon^{\nearrow}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, S, V)}\ (\textbf{if}_{false})$$

$$\frac{Mon^{\nearrow}(P_1, S \wedge (b \vee \mathcal{B}(b)), V) \quad Mon^{\nearrow}(P_2, S \wedge (\neg b \vee \mathcal{B}(b)), V) \quad Boundary(S \wedge \mathcal{B}(b), V, P_1, P_2)}{Mon^{\nearrow}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, S, V)}\ (\textbf{if})$$

Fig. 3. Verifying the non-decreasing program property of if-statements.

$x \geq 0 \wedge y \geq 0$, we can apply rule (**assign**) for P_1 and derive $Mon^{\nearrow}(x := x^2 + y^2, \hat{S}, \{x, y, z\})$. The expression $2y$ of P_2 is non-decreasing at any inputs, therefore by (**assign**) we can derive the judgment $Mon^{\nearrow}(y := 2y, true, \{x, y, z\})$. As the Hoare triple $\{\hat{S}\}P_1\{true\}$ trivially holds, by applying rule (**seq**) on the first two programs we obtain $Mon^{\nearrow}(P_1; P_2, \hat{S}, \{x, y, z\})$. The function $(\!|x + y|\!)$ of program P_3 represents a non-decreasing plane over $\mathbb{R}^3$. P_3 uses only variables in $\{x, y, z\}$ and it is clearly non-decreasing as $\nabla(\!|x + y|\!) = (1, 1)^T$. Therefore, by (**assign**), we can safely derive $Mon^{\nearrow}(z := x + y, true, \{x, y, z\})$. Again, with rule (**seq**) we join the program $P_1; P_2$ with P_3 by deriving $Mon^{\nearrow}(P_1; P_2; P_3, \hat{S}, \{x, y, z\})$. Finally, as $S \subseteq \hat{S}$, rule (**weaken**) concludes the overall derivation of P for the non-convex set S: $Mon^{\nearrow}(P_1; P_2; P_3, S, \{x, y, z\})$. ♦

3.3 If-branches

Challenges arise when programs contain if-statements or loops. In fact, similar to the analysis of program continuity [Chaudhuri et al. 2010] and differentiability [Beck and Fischer 1994], the main source of non-monotone behaviors are branches.

Let us start by analyzing the if-statement **if** b **then** P_1 **else** P_2. The trivial cases here correspond to guards b that are always satisfied (resp. not satisfied) by the considered input states S, i.e., $[\![b]\!]S = S$ (resp. $[\![b]\!]S = \emptyset$): the conclusion is the analysis of P_1 (resp. P_2). This is formalized by rule ($\textbf{if}_{\text{true}}$) (resp. ($\textbf{if}_{\text{false}}$)) of Fig. 3. By considering now the non-trivial cases ($[\![b]\!]S \neq S \wedge [\![b]\!]S \neq \emptyset$), we would like to prove the V-non-decreasing property of **if** b **then** P_1 **else** P_2 provided that both P_1 and P_2 are V-non-decreasing. Unfortunately, these assumptions are not sufficient to guarantee the overall monotonicity of the if-statement. This is because two comparable states could flow along different branches, potentially resulting in non-monotone behavior.

Example 3.5. Consider the following program

$$P \ : \ \textbf{if } x + y - 2 \leq 0 \vee 2x + y - 3 \leq 0 \textbf{ then } y := x + 10 \textbf{ else } x := x^2 + 3$$

If we consider all inputs having $x \geq 0$, then both paths lead to non-decreasing programs: $x := x^2 + 3$ is $\{x, y\}$-non-decreasing for all $x \geq 0$, and $y := x + 10$ is always $\{x, y\}$-non-decreasing. The question is: because both branches are $\{x, y\}$-non-decreasing on all states $\mathbb{R}^2$ satisfying $x \geq 0$, can we also conclude that P is $\{x, y\}$-non-decreasing over all states $\mathbb{R}^2$ satisfying $x \geq 0$? Consider two comparable input states $\sigma_1 = (0, 0)$ and $\sigma_2 = (2, 2)$ such that $\sigma_1 \leq^{\{x,y\}} \sigma_2$. Then, if P is $\{x, y\}$-non-decreasing, we would expect that $[\![P]\!]\sigma_1 \leq^{\{x,y\}} [\![P]\!]\sigma_2$. But this is not true, as $[\![P]\!]\sigma_1 = (0, 10) \not\leq^{\{x,y\}} (7, 2) = [\![P]\!]\sigma_2$. Note that σ_1 follows the **then** branch while σ_2 follows the **else** branch. ♦

The optimal approach here would involve verifying that comparable states following different branches, do not violate monotonicity, i.e., $\forall \sigma_1, \sigma_2$ such that $\sigma_1 \leq^V \sigma_2$ and $[\![b]\!]\sigma_1 = \sigma_1 \wedge [\![b]\!]\sigma_2 = \emptyset$ (resp. $[\![b]\!]\sigma_1 = \emptyset \wedge [\![b]\!]\sigma_2 = \sigma_2$) it must hold that $[\![P_1]\!]\sigma_1 \leq^V [\![P_2]\!]\sigma_2$ (resp. $[\![P_2]\!]\sigma_1 \leq^V [\![P_1]\!]\sigma_2$). This checking phase can be simplified when the guard b is a *positive linear guard*: all Boolean predicates occurring in b either have the form $e \lessdot 0$ or they all have the form $e \gtrdot 0$, and $e = 0$ is a *positive linear equation*, i.e., the expression e can be written as $e = \sum_{i=1}^{|Var(e)|} v_i x_i + k$ with $v \in \mathbb{I}_{\geq 0}$, $k \in \mathbb{I}$.

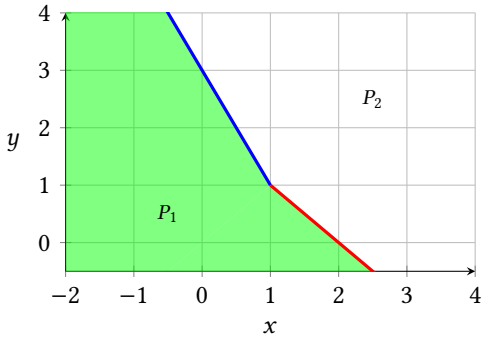

Fig. 4. Representation of the disjunction of the Boolean guard $x + y - 2 \leq 0$, in red, and $2x + y - 3 \leq 0$, in blue, of Example 3.5. All states satisfying the guard are in the green region, while the white area contains the states that do not satisfy the Boolean guard.

Example 3.6. The guard $x + y - 2 \leq 0 \vee 2x + y - 3 \leq 0$ of Example 3.5 is positive linear because it is a disjunction of two predicates, $x + y - 2 \leq 0$ and $2x + y - 3 \leq 0$, both having the same inequality, namely $\leqslant$, and the equations $x + y - 2 = 0$ and $2x + y - 3 = 0$ are positive linear, both representing a non-increasing line in $\mathbb{R}^2$. ♦

Let $Lin^+_{\leqslant}$, $Lin^+_{\geqslant}$ be the sets of all, respectively, positive linear guards having only predicates $\leqslant$, positive linear guards having only predicates $\geqslant$, and let $Lin^+ \stackrel{def}{=} Lin^+_{\leqslant} \cup Lin^+_{\geqslant}$. Then the following topological properties are satisfied for all $b \in Lin^+$:

(1) the guard b divides the Euclidean space $\mathbb{I}^{|Var(P)|}$ in *two regions*: one region populated by states that satisfy b and the other region populated by states that satisfy $\neg b$;
(2) if $b \in Lin^+_{\leqslant}$ (resp. $b \in Lin^+_{\geqslant}$) then every state that makes b true is either less or equal (resp. greater or equal) or not comparable for variables $Var(P)$ with respect to the states that do not satisfy b, i.e., $\forall \sigma_1, \sigma_2$: if $[\![b]\!]\sigma_1 = \sigma_1 \wedge [\![b]\!]\sigma_2 = \varnothing$ then either $\sigma_1 \leq^{Var(P)} \sigma_2$ (resp. $\sigma_1 \geq^{Var(P)} \sigma_2$), or σ_1 and σ_2 are not comparable.

Example 3.7. The graphical representation of $x + y - 2 \leq 0 \vee 2x + y - 3 \leq 0 \in Lin^+_{\leqslant}$ of Example 3.5 is depicted in Fig. 4. By selecting any two states σ_1, σ_2 such that σ_1 is in the green area (i.e., satisfying the guard) while σ_2 is in the white area (i.e., not satisfying the guard), we can be certain that either $\sigma_1 \leq^{\{x,y\}} \sigma_2$ or they are not comparable, in other words, it is never the case that σ_2 is less than σ_1. ♦

In this scenario, provided that P_1 and P_2 are V-non-decreasing, in order to conclude the overall monotonicity of **if** b **then** P_1 **else** P_2, it is sufficient to check the behavior of branches P_1 and P_2 at *boundary states*. Given $b \in Lin^+$ we formally define the set $\mathcal{B}(b)$ of boundary states as follows:

$$\mathcal{B}(b) \stackrel{def}{=} \begin{cases} e = 0 & \text{if } b = e \leqslant 0 \text{ or } b = e \geqslant 0, \\ \mathcal{B}(b_1) \vee \mathcal{B}(b_2) & \text{if } b = b_1 \vee b_2 \text{ or } b = b_1 \wedge b_2. \end{cases}$$

Note that $\mathcal{B}(b)$ represents an overapproximation of the true boundary states of b.

Example 3.8. By considering again Example 3.5, the boundary states here are $\mathcal{B}(x+y-2 \leq 0 \vee 2x+y-3 \leq 0) = x+y-2 = 0 \vee 2x+y-3 = 0$, namely, the set of points $\{(x, y) \mid x+y-2 = 0 \vee 2x+y-3 = 0\}$ solving either the equation $x + y - 2 = 0$ or the equation $2x + y - 3 = 0$. Note that this set contains more states than those actually on the guard $x + y - 2 = 0 \vee 2x + y - 3 = 0$ (the states on the blue and red lines of Fig. 4). ♦

Rule (**if**) contains all the necessary pre-conditions for proving the non-decreasing behavior of if-statements. Firstly, we have to prove that P_1 is V-non-decreasing at all states satisfying $S \wedge (b \vee \mathcal{B}(b))$, namely, the set of all program states in S that either satisfy b or are boundary states. The same applies for program P_2 at states $S \wedge (\neg b \vee \mathcal{B}(b))$. Finally, for the case $b \in Lin^+_{\leqslant}$, it is sufficient to check that the result of the computation of the true-branch P_1 cannot exceed the result of the computation along the false-branch P_2 on boundary states that are also in S, i.e., it must hold $\forall \sigma \in S \wedge \mathcal{B}(b) : [\![P_1]\!]\sigma \leq^V [\![P_2]\!]\sigma$. This condition is encoded by the following predicate:

$$Boundary(S \wedge \mathcal{B}(b), V, P_1, P_2) \overset{def}{\Leftrightarrow} \begin{array}{c} b \in Lin^+_{\leqslant} \wedge \forall \sigma \in S \wedge \mathcal{B}(b).\ [\![P_1]\!]\sigma \leq^V [\![P_2]\!]\sigma \\ \vee \\ b \in Lin^+_{\geqslant} \wedge \forall \sigma \in S \wedge \mathcal{B}(b).\ [\![P_1]\!]\sigma \geq^V [\![P_2]\!]\sigma \end{array}$$

Note that the predicate *Boundary* treats also the case of $b \in Lin^+_{\geqslant}$: in this scenario, the execution of P_2 must not exceed P_1 on all states $\sigma \in S \wedge \mathcal{B}(b)$. This is the intuition behind rule (**if**) on Fig. 3. Practically, checking the above predicate could be done automatically by exploiting, e.g., a static analyzer based on abstract interpretation (e.g., [Cousot et al. 2005]).

