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:

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
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 $\llbracket \cdot \rrbracket_{\mathcal{A}}$ 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, $[P]S \subseteq Spec \iff [P]_{\mathcal{A}}\alpha_{\mathcal{A}}(S) \leq_{\mathcal{A}} \alpha_{\mathcal{A}}(Spec)$. 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) $\llbracket \cdot \rrbracket_{\mathcal{A}}^{\alpha_{\mathcal{A}}} = \alpha_{\mathcal{A}} \circ [P] \circ \gamma_{\mathcal{A}}$ 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 $\llbracket \cdot \rrbracket_{\mathcal{A}}^{\alpha_{\mathcal{A}}}$ is more precise than any other abstract interpreter $\llbracket \cdot \rrbracket_{\mathcal{A}}$ 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 $[P]_{\mathcal{A}}^{\alpha_{\mathcal{A}}} = [Q]_{\mathcal{A}}^{\alpha_{\mathcal{A}}}$, 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
\[
\text{ReLU} \overset{\text{def}}{=} \text{if } x \leq 0 \text{ then } x := 0 \text{ else } x := x
\]
also known as ReLU in artificial neural networks [Nair and Hinton 2010], that filters the input below 0. Consider the abstract domain of intervals \(\text{Int}\), where sets of integers \(S \subseteq \mathbb{Z}\) are abstracted by their bounds, i.e., the least interval \(\alpha_{\text{int}}(S) = [l, u]\) 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 \([\text{ReLU}]_{\text{int}}([l, u]) = [l', u']\) 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, \([\text{ReLU}]_{\text{int}}\) is complete: \(\alpha_{\text{int}}([\text{ReLU}]_{\text{int}}(S)) = [\text{ReLU}]_{\text{int}}(S)\). This means that the bounds computed by ReLU on \(S\) are not altered if we run \([\text{ReLU}]_{\text{int}}\) 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: \(0 \in \alpha_{\text{int}}([\text{ReLU}]_{\text{int}}(S)) \Leftrightarrow 0 \in [\text{ReLU}]_{\text{int}}\alpha_{\text{int}}(S)\). In this case \([\text{ReLU}]_{\text{int}}\) corresponds to the BCA \([\text{ReLU}]_{\text{int}}^a\).

This is not the case for the program
\[
\text{ABS} \overset{\text{def}}{=} \text{if } x \geq 0 \text{ then } x := x \text{ else } x := -x
\]
computing the absolute value of \(x\). In this case, even the BCA on \(\text{Int}\) may report a false-alarm for \(e = f(x)/\text{ABS}(x)\). For instance, with \(S = \{-7, 7\}\), we have: \(\alpha_{\text{int}}([\text{ABS}]_{\text{int}}(\{-7, 7\})) = [7, 7]\), while \([\text{ABS}]_{\text{int}}\alpha_{\text{int}}(\{-7, 7\}) = [0, 7]\) and \([7, 7] <_{\text{int}} [0, 7]\). Hence, even if \(0 \notin \alpha_{\text{int}}([\text{ABS}]_{\text{int}}(S))\), interval analysis may return a potential false-alarm for \(e\), namely \(0 \in [\text{ABS}]_{\text{int}}\alpha_{\text{int}}(S)\). Consequently, any sound approximation \([\text{ABS}]_{\text{int}}\) of \([\text{ABS}]_{\text{int}}^a\) 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 \([\text{ABS}]_{\text{int}}^a\) holds. For example, with \(S_1 = \{1, 3, 6\}\) and \(S_2 = \{-2, -5\}\) we get
\[
\begin{align*}
\alpha_{\text{int}}([\text{ABS}]_{\text{int}}(S_1)) &= [1, 6] = [\text{ABS}]_{\text{int}}^a\alpha_{\text{int}}(S_1) \\
\alpha_{\text{int}}([\text{ABS}]_{\text{int}}(S_2)) &= [2, 5] = [\text{ABS}]_{\text{int}}^a\alpha_{\text{int}}(S_2)
\end{align*}
\]
Observe that in these latter cases \(\text{ABS}\) 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: \textit{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?}

\textbf{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 numerical information.
of inputs. The proof judgments have the form $\text{Mon}^\prec (P, S, V)$ meaning that program $P$ is non-decreasing on the set of inputs $S$ with respect to variables in $V$ (the judgments $\text{Mon}^\succ (P, S, V)$ 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 $\sum_{i=1}^{\text{Var}(e)} a_i x_i + k$ where $x_i$ is a program variable, $v \in \mathbb{R}_{\geq 0}$, $k \in \mathbb{R}$ and $|\text{Var}(e)|$ 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 $[\text{ReLU}]_{\text{Int}}$ 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 $\equiv$ 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 ReLU$^w \overset{df}{=} \text{while } x < 0 \text{ do } x := x + 1$ 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 $T$, $\varphi(S)$ denotes the powerset of $S$, the symbol $\emptyset$ 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 $I \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ 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 $S \subseteq \mathbb{N}^n$ 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 $\equiv$. 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 $(L, \leq_L)$. 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 $\lor_L X$, and a unique meet, also called greatest lower bound (glb), denoted $\land_L X$. The tuple $(L, \leq_L, \lor_L, \land_L, \top_L, \bot_L)$, where $\top_L$ and $\bot_L$ are, respectively, the greatest (top) and least (bottom) elements in $L$, while $\lor_L$ and $\land_L$ 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 $f : L \rightarrow L$ over a poset $(L, \leq_L)$ 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 $f_1 : L_1 \rightarrow L_2$, $f_2 : L_2 \rightarrow L_3$ is denoted by $f_2 \circ f_1 : L_1 \rightarrow L_3$. A function $f : L_1 \rightarrow L_2$ between complete lattices is additive (resp. co-additive) if for all $Y \subseteq L_1$, $f(\lor_L Y) = \lor_L f(Y)$ (resp. $f(\land_L Y) = \land_L f(Y)$). The Knaster-Tarski theorem guarantees that if $L$ is a complete lattice and $f : L \rightarrow L$ 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 $\text{lfp}(f)$ (resp. $\text{gfp}(f)$). Moreover, if $f : L \rightarrow L$ is (Scott) continuous, i.e., $f$ preserves lubs of chains in $L$, then $\text{lfp}(f) = \lor_{n \in \mathbb{N}} f^n(\bot_L)$, where, for all $n \in \mathbb{N}$ and $x \in L$, $f^n$ is inductively defined by: $f^0(x) \overset{def}{=} x$ and $f^{n+1}(x) \overset{def}{=} f(f^n(x))$.

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

Proc. ACM Program. Lang., Vol. 8, No. POPL, Article 55. Publication date: January 2024.
A single store update is written abbreviate operations like

\[ \text{skip} \] def \[ S \]
\[ x := e ] \{ \sigma[x \mapsto \langle e \rangle \sigma] | \sigma \in S \}
\[ [P_1; P_2] S \quad [P_2][P_1] S \]
\[ \text{if } b \text{ then } P_1 \text{ else } P_2 \] \{ \[ b \] \} \cup \{ \neg b \} S
\[ \text{while } b \text{ do } P \] \{ \neg b \}(\text{fp}(\lambda T. \ S \cup [P][b] T))

Fig. 1. Collecting denotational semantics of Prog.

defined as follows:

\[ \text{AExp} \ni e ::= v \in \mathbb{I} | x \in \text{Var} | e + e | e - e | e \times e \]
\[ \text{BExp} \ni b ::= \text{true} | \text{false} | e \times 0 | e \times 0 | b \land b | b \lor b \]
\[ \text{Prog} \ni P ::= \text{skip} | x ::= e | P_1; P_2 |
\[ \text{if } b \text{ then } P_1 \text{ else } P_2 | \text{while } b \text{ do } P \]

where \( \in \{\lt, \leq, \gt, \geq\} \) and, by abusing notation, \( \text{Var} \) is both used to denote a denumerable set of variables and, when applied to a program \( P \in \text{Prog} \), denotes the (finite) set of variables in the text of \( P \), namely, \( \text{Var} : \text{Prog} \rightarrow \phi(\text{Var}) \). Similarly, when applied to arithmetic and Boolean expressions, \( \text{Var}(e) \) and \( \text{Var}(b) \) denote the variables appearing in those expressions. We sometimes abbreviate operations like \( 2 * x \) and \( x * x \), relative to, \( 2x \) and \( x^2 \). From now on and in the rest of the paper, whenever we talk about a program \( P \in \text{Prog} \), we assume \( |\text{Var}(P)| = n \), unless otherwise specified. A store \( \sigma_P \) for \( P \) is a total function from variables in the text of \( P \) to their values, namely, \( \sigma_P : \text{Var}(P) \rightarrow \mathbb{I} \). A store \( \sigma_P \) 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 \( \text{Var}(P) = \{x_1, \ldots, x_n\} \), \( \sigma(x_i) = v_i \), therefore \( \llbracket \text{Var}(P) \rrbracket \) is the set of all possible stores for \( P \). Most of the examples shown in the paper consider programs with \( \text{Var}(P) = \{x, y, z\} \), so that, a tuple like \((10, 2, 5) \in \mathbb{I}^3 \) corresponds to the store \( \sigma_P \) 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 \( \sigma \) when it is clear from context. The semantics of arithmetic and Boolean expressions of \( P \) is defined by the functions, respectively, \( \langle e \rangle : \mathbb{I}^n \rightarrow \mathbb{I} \) and \( \langle b \rangle : \mathbb{I}^n \rightarrow \{\text{true}, \text{false}\} \) whose definitions are straightforward and therefore omitted. The collecting semantics of arithmetic and Boolean expressions is respectively defined by the functions \( \llbracket e \rrbracket : \phi(\mathbb{I}^n) \rightarrow \phi(\mathbb{I}) \) and \( \llbracket b \rrbracket : \phi(\mathbb{I}^n) \rightarrow \phi(\mathbb{I}^n) \). The collecting denotational program semantics is \( [P] : \phi(\mathbb{I}^n) \rightarrow \phi(\mathbb{I}^n) \) and it is defined in Fig. 1, where the operator \( \neg b \) transforms the Boolean expression \( b \) into its negate. It is the standard predicate transformer semantics (also called strongest postcondition semantics) since \( [P] S \in \phi(\mathbb{I}^n) \) turns out to be the strongest store predicate for the store precondition \( S \in \phi(\mathbb{I}^n) \). The terminology "collecting semantics" comes from the fact that for all \( P \in \text{Prog} \), \( [P] : \phi(\llbracket \text{Var}(P) \rrbracket) \rightarrow \phi(\llbracket \text{Var}(P) \rrbracket) \) is an additive function on the complete lattice \( \langle \phi(\llbracket \text{Var}(P) \rrbracket) \rangle \subseteq \llbracket \cup \bigcap \llbracket \text{Var}(P) \rrbracket, \emptyset \rangle \), so that \( [P] S = \cup_{\sigma \in S} [P] \{\sigma\} \). 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 \((C \leq_C)\) and \((A, \leq_A)\), which are related by abstraction and concretization maps, \(\alpha_A : C \to A\) and \(\gamma_A : A \to C\), that give rise to a GC \((\alpha_A, C, A, \gamma_A)\), that is, for all \(a \in A\) and \(c \in C\): \(\alpha_A(c) \leq a \iff c \leq_A \gamma_A(a)\), where we use the subscript to functions \(\alpha_A\) and \(\gamma_A\) in order to emphasize the abstract domain \(A\) considered. A GC is a GI when \(\alpha_A \circ \gamma_A = \lambda x. x\). Let us recall some basic properties of a GC \((\alpha_A, C, A, \gamma_A)\): (1) \(\alpha_A\) is additive and \(\gamma_A\) is co-additive; (2) \(\gamma_A \circ \alpha_A : C \to C\) 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 \(\text{Abs}(C)\) to denote all the possible abstractions of a concrete domain \(C\), where \(A \in \text{Abs}(C)\) means that \(A\) 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 \(A\) whenever \(\gamma_A(\alpha_A(c)) = c\). If we consider two abstract domains \(A_1, A_2 \in \text{Abs}(C)\) then \(A_1\) is a more precise abstraction than \(A_2\), or, equivalently, \(A_2\) abstracts \(A_1\), if and only if for all \(c \in C\), \(\gamma_{A_1}(\alpha_{A_1}(c)) \leq C \gamma_{A_2}(\alpha_{A_2}(c))\), and it is denoted by \(A_1 \leq_{\text{Abs}(C)} A_2\). An abstract domain \(A \in \text{Abs}(C)\) is said to be trivial when \(A = C\), namely, it is isomorphic to the concrete domain \(C\) (i.e., \(\gamma_A \circ \alpha_A\) is the identity function).

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

\[
\alpha_{A^n}(S) \overset{def}{=} \begin{cases} 
\bot_{A^n} & \text{if } S = \emptyset, \\
(a_1, \ldots, a_n) \in A^n & \text{where } a_i \overset{def}{=} \alpha_A(\{\sigma(x_i) \mid \sigma \in S\}) \text{ otherwise}
\end{cases}
\]

\[
\gamma_{A^n}(\sigma^#) \overset{def}{=} \begin{cases} 
\emptyset & \text{if } \sigma^# = \bot_{A^n}, \\
\{(u_1, \ldots, u_n) \in I^n \mid u_i \in \gamma_A(\sigma^#(x_i))\} & \text{otherwise.}
\end{cases}
\]

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

Example 2.2. The classical pedagogical examples include the abstract domains \(\text{Sign} \overset{def}{=} \{1, -1, 0, +, \emptyset\}\) and \(\text{Parity} \overset{def}{=} \{\mathbb{Z}, \text{even}, \text{odd}, \emptyset\}\) for, respectively, sign and parity analysis of numerical variables. These are straightforward non-relational abstractions of \((\phi(I), \subseteq)\) [Cousot and Cousot 1976], namely, \(\text{Sign} \in \text{Abs}(\phi(I))\) and \(\text{Parity} \in \text{Abs}(\phi(\mathbb{Z}))\), where the order relation \(\leq_{\text{Sign}}\) is defined as \(\emptyset \leq_{\text{Sign}} 0 <_{\text{Sign}} - <_{\text{Sign}} 1 \text{ and } \emptyset <_{\text{Parity}} 0 <_{\text{Parity}} + <_{\text{Parity}} 1\), while \(\emptyset \leq_{\text{Parity}} \text{even} <_{\text{Parity}} \mathbb{Z}\) and

\footnote{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.}
\( \emptyset <_{\text{Parity}} \text{odd} <_{\text{Parity}} \mathbb{Z} \). The abstraction maps \( \alpha_{\text{Sign}} : \varnothing(\mathbb{I}) \rightarrow \text{Sign} \) and \( \alpha_{\text{Parity}} : \varnothing(\mathbb{Z}) \rightarrow \text{Parity} \) are defined as follows:

\[
\alpha_{\text{Sign}}(X) \equiv \begin{cases} 
\emptyset & \text{if } X = \emptyset, \\
0 & \text{if } X = \{0\}, \\
+ & \text{if } \forall x \in X. \, x \geq 0, \\
- & \text{if } \forall x \in X. \, x \leq 0, \\
\| & \text{otherwise}
\end{cases}
\]

\[
\alpha_{\text{Parity}}(X) \equiv \begin{cases} 
\emptyset & \text{if } X = \emptyset, \\
0 & \text{even if } \forall x \in X. \, x \text{ mod } 2 = 0, \\
\text{odd} & \text{if } \forall x \in X. \, x \text{ mod } 2 \neq 0, \\
\mathbb{Z} & \text{otherwise}
\end{cases}
\]

where mod is the integer modulo operation.