Example 3.9. By using the rules presented in this section, we can prove that the program ReLU : **if** $x \leq 0$ **then** $x := 0$ **else** $x := x$ is $\{x\}$-non-decreasing at all inputs $\mathbb{R}$. Clearly, the guard is positive linear $x \leq 0 \in Lin^+_{\leqslant}$. The boundary state is $\mathcal{B}(x \leq 0) = x = 0$, namely, the store $\hat{\sigma}(x) = 0$, thus $true \wedge (x \leq 0 \vee x = 0)$ can be simplified into $x \leq 0$, and $true \wedge (x > 0 \vee x = 0)$ into $x \geq 0$. By applying rule (**assign**) to the true-branch and false-branch we can derive, respectively, $Mon^{\nearrow}(x := 0, x \leq 0, \{x\})$ and $Mon^{\nearrow}(x := x, x \geq 0, \{x\})$. Moreover, the predicate $Boundary(x = 0, \{x\}, x := 0, x := x)$ holds, indeed $[\![x := 0]\!]\hat{\sigma} = \hat{\sigma} \leq \hat{\sigma} = [\![x := x]\!]\hat{\sigma}$. Hence, we can apply rule (**if**) and derive $Mon^{\nearrow}$(**if** $x \leq 0$ **then** $x := 0$ **else** $x := x, true, \{x\}$), proving that ReLU is $\{x\}$-non-decreasing. Intuitively, by proving that $Boundary(x = 0, \{x\}, x := 0, x := x)$ holds, together with $Mon^{\nearrow}(x := 0, x \leq 0, \{x\})$ and $Mon^{\nearrow}(x := x, x \geq 0, \{x\})$, guarantee that it is never the case that $[\![\text{ReLU}]\!]\sigma_1 > [\![\text{ReLU}]\!]\sigma_2$ starting from two stores $\sigma_1 \leq \sigma_2$ that follows different branches. Indeed, since $\sigma_1 \leq \hat{\sigma} \leq \sigma_2$ and both $Mon^{\nearrow}(x := 0, x \leq 0, \{x\})$ and $Mon^{\nearrow}(x := x, x \geq 0, \{x\})$ hold, we are sure that $[\![\text{ReLU}]\!]\sigma_1 \leq [\![\text{ReLU}]\!]\hat{\sigma} \leq [\![\text{ReLU}]\!]\sigma_2$, therefore we can conclude $[\![\text{ReLU}]\!]\sigma_1 \leq [\![\text{ReLU}]\!]\sigma_2$. ♦

Example 3.10. We have already seen that the if-program P in Example 3.5 is not $\{x, y\}$-non-decreasing on $x \geq 0$. Here the predicate $Boundary(x \geq 0 \wedge \mathcal{B}(b), \{x, y\}, y := x + 10, x := x^2 + 3)$, where b is the guard of the if-statement, is false because the value of variable y grows faster when executing P_1 rather than when executing P_2: consider for instance the boundary state $(1, 1)$, $[\![P_1]\!](1, 1) = (1, 11) \not\leq^{\{x,y\}} (4, 1) = [\![P_2]\!](1, 1)$. However, we can prove that P is $\{x\}$-non-decreasing for all inputs $S = \{(x, y) \mid 0 \leq x \leq 10 \wedge y \geq 0\}$ over $\mathbb{R}^2$. Let us construct the proof with our proof system. The condition $S \wedge (b \vee \mathcal{B}(b))$ can be simplified into $S \wedge b$ since b already includes the boundary states. For the true-branch, since $y \notin \{x\}$, we apply (**empty**$_{var}$) and derive $Mon^{\nearrow}(y := x+10, S \wedge b, \{x\})$. For the false-branch, as both the predicates $Conv(S \wedge (\neg b \vee \mathcal{B}(b)))$ and $S \wedge (\neg b \vee \mathcal{B}(b)) \Rightarrow \nabla(\!|x^2 + 3|\!) = 2x \geq 0$ holds, we can derive $Mon^{\nearrow}(x := x^2+3, S \wedge (\neg b \vee \mathcal{B}(b)), \{x\})$ by applying (**assign**). The guard is positive linear, therefore the only remaining assumption to verify is the non-deceasing property on boundary states, namely: $Boundary(S \wedge \mathcal{B}(b), \{x\}, y := x+10, x := x^2 + 3)$. This final predicate can also be checked with the assistance of an abstract interpreter $[\![\cdot]\!]_{\mathsf{Int}}$ on the abstract domain of intervals Int where the abstract sum $+_{\mathsf{Int}}$ between two intervals is defined as $[a, b] +_{\mathsf{Int}} [c, d] \overset{def}{=} [a + c, b + d]$ while the abstract multiplication $\times_{\mathsf{Int}}$ is defined as $[a, b] \times_{\mathsf{Int}} [c, d] \overset{def}{=} [min(\{ac, ad, bc, bd\}), max(\{ac, ad, bc, bd\})]$. The set of states satisfying $S \wedge \mathcal{B}(b)$ can be overapproximated by the abstract state $([0, 2], [0, 3]) \in \mathsf{Int}^2$ which corresponds to all the program states $\{\sigma \in \mathbb{R}^2 \mid \sigma(x) \in [0, 2] \wedge \sigma(y) \in [0, 3]\}$. Then by running both branches on the

$$\frac{S \Rrightarrow \neg b}{Mon^{\nearrow}(\textbf{while } b \textbf{ do } R, S, V)}\ (\textbf{while}_{false}) \qquad \frac{\{I \wedge b\}R\{I\} \quad Mon^{\nearrow}(R, I \wedge b, V) \quad Limit(b, R, I \vee S, V)}{Mon^{\nearrow}(\textbf{while } b \textbf{ do } R, S, V)}\ (\textbf{while})$$

Fig. 5. Verifying the non-decreasing program property of while-loops.

abstract interpreter, we get:

$$[\![P_1]\!]_{\mathsf{Int}}([0,2],[0,3]) = ([0,2],[10,12]) \ \leq^{\{x\}}_{\mathsf{Int}} \ ([3,7],[0,3]) = [\![P_2]\!]_{\mathsf{Int}}([0,2],[0,3])$$

where $\leq^{\{x\}}_{\mathsf{Int}}$, as for the concrete stores, indicates that we are comparing the x-component only. Since the two intervals $[0,2] \leq_{\mathsf{Int}} [3,7]$ are not overlapping, we are sure that the values of x on boundary states after executing P_1 are always less than the values of x after executing P_2. This means that the predicate $Boundary(S \wedge \mathcal{B}(b), \{x\}, y := x + 10, x := x^2 + 3)$ holds. We can conclude by rule (**if**): $Mon^{\nearrow}(P, 0 \leq x \leq 10 \wedge y \geq 0, \{x\})$ thus ensuring that variable x is computed monotonically non-decreasing by the if-statement along all states in S. ♦

3.4 Loops

Similar to if-statements, loops can easily break the monotonicity property, even when the loop body is monotone. This is because, given two comparable states $\sigma \leq^V \sigma'$, an execution starting from σ may terminate earlier or later than the one starting from σ', resulting in a potentially different outcome.

Example 3.11. Let us consider, for example, the while-loop **while** $x < 3$ **do** $x := x + 2$. Although the assignment $x := x + 2$ is inherently non-decreasing for any input, the overall loop does not preserve the non-decreasing property. In fact, when the input is 0, the loop terminates after two iterations, whereas for input 1, it only requires one iteration to terminate. As a result, at the end of the execution, the program with input 0 surpasses the execution with input 1:

$$[\![\textbf{while } x < 3 \textbf{ do } x := x + 2]\!](0) = 4 \not\leq 3 = [\![\textbf{while } x < 3 \textbf{ do } x := x + 2]\!](1). \qquad ♦$$

Since each iteration of **while** b **do** R can be viewed as an execution of **if** b **then** R **else skip**, when guards are positive linear we may think that the monotonicity of loops could be proved by employing the technique of the (**if**) rule. Regrettably, although sound, the use of boundary states $\mathcal{B}(b)$ on loops is too weak as it fails to establish the monotonicity even for simple loops like **while** $x < 0$ **do** $x := x + 1$. Indeed, for this example, requiring $[\![x = x + 1]\!](0) \leq [\![\textbf{skip}]\!](0)$ for the boundary state $x = 0$, is equivalent to demanding that the loop body does not modify variable x.

For this reason, we need to refine the definition of boundary states for loops: instead of considering a set of states, we now consider a set of pairs of states (σ_1, σ_2), which we refer to as *limit states*. Intuitively, in the case of $b \in Lin^+_{\leqslant}$, the states (σ_1, σ_2) are limit states for the while-loop **while** b **do** R, when σ_1 enters the loop, σ_2 does not enter the loop and $\sigma_1 \leq^V \sigma_2$. We generalize this reasoning by defining the following two sets based on a program P, set of inputs S, variables V and, respectively, $b \in Lin^+_{\leqslant}$ and $b \in Lin^+_{\geqslant}$:

$$\mathcal{L}_{\leqslant}(b, P, S, V) \stackrel{def}{=} \{(\sigma_1, \sigma_2) \mid \sigma_1 \in S \wedge b, \sigma_2 \in S \wedge \neg b, \sigma_1 \leq^V \sigma_2\}$$

$$\mathcal{L}_{\geqslant}(b, P, S, V) \stackrel{def}{=} \{(\sigma_1, \sigma_2) \mid \sigma_1 \in S \wedge \neg b, \sigma_2 \in S \wedge b, \sigma_1 \leq^V \sigma_2\}$$

Given a while-loop **while** b **do** R such that $b \in Lin^+_{\leqslant}$, a set of input states S, set of variables V, and a loop invariant I, the set of pairs of states $\mathcal{L}_{\leqslant}(b, R, I \vee S, V)$ identifies our intuition of limit states. In this case, when $b \in Lin^+_{\leqslant}$, in order to conclude the overall non-decreasing behavior of the loop, we need to verify two conditions: (1) the loop body R must be V-non-decreasing on $I \wedge b$, and (2) for every pair of limit states $(\sigma_1, \sigma_2) \in \mathcal{L}_{\leqslant}(b, R, I \vee S, V)$, the execution of R with input the state

σ_1 must not exceed σ_2, namely it must hold that $[\![R]\!]\sigma_1 \leq^V \sigma_2$. These two conditions will ensure that, starting from two comparable stores $\sigma \leq \sigma'$ entering the loop, each execution of the body R will not alter their order and, even if at some point, e.g., at the n-th iteration, $[\![R^n]\!]\sigma'$ exits the loop, where R^n is the sequential composition of R n-times, the successive iteration of σ until it exists the loop, will not break the order. The condition (2) and its dual case when $b \in Lin^+_{\geqslant}$, are treated by the following predicate:

$$Limit(b,P,S,V) \overset{def}{\Leftrightarrow} \begin{array}{l} b \in Lin^+_{\leqslant} \ \wedge \ \forall(\sigma_1,\sigma_2) \in \mathcal{L}_{\leqslant}(b,P,S,V).\ [\![P]\!]\sigma_1 \leq^V \sigma_2 \\ \qquad\qquad \vee \\ b \in Lin^+_{\geqslant} \ \wedge \ \forall(\sigma_1,\sigma_2) \in \mathcal{L}_{\geqslant}(b,P,S,V).\ \sigma_1 \leq^V [\![P]\!]\sigma_2 \end{array}$$

This is the intuition underlying rule (**while**) in Fig. 5. The premise $\{I \wedge b\}R\{I\}$ corresponds to the Hoare triple which states that I is a loop invariant. Abstract interpretation can be employed to automatically detect a sound invariant. Rule ($\mathbf{while}_{false}$) addresses the straightforward scenario where none of the states in S enter the loop. Practically, checking the validity of the predicate $Limit(b,P,S,V)$, can be semi-automated by using modern automatic theorem provers or SMT solvers.

Example 3.12. Consider the program $\mathrm{ReLU}^w \overset{def}{=}$ **while** $x < 0$ **do** $x := x + 1$ which implements the ReLU function on integers by using a while-loop instead of an if-statement. Clearly, as ReLU^w is semantically equivalent to the ReLU program of Example 3.9 on integer inputs, ReLU^w is non-decreasing over $\mathbb{Z}$. We want to prove it by exploiting rule (**while**) of our proof system. Let us consider the invariant $I : x \leq 0$. By rule (**assign**), we first prove that the loop body is non-decreasing: $Mon^{\nearrow}(x := x+1, x < 0, \{x\})$. Since $x < 0 \in Lin^+_{\leqslant}$, the set of all pair of limit states is $\mathcal{L}_{\leqslant}(x < 0, x := x+1, \mathbb{Z}, \{x\}) = \{(m,n) \mid m,n \in \mathbb{Z} \wedge m < 0 \wedge n \geq 0\}$. Then, clearly, for all $(m,n) \in \mathcal{L}_{\leqslant}(x < 0, x := x+1, \mathbb{Z}, \{x\})$, $[\![x := x+1]\!](m) \leq n$ holds, therefore the predicate $Limit(x < 0, x := x+1, \mathbb{Z}, \{x\})$ is true. All the premises are satisfied and rule (**while**) concludes $Mon^{\nearrow}(\mathrm{ReLU}^w, true, \{x\})$. ♦

Example 3.13. Let us consider the program Fact that calculates the factorial of a natural number $x \in \mathbb{N}$ and stores the result in the variable f:

$$\mathrm{Fact} \ : \ i := 1;\ f := 1;\ \mathbf{while}\ i - x \leq 0\ \mathbf{do}\ f := f * i;\ i := i + 1$$

Let the triple $\sigma = (x,i,f)$ represents a program state of Fact, we want to prove that Fact is monotonically non-decreasing on all its variables at all input states satisfying the loop invariant $I = \{(x,i,f) \mid f = (i-1)!\}$ where $n! \overset{def}{=} n * (n-1) * \cdots * 1$ is the mathematical definition of factorial for $n \in \mathbb{N}$. For the first two assignments before the loop, it is easy to derive $Mon^{\nearrow}(i := 1; f := 1, true, \{x,i,f\})$ by rules (**assign**) and (**seq**). Then, by rule (**weaken**), we soundly restrict the set of input states to the set I: $Mon^{\nearrow}(i := 1; f := 1, I, \{x,i,f\})$. Let us now analyze the while-loop. Note that I is a loop invariant, indeed it is easy to verify that the Hoare triple $\{I \wedge (i \leq x)\} f := f * i;\ i := i+1 \{I\}$ holds. As we are working on natural numbers and both assignments of the body of the loop are non-decreasing over $\mathbb{N}^3$, we can easily infer $Mon^{\nearrow}(f := f * i; i := i+1, true, \{x,i,f\})$ by rules (**assign**) and (**seq**). Further, we use rule (**weaken**) to derive $Mon^{\nearrow}(f := f * i; i := i+1, I \wedge (i \leq x), \{x,i,f\})$, as $I \wedge (i \leq x) \Rightarrow true$ trivially holds. Since the guard $i - x \leq 0 \in Lin^+_{\leqslant}$, it remains to check whether the predicate $Limit(i - x \leq 0, f := f * i; i := i+1, I, \{x,i,f\})$ holds or not for the limit states. Given a pair of limit states $(\sigma_1,\sigma_2) \in \mathcal{L}_{\leqslant}(i - x \leq 0, f := f * i; i := i+1, I, \{x,i,f\})$, we know that: $\sigma_1 \leq \sigma_2$, $\sigma_1 \in \{(x,i,f) \mid I \wedge (i \leq x)\}$ and $\sigma_2 \in I \wedge (i > x)$. Then, clearly $[\![f := f * i; i := i+1]\!]\sigma_1 \leq \sigma_2$ holds for all pair of limit states since a further execution of the loop body on σ_1 cannot exceed σ_2. Thanks to the validity of the premises, by rule (**while**) we can derive

$Mon^{\nearrow}(\textbf{while } i - x \leq 0 \textbf{ do } f := f * i;\ i := i + 1, I, \{x, i, f\})$. Finally, with rule (**seq**) we combine $Mon^{\nearrow}(i := 1; f := 1, I, \{x, i, f\})$ with this last derivation, and conclude $Mon^{\nearrow}(\text{Fact}, I, \{x, i, f\})$. ♦

The proof system specified by the rules in Fig. 2, 3 and 5, is sound, as stated by the following theorem.

Theorem 3.14. $\vdash Mon^{\nearrow}(P, S, V) \implies Mon^{\nearrow}(P, S, V)$.

Proof. (**skip**): Since $\mathcal{D}_{\textbf{skip}} = \mathbb{I}^n$, for any $\sigma_1, \sigma_2 \in S$ such that $\sigma_1 \preceq^V \sigma_2$, we get $[\![\textbf{skip}]\!]\sigma_1 = \sigma_1 \preceq^V \sigma_2 = [\![\textbf{skip}]\!]\sigma_2$, therefore $Mon^{\nearrow}(\textbf{skip}, S, V)$ holds for any $S \subseteq \mathbb{I}^n$ and $V \subseteq Var$.

(**empty**$_{in}$): S is *false* is equivalent to the emptyset of states $S = \emptyset$. The monotonicity condition is trivially satisfied as there are no states to check, therefore $Mon^{\nearrow}(P, \mathit{false}, V)$ holds for all $P \in \mathsf{Prog}$ and $V \in Var$.

(**empty**$_{var}$): Assume $Assign(P) \cap V = \emptyset$, namely either P does not modify any variable in V or $V = \emptyset$. In the first case, for every $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_P$ and for every $x \in Var(P) \cap V$ such that $\sigma_1(x) \leq \sigma_2(x)$ we get $([\![P]\!]\sigma_1)(x) = ([\![P]\!]\sigma_2)(x)$, i.e., $Mon^{\nearrow}(P, S, V)$ holds, while if $V = \emptyset$ then there are no variables to check monotonicity, therefore the program is monotone for all $S \subseteq \mathbb{I}^n$.

(**assign**): $\mathcal{D}_{x:=e} = \mathbb{I}^{|Var(x:=e)|}$ since every assignment command is always terminating. Assume $Var(e) \subseteq V$, $Conv(S)$, $S \Rightarrow \nabla(\!|e|\!) \geq \mathbf{0}_{|Var(e)|\times 1}$ and consider any $\sigma_1, \sigma_2 \in S$ such that $\sigma_1 \preceq^V \sigma_2$. Then, we get the following implications:

$$
\begin{aligned}
S \Rightarrow \nabla(\!|e|\!) \geq \mathbf{0}_{|Var(e)|\times 1} &\implies [\text{by } \sigma_1, \sigma_2 \in S] \\
\nabla(\!|e|\!)\sigma_1 \geq \mathbf{0}_{|Var(e)|\times 1} \wedge \nabla(\!|e|\!)\sigma_2 \geq \mathbf{0}_{|Var(e)|\times 1} &\implies [\text{by } \sigma_1 \preceq^V \sigma_2,\ Var(e) \subseteq V \text{ and } Conv(S)] \\
(\!|e|\!)\sigma_1 \leq (\!|e|\!)\sigma_2 &\implies [\text{by Definition of } [\![x := e]\!]] \\
[\![x := e]\!]\sigma_1 \preceq^V [\![x := e]\!]\sigma_2 &\implies [\text{by Definition 3.1}] \\
Mon^{\nearrow}(x := e, S, V)
\end{aligned}
$$

(**weaken**): If $S' = S$ then trivially $Mon^{\nearrow}(P, S', V)$ holds. Assume $Mon^{\nearrow}(P, S, V)$ and $S' \subset S$. By assumption, the non-decreasing property holds for all states $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_P$. Since $(S' \cap \mathcal{D}_P) \subset (S \cap \mathcal{D}_P)$, then it must also hold for all states $\sigma_1, \sigma_2 \in (S' \cap \mathcal{D}_P)$, therefore $Mon^{\nearrow}(P, S', V)$ is true.

(**seq**): Let us assume $Mon^{\nearrow}(P_1, S_1, V)$, $\{S_1\}P_1\{S_2\}$ and $Mon^{\nearrow}(P_2, S_2, V)$. Since $\mathcal{D}_{P_1;P_2} \subseteq \mathcal{D}_{P_1}$, we know that $(S_1 \cap \mathcal{D}_{P_1;P_2}) \subseteq (S_1 \cap \mathcal{D}_{P_1})$. Therefore, for all $\sigma_1, \sigma_2 \in S_1 \cap \mathcal{D}_{P_1;P_2}$, we get the following implications:

$$
\begin{aligned}
\sigma_1 \preceq^V \sigma_2 &\implies [\text{by } Mon^{\nearrow}(P_1, S_1, V)] \\
[\![P_1]\!]\sigma_1 \preceq^V [\![P_1]\!]\sigma_2 &\implies [\text{by } \{S_1\}P_1\{S_2\} \text{ and } Mon^{\nearrow}(P_2, S_2, V)] \\
[\![P_2]\!][\![P_1]\!]\sigma_1 \preceq^V [\![P_2]\!][\![P_1]\!]\sigma_2 &\implies [\text{by } [\![P_2]\!] \circ [\![P_1]\!] = [\![P_1; P_2]\!]] \\
[\![P_1; P_2]\!]\sigma_1 \preceq^V [\![P_1; P_2]\!]\sigma_2 &\implies [\text{by Definition 3.1}] \\
Mon^{\nearrow}(P_1; P_2, S_1, V)
\end{aligned}
$$

(**if**$_{true}$),(**if**$_{false}$): Since $S \Rightarrow b$ implies that $[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]S = [\![P_1]\!]S$, then by the assumption $Mon^{\nearrow}(P_1, S, V)$, we can conclude $Mon^{\nearrow}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, S, V)$. A similar reasoning can be used to prove (**if**$_{false}$).

(**if**): We assume the following: $Mon^{\nearrow}(P_1, S \wedge (b \vee \mathcal{B}(b)), V)$, $Mon^{\nearrow}(P_2, S \wedge (\neg b \vee \mathcal{B}(b)), V)$ and $Boundary(S \wedge \mathcal{B}(b), V, P_1, P_2)$. Let us consider the non-trivial case where $Var(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) \cap V \neq \emptyset$. Note that we do not care about the case $= \emptyset$ as if we are able to prove non-decreasing the if-statement, the result is still sound (see rule (**empty**$_{var}$)). Suppose $b \in Lin^{+}_{\leqslant}$ (the case $b \in$

$Lin^{+}_{\geqslant}$ is similar). Given $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_{\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2}$ such that $\sigma_1 \leq^V \sigma_2$, there are three cases to verify: (1) $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1, \sigma_2\}$, (2) $[\![b]\!]\{\sigma_1, \sigma_2\} = \emptyset$, and (3) $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1\}$. Note that, since $Var(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2) \cap V \neq \emptyset$, $b \in Lin^{+}_{\leqslant}$ and $\sigma_1 \leq^V \sigma_2$, it is not feasible for the case $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_2\}$ to occur. We analyze each of them:

(1) case $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1, \sigma_2\}$: from the assumption $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_{\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2}$ and by $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1, \sigma_2\}$, we get $\sigma_1, \sigma_2 \in S \wedge (b \vee \mathcal{B}(b))$. Then by $Mon^{\nearrow}(P_1, S \wedge (b \vee \mathcal{B}(b)), V)$ and $\sigma_1 \leq^V \sigma_2$, we obtain $[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_1 = [\![P_1]\!]\sigma_1 \leq^V [\![P_1]\!]\sigma_2 = [\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_2$;

(2) case $[\![b]\!]\{\sigma_1, \sigma_2\} = \emptyset$: from the assumption $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_{\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2}$ and by $[\![b]\!]\{\sigma_1, \sigma_2\} = \emptyset$, we get $\sigma_1, \sigma_2 \in S \wedge (\neg b \vee \mathcal{B}(b))$. Then by $Mon^{\nearrow}(P_2, S \wedge (\neg b \vee \mathcal{B}(b)), V)$ and $\sigma_1 \leq^V \sigma_2$, we obtain $[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_1 = [\![P_2]\!]\sigma_1 \leq^V [\![P_2]\!]\sigma_2 = [\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_2$;

(3) case $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1\}$: by $Boundary(S \wedge \mathcal{B}(b), V, P_1, P_2)$ we know that $\forall \sigma \in S \wedge \mathcal{B}(b)$: $[\![P_1]\!]\sigma \leq^V [\![P_2]\!]\sigma$. Moreover, consider a boundary state $\sigma_{\mathcal{B}} \in S \wedge \mathcal{B}(b)$ between σ_1 and σ_2, namely such that $\sigma_1 \leq^V \sigma_{\mathcal{B}} \leq^V \sigma_2$. Then:

$$
\begin{aligned}
[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_1 &= [\text{by } [\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1\}] \\
[\![P_1]\!]\sigma_1 &\leq^V [\text{by } Boundary(S \wedge \mathcal{B}(b), V, P_1, P_2)] \\
[\![P_1]\!]\sigma_{\mathcal{B}} &\leq^V [\text{by Definition of } \mathcal{B}(b)] \\
[\![P_2]\!]\sigma_{\mathcal{B}} &\leq^V [\text{by } Boundary(S \wedge \mathcal{B}(b), V, P_1, P_2)] \\
[\![P_2]\!]\sigma_2 &= [\text{by } [\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1\}] \\
[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_2 &
\end{aligned}
$$

By the premises of rule (**if**) we ended $[\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_1 \leq^V [\![\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2]\!]\sigma_2$ for all $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_{\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2}$ such that $\sigma_1 \leq^V \sigma_2$, therefore, by Definition 3.1, the predicate $Mon^{\nearrow}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, S, V)$ holds.

($\textbf{while}_{false}$): $S \Longrightarrow \neg b$ implies that $[\![\textbf{while } b \textbf{ do } R]\!]S = S$, therefore $Mon^{\nearrow}(\textbf{while } b \textbf{ do } R, S, V)$.