**Example 2.3.** The interval abstraction \( \text{Int} \) \([\text{Cousot and Cousot 1976}]\) 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 \( I^* \equiv \mathbb{I} \cup \{-\infty, +\infty\} \) and assume that the standard ordering \( \leq \) on \( \mathbb{I} \) is extended to \( I^* \) in the usual way. Hence \( \text{Int} \) is endowed with the standard ordering \( \leq_{\text{Int}} \) induced by the interval containment gives rise to a complete lattice, where \( \bot_{\text{Int}} \) is the bottom element and \( \top_{\text{Int}} \equiv [-\infty, +\infty] \) is the top element. We have that \( \text{Int} \in \text{Abs}(\varnothing(\mathbb{I})) \). Consider the function \( \text{min} : \varnothing(\mathbb{I}) \rightarrow \mathbb{I}^* \) defined as \( \text{min}(S) \equiv x \) if there exists \( x \in S \) such that for all \( y \in S \, x \leq y \), while \( \text{min}(S) \equiv -\infty \) otherwise, and the function \( \text{max} : \varnothing(\mathbb{I}) \rightarrow \mathbb{I}^* \) dually defined. The abstraction map \( \alpha_{\text{Int}} : \varnothing(\mathbb{I}) \rightarrow \text{Int} \) is defined by:

\[
\alpha_{\text{Int}}(X) \equiv \begin{cases} 
\bot_{\text{Int}} & \text{if } X = \emptyset, \\
[\text{min}(X), \text{max}(X)] & \text{otherwise}
\end{cases}
\]

Note that \( \alpha_{\text{Int}} \) preserves arbitrary unions in \( \varnothing(\mathbb{I}) \) and therefore gives rise to a GI.

**Abstract Interpretation.** Let \( f : C \rightarrow C \) be a concrete monotone (transfer) function (to keep notation simple we consider unary functions) and let \( f^{\#} : \mathcal{A} \rightarrow \mathcal{A} \) be a corresponding abstract (transfer) function defined on some abstraction \( \mathcal{A} \in \text{Abs}(C) \). Then, \( f^{\#} \) is a correct (or sound) approximation of \( f \) on \( \mathcal{A} \) when \( \alpha_{\mathcal{A}} \circ f \leq_{\mathcal{A}} f^{\#} \circ \alpha_{\mathcal{A}} \) holds. If \( f^{\#} \) is correct for \( f \) then least fixpoint correctness holds, that is, \( \alpha_{\mathcal{A}}(\text{lfp}(f)) \leq_{\mathcal{A}} \text{lfp}(f^{\#}) \) 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 \( f^{\#} : \mathcal{A} \rightarrow \mathcal{A} \) defined as \( f^{\#} \equiv \alpha_{\mathcal{A}} \circ f \circ \gamma_{\mathcal{A}} \) is called the best correct approximation (BCA for short) of \( f \) on \( \mathcal{A} \).

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

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

This definition of completeness is taken from the local completeness notion introduced by \( \text{Bruni et al. 2021, 2023} \), which is a weakening of the standard notion of completeness requiring Definition 2.5 to hold over all possible inputs \( c \in C \) \([\text{Cousot 2021; Giacobazzi et al. 2000}]\). 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 \( f^{\#} \) is an abstract transfer function on \( \mathcal{A} \) used in some static program analysis algorithm, completeness encodes an optimal precision for \( f^{\#} \) at input \( c \), meaning that the

---

Proc. ACM Program. Lang., Vol. 8, No. POPL, Article 55. Publication date: January 2024.
abstract behavior of $f^\#$ 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 $f^\#$ of $f$ on some $\mathcal{A} \in \text{Abs}(C)$ and $c \in C$ only depends upon the concrete function $f$, the abstraction $\mathcal{A}$ and the input $c$, that is, $f^\#(c)$ 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 $f^\#$ over an input $c \in C$ holds if and only if $\alpha_\mathcal{A}(f(c)) = \alpha_\mathcal{A}(f(\gamma_\mathcal{A}(\alpha_\mathcal{A}(c)))) = f^\alpha(\alpha_\mathcal{A}(c)) = f^\#(\alpha_\mathcal{A}(c))$.

**Completeness and Static Verification.** The abstract interpreter applied to a program $P \in \text{Prog}$ is specified by the function $[P]_\mathcal{A} : \mathcal{A} \rightarrow \mathcal{A}$, where $\mathcal{A} \in \text{Abs}(\wp(\text{Var}(P)))$ 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 $[P]_\mathcal{A}^\alpha : \mathcal{A} \rightarrow \mathcal{A}$ the abstract interpreter given by the BCA and defined as $[P]_\mathcal{A}^\alpha \overset{def}{=} \alpha_\mathcal{A} \circ [P] \circ \gamma_\mathcal{A}$. 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 $[P]_\mathcal{A}^\alpha(S)$ is complete over the input $S \in \wp(\text{Var}(P))$, it is sufficient to show that $[P]_\mathcal{A}^\alpha[S] = [P]_\mathcal{A}^\alpha(S) = \alpha_\mathcal{A}(\gamma_\mathcal{A}S)$. 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 $P \in \text{Prog}$, an input $S \subseteq \mathbb{N}$ and a safety property (also called correctness property) $\ spec \subseteq \mathbb{N}$ representable in our chosen abstract domain $\mathcal{A}$, the aim of a static verification $[P]_\mathcal{A}^\alpha[S] \subseteq \ spec$ 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 $\alpha_\mathcal{A}([P]S) = [P]_\mathcal{A}^\alpha(S)$, then proving $[P]S \subseteq \ spec$ by executing the program is the same as checking whether $[P]_\mathcal{A}^\alpha[S] \subseteq \ spec$ 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 $S \subseteq \mathbb{N}$ and $\ spec \subseteq \mathbb{N}$ is representable in $\mathcal{A}$, then the following holds: $[P]S \subseteq \ spec \Leftrightarrow [P]_\mathcal{A}^\alpha(S) \subseteq \ spec$.

**Proof.** Let $\mathcal{A} \in \text{Abs}(\wp(\mathbb{N}))$, $[P]_\mathcal{A}$ 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:

$$[P]_\mathcal{A}^\alpha(S) \subseteq \ spec \ \Rightarrow \ \alpha_\mathcal{A}(\ spec) \ \Rightarrow \ \text{[by Soundness of } [\cdot]_\mathcal{A}]$$

$$\alpha_\mathcal{A}(\ [P]S) \subseteq \ spec \ \Rightarrow \ \gamma_\mathcal{A} \text{ monotone}$$

$$\gamma_\mathcal{A}(\alpha_\mathcal{A}(\ [P]S)) \subseteq \ spec \ \Rightarrow \ \gamma_\mathcal{A}(\alpha_\mathcal{A}(\ spec)) \ \Rightarrow \ \text{Spec} = \gamma_\mathcal{A}(\alpha_\mathcal{A}(\ spec))$$

$$\gamma_\mathcal{A}(\alpha_\mathcal{A}(\ [P]S)) \subseteq \ spec \ \Rightarrow \ \gamma_\mathcal{A} \circ \alpha_\mathcal{A} \text{ closure operator}$$