(**while**): Assume $\{I \wedge b\}R\{I\}$, namely I is a loop invariant, and both predicates $Mon^{\nearrow}(R, I \wedge b, V)$ and $Limit(b, R, I \vee S, V)$ hold. Furthermore, assume that $b \in Lin^{+}_{\leqslant}$ (the case $b \in Lin^{+}_{\geqslant}$ is similar). Given two states $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_{\textbf{while } b \textbf{ do } R}$ such that $\sigma_1 \leq^V \sigma_2$, there are three cases to consider:

(1) case $[\![b]\!]\{\sigma_1, \sigma_2\} = \emptyset$: $[\![\textbf{while } b \textbf{ do } R]\!]\sigma_1 = \sigma_1 \leq_V \sigma_2 = [\![\textbf{while } b \textbf{ do } R]\!]\sigma_2$;

(2) case $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1\}$: because $\sigma_1 \in \mathcal{D}_{\textbf{while } b \textbf{ do } R}$, this implies that $\exists n > 0$ such that $[\![b]\!][\![R^{n-1}]\!]\sigma_1 \neq \emptyset$ and $[\![b]\!][\![R^{n}]\!]\sigma_1 = \emptyset$. We prove, by induction on i, that for all $1 \leq i \leq n$ $[\![R^i]\!]\sigma_1 \leq^V \sigma_2$ holds. Base case $i = 1$: $[\![R]\!]\sigma_1 \leq^V \sigma_2$ is true since $\sigma_2 \in I \vee S$, $(\sigma_1, \sigma_2) \in \mathcal{L}_{\leqslant}(b, R, I \vee S, V)$ and $Limit(b, R, I \vee S, V)$ holds by assumption . Inductive step: assume the statement holds for $i = n - 1$, namely, $[\![R^{n-1}]\!]\sigma_1 \leq^V \sigma_2$. Then, since $[\![b]\!][\![R^{n-1}]\!]\sigma_1 \neq \emptyset$, $[\![b]\!]\sigma_2 = \emptyset$ and by the inductive hypothesis $[\![R^{n-1}]\!]\sigma_1 \leq^V \sigma_2$, we get that these are limit states, i.e., $([\![R^{n-1}]\!]\sigma_1, \sigma_2) \in \mathcal{L}_{\leqslant}(b, R, I \vee S, V)$. By $Limit(b, R, I \vee S, V)$, we can conclude $[\![R]\!][\![R^{n-1}]\!]\sigma_1 = [\![R^n]\!]\sigma_1 \leq^V \sigma_2$. Therefore, we have proved that for all $1 \leq i \leq n$, $[\![R^i]\!]\sigma_1 \leq^V \sigma_2$ holds. Finally, since $[\![b]\!][\![R^n]\!]\sigma_1 = \emptyset$, we conclude $[\![\textbf{while } b \textbf{ do } R]\!]\sigma_1 = [\![R^n]\!]\sigma_1 \leq^V \sigma_2$;

(3) case $[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1, \sigma_2\}$: because $\sigma_1, \sigma_2 \in \mathcal{D}_{\textbf{while } b \textbf{ do } R}$, this implies that $\exists n_1, n_2 > 0$, $n_2 \leq n_1$ such that $[\![b]\!][\![R^{n_1-1}]\!]\sigma_1 \neq \emptyset$ and $[\![b]\!][\![R^{n_1}]\!]\sigma_1 = \emptyset$, while $[\![b]\!][\![R^{n_2-1}]\!]\sigma_2 \neq \emptyset$ and $[\![b]\!][\![R^{n_2}]\!]\sigma_2 = \emptyset$. For all $1 \leq i \leq n_2$, we know $[\![R^{i-1}]\!]\sigma_1, [\![R^{i-1}]\!]\sigma_2 \in I \wedge b$ therefore, by assumption $Mon^{\nearrow}(R, I \wedge b, V)$, we derive that $[\![R^i]\!]\sigma_1 \leq^V [\![R^i]\!]\sigma_2$. Consequently, at the n_2-iteration of the loop with input σ_2, we have (1) $[\![R^{n_2}]\!]\sigma_1 \leq^V [\![R^{n_2}]\!]\sigma_2$ and $[\![b]\!][\![R^{n_2}]\!]\sigma_2 = \emptyset$. To terminate the proof it is sufficient to conclude, by induction on i, that for all $n_2 \leq i \leq n_1$, $[\![R^i]\!]\sigma_1 \leq^V \hat{\sigma} = [\![R^{n_2}]\!]\sigma_2$ holds. This proof follows exactly the same steps of the proof by induction of the previous case ($[\![b]\!]\{\sigma_1, \sigma_2\} = \{\sigma_1\}$) by considering $\hat{\sigma}$ in place of σ_2, and the

base case shifted to $i = n_2$ (which trivially holds by (1)). Finally, from $[\![b]\!][\![R^{n_1}]\!]\sigma_1 = \varnothing$ and $[\![b]\!][\![\hat{\sigma}]\!] = \varnothing$, we conclude $[\![\textbf{while } b \textbf{ do } R]\!]\sigma_1 = [\![R^{n_1}]\!]\sigma_1 \leq^V \hat{\sigma} = [\![R^{n_2}]\!]\sigma_2 = [\![\textbf{while } b \textbf{ do } R]\!]\sigma_2$.

By using the premises of rule (**while**) we ended $[\![\textbf{while } b \textbf{ do } R]\!]\sigma_1 \leq^V [\![\textbf{while } b \textbf{ do } R]\!]\sigma_2$ for all $\sigma_1, \sigma_2 \in S \cap \mathcal{D}_{\textbf{while } b \textbf{ do } R}$ such that $\sigma_1 \leq^V \sigma_2$, therefore, by Definition 3.1, $Mon^{\nearrow}(\textbf{while } b \textbf{ do } R, S, V)$ holds. □

4 ANALYSIS OF MONOTONE PROGRAMS

In this section, we show how monotonicity plays a central role in verifying numerical properties of variables by abstract interpretation. In particular, we identify certain structural properties of abstract domains and inputs that *guarantee the existence of a complete abstract interpreter* when analyzing variables that exhibit monotonic behavior in the program under analysis.

An abstract interpreter $[\![\cdot]\!]_{\mathcal{A}}$ is said to be complete for a program $P \in \mathsf{Prog}$ and input $S \subseteq \mathbb{I}^n$ when the equality $\alpha_{\mathcal{A}}([\![P]\!]) = [\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S)$ holds. Note that this standard notion of completeness on abstract interpreters refers to all program variables used by P. However, when dealing with non-relational abstractions, completeness can be specified with respect to a set of variables $V \subseteq Var$ which may be a subset of the variables actually utilized by the program under consideration. This gives rise to the notion of V*-completeness*.

*Definition 4.1 (**V-Completeness**).* Let us consider a non-relational abstraction $\mathcal{A} \in \mathsf{Abs}(\wp(\mathbb{I}^n))$, a program $P \in \mathsf{Prog}$, an input $S \subseteq \mathbb{I}^n$ and a set of variables $V \subseteq Var$. We say that the abstract interpretation $[\![P]\!]_{\mathcal{A}}$ of program P is V*-complete* at S whenever the following condition holds for all $x \in V \cap Var(P)$:

$$(\alpha_{\mathcal{A}}([\![P]\!]S))(x) = ([\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S))(x)$$

or, equivalently, $\alpha_{\mathcal{A}}([\![P]\!]S) =^V [\![P]\!]_{\mathcal{A}}\alpha_{\mathcal{A}}(S)$. ■

Essentially, the V-completeness property of abstract interpreters focuses on the analysis precision of a specific set of variables, namely, the variables in V. As the standard notion of completeness is equivalent to require $Var(P)$-completeness, if $[\![P]\!]_{\mathcal{A}}$ is complete then it is also V-complete for all $V \subseteq Var$, while if $[\![P]\!]_{\mathcal{A}}$is V-complete for $V \subset Var(P)$ then it may be not complete.

Example 4.2. Consider the abstract domain $\mathsf{Sign} \stackrel{def}{=} \{\mathbb{Z}, -, 0, +, \varnothing\}$ for integer sign analysis presented in Example 2.2. Suppose that the abstract sum operation $\oplus$ on $[\![\cdot]\!]_{\mathsf{Sign}}$ is soundly implemented as follows: $+ \oplus + = +$; if $\star \in \{\mathbb{Z}, -, 0, +\}$ then $\star \oplus 0 = \star$, $\varnothing \oplus 0 = 0$, $+ \oplus - = \mathbb{Z}$, and if $\star \in \{\mathbb{Z}, -, 0, +, \varnothing\}$ then $\mathbb{Z} \oplus \star = \mathbb{Z}$. Consider the program $P : x := -1;\ y := 1;\ x := x + y;\ y := y + 1$. Then, for any input, $[\![P]\!]_{\mathsf{Sign}}$ is $\{y\}$-complete but not complete, i.e., not $\{x, y\}$-complete. Indeed, for all $S \in \wp(\mathbb{Z}^2)$ we have $\alpha_{\mathsf{Sign}}([\![P]\!]S) = (0, +) <_{\mathsf{Sign}} (\mathbb{Z}, +) = [\![P]\!]_{\mathsf{Sign}}\alpha_{\mathsf{Sign}}(S)$, but $(\alpha_{\mathsf{Sign}}([\![P]\!]S))(y) = + = ([\![P]\!]_{\mathsf{Sign}}\alpha_{\mathsf{Sign}}(S))(y)$. ♦

It is possible to relate monotonicity with the precision of program analysis when certain structural properties on the considered set of inputs and on the abstract domain are met. To this end, we introduce the notions of: (1) V*-bounded* input, (2) Int*-abstractable* domain and (3) V*-convex* domain.

Given a program $P \in \mathsf{Prog}$, a set $V \subseteq Var$ of variables and a set of inputs $S \subseteq \mathbb{I}^n$, we define

$$min^V S \stackrel{def}{=} \{\sigma \in S \mid \forall x \in V \cap Var(P).\, \sigma(x) = min(\{\rho(x) \mid \rho \in S\})\}$$

as the minimum stores in S that assign to every variable x in V and in the text of P, the minimum value assumed by x over S. This means that, when $min^V S \neq \varnothing$, then we can find a minimum store $\hat{\sigma} \in S$ for the variables in V such that for all $\sigma \in S$, $\hat{\sigma} \leq^V \sigma$. For instance, if $Var(P) = V = \{x, y\}$, $S = \{(0, 1), (3, 4), (1, 2)\}$, $S' = \{(1, 3), (2, 0), (4, 4)\}$, then $min^V S = \{(0, 1)\}$ while $min^V S' = \varnothing$. The set max^V of maximum stores in S for variables in V, is dually defined.

*Definition 4.3 (**V-Bounded input**).* The input set S is said to be V*-bounded* for the program P when the following two conditions are satisfied: (i) $[\![P]\!] min^V S \neq \varnothing$, and (ii) $[\![P]\!] max^V S \neq \varnothing$. ■

Intuitively, an input set S is V-bounded for the program P when S contains a minimum and a maximum store according to, respectively, $min^V S$ and $max^V S$, and P terminates for at least one store in $min^V S$ and $max^V S$.

While the notion of V-boundness depends on the input (and on the program), the next two definitions rely more on abstract domains.

*Definition 4.4 (**Int-Abstractable domain**).* We say that an abstract domain $\mathcal{A} \in \mathsf{Abs}(\wp(\mathbb{I}^n))$ is Int*-abstractable* whenever $\mathcal{A} \leq_{\mathsf{Abs}(\wp(\mathbb{I}^n))} \mathsf{Int}$, i.e., $\mathcal{A}$ can exactly represent intervals. ■

For instance, Sign is not Int-abstractable, while it is the case for Int, Zone, Oct $\in \mathsf{Abs}(\wp(\mathbb{I}^n))$ namely Intervals, Zones and Octagons abstract domains [Miné 2001a,b, 2017].

*Definition 4.5 (**V-Convexity**).* An abstract domain $\mathcal{A} \in \mathsf{Abs}(\wp(\mathbb{I}^n))$ is V*-convex* at S for a program $P \in \mathsf{Prog}$, set of variables $V \subseteq Var$ and input set $S \in \wp(\mathbb{I}^n)$, if and only if for all $x \in V \cap Var(P)$, $(\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S)))(x) \in \wp(\mathbb{I})$ forms a convex set. ■

Namely, the abstract set of values assumed by the program variable x at the end of the concrete execution of P with input S, formally $(\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S)))(x)$, must form a convex set, i.e. it must have no holes, and this must hold for all variables in V that are also in the text of P. Of course, abstract domains composed by only convex representations of elements of $\wp(\mathbb{I})$, e.g. Int and Sign, are V-convex for all $P \in \mathsf{Prog}$, $V \subseteq Var$ and $S \subseteq \mathbb{I}^n$. This does not hold in general for abstract domains composed by also non-convex abstract elements, e.g., Parity. However, it may happen that non-convex abstractions are V-convex for some program P and input S.