$$[P]S \subseteq \ spec$$

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

$$[P]S \subseteq \ spec \ \Rightarrow \ \text{[by } \alpha_\mathcal{A} \text{ monotone]}$$

$$\alpha_\mathcal{A}(\ [P]S) \leq \alpha_\mathcal{A}(\ spec) \ \Rightarrow \ \text{[by } [P]_\mathcal{A} \text{ complete at } S]$$

$$[P]_\mathcal{A}^\alpha(S) \leq \ spec$$

$\square$
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 $P \in \text{Prog}$, its denotational semantics $\llbracket P \rrbracket$ over singletons $\sigma \in \mathcal{I}^{\text{Var}(P)}$ can be considered as a function from $(\mathcal{I}^{\text{Var}(P)})^n$ to $(\mathcal{I}^{\text{Var}(P)})^n$ 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 $V \subseteq \text{Var}$ of program variables. Let $P \in \text{Prog}$ be a program with $|\text{Var}(P)| = n$ variables, and $\leq^V$ be the elementwise inequality only for variables in $V$ and that also appear in $\text{Var}(P)$, formally, $\forall \sigma_1, \sigma_2 \in \mathcal{I}^n$:

$$\sigma_1 \leq^V \sigma_2 \iff \forall x \in V \cap \text{Var}(P). \sigma_1(x) \leq \sigma_2(x)$$

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

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

$$\sigma_1 \leq^V \sigma_2 \implies \llbracket P \rrbracket \sigma_1 \leq^V \llbracket P \rrbracket \sigma_2 \quad \text{(resp. } \llbracket P \rrbracket \sigma_1 \geq^V \llbracket P \rrbracket \sigma_2)$$

$P$ is called $V$-monotone at $S \subseteq \mathcal{I}^n$ 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 $V \subseteq \text{Var}$ 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: $\llbracket P \rrbracket \sigma_1 \leq^{\{x,y,z\}} \llbracket P \rrbracket \sigma_2$. 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 $\llbracket P \rrbracket (1, 0, 0) = (2, 1, 3) \not\leq^{\{x,z\}} (4, -9, -5) = \llbracket P \rrbracket (2, -10, 0)$. ♦

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 $\llbracket \text{Var}(P) \rrbracket$. 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 \( \text{Mon}(P, S, V) \) meaning that program \( P \) is \( V \)-monotone w.r.t. the set of variables \( V \subseteq \text{Var} \) and over a set of inputs \( S \). We write \( \vdash \text{Mon}(P, S, V) \) when the proof system allows us to formalize a derivation for the monotonicity judgement \( \text{Mon}(P, S, V) \). We will show that the proposed proof system is sound, namely, \( \vdash \text{Mon}(P, S, V) \) implies that \( \text{Mon}(P, S, V) \) holds. Our proof system computes an underapproximation of the sets of monotone variables:

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

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 \( \text{Mon} \neg(P, S, V) \) (the non-increasing case \( \text{Mon} \neg(P, S, V) \) 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 \( \sigma_1 \leq^V \sigma_2 \Rightarrow \llbracket x := e \rrbracket \sigma_1 \leq^V \llbracket x := e \rrbracket \sigma_2 \). 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., \( \text{Var}(e) \subseteq V \), (2) the set of inputs \( S \) is convex, and (3) the gradient of the function \( \llbracket e \rrbracket : [\text{Var}(e)] \rightarrow \mathbb{I} \), i.e., the column vector of all partial derivatives of \( \llbracket e \rrbracket \) denoted

\[
\nabla(\llbracket e \rrbracket) \equiv \left( \frac{\partial(\llbracket e \rrbracket)}{\partial x_1}, \ldots, \frac{\partial(\llbracket e \rrbracket)}{\partial |\text{Var}(e)|} \right)^T
\]

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 \( \llbracket e \rrbracket \) are differentiable over \( [\text{Var}(e)] \), namely, the gradient \( \nabla(\llbracket e \rrbracket) \) is always defined and can be obtained by symbolic differentiation. By requiring \( \nabla(\llbracket e \rrbracket) \geq 0_{|\text{Var}(e)| \times 1} \), where \( 0_{|\text{Var}(e)| \times 1} \) is the column vector of all 0s having number of rows equal to \( |\text{Var}(e)| \), the result is a system of constraints on \( \text{Var}(e) \), limiting the input states to only those that make the gradient \( \nabla(\llbracket e \rrbracket) \) non-negative [Trench 2013].

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

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

\footnote{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 \( \emptyset \) by \textit{false}.}
that both $S$ forms a convex set, denoted by the predicate $\text{Conv}(S)$, and all the states in $S$ satisfies $\nabla \{e\} \geq 0_{\{\text{Var}(e)\times 1\}}$ (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 (empty$_{in}$)), or the program $P$ does not modify any variables of $V$. This last condition is stated by rule (empty$_{var}$) where $\text{Assign}(P)$ 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 $S' \subseteq S \land \text{Conv}(S)$, 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 $f : \langle L_1, \leq L_1 \rangle \rightarrow \langle L_2, \leq L_2 \rangle$ and $g : \langle L_2, \leq L_2 \rangle \rightarrow \langle L_3, \leq L_3 \rangle$, gives as result a non-decreasing function $g \circ f : \langle L_1, \leq L_1 \rangle \rightarrow \langle L_3, \leq L_3 \rangle$. 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 $S = \{(x, y, z) \mid ((0 \leq x \leq 1) \land (0 \leq y \leq 1)) \lor ((1 \leq x \leq 2) \land (1 \leq y \leq 2))\}$ by deriving the judgment $\text{Mon}^\sim (P; S, \{x, y, z\})$. 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 $\hat{S} = \{(x, y, z) \mid (0 \leq x \leq 2) \land (0 \leq y \leq 2)\}$ 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 $x \geq 0 \land y \geq 0$. Since all the states in the convex set $\hat{S}$ satisfy also...
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 (iftrue) (resp. (iffalse)) of Fig. 3. By considering now the non-trivial cases (\( [b] S \neq S \land [b] S \neq \emptyset \)), 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

\[
P : \text{if } x + y - 2 \leq 0 \lor 2x + y - 3 \leq 0 \text{ then } y := x + 10 \text{ else } x := x^2 + 3
\]

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 \( \sigma_1 \) follows the then branch while \( \sigma_2 \) 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 \( \sigma_1 \leq_{\{x,y\}} \sigma_2 \) and \( [b] \sigma_1 = \sigma_1 \land [b] \sigma_2 = \emptyset \) (resp. \( [b] \sigma_1 = \emptyset \land [b] \sigma_2 = \sigma_2 \)) it must hold that \( [P_1] \sigma_1 \leq_{\{x,y\}} [P_2] \sigma_2 \) (resp. \( [P_2] \sigma_1 \leq_{\{x,y\}} [P_1] \sigma_2 \)). 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 \( e < 0 \) or they all have the form \( e > 0 \), and \( e = 0 \) is a **positive linear equation**, i.e., the expression \( e \) can be written as \( e = \sum_{i=1}^{\text{Var}(e)} v_i x_i + k \) with \( v \in \mathbb{Z}_{\geq 0}, k \in \mathbb{Z} \).
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 \( \leq \), 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 \( \text{Lin}_< \), \( \text{Lin}_> \) be the sets of all, respectively, positive linear guards having only predicates \( \leq \), positive linear guards having only predicates \( \geq \), and let \( \text{Lin}^* \triangleq \text{Lin}_< \cup \text{Lin}_> \). Then the following topological properties are satisfied for all \( b \in \text{Lin}^* \):

1. The guard \( b \) divides the Euclidean space \( \mathbb{R}^{|\text{Var}(P)|} \) in two regions: one region populated by states that satisfy \( b \) and the other region populated by states that satisfy \( \lnot b \);
2. If \( b \in \text{Lin}_< \) (resp. \( b \in \text{Lin}_> \)) then every state that makes \( b \) true is either less or equal (resp. greater or equal) or not comparable for variables \( \text{Var}(P) \) with respect to the states that do not satisfy \( b \), i.e., \( \forall \sigma_1, \sigma_2 : [b] \sigma_1 = \sigma_1 \land [b] \sigma_2 = \emptyset \) then either \( \sigma_1 \leq_{\text{Var}(P)} \sigma_2 \) (resp. \( \sigma_1 \geq_{\text{Var}(P)} \sigma_2 \)), or \( \sigma_1 \) and \( \sigma_2 \) are not comparable.