Example 4.6. The integer congruence abstract domain $\mathsf{Congr} \in \mathsf{Abs}(\wp(\mathbb{Z}))$ [Granger 1989] defined as $\mathsf{Congr} \stackrel{def}{=} \{a\mathbb{Z} + b \mid a \in \mathbb{N}, b \in \mathbb{Z}\} \cup \{\bot_{\mathsf{Congr}}\}$, contains abstract elements having the form $a\mathbb{Z} + b$ such that $\gamma_{\mathsf{Congr}}(a\mathbb{Z} + b) \stackrel{def}{=} \{ak + b \mid k \in \mathbb{Z}\}$ and $\alpha_{\mathsf{Congr}}(S) \stackrel{def}{=} \bigvee^{c \in S}_{\mathsf{Congr}}(0\mathbb{Z} + c)$. The parity domain Parity is a special case of this domain where $a = 2$. Congr contains convex properties, e.g. $\gamma_{\mathsf{Congr}}(1\mathbb{Z} + 0) = \mathbb{Z}$ or singletons $\gamma_{\mathsf{Congr}}(0\mathbb{Z} + b) = \{b\}$, and non-convex properties. For example, Congr is $\{x\}$-convex for the assignment $x := 1$ regardless of the input $S \subseteq \mathbb{Z}$ since $\gamma_{\mathsf{Congr}}(\alpha_{\mathsf{Congr}}([\![x := 1]\!]S))(x) = \{1\}$ and $\{1\}$ is clearly a convex set. While, for instance, it is not $\{x\}$-convex for the assignment $x := x * 2 + 1$ at the input $\{2, 3\}$ as $\gamma_{\mathsf{Congr}}(\alpha_{\mathsf{Congr}}([\![x := x * 2 + 1]\!]\{2, 3\})) = \gamma_{\mathsf{Congr}}(2\mathbb{Z} + 1) = \{1, 3, 5, \dots\}$ contains many holes, namely, all the even numbers. ♦

Finally, we introduce the V-complete-analyzability property which identifies the class of programs admitting a V-complete analysis over a non-relational abstract domain $\mathcal{A}$ for input S.

*Definition 4.7 (**V-Complete-analyzability**).* A program $P \in \mathsf{Prog}$ is said to be V*-complete-analyzable* for the non-relational abstraction $\mathcal{A} \in \mathsf{Abs}(\wp(\mathbb{I}^n))$, variables in $V \subseteq Var$ and input $S \subseteq \mathbb{I}^n$ if and only if there exists an abstract interpreter $[\![\cdot]\!]_{\mathcal{A}}$ such that $[\![P]\!]_{\mathcal{A}}$ is V-complete at S. ■

We use the predicate $Compl^{\mathcal{A}}(P, S, V)$ to indicate that P is V-complete-analyzable for the abstract domain $\mathcal{A}$ at input S. For instance, P in Example 4.2 and ReLU defined in Section 1 are, respectively, $\{y\}$- and $\{x\}$-complete-analyzable for, respectively, Sign and Int as the defined analyses $[\![P]\!]_{\mathsf{Sign}}$ and $[\![\mathrm{ReLU}]\!]_{\mathsf{Int}}$ are, respectively, $\{y\}$- and $\{x\}$-complete for all inputs. The following result is a straightforward consequence of Theorem 2.6.

Lemma 4.8. *$Compl^{\mathcal{A}}(P, S, V) \Leftrightarrow [\![P]\!]^{\alpha}_{\mathcal{A}}$ is V-complete at S.* □

Therefore, in order to prove that a program P is V-complete-analyzable for a non-relational abstraction $\mathcal{A}$ and input S it is sufficient to prove that the BCA $[\![P]\!]^{\alpha}_{\mathcal{A}}$ is V-complete at S. The following proposition outlines the trivial cases in which the V-complete-analyzability property always holds for any program.

Lemma 4.9. *Let $P \in \mathsf{Prog}$. If one of the following holds:*

(i) $\mathcal{A}$ is trivial, namely, $\mathcal{A} = \wp(\mathbb{I}^n)$;

(ii) $V \cap Var(P) = \varnothing$;

(iii) S is representable in $\mathcal{A}$, namely, $S = \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$;

then $Compl^{\mathcal{A}}(P, S, V)$ is true.

Proof. (i) If $\mathcal{A}$ is trivial then $\gamma_{\mathcal{A}} \circ \alpha_{\mathcal{A}} = \lambda x.x$, therefore the BCA is exactly the concrete semantics $[\![\cdot]\!]$.

(ii) By Definition 4.1, when V does not contain variables in the text of P then any abstract interpreter $[\![P]\!]_{\mathcal{A}}$ is V-complete at all inputs S.

(iii) Suppose $S = \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$. Then, for every $\mathcal{A}$ and P, $\alpha_{\mathcal{A}}([\![P]\!]S) = \alpha_{\mathcal{A}}([\![P]\!]\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))) = [\![P]\!]^{\alpha}_{\mathcal{A}}\alpha_{\mathcal{A}}(S)$, i.e., the BCA $[\![P]\!]^{\alpha}_{\mathcal{A}}$ is complete at S and therefore V-complete for any $V \subseteq Var$. By Lemma 4.8, this implies that P is V-complete-analyzable. □

We are interested in studying the non-trivial cases, namely, when the abstract domain $\mathcal{A}$ differs from the concrete domain $\wp(\mathbb{I}^n)$, the set of variables V includes at least one variable in the text of the considered program P, and the input under inspection is not representable in $\mathcal{A}$.

We have now all the ingredients to state the main result of this section.

Theorem 4.10. *Let $P \in \mathsf{Prog}$, $V \subseteq Var$, $\mathcal{A} \in \mathsf{Abs}(\wp(\mathbb{I}^n))$ non-relational, and assume the following:* (1) *S is V-bounded,* (2) *$\mathcal{A}$ is* Int*-abstractable, and* (3) *$\mathcal{A}$ is V-convex at S for P. Then, the following implication holds:*

$$Mon(P, \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)), V) \implies Compl^{\mathcal{A}}(P, S, V)$$

Proof. We prove the implication by showing that, by the four assumptions (1),(2),(3) and monotonicity, the BCA $[\![P]\!]^{\alpha}_{\mathcal{A}}$ is V-complete at S, thus, by Lemma 4.8, proving the possibility to build a V-complete analysis of variables in V at S. Lemma 4.9 already provides us a proof for the trivial cases. Let us consider the non-trivial cases where both $\mathcal{A} \neq \wp(\mathbb{I}^n)$, $V \cap Var(P) \neq \varnothing$ and $S \subset \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$ hold. Let $\tilde{V} = V \cap Var(P)$. The proof is made by contradiction. We assume:

(0) $\mathcal{A}$ non-relational;
(1) $min^{\tilde{V}}S$ and $max^{\tilde{V}}S$ exist in S, and $[\![P]\!]min^{\tilde{V}}S$, $[\![P]\!]max^{\tilde{V}}S \neq \varnothing$;
(2) $\mathcal{A}$ can exactly represents intervals;
(3) for all $x \in \tilde{V}$, the set $(\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S)))(x)$ is convex;
(4) P is V-monotone at $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$, namely, for all $x \in \tilde{V}$ either it holds
  i) $\forall \sigma_1, \sigma_2 \in \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) \cap \mathcal{D}_P.\ (\sigma_1(x) \leq \sigma_2(x) \Rightarrow ([\![P]\!]\sigma_1)(x) \leq ([\![P]\!]\sigma_2)(x))$ or
  ii) $\forall \sigma_1, \sigma_2 \in \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) \cap \mathcal{D}_P.\ (\sigma_1(x) \leq \sigma_2(x) \Rightarrow ([\![P]\!]\sigma_1)(x) \geq ([\![P]\!]\sigma_2)(x))$;
(5) $Compl^{\mathcal{A}}(P, S, V)$ does not hold, namely, $\exists x \in \tilde{V}\colon \alpha_{\mathcal{A}}([\![P]\!]S) <^{\{x\}}_{\mathcal{A}} [\![P]\!]^{\alpha}_{\mathcal{A}}\alpha_{\mathcal{A}}(S)$.

By (2), we know that all the spurious elements added by $\mathcal{A}$ are contained in the Int abstraction, namely, for all $I \in \wp(\mathbb{I}^n)$, $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(I)) \subseteq \gamma_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(I))$. This implies that, because by (1) S has a minimum and maximum element w.r.t. variables in $\tilde{V}$, minimum and maximum elements are not altered by $\mathcal{A}$, namely, $min^{\tilde{V}}S =^{\tilde{V}} min^{\tilde{V}}\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$ and $max^{\tilde{V}}S =^{\tilde{V}} max^{\tilde{V}}\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$. Let us consider the case where P is V-non-decreasing at $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$ (the non-increasing case is dual).

Starting from assumptions (1) and (2), we get:

$$\begin{aligned}
min^{\tilde{V}} S =^{\tilde{V}} min^{\tilde{V}} \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) &\Rightarrow \\
[\![P]\!] min^{\tilde{V}} S =^{\tilde{V}} [\![P]\!] min^{\tilde{V}} \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) &\Rightarrow \text{[by (4)]} \\
(a)\ min^{\tilde{V}} [\![P]\!] S =^{\tilde{V}} min^{\tilde{V}} [\![P]\!] \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) &\leq^{\tilde{V}} \text{[by (1)]} \\
max^{\tilde{V}} S =^{\tilde{V}} max^{\tilde{V}} \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) &\Rightarrow \\
[\![P]\!] max^{\tilde{V}} S =^{\tilde{V}} [\![P]\!] max^{\tilde{V}} \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) &\Rightarrow \text{[by (4)]} \\
max^{\tilde{V}} [\![P]\!] S =^{\tilde{V}} max^{\tilde{V}} [\![P]\!] \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)) &\Rightarrow \text{[by } (a),(2)] \\
\forall x \in \tilde{V}.\, [(min^{\{x\}} [\![P]\!] S)(x), (max^{\{x\}} [\![P]\!] S)(x)] &= \\
[(min^{\{x\}} [\![P]\!] \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)))(x), (max^{\{x\}} [\![P]\!] \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)))(x)] &\Rightarrow \text{[by (3)]} \\
\forall x \in \tilde{V}.\, \gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!] S)) =^{\{x\}} \gamma_{\mathcal{A}}([\![P]\!]^{\alpha}_{\mathcal{A}} \alpha_{\mathcal{A}}(S)) &
\end{aligned}$$

Note that, $\mathcal{A}$ non-relational and V-convex at S means that $(\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S)))(x)$ corresponds to the interval $[(min^{\{x\}}[\![P]\!]S)(x), (max^{\{x\}}[\![P]\!]S)(x)]$ for all the considered variables in $\tilde{V}$, i.e., all the 1-dimensional lines having $(min^{\{x\}}[\![P]\!]S)(x)$ as minimum element and $(max^{\{x\}}[\![P]\!]S)(x)$ as maximum element, while the interval $[(min^{\{x\}}[\![P]\!]\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)))(x), (max^{\{x\}}[\![P]\!]\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S)))(x)]$ corresponds to $(\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S)))(x)$ for all $x \in \tilde{V}$ because of the soundness property of abstract interpretation. Therefore, by the last derivation, we can conclude $\alpha_{\mathcal{A}}([\![P]\!]S) =^{V} [\![P]\!]^{\alpha}_{\mathcal{A}} \alpha_{\mathcal{A}}(S)$, i.e., $Compl^{\mathcal{A}}(P, S, V)$ holds, contradicting (5). □

Theorem 4.10 identifies specific conditions on the input states and on the abstract domain that guarantee the existence of a program analysis that accurately captures the full behavior of a program. More specifically, under the three assumptions of Theorem 4.10, the V-monotonicity of program P over the inputs $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$ is a sufficient condition to ensure the V-complete-analyzability of program P at S over the abstract domain $\mathcal{A}$. This result reveals a connection between the extensional property of V-monotonicity in programs and the completeness property of an abstract interpreter $[\![\cdot]\!]_{\mathcal{A}}$: the class of monotone programs represents a particular case over which it is possible to precisely (i.e., with no false positives and no false negatives) prove through $[\![\cdot]\!]_{\mathcal{A}}$ all properties expressible in the abstract domains and over the inputs that satisfy the three conditions of Theorem 4.10.

Example 4.11. Monotone activation functions in neural networks (see, e.g., [Albarghouthi 2021]) are used to add non-linearity to the function computed by a neuron. The following three programs[3] implement three well known activation functions:

$$\begin{aligned}
\text{Signum} &\stackrel{def}{=} \textbf{if } x < 0 \textbf{ then } x := -1 \\
&\qquad\qquad \textbf{else if } x > 0 \textbf{ then } x := 1 \textbf{ else } x := 0 \\
\text{ReLU} &\stackrel{def}{=} \textbf{if } x \leq 0 \textbf{ then } x := 0 \textbf{ else } x := x \\
\text{SiL} &\stackrel{def}{=} \frac{x}{1 + e^{-x}}
\end{aligned}$$

The Signum function maps negative input values to -1. The Sigmoid-weighted Linear Unit (SiL) [Elfwing et al. 2018] is used for neural network function approximation in reinforcement learning [Ramachandran et al. 2018]. Their input-output relation is represented in Fig. 6. Let us

[3]We assume that SiL is an implementation in Prog using only variable x.