Example 3.7. The graphical representation of \( x + y - 2 \leq 0 \lor 2x + y - 3 \leq 0 \) is depicted in Fig. 4. By selecting any two states \( \sigma_1, \sigma_2 \) such that \( \sigma_1 \) is in the green area (i.e., satisfying the guard) while \( \sigma_2 \) 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 \( \sigma_2 \) is less than \( \sigma_1 \).

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 \( b \in \text{Lin}^* \) we formally define the set \( \mathcal{B}(b) \) of boundary states as follows:

\[
\mathcal{B}(b) \triangleq \begin{cases} 
\emptyset & \text{if } b = e \land b = e > 0, \\
\mathcal{B}(b_1) \lor \mathcal{B}(b_2) & \text{if } b = b_1 \lor b_2 \text{ or } b = b_1 \land b_2.
\end{cases}
\]

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 \( \mathcal{B}(x + y - 2 \leq 0 \lor 2x + y - 3 \leq 0) = x + y - 2 = 0 \lor 2x + y - 3 = 0 \), namely, the set of points \( \{(x, y) \mid x + y - 2 = 0 \lor 2x + y - 3 = 0\} \) 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 \( x + y - 2 = 0 \lor 2x + y - 3 = 0 \) (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 \( S \land (b \lor B(b)) \), 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 \( S \land (\neg b \lor B(b)) \). Finally, for the case \( b \in Lin^*_\uparrow \), 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

\[
\forall \sigma \in S \land B(b) : [P_1]\sigma \leq V [P_2]\sigma.
\]

This condition is encoded by the following predicate:

\[
\text{Boundary}(S \land B(b), V, P_1, P_2) \overset{\text{def}}{\iff} b \in Lin^*_\uparrow \land \forall \sigma \in S \land B(b). [P_1]\sigma \leq V [P_2]\sigma \\
\quad \lor \quad b \in Lin^*_\downarrow \land \forall \sigma \in S \land B(b). [P_1]\sigma \geq V [P_2]\sigma
\]

Note that the predicate \( \text{Boundary} \) treats also the case of \( b \in Lin^*_\downarrow \); in this scenario, the execution of \( P_2 \) must not exceed \( P_1 \) on all states \( \sigma \in S \land B(b) \). 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 \( \text{ReLU} : \) 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 \( x \leq 0 \in Lin^*_\uparrow \). The boundary state is \( B(x \leq 0) = x = 0 \), namely, the store \( \hat{\sigma}(x) = 0 \), thus \( true \land (x \leq 0 \lor x = 0) \) can be simplified into \( x \leq 0 \), and \( true \land (x > 0 \lor x = 0) \) into \( x \leq 0 \). By applying rule (assign) to the true-branch and false-branch we can derive, respectively, \( \text{Mon}^\leftarrow(x := 0, x \leq 0, \{x\}) \) and \( \text{Mon}^\leftarrow(x := x, x \geq 0, \{x\}) \). Moreover, the predicate \( \text{Boundary}(x = 0, \{x\}, x := 0, x := x) \) holds, indeed \([x := 0]\hat{\sigma} = \hat{\sigma} = [x := x]\hat{\sigma} \). Hence, we can apply rule (if) and derive \( \text{Mon}^\leftarrow(\text{if} \ x \leq 0 \ \text{then} \ x := 0 \ \text{else} \ x := x, true, \{x\}) \), proving that \( \text{ReLU} \) is \{\( x \}\)-non-decreasing. Intuitively, by proving that \( \text{Boundary}(x = 0, \{x\}, x := 0, x := x) \) holds, together with \( \text{Mon}^\leftarrow(x := 0, x \leq 0, \{x\}) \) and \( \text{Mon}^\leftarrow(x := x, x \geq 0, \{x\}) \), guarantee that it is never the case that \([\text{ReLU}]_1 \leq [\text{ReLU}]_2 \) 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 \( \text{Mon}^\leftarrow(x := 0, x \leq 0, \{x\}) \) and \( \text{Mon}^\leftarrow(x := x, x \geq 0, \{x\}) \) hold, we are sure that \([\text{ReLU}]_1 \leq [\text{ReLU}]\leq [\text{ReLU}]_2 \), therefore we can conclude \([\text{ReLU}]_1 \leq [\text{ReLU}]_2 \).

**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 \( \text{Boundary}(x \geq 0 \land B(b), \{x, y\}, y := x + 10, x := x^2 + 3) \), 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) \neq \{(x, y) | 0 \leq x \leq 10 \land y \geq 0 \} \) over \( \mathbb{R}^2 \). Let us construct the proof with our proof system. The condition \( S \land (b \lor B(b)) \) can be simplified into \( S \land b \) since \( b \) already includes the boundary states. For the true-branch, since \( y \notin \{x\} \), we apply (empty\_out) and derive \( \text{Mon}^\leftarrow(y := x + 10, S \land b, \{x\}) \). For the false-branch, as both the predicates \( \text{Cono}(S \land (\neg b \lor B(b))) \) and \( S \land (\neg b \lor B(b)) \) hold, we can derive \( \text{Mon}^\leftarrow(x := x^2 + 3, S \land (\neg b \lor B(b)), \{x\}) \) by applying (assign). The guard is positive linear, therefore the only remaining assumption to verify is the non-decreasing property on boundary states, namely: \( \text{Boundary}(S \land B(b), \{x\}, y := x + 10, x := x^2 + 3) \). This final predicate can also be checked with the assistance of an abstract interpreter \( [\cdot]_\text{int} \) on the abstract domain of intervals \( \text{Int} \) where the abstract sum \( +_\text{int} \) between two intervals is defined as \( [a, b] +_\text{int} [c, d] \overset{\text{def}}{=} [a + c, b + d] \) while the abstract multiplication \( \times_\text{int} \) is defined as \( [a, b] \times_\text{int} [c, d] \overset{\text{def}}{=} [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \). The set of states satisfying \( S \land B(b) \) can be overapproximated by the abstract states \( \{(0, 2), (0, 3)\} \in \text{Int}^2 \) which corresponds to all the program states \( \{\sigma \in \mathbb{R}^2 | \sigma(x) \in [0, 2] \land \sigma(y) \in [0, 3]\} \). Then by running both branches on the
\[
\begin{align*}
S \Rightarrow \neg b & \\
\text{Mon}^{-}((\text{while } b \text{ do } R, S, V)) & \quad (\text{while}_{\text{false}}) \quad \{I \land b\} R(I) \\
\text{Mon}^{-}((\text{while } b \text{ do } R, S, V)) & \quad \text{(while)} \quad \text{Mon}^{-}(R, I \land b, V) \\
\text{Limit}(b, R, I \lor S, V) & \\
\text{Mon}^{-}((\text{while } b \text{ do } R, S, V)) &
\end{align*}
\]

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

abstract interpreter, we get:

\[
\llbracket P_1 \rrbracket_{\text{int}}([0, 2], [0, 3]) = ([0, 2], [10, 12]) \leq_{\text{int}}^x ([3, 7], [0, 3]) = \llbracket P_2 \rrbracket_{\text{int}}([0, 2], [0, 3])
\]

where \( \leq_{\text{int}}^x \), as for the concrete stores, indicates that we are comparing the \( x \)-component only. Since the two intervals \([0, 2] \leq_{\text{int}} [3, 7]\) 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 \( \text{Boundary}(S \land B(b), \{x\}, y := x + 10, x := x^2 + 3) \) holds. We can conclude by rule (if): \( \text{Mon}^{-}(P, 0 \leq x \leq 10 \land y \geq 0, \{x\}) \) 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 \( \sigma \) may terminate earlier or later than the one starting from \( \sigma' \), resulting in a potentially different outcome.

**Example 3.11.** Let us consider, for example, the while-loop \( \text{while } x < 3 \text{ do } x := x + 2 \). 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:

\[
\llbracket \text{while } x < 3 \text{ do } x := x + 2 \rrbracket(1) = 4 \not\leq 3 = \llbracket \text{while } x < 3 \text{ do } x := x + 2 \rrbracket(1)
\]

Since each iteration of while \( b \text{ do } R \) can be viewed as an execution of if \( b \text{ then } R \text{ else } \text{skip} \), 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 \( B(b) \) on loops is too weak as it fails to establish the monotonicity even for simple loops like while \( x < 0 \text{ do } x := x + 1 \). Indeed, for this example, requiring \( \llbracket x = x + 1 \rrbracket(0) \leq \llbracket \text{skip} \rrbracket(0) \) 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 \( (\sigma_1, \sigma_2) \), which we refer to as limit states. Intuitively, in the case of \( b \in \text{Lin}^* \), the states \( (\sigma_1, \sigma_2) \) are limit states for the while-loop while \( b \text{ do } R \), when \( \sigma_1 \) enters the loop, \( \sigma_2 \) 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, \( b \in \text{Lin}^* \) and \( b \in \text{Lin}^* \):

\[
\mathcal{L}_\leq(b, P, S, V) \overset{\text{def}}{=} \{ (\sigma_1, \sigma_2) \mid \sigma_1 \in S \land b, \sigma_2 \in S \land \neg b, \sigma_1 \leq^V \sigma_2 \}
\]

\[
\mathcal{L}_\geq(b, P, S, V) \overset{\text{def}}{=} \{ (\sigma_1, \sigma_2) \mid \sigma_1 \in S \land \neg b, \sigma_2 \in S \land b, \sigma_1 \leq^V \sigma_2 \}
\]

Given a while-loop while \( b \text{ do } R \) such that \( b \in \text{Lin}^* \), a set of input states \( S \), set of variables \( V \), and a loop invariant \( I \), the set of pairs of states \( \mathcal{L}_\leq(b, R, I \lor S, V) \) identifies our intuition of limit states. In this case, when \( b \in \text{Lin}^* \), 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 \( I \land b \), and (2) for every pair of limit states \( (\sigma_1, \sigma_2) \in \mathcal{L}_\leq(b, R, I \lor S, V) \), the execution of \( R \) with input the state
\( \sigma_1 \) must not exceed \( \sigma_2 \), 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 \( \sigma \) until it exists the loop, will not break the order. The condition (2) and its dual case when \( b \in \text{Lin}_2^* \), are treated by the following predicate:

\[
\text{Limit}(b, P, S, V) \overset{\text{def}}{=} (b \in \text{Lin}_2^* \land \forall (\sigma_1, \sigma_2) \in \mathcal{L}_\prec(b, P, S, V). [P] \sigma_1 \leq V \sigma_2) \lor (b \in \text{Lin}_\prec^* \land \forall (\sigma_1, \sigma_2) \in \mathcal{L}_\succ(b, P, S, V). \sigma_1 \leq V [P] \sigma_2)
\]

This is the intuition underlying rule (\textbf{while}) in Fig. 5. The premise \( \{I \land b\} R \{I\} \) 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 (\textbf{while}_{\text{false}}) addresses the straightforward scenario where none of the states in \( S \) enter the loop. Practically, checking the validity of the predicate \( \text{Limit}(b, P, S, V) \), can be semi-automated by using modern automatic theorem provers or SMT solvers.

**Example 3.12.** Consider the program \( \text{ReLU}^w \overset{\text{def}}{=} \text{while } x < 0 \text{ do } x := x + 1 \text{ which implements the ReLU function on inputs by using a while-loop instead of an if-statement. Clearly, as } \text{ReLU}^w \text{ is semantically equivalent to the ReLU program of Example 3.9 on integer inputs, } \text{ReLU}^w \text{ is non-decreasing over } \mathbb{Z}. \text{ We want to prove it by exploiting rule (\textbf{while}) of our proof system. Let us consider the invariant } I : x \leq 0. \text{ By rule (\textbf{assign}), we first prove that the loop body is non-decreasing: } \text{Mon}^\prec(x := x + 1, x < 0, \{x\}). \text{ Since } x < 0 \in \text{Lin}_2^*, \text{ the set of all pair of limit states is } \mathcal{L}_\prec(x < 0, x := x + 1, \mathbb{Z}, \{x\}) = \{(m, n) \mid m, n \in \mathbb{Z} \land m < 0 \land n \geq 0\}. \text{ Then, clearly, for all } (m, n) \in \mathcal{L}_\prec(x < 0, x := x + 1, \mathbb{Z}, \{x\}), \llbracket x := x + 1 \rrbracket (m) \leq n \text{ holds, therefore the predicate } \text{Limit}(x < 0, x := x + 1, \mathbb{Z}, \{x\}) \text{ is true. All the premises are satisfied and rule (\textbf{while}) concludes } \text{Mon}^\prec(\text{ReLU}^w, \text{true}, \{x\}). \star\)

**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 \( f \):

\[
\text{Fact : } i := 1; f := 1; \text{while } i - x \leq 0 \text{ do } f := f \ast i; i := i + 1
\]

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 \( n! \overset{\text{def}}{=} n \ast (n - 1) \ast \cdots \ast 1 \) is the mathematical definition of factorial for \( n \in \mathbb{N} \). For the first two assignments before the loop, it is easy to derive \( \text{Mon}^\prec(i := 1; f := 1, \text{true}, \{x, i, f\}) \) by rules (\textbf{assign}) and (\textbf{seq}). Then, by rule (\textbf{weaken}), we soundly restrict the set of input states to the set \( I : \text{Mon}^\prec(i := 1; f := 1, I, \{x, i, f\}) \). Let us now analyze the while-loop. Note that \( I \) is a loop invariant, indeed it is easy to verify that the Hoare triple \( \{I \land (i \leq x)\} f := f \ast i; i := i + 1\{I\} \) 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 \( \text{Mon}^\prec(f := f \ast i; i := i + 1, \text{true}, \{x, i, f\}) \) by rules (\textbf{assign}) and (\textbf{seq}). Further, we use rule (\textbf{weaken}) to derive \( \text{Mon}^\prec(f := f \ast i; i := i + 1, I \land (i \leq x), \{x, i, f\}) \), as \( I \land (i \leq x) \Rightarrow \text{true} \) trivially holds. Since the guard \( i - x \leq 0 \in \text{Lin}_2^* \), it remains to check whether the predicate \( \text{Limit}(i - x \leq 0, f := f \ast i; i := i + 1, I, \{x, i, f\}) \) holds or not for the limit states. Given a pair of limit states \( (\sigma_1, \sigma_2) \in \mathcal{L}_\prec(i - x \leq 0, f := f \ast i; i := i + 1, I, \{x, i, f\}) \), we know that: \( \sigma_1 \leq \sigma_2, \sigma_1 \in \{(x, i, f) \mid I \land (i \leq x)\} \) and \( \sigma_2 \in I \land (i > x) \). Then, clearly \( [f := f \ast i; i := i + 1] \sigma_1 \leq \sigma_2 \) holds for all pair of limit states since a further execution of the loop body on \( \sigma_1 \) cannot exceed \( \sigma_2 \). Thanks to the validity of the premises, by rule (\textbf{while}) we can derive
Mon′(while \( i - x \leq 0 \) do \( f := f \ast i; \ i := i + 1, I, \{x, i, f\} \)). Finally, with rule (seq) we combine Mon′(\( i := 1; f := 1, I, \{x, i, f\} \)) with this last derivation, and conclude Mon′(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. \( \vdash \text{Mon′}(P, S, V) \Rightarrow \text{Mon′}(P, S, V) \).