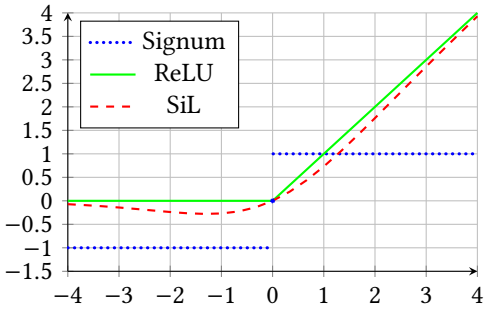

Fig. 6. Signum, ReLU and SiL activation functions.

consider the interval abstract domain Int on reals $\mathbb{R}$, which has been extensively used for verifying properties of neural networks for, e.g., image classification [Gehr et al. 2018; Gowal et al. 2019], natural-language processing [Huang et al. 2019], and cyber-physical systems [Wang et al. 2018]. It is easy to note that both Signum and ReLU are $\{x\}$-non-decreasing programs and they can be easily verified with the proof system defined in Section 3. This guarantees the existence of a complete analysis on Int for all possible bounded inputs since $[\![\text{Signum}]\!]^{\alpha}_{\mathsf{Int}}$ and $[\![\text{ReLU}]\!]^{\alpha}_{\mathsf{Int}}$ are complete.

Furthermore, if the overall neural network is monotone, for example, when it consists of non-decreasing activation functions and each neuron implements a non-decreasing function, then Theorem 4.10 assures us the possibility to implement a complete static analyzer on bounded inputs for the Int abstract domain and, more generally, for any non-relational abstraction $\mathcal{A}$ that satisfies Definition 4.4 and 4.5. This analyzer can accurately verify safety properties [Amodei et al. 2016] representable in $\mathcal{A}$, at bounded sets of inputs. This result is not guaranteed when the overall neural network is not monotone, namely, when either a neuron or the activation functions used are not monotone, such as SiL. For instance, suppose that we want to check whether the variable x at the end of the execution of SiL on the input set $\{-4, 0\}$ ranges in the interval $[-0.1, 0]$, and suppose we would like to check this specification using an abstract interpreter over Int able to answer this question with no imprecision. The concrete evaluation returns $\alpha_{\mathsf{Int}}([\![\text{SiL}]\!]\{-4, 0\}) \simeq [-0.07, 0]$, while the BCA on Int outputs $[\![\text{SiL}]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(\{-4, 0\}) \simeq [-0.28, 0]$. Note that $[\![\text{SiL}]\!]^{\alpha}_{\mathsf{Int}}$ leads to a false-alarm since $[\![\text{SiL}]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(\{-4, 0\}) \not\leq_{\mathsf{Int}} [-0.1, 0]$ even if the specification is satisfied: $\alpha_{\mathsf{Int}}([\![\text{SiL}]\!]\{-4, 0\}) \leq_{\mathsf{Int}} [-0.1, 0]$. This result excludes the possibility to define a complete analysis $[\![\text{SiL}]\!]_{\mathsf{Int}}$ over $\{-4, 0\}$. Nevertheless, if we consider an input set where SiL is monotone, e.g. $\{0, 4\}$, we have completeness: $\alpha_{\mathsf{Int}}([\![\text{SiL}]\!]\{0, 4\}) \simeq [0, 3.93] \simeq [\![\text{SiL}]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(\{0, 4\})\}$, thus admitting the possibility to build a complete analysis for this input. ♦

The converse of Theorem 4.10 does not hold, pointing out that the monotone condition is stronger than the notion of V-complete-analyzability, as shown by the following example.

Example 4.12. Consider the following program P : **if** $x \neq 3$ **then** $x := x$ **else** $x := x - 2$. It is easy to note that this program is not monotone: it is not non-decreasing since $2 < 3$ but $[\![P]\!](2) = 2 > 1 = [\![P]\!](3)$, and not non-increasing since $2 < 4$ but $[\![P]\!](2) = 2 < 4 = [\![P]\!](4)$. Consider the input set $\{1, 5\}$ and the $\mathsf{Int} \in \mathsf{Abs}(\wp(\mathbb{Z}))$ abstraction. Clearly, Int and $\{1, 5\}$ satisfy the assumptions of Theorem 4.10. However, $\alpha_{\mathsf{Int}}([\![P]\!]\{1, 5\}) = [1, 5] = [\![P]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(\{1, 5\})$, namely $[\![P]\!]^{\alpha}_{\mathsf{Int}}$ is complete at $\{1, 5\}$, therefore P is complete-analyzable at $\{1, 5\}$ even if P is not monotone. ♦

In the next three examples we show the reasons why the three assumptions of Theorem 4.10 are necessary. Assumptions (1), requiring that S is V-bounded, and (2), requiring that $\mathcal{A}$ is Int-abstractable, are strictly correlated: if S is V-bounded and P is V-non-decreasing, then $[\![P]\!]S$ is

V-bounded and its minimum and maximum are precisely $[\![P]\!] min^V S$ and $[\![P]\!] max^V S$ (or reversed in case of non-increasing programs). Moreover, when $\mathcal{A}$ is Int-abstractable then all the spurious elements added by the abstraction over S are enclosed by the minimum and maximum of S. Indeed, the two assumptions are, in a sense, complementary: if S is not V-bounded, then even if $\mathcal{A} \leq_{\mathsf{Abs}(\wp(\mathbb{I}^n))} \mathsf{Int}$, $\mathcal{A}$ could add new maximum and minimum values for a variable $x \in V$ at $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$ which may lead to incompleteness. This reasoning holds also for S V-bounded and $\mathcal{A}$ not representing intervals. The following example shows these corner cases.

Example 4.13. Consider the following program:

$$P \ : \ \textbf{if } x + y - 8 \geq 0 \textbf{ then } x := 1;\ y := 1 \textbf{ else } x := -1;\ y := -1$$

We analyze this program over the interval abstraction $\mathsf{Int} \in \mathsf{Abs}(\wp(\mathbb{Z}^2))$ on the input $S = \{(4,4),(3,5)\}$. It is easy to note that $\{(4,4),(3,5)\}$ is not $\{x,y\}$-bounded since the minimum $(3,4)$ for both x, y is not in S. The abstraction of $\{(4,4),(3,5)\}$ into Int^2 corresponds to $([3,4],[4,5])$, i.e., the 2-dimensional rectangle including $(4,4)$ and $(3,5)$. Therefore, $\gamma_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(\{(4,4),(3,5)\}))$ has new minimum and maximum elements for $\{x,y\}$, namely, $min^{\{x,y\}}\gamma_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(\{(4,4),(3,5)\})) = \{(3,4)\}$ and $max^{\{x,y\}}\gamma_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(\{(4,4),(3,5)\})) = \{(4,5)\}$. By using rules in Fig. 2 and 3, it is easy to verify that P is $\{x,y\}$-non-decreasing over $\gamma_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(\{(4,4),(3,5)\}))$ as the guard is positive linear and the two branches implies that $(-1,-1) <^{\{x,y\}} (1,1)$ so the predicate *Boundary* is trivially true on boundary states. However, $Compl^{\mathsf{Int}}(P, \{(4,4),(3,5)\}, \{x,y\})$ does not hold:

$$\begin{aligned}
\alpha_{\mathsf{Int}}([\![P]\!]S) &= \alpha_{\mathsf{Int}}(\{(1,1)\}) = ([1,1],[1,1]) \\
[\![P]\!]^{\alpha}_{\mathsf{Int}}\alpha_{\mathsf{Int}}(S) &= \alpha_{\mathsf{Int}}([\![P]\!]\{(4,4),(3,5),(3,4),(4,5)\}) \\
&= \alpha_{\mathsf{Int}}(\{(1,1),(-1,-1)\}) = ([-1,1],[-1,1]).
\end{aligned}$$

By $([1,1],[1,1]) <_{\mathsf{Int}} ([-1,1],[-1,1])$, we can deduce the $\{x,y\}$-incompleteness of $[\![P]\!]^{\alpha}_{\mathcal{A}}$ at $\{(4,4),(3,5)\}$. In this example, the minimal point $(3,4)$ added by Int caused the incompleteness.

Consider now a slightly change in the Boolean condition of P:

$$P' \ : \ \textbf{if } x + y - 7 \geq 0 \textbf{ then } x := 1;\ y := 1 \textbf{ else } x := -1;\ y := -1$$

We analyze this new program over the $\mathsf{Sign} \in \mathsf{Abs}((\mathbb{Z}^2))$ abstraction with input $S' = \{(3,4),(4,5)\}$. Clearly, Sign is not Int-abstractable as for instance $\gamma_{\mathsf{Sign}}(\alpha_{\mathsf{Sign}}(\{1,4\})) = \{0,1,2,3,4,5,\dots\} \not\subseteq \gamma_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(\{1,4\})) = \{1,2,3,4\}$. Here, even if S' is $\{x,y\}$-bounded, Sign adds also the spurious elements on the left of $min^{\{x,y\}}S = \{(3,4)\}$, therefore capturing another output property, the negative numbers $(-)$ in this case, as shown by the following evaluations:

$$\begin{aligned}
\alpha_{\mathsf{Sign}}([\![P']\!]S') &= \alpha_{\mathsf{Sign}}(\{(1,1)\}) = (+,+) \\
[\![P']\!]^{\alpha}_{\mathsf{Sign}}\alpha_{\mathsf{Sign}}(S') &= \alpha_{\mathsf{Sign}}([\![P']\!]\{(\mathbb{Z}_{\geq 0},\mathbb{Z}_{\geq 0})\}) = (\mathbb{Z},\mathbb{Z})
\end{aligned}$$

therefore we can conclude that P' is not $\{x,y\}$-complete analyzable at $\{(3,4),(4,5)\}$. ♦

As regarding assumption (3), we require V-convexity of the abstraction on the considered input S intuitively because there must be no "holes" on the property which captures the output of P at S otherwise there could exist monotone functions which "exploit" a point in that hole to make the function incomplete.

Example 4.14. Consider the abstract domain $\mathsf{P} \sqcap \mathsf{I} \in \mathsf{Abs}(\wp(\mathbb{Z}))$ which is the reduced product of Parity and Int abstract domains. $\mathsf{P} \sqcap \mathsf{I}$ represents properties which are the intersection between intervals and even or odd numbers. For example, $[0,10]_{even}$ represents $\{0,2,4,6,8,10\}$, that is, all the even numbers in the interval $[0,10]$. Clearly, this abstract domain can represent non-convex

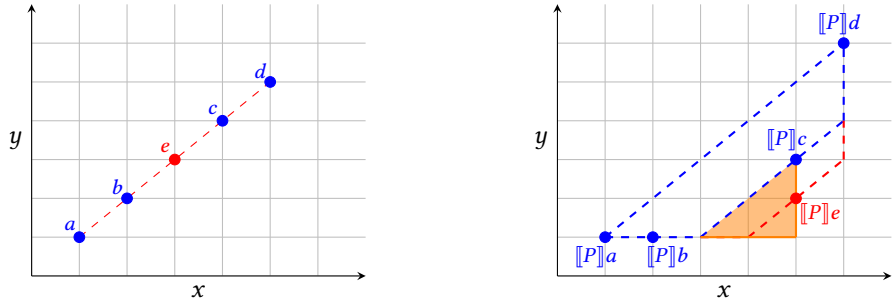

Fig. 7. On the left, the 2-dimensional representation of $S = \{a, b, c, d\}$ (blue points) and $\gamma_{\text{Zone}}(\alpha_{\text{Zone}}(S))$ which adds e (the red point). On the right, the monotone transformation P of S where the region outlined by the blue dashed line represents $\gamma_{\text{Zone}}(\alpha_{\text{Zone}}([\![P]\!]S))$, while the red dashed line corresponds to the region added by $\gamma_{\text{Zone}}([\![P]\!]^{\alpha}_{\text{Zone}}(S))$.