Proof. (skip): Since \( \mathcal{D}_\text{skip} = \emptyset \), for any \( \sigma_1, \sigma_2 \in S \) such that \( \sigma_1 \leq V \sigma_2 \), we get \([\text{skip}]\sigma_1 = \sigma_1 \leq V \sigma_2 = [\text{skip}]\sigma_2 \), therefore Mon′(skip, S, V) holds for any \( S \subseteq \mathbb{P} \) and \( V \subseteq \text{Var} \).

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

(empty\(_\text{var}\)): 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 \( x \in \text{Var}(P) \cap V \) such that \( \sigma_1(x) \leq \sigma_2(x) \) we get \((\llbracket P \rrbracket \sigma_1)(x) = (\llbracket P \rrbracket \sigma_2)(x) \), i.e., Mon′(P, S, V) holds, while if \( V = \emptyset \) then there are no variables to check monotonicity, therefore the program is monotone for all \( S \subseteq \mathbb{P} \).

(assign): \( \mathcal{D}_\text{as} = \emptyset \llbracket \text{Var}(x:=e) \rrbracket \) since every assignment command is always terminating. Assume \( \text{Var}(e) \subseteq V \), Conv(S), \( S \Rightarrow \exists e \geq 0 | \text{Var}(e) \cap 1 \) and consider any \( \sigma_1, \sigma_2 \in S \) such that \( \sigma_1 \leq V \sigma_2 \). Then, we get the following implications:

\[
S \Rightarrow \exists e \geq 0 | \text{Var}(e) \cap 1 \Rightarrow [\sigma_1, \sigma_2 \in S]
\]

\[
\exists e \geq 0 | \text{Var}(e) \cap 1 \ \land \ [\exists e \geq 0 | \text{Var}(e) \cap 1] \Rightarrow [\exists \sigma_1 \leq V \sigma_2, \text{Var}(e) \subseteq V \text{ and Conv}(S)]
\]

\[
[\exists e \geq 0 | \text{Var}(e) \cap 1] \Rightarrow [\exists \sigma_1 \leq V \sigma_2] \Rightarrow [\exists \sigma_1 \leq V \sigma_2] \Rightarrow [\exists \sigma_1 \leq V \sigma_2] \Rightarrow [\exists \sigma_1 \leq V \sigma_2] \Rightarrow [\exists \sigma_1 \leq V \sigma_2] \Rightarrow [\exists \sigma_1 \leq V \sigma_2] \Rightarrow [\exists \sigma_1 \leq V \sigma_2]
\]

\[
\text{Mon′}(x := e, S, V)
\]

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

(seq): Let us assume Mon′(P\(_1\), S\(_1\), V\(_1\)), \( \{S\_1\} P\(_1\) \{S\_1\} \) and Mon′(P\(_2\), S\(_2\), V\(_2\)). Since \( \mathcal{D}_{P\_1, P\_2} \subseteq \mathcal{D}_{P\_1, P\_2} \), we know that \( (S\_1 \cap \mathcal{D}_{P\_1, P\_2}) \subseteq (S\_1 \cap \mathcal{D}_{P\_1, P\_2}) \). Therefore, for all \( \sigma_1, \sigma_2 \in S \cap \mathcal{D}_{P\_1, P\_2} \), we get the following implications:

\[
\sigma_1 \leq V \sigma_2 \Rightarrow [\text{Mon′}(P\(_1\), S\(_1\), V\(_1\))]
\]

\[
[P\(_1\)]\sigma_1 \leq V [P\(_1\)]\sigma_2 \Rightarrow [\{S\_1\} P\(_1\) \{S\_1\}] \text{ and Mon′}(P\(_2\), S\(_2\), V\(_2\))]
\]

\[
[P\(_2\)]\sigma_1 \leq V [P\(_2\)]\sigma_2 \Rightarrow [\text{Mon′}(P\(_2\), S\(_2\), V\(_2\))]
\]

\[
[P\(_1\); P\(_2\)]\sigma_1 \leq V [P\(_1\); P\(_2\)]\sigma_2 \Rightarrow [\text{Mon′}(P\(_1\), P\(_2\), S\(_1\), V\(_1\))]
\]

(if\(_\text{true}\)).(if\(_\text{false}\)): Since \( S \Rightarrow B \) implies that \([if \ b \ then \ P\(_1\) \ else \ P\(_2\)]S = [P\(_1\)] \), then by the assumption Mon′(P\(_1\), S, V), we can conclude Mon′(if \( b \ then \ P\(_1\) \ else P\(_2\), S, V). A similar reasoning can be used to prove (if\(_\text{false}\)).

(if): We assume the following: Mon′(P\(_1\), S \( \land \) \( ( b \lor B(b) ) \), V), Mon′(P\(_2\), S \( \land \) \( ( \neg b \lor B(b) ) \), V) and Boundary(S \( \land \) B(b), V, P\(_1\), P\(_2\)). Let us consider the non-trivial case where Var(if \( b \ then \ P\(_1\) \ 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\(_\text{var}\))). Suppose \( b \in \text{Lin}_\text{c} \) (the case \( b \in \text{Lin}_\text{c} \).
Lin’ is similar). Given \( \sigma_1, \sigma_2 \in S \cap D_{bf} \) 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 \( \text{Var}(\text{if} b \text{ then } P_1 \text{ else } P_2) \cap V \neq \emptyset \), \( b \in \text{Lin’}_s \) 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 \( \sigma_1, \sigma_2 \in S \cap D_{bf} \) such that \( \sigma_1 \leq V \sigma_2 \), we get \( \sigma_1, \sigma_2 \in S \cap (b \lor B(b)) \). Then by \( \text{Mon'}(P_1, S \cap (b \lor B(b)), V) \) and \( \sigma_1 \leq V \sigma_2 \), we obtain \([if \ b \ then \ P_1 \ else \ P_2]\sigma_1 = [P_1]\sigma_1 \leq V [P_1]\sigma_2 = [if \ b \ then \ P_1 \ else \ P_2]\sigma_2 \).

2. Case \([b]\{\sigma_1, \sigma_2\} = \emptyset \) from the assumption \( \sigma_1, \sigma_2 \in S \cap D_{bf} \) such that \( \sigma_1 \leq V \sigma_2 \), we get \( \sigma_1, \sigma_2 \in S \cap (-b \lor B(b)) \). Then by \( \text{Mon'}(P_2, S \cap (-b \lor B(b)), V) \) and \( \sigma_1 \leq V \sigma_2 \), we obtain \([if \ b \ then \ P_1 \ else \ P_2]\sigma_1 = [P_2]\sigma_1 \leq V [P_2]\sigma_2 = [if \ b \ then \ P_1 \ else \ P_2]\sigma_2 \).