properties, such as $[0, 10]_{even}$. However, $\mathsf{P} \sqcap \mathsf{I}$ is Int-abstractable because it can represent exactly all the intervals. Let us consider the following program:

$$Q \ : \ \textbf{if } x \bmod 2 = 0 \vee x = 3 \textbf{ then skip else } x := x + 1$$

where we assume the mod operator is defined in Prog. Let us consider the set of inputs $\{2, 5\}$. It is easy to note that Q is non-decreasing over $\gamma_{\mathsf{P}\sqcap\mathsf{I}}(\alpha_{\mathsf{P}\sqcap\mathsf{I}}(\{2, 5\}))$. Moreover, by the evaluations

$$\alpha_{\mathsf{P}\sqcap\mathsf{I}}([\![Q]\!]\{2, 5\}) = \alpha_{\mathsf{P}\sqcap\mathsf{I}}(\{2, 6\}) = [2, 6]_{even}$$
$$[\![Q]\!]^{\alpha}_{\mathsf{P}\sqcap\mathsf{I}}\alpha_{\mathsf{P}\sqcap\mathsf{I}}(\{2, 5\}) = \alpha_{\mathsf{P}\sqcap\mathsf{I}}(\{2, 3, 4, 6\}) = [2, 6]$$

we can conclude that $\mathsf{P} \sqcap \mathsf{I}$ is not $\{x\}$-convex at $\{2, 5\}$, since $[2, 6]_{even}$ has many holes, namely, all the odd numbers in $[2, 6]$. Q is monotone over $\gamma_{\mathsf{P}\sqcap\mathsf{I}}(\alpha_{\mathsf{P}\sqcap\mathsf{I}}(\{2, 5\}))$ but $Compl^{\mathsf{P}\sqcap\mathsf{I}}(Q, \{2, 5\}, \{x\})$ does not hold as $[\![P]\!]^{\alpha}_{\mathsf{P}\sqcap\mathsf{I}}$ is not complete at $\{2, 5\}$: $[2, 6]_{even} <_{\mathsf{P}\sqcap\mathsf{I}} [2, 6]$. ♦

5 ON THE RELATIONAL ABSTRACT DOMAINS

So far we have exclusively focused on non-relational abstractions. A follow-up question could be whether it is possible to extend the result of Theorem 4.10 to relational abstractions able to infer affine inequalities. These include, e.g., Zones and Octagons abstract domains $\text{Zone}, \text{Oct} \in \text{Abs}(\wp(\mathbb{I}^n))$ which are able to express constraints with two variables: $x - y \leq k$ for Zone, while $\pm x \pm y \leq k$ for Oct with $k \in \mathbb{I}$. Unfortunately, the answer is negative. As a first observation, because relational abstractions consider, as the name suggest, relations between variables, it is no longer possible to employ the notion of completeness modulo a set of variables (Definition 4.1). Instead, the standard notion of completeness is considered. Let us look at the case of 2-dimensional inputs, namely when $Var(P) = \{x, y\}$. Firstly, note that the 2-dimensional abstraction shape of a relational Int-abstractable abstract domain able to infer affine inequalities over a bounded set of inputs S, will consist of edges having the form $x = 0$ and $y = mx + k$ with $m, k \geq 0$. This is because, when considering bounded sets, S has minimum and maximum points, $\mathcal{A}$ is at least as precise as Int, and, by assuming monotonicity, $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]S))$ and $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}([\![P]\!]\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))))$ do not modify the order in $\gamma_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))$. So, for example, the Oct abstraction of a monotonic program over an input will result in the same shape as the Zone abstraction. However, it turns out that even monotonic programs can violate the completeness property over a bounded set S for a relational abstraction. The following is a counterexample of Theorem 4.10 for the $\text{Zone} \in \text{Abs}(\wp(\mathbb{Z}^2))$ abstract domain.

Example 5.1. Consider a set $S = \{a, b, c, d\}$ of four symbolic points in two dimensions along the line $y = x$ depicted on the left of Fig. 7. This set has minimum at a and maximum at d. The Zone abstraction $\gamma_{\mathsf{Zone}}(\alpha_{\mathsf{Zone}}(S))$ adds one spurious element e between b and c. The figure on the right represents a monotone transformation $[\![P]\!]$ of the points in S: essentially, the minimum and maximum are left invariant, while b and c points are monotonically moved downward, keeping the $\leq$-order unchanged since $[\![P]\!]a \leq [\![P]\!]b \leq [\![P]\!]c \leq [\![P]\!]d$. The trapezoid having dashed blue edges represents the concrete abstraction $\gamma_{\mathsf{Zone}}(\alpha_{\mathsf{Zone}}([\![P]\!]S))$. If P transforms the spurious point e generated by the abstraction, as the red point on the right of Fig. 7, namely $[\![P]\!]e$, then program P is still monotonic over $\gamma_{\mathsf{Zone}}(\alpha_{\mathsf{Zone}}(S))$ since $[\![P]\!]b \leq [\![P]\!]e \leq [\![P]\!]c$. However, the shape outlined by $\gamma_{\mathsf{Zone}}([\![P]\!]^{\alpha}_{\mathsf{Zone}}(S))$ differs from the dashed blue trapezoid because of $[\![P]\!]e$: this point adds the region surrounded by the red dashed line. This implies that, even if P is non-decreasing over $\gamma_{\mathsf{Zone}}(\alpha_{\mathsf{Zone}}(S))$, $[\![P]\!]^{\alpha}_{\mathsf{Zone}}$ is not complete at S. In fact, it turns out that P could move e to any point in the orange region of Fig. 7 in order to keep monotonicity and break completeness. ♦

Relational abstract domains may disclose hidden relations between variables which may violate the complete-analyzability property over a bounded set S, even when all program variables behave monotonically. A better understanding of which operators (or weakening) preserve the complete-analyzability property is necessary, which we leave as a future work.

6 RELATED WORK

In the literature, there are many works exploring the role of monotonicity in programming languages (mostly referring to the non-decreasing case only). One of the most prominent and classical use of monotonicity is to guarantee the existence of a minimal fixpoint and termination of functions when certain conditions are satisfied. This goes back to the Knaster-Tarski fixpoint theorem [Tarski 1955] and its extensive uses, e.g., in program analysis [Nielson et al. 2015], or as the bases for defining programming languages such as Datafun [Arntzenius and Krishnaswami 2016] and Flix [Madsen et al. 2016]. In particular, Arntzenius and Krishnaswami [2016] track monotonicity of the functional language Datafun with the aid of a type system which has some similarities with our proof system. In Datafun variables could be declared monotone by users guiding the type system in recognizing monotone functions. Conversely, our aim is to inductively find monotone variables starting from a piece of code without any prior knowledge. Their type system differs from ours as their definition of monotonicity is slightly different: in their case a program is V-monotone when for every pair of stores that share the same values of variables not in V, the order of variables in V is preserved after the execution of the program (while we do not have any constraints on the variables not in V). Moreover, their rules for the if-statement consider two simple cases: the guard does not use monotone variables, or the false-branch corresponds to the least element.

Monotonicity is of key importance also in separation logic [Ahman et al. 2018; Pilkiewicz and Pottier 2011; Timany and Birkedal 2021] for reasoning about concurrent programs. Here monotonicity is used with respect to some relation as requirement on modules, for example a program module may need to know that the computation performed on the shared memory by other modules always amounts to progress in some monotone way. This is in contrast with the goal of our proof system where monotonicity is not a requirement but it is a property of variables that we want to discover in order to derive the complete-analyzability property.

Other uses of monotonicity lie also in distributed programming [Alvaro et al. 2011; Conway et al. 2012] to guarantee that nodes in a distributed database eventually agree [Vogels 2009]. Conway et al. [2012] propose a simple analysis for identifying possible non-monotone code locations of programs written in Bloom^L (an extension of the Bloom [Alvaro et al. 2011] declarative programming language incorporating a join-semilattice L with built-in monotone functions). The monotonicity analysis

here is carried out through a predicate dependency graph which is based on a simple program syntax test: locations where an asynchronously computed value is consumed by a non-monotone operator, are classified “at risk". Monotone operations are just a list of monotone functions defined by the lattices used by the current program.

The mathematical notions of continuity and differentiability are related to monotonicity, even when considering programs. Continuity has been extensively studied for programs [Chaudhuri et al. 2010] since it is a precursor for verifying their robustness [Chaudhuri et al. 2011]. Although monotonicity and continuity are different properties, the proof system defined in [Chaudhuri et al. 2010] for verifying continuity has some similarities with ours for monotonicity: both if rules consider boundary states for proving the respective property on branches. Chaudhuri et al. [2010] check that the true- and false-branches agree at boundary states on the considered variables (called V-equivalence), while for those states we check that the order is preserved (thanks to the positive linearity assumption of the guard). Furthermore, program continuity could be used as a sufficient condition for proving monotonicity on branches: roughly, by considering the non-decreasing case, if both branches are non-decreasing and the overall if-statement is proved to be continuous, then the if-statement is also non-decreasing as the V-equivalence on borders ensures the order of boundary states is not violated. Also differentiability has been studied in programming languages [Abadi and Plotkin 2020; Beck and Fischer 1994; Ehrhard and Regnier 2003; Elliott 2018] as it plays a prominent role in modern machine learning. Although the differentiability property can be exploited for verifying monotonicity of expressions (see (**assign**) rule), our proof system does not require differentiability of the branches, as this would be a stronger condition (there are non-differentiable functions that are monotone, e.g., ReLU).

The Interval Universal Approximation Theorem (IUA) [Baader et al. 2020; Wang et al. 2022] relates (computable) continuous function to provably robust neural networks through interval analysis. IUA guarantees that it is possible to find a provably robust neural network approximating a continuous function. We are relating, instead, the (extensional property) monotonicity of a program P (namely, a computable monotone function) with the existence of a provably complete analysis over it on a restricted class of non-relational domains and inputs, without passing through the phase of finding a new complete-analyzable program P' semantically equivalent to P. Nevertheless, these two results may hide interesting connections which deserve further investigations.

To the best of our knowledge, this is the first time that a relation between monotonicity (either non-decreasing or non-increasing) and the precision of program analysis has been established. Previous work in program analysis (e.g., by Giacobazzi et al. [2015] and Campion et al. [2022b]) shows that the precision of a program analyzer is strictly correlated to how programs are written. Our definition of V-completeness is based on the well known notion of completeness [Giacobazzi et al. 2000] and local completeness [Bruni et al. 2021, 2023]. Although it has been proved that completeness and its weakening are non-decidable program properties [Campion et al. 2022b; Giacobazzi et al. 2015], we showed that if a program is monotone then this fact is a sufficient condition for proving completeness of the BCA over a specific family of abstractions and inputs.

7 DISCUSSION

Completeness in abstract interpretation is influenced not only by how programs are written but also by what they compute. It is therefore an interesting mix of intensional and extensional program properties that impact the precision of program analysis. While the intensional nature of completeness in abstract interpretation has been recently investigated [Bruni et al. 2020; Giacobazzi et al. 2015], less is known about its extensional nature. We established a relation between extensional program properties and the precision of program analysis. Monotone programs are an interesting case study delimiting the precision of some non-relational abstractions. On sub-components that

behave monotonically, specifications *Spec* expressible on abstract domains satisfying the hypothesis of Theorem 4.10 can be proved with no false alarms (Theorem 2.7) by using some computationally less expensive non-relational abstract domains (e.g., Int). In case *Spec* is not expressible, we can always overapproximate *Spec* in that domain. Our result ensures the existence of a complete abstract interpreter that generates no additional false alarms with respect to this overapproximation over bounded inputs. Furthermore, the complete-analyzability property could support static analysis in the context of obfuscated programs [Collberg and Nagra 2009; Kinder 2012; Wagner 2019], e.g., in malware analysis where malware tend to conceal their behavior using obfuscation techniques [Campion et al. 2021; Dalla Preda et al. 2015; Moser et al. 2007; You and Yim 2010]. Being able to discover program variables that behave monotonically grants us the possibility to precisely analyze them and, thus, gain a better understanding of the malware's behavior.

As future work, we plan to investigate the relation between the recently introduced notion of partial completeness [Campion et al. 2022a,b, 2023] and monotonocity. Allowing a limited non-monotone behavior of a program over certain variables may be related to a weakening of the notion of complete-analyzability, thus admitting a partial completeness of the BCA. Moreover, it could be interesting to formalize a proof system able to underapproximate the set of complete-analyzable programs for an abstraction and a set of inputs without passing through monotonicity, as done for, e.g., the local completeness [Ascari et al. 2023; Bruni et al. 2023; Milanese and Ranzato 2022].

The properties of boundedness and convexity, both depending on the program under inspection, would require dedicated analysis. Although our current work primarily centers on the monotonicity property, we recognize the importance of exploring these aspects in future investigations. The boundness analysis involves the study on how to find minimum/maximum points over a set of inputs and to check termination over them. Convexity may require another analysis in the style of Section 3 even though there are well known abstract domains composed by only convex properties that trivially satisfy convexity (e.g., Intervals).

Although Theorem 4.10 considers non-relational abstractions only, we may think of partitioning the set of program variables in blocks containing monotone variables only and therefore obtaining a "weakly relational" abstract domain involving relations among only the variables in these blocks. This could potentially be related to the methods used for decomposing relational numerical abstract shapes, as considered in [Cousot et al. 2019; Singh et al. 2018].