3. Case \([b]\{\sigma_1, \sigma_2\} = \{\sigma_1\} \) by \( \text{Boundary}(S \cap B(b), V, P_1, P_2) \) that we know \( \forall \sigma \in S \cap B(b): [P_1]\sigma \leq V [P_2]\sigma \). Moreover, consider a boundary state \( \sigma_B \in S \cap B(b) \) between \( \sigma_1 \) and \( \sigma_2 \), namely such that \( \sigma_1 \leq V \sigma_B \leq V \sigma_2 \). Then:

\[ [if \ b \ then \ P_1 \ else \ P_2]\sigma_1 = [by \ [b]\{\sigma_1, \sigma_2\} = \{\sigma_1\}] [P_1]\sigma_1 \leq V [by \ \text{Boundary}(S \cap B(b), V, P_1, P_2)] [P_1]\sigma_B \leq V [by \ \text{Definition of B(b)}] [P_2]\sigma_B \leq V [by \ \text{Boundary}(S \cap B(b), V, P_1, P_2)] [P_2]\sigma_2 = [by \ [b]\{\sigma_1, \sigma_2\} = \{\sigma_1\}] [if \ b \ then \ P_1 \ else \ P_2]\sigma_2 \]

By the premises of rule (if) we end \([if \ b \ then \ P_1 \ else \ P_2]\sigma_1 \leq V \sigma_2 \), therefore \( \text{Mon'}(\text{if} \ b \text{ then } P_1 \text{ else } P_2, S, V) \) holds.

(proof): Assume \(\{I \land b\}R(I)\), namely \( I \) is a loop invariant, and both predicates \( \text{Mon'}(R, I \land b, V) \) and \( \text{Limit}(b, R, I \land S, V) \) hold. Furthermore, assume that \( b \in \text{Lin’}_s \) (the case \( b \in \text{Lin’}_s \) is similar).

Given two states \( \sigma_1, \sigma_2 \in S \cap D_{bf} \) then \( b \) such that \( \sigma_1 \leq V \sigma_2 \), there are three cases to consider:

1. Case \([b]\{\sigma_1, \sigma_2\} = \emptyset \) \( [while \ b \ do \ R] \sigma_1 \leq V \sigma_2 \) \( [while \ b \ do \ R] \sigma_2 \);

2. Case \([b]\{\sigma_1, \sigma_2\} = \{\sigma_1\} \) because \( \sigma_1 \in D_{while \ b \ do \ R} \), this implies that \( E \exists n > 0 \) 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 \( 1 \leq n \leq k \), \( \sigma_1 \leq V \sigma_2 \) holds. Base case \( i = 1 \): \([R]\sigma_1 \leq V \sigma_2 \) is true since \( \sigma_2 \in I \land S \), \( (\sigma_1, \sigma_2) \in L_{bf}(b, I \land S, V) \) and \( \text{Limit}(b, R, I \land S, V) \) 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 \([b][R^{n-1}]\sigma_1 \neq \emptyset \), \([b]_{bf} \sigma_1 = \emptyset \) and by the inductive hypothesis \([R^{n-1}]\sigma_1 \leq V \sigma_2 \), we get that these are limit states, i.e., \( [R^{n-1}]\sigma_1, \sigma_2 \in L_{bf}(b, R, I \land S, V) \). By \( \text{Limit}(b, R, I \land S, V) \), 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 \( [while \ b \ do \ R] \sigma_1 \leq V \sigma_2 \);
base case shifted to \( i = n_2 \) (which trivially holds by (1)). Finally, from \([\bar{b}] [R^{n_1}] \sigma_1 = \emptyset\) and \([b] [\hat{\sigma}] = \emptyset\), we conclude \([\text{while } b \text{ do } R] \sigma_1 = [R^{n_1}] \sigma_1 \leq^V \hat{\sigma} = [R^{n_1}] \sigma_2 = [\text{while } b \text{ do } R] \sigma_2\).

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

\[\square\]

### 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 \( P \in \text{Prog} \) and input \( S \subseteq \mathcal{I}^n \) 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 \( V \subseteq \text{Var} \) 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 \( \mathcal{A} \in \text{Abs}(\wp(\mathcal{I}^n)) \), a program \( P \in \text{Prog} \), an input \( S \subseteq \mathcal{I}^n \) and a set of variables \( V \subseteq \text{Var} \). We say that the abstract interpretation \([P]_\mathcal{A}\) of program \( P \) is \( V\)-complete at \( S \) whenever the following condition holds for all \( x \in V \cap \text{Var}(P)\):

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

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

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 \( \text{Var}(P)\)-completeness, if \([P]_\mathcal{A}\) is complete then it is also \( V\)-complete for all \( V \subseteq \text{Var} \), while if \([P]_\mathcal{A}\) is \( V\)-complete for \( V \subseteq \text{Var}(P) \) then it may be not complete.

**Example 4.2.** Consider the abstract domain \( \text{Sign} \overset{\text{def}}{=} \{ \mathbb{Z}, - , 0 , + , \emptyset \} \) for integer sign analysis presented in Example 2.2. Suppose that the abstract sum operation \( \oplus \) on \([\cdot]_{\text{Sign}}\) is soundly implemented as follows: \( + \oplus + = + \); if \( \star \in \{ \mathbb{Z}, - , 0 , + \} \) then \( \star \oplus 0 = \star, \emptyset \oplus 0 = 0, + \oplus - = \mathbb{Z} \), and if \( \star \in \{ \mathbb{Z}, - , 0 , + , \emptyset \} \) 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, \([P]_{\text{Sign}}\) is \( \{y\}\)-complete but not complete, i.e., not \{x, y\}-complete. Indeed, for all \( S \in \wp(\mathbb{Z}^2) \) we have \( \alpha_{\text{Sign}} ([P]_S) = (0, +) \triangleleft_{\text{Sign}} (\mathbb{Z}, +) = ([P]_{\text{Sign}} \alpha_{\text{Sign}} (S)) (y) = + = ([P]_{\text{Sign}} \alpha_{\text{Sign}} (S)) (y) \).

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 \( P \in \text{Prog} \), a set \( V \subseteq \text{Var} \) of variables and a set of inputs \( S \subseteq \mathcal{I}^n \), we define

\[
\min^V S \overset{\text{def}}{=} \{ \sigma \in S \mid \forall x \in V \cap \text{Var}(P), \sigma(x) = \min (\{ \rho(x) \mid \rho \in S \}) \}
\]
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 \( \min^V S \neq \emptyset \), 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 \( \text{Var}(P) = V = \{ x , y \}, S = \{ (0, 1), (3, 4), (1, 2) \}, S' = \{ (1, 3), (2, 0), (4, 4) \} \), then \( \min^V S = \{ (0, 1) \} \) while \( \min^V S' = \emptyset \). 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) \([P]^{min} S \neq \emptyset\), and (ii) \([P]^{max} S \neq \emptyset\).

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 \( V \)-boundedness 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 \( \mathcal{A} \in \text{Abs}(\varphi(I^n)) \) is Int-abstractable whenever \( \mathcal{A} \leq_{\text{Abs}(\varphi(I))} \text{Int} \), i.e., \( \mathcal{A} \) can exactly represent intervals.

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

Definition 4.5 (V-Convexity). An abstract domain \( \mathcal{A} \in \text{Abs}(\varphi(I^n)) \) is \( V \)-convex at \( S \) for a program \( P \in \text{Prog} \), set of variables \( V \subseteq \text{Var} \) and input set \( S \in \varphi(I^n) \), if and only if for all \( x \in V \cap \text{Var}(P) \), \( (\gamma_A(\alpha_A([P]S)))(x) \in \varphi(I) \) 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 \( (\gamma_A(\alpha_A([P]S)))(x) \), 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 \( \varphi(I) \), e.g. Int and Sign, are \( V \)-convex for all \( P