In our current development, which does not involve data structures like arrays or pointers, the main source of imprecision in the monotonicity verifier lies within branches and loops: analyzing the validity of the predicates *Boundary* and *Limit* might be complex, and, additionally, they both may turn false even if the program is V-monotone. This may depend on some factors such as the presence of non-linear Boolean guards in if-statements and loops as we have not identified an efficient method to handle them without executing the program under analysis. Further investigation is required to fine-tune the verifier's precision in this direction.

There is a considerable amount of research on algorithmic problems for the generation of ranking functions (e.g., see [Almagor et al. 2021; Ben-Amram et al. 2019; Ben-Amram and Genaim 2014]) for proving loop termination which is also strongly connected with program monotonicity. For instance, it would be interesting to modify the proposed proof system in order to consider a notion of strict V-monotonicity (namely, either V-increasing or V-decreasing). Suppose the guard of the loop has the form $b \in Lin^{+}_{\geqslant}$ and $Var(b) \subseteq V$. In such case, when the loop body is proven to be V-decreasing at S and every input in S has at least a comparable state in S (formally $\forall \sigma \in S.\, \exists \sigma' \in S.\, \sigma < \sigma' \vee \sigma > \sigma'$), it can be regarded as evidence for the existence of a ranking function among program states S, thus a sufficient condition to establish termination. Moreover, rather than confining monotonicity to integer numbers, it would be interesting to extend its definition to ordinals, enabling the discovery of ordinal-based ranking functions for conditional termination proofs. [Urban and Miné 2014a,b].

ACKNOWLEDGMENTS

We express our gratitude to the anonymous reviewers of POPL2024 for their valuable insights and the suggested directions for future work. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-23-1-0544 and the French PEPR Intelligence Artificielle SAIF project (ANR-23-PEIA-0006).

REFERENCES

Martín Abadi and Gordon D. Plotkin. 2020. A simple differentiable programming language. *Proc. ACM Program. Lang.* 4, POPL (2020), 38:1–38:28. https://doi.org/10.1145/3371106

Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.* 2, POPL (2018), 65:1–65:30. https://doi.org/10.1145/3158153

Aws Albarghouthi. 2021. Introduction to Neural Network Verification. *Found. Trends Program. Lang.* 7, 1-2 (2021), 1–157. https://doi.org/10.1561/2500000051

Shaull Almagor, Toghrul Karimov, Edon Kelmendi, Joël Ouaknine, and James Worrell. 2021. Deciding ω-regular properties on linear recurrence sequences. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–24. https://doi.org/10.1145/3434329

Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. 2016. Concrete Problems in AI Safety. *CoRR* abs/1606.06565 (2016). arXiv:1606.06565 http://arxiv.org/abs/1606.06565

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 214–227. https://doi.org/10.1145/2951913.2951948

Flavio Ascari, Roberto Bruni, and Roberta Gori. 2023. Logics for Extensional, Locally Complete Analysis via Domain Refinements. In *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 1–27. https://doi.org/10.1007/978-3-031-30044-8_1

Maximilian Baader, Matthew Mirman, and Martin T. Vechev. 2020. Universal Approximation with Certified Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=B1gX8kBtPr

Thomas Beck and Herbert Fischer. 1994. The if-problem in automatic differentiation. *J. Comput. Appl. Math.* 50, 1-3 (1994), 119–131. https://doi.org/10.1016/0377-0427(94)90294-1

Amir M. Ben-Amram, Jesús J. Doménech, and Samir Genaim. 2019. Multiphase-Linear Ranking Functions and Their Relation to Recurrent Sets. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 459–480. https://doi.org/10.1007/978-3-030-32304-2_22

Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4 (2014), 26:1–26:55. https://doi.org/10.1145/2629488

Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2011. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Softw. Eng. Notes* 36, 1 (2011), 1–8. https://doi.org/10.1145/1921532.1921553

Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2015. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. *Found. Trends Program. Lang.* 2, 2-3 (2015), 71–190. https://doi.org/10.1561/2500000002

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 (2020), 28:1–28:28. https://doi.org/10.1145/3371096

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470608

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2022. Abstract interpretation repair. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 426–441. https://doi.org/10.1145/3519939.3523453

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A Correctness and Incorrectness Program Logic. *J. ACM* 70, 2 (2023), 15:1–15:45. https://doi.org/10.1145/3582267

Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2022a. On the Properties of Partial Completeness in Abstract Interpretation. In *Proceedings of the 23rd Italian Conference on Theoretical Computer Science, ICTCS 2022, Rome, Italy, September 7-9, 2022 (CEUR Workshop Proceedings, Vol. 3284)*, Ugo Dal Lago and Daniele Gorla (Eds.). CEUR-WS.org, 79–85. http://ceur-ws.org/Vol-3284/8665.pdf

Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2021. Learning metamorphic malware signatures from samples. *J. Comput. Virol. Hacking Tech.* 17, 3 (2021), 167–183. https://doi.org/10.1007/s11416-021-00377-z

Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2022b. Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498721

Marco Campion, Caterina Urban, Mila Dalla Preda, and Roberto Giacobazzi. 2023. A Formal Framework to Measure the Incompleteness of Abstract Interpretations. In *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14284)*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer, 114–138. https://doi.org/10.1007/978-3-031-44245-2_7

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 57–70. https://doi.org/10.1145/1706299.1706308

Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. 2011. Proving programs robust. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 102–112. https://doi.org/10.1145/2025113.2025131

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

Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, Michael J. Carey and Steven Hand (Eds.). ACM, 1. https://doi.org/10.1145/2391229.2391230

Patrick Cousot. 2021. *Principles of Abstract Interpretation.* The MIT Press, Cambridge, Mass.

Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*. Dunod, Paris, 106–130. https://doi.org/10.1145/390019.808314

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *J. Log. Comput.* 2, 4 (1992), 511–547. https://doi.org/10.1093/logcom/2.4.511

Patrick Cousot and Radhia 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, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 2:1–2:10. https://doi.org/10.1145/2603088.2603165

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. A^2I: abstract^2 interpretation. *Proc. ACM Program. Lang.* 3, POPL (2019), 42:1–42:31. https://doi.org/10.1145/3290355

M. Dalla Preda and R. Giacobazzi. 2005. Control Code Obfuscation by Abstract Interpretation. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, 301–310. https://doi.org/10.1109/SEFM.2005.13

Mila Dalla Preda, Roberto Giacobazzi, and Saumya K. Debray. 2015. Unveiling metamorphism by abstract interpretation of code properties. *Theor. Comput. Sci.* 577 (2015), 74–97. https://doi.org/10.1016/j.tcs.2015.02.024

M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. 2006. Opaque Predicates Detection by Abstract Interpretation. In *Proc. of the 11th Internat. Conf. on Algebraic Methodology and Software Technology* (*AMAST '06*) *(Lecture Notes in Computer Science, Vol. 4019)*. Springer-Verlag, 81–95.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

Thomas Ehrhard and Laurent Regnier. 2003. The differential lambda-calculus. *Theor. Comput. Sci.* 309, 1-3 (2003), 1–41. https://doi.org/10.1016/S0304-3975(03)00392-X

Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2018. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks* 107 (2018), 3–11. https://doi.org/10.1016/j.neunet.2017.12.012

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proc. ACM Program. Lang.* 2, ICFP (2018), 70:1–70:29. https://doi.org/10.1145/3236765

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 3–18. https://doi.org/10.1109/SP.2018.00058

Roberto Giacobazzi. 2008. Hiding Information in Completeness Holes: New Perspectives in Code Obfuscation and Watermarking. In *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, Antonio Cerone and Stefan Gruner (Eds.). IEEE Computer Society, 7–18. https://doi.org/10.1109/SEFM.2008.41

Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 261–273. https://doi.org/10.1145/2676726.2676987

Roberto Giacobazzi and Isabella Mastroeni. 2012. Making Abstract Interpretation Incomplete: Modeling the Potency of Obfuscation. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7460)*, Antoine Miné and David Schmidt (Eds.). Springer, 129–145. https://doi.org/10.1007/978-3-642-33125-1_11

Roberto Giacobazzi, Isabella Mastroeni, and Mila Dalla Preda. 2017. Maximal incompleteness as obfuscation potency. *Formal Aspects Comput.* 29, 1 (2017), 3–31. https://doi.org/10.1007/s00165-016-0374-2

Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. 2000. Making abstract interpretations complete. *J. ACM* 47, 2 (2000), 361–416. https://doi.org/10.1145/333979.333989

Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Arthur Mann, and Pushmeet Kohli. 2019. Scalable Verified Training for Provably Robust Image Classification. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*. IEEE, 4841–4850. https://doi.org/10.1109/ICCV.2019.00494

P. Granger. 1989. Static analysis of arithmetical congruences. *Intern. J. Computer Math.* 30 (1989), 165–190. https://doi.org/10.1080/00207168908803778

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Po-Sen Huang, Robert Stanforth, Johannes Welbl, Chris Dyer, Dani Yogatama, Sven Gowal, Krishnamurthy Dvijotham, and Pushmeet Kohli. 2019. Achieving Verified Robustness to Symbol Substitutions via Interval Bound Propagation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, 4081–4091. https://doi.org/10.18653/v1/D19-1419

Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 61–70. https://doi.org/10.1109/WCRE.2012.16

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to flix: a declarative language for fixed points on lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 194–208. https://doi.org/10.1145/2908080.2908096

Marco Milanese and Francesco Ranzato. 2022. Local Completeness Logic on Kleene Algebra with Tests. In *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13790)*, Gagandeep Singh and Caterina Urban (Eds.). Springer, 350–371. https://doi.org/10.1007/978-3-031-22308-2_16

Antoine Miné. 2001a. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2053)*, Olivier Danvy and Andrzej Filinski (Eds.). Springer, 155–172. https://doi.org/10.1007/3-540-44978-7_10

Antoine Miné. 2001b. The Octagon Abstract Domain. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, Elizabeth Burd, Peter Aiken, and Rainer Koschke (Eds.). IEEE Computer

Society, 310. https://doi.org/10.1109/WCRE.2001.957836

Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372. https://doi.org/10.1561/2500000034

Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*. IEEE Computer Society, 421–430. https://doi.org/10.1109/ACSAC.2007.21

Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 807–814. https://icml.cc/Conferences/2010/papers/432.pdf

Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis.* springer.

Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 13–25. https://doi.org/10.1145/3209108.3209109

Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, Stephanie Weirich and Derek Dreyer (Eds.). ACM, 73–86. https://doi.org/10.1145/1929553.1929565

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2018. Searching for Activation Functions. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Hkuq2EkPf

H. G. Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. https://doi.org/10.2307/1990888

Xavier Rival and Kwangkeun Yi. 2020. *Introduction to static analysis: an abstract interpretation perspective.* Mit Press.

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. https://doi.org/10.1145/3188720

Dana S Scott and Christopher Strachey. 1971. *Toward a mathematical semantics for computer languages*. Vol. 1. Oxford University Computing Laboratory, Programming Research Group Oxford.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.* 2, POPL (2018), 55:1–55:28. https://doi.org/10.1145/3158143

Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics* 5, 2 (1955), 285–309.

Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 91–104. https://doi.org/10.1145/3437992.3439931

William F Trench. 2013. *Introduction to real analysis.* Faculty Authored and Edited Books & CDs. 7.

Caterina Urban and Antoine Miné. 2014a. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 412–431. https://doi.org/10.1007/978-3-642-54833-8_22

Caterina Urban and Antoine Miné. 2014b. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8723)*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer, 302–318. https://doi.org/10.1007/978-3-319-10936-7_19

Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44. https://doi.org/10.1145/1435417.1435432

Rusty Wagner. 2019. Modern Static Analysis of Obfuscated Code. In *Proceedings of the 3rd ACM Workshop on Software Protection* (London, United Kingdom) *(SPRO'19)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3338503.3357718

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 1599–1614. https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

Zi Wang, Aws Albarghouthi, Gautam Prakriya, and Somesh Jha. 2022. Interval universal approximation for neural networks. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498675

Glynn Winskel. 1993. *The formal semantics of programming languages - an introduction.* MIT Press.

Zaifu Yang. 2009. Discrete fixed point analysis and its applications. *Journal of fixed point theory and applications* 6 (2009), 351–371. https://doi.org/10.1007/s11784-009-0130-9

Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *Proceedings of the Fifth International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2010, November 4-6, 2010,*

Fukuoka Institute of Technology, Fukuoka, Japan (In conjunction with the 3PGCIC-2010 International Conference). IEEE Computer Society, 297–300. https://doi.org/10.1109/BWCCA.2010.85

Received 2023-07-11; accepted 2023-11-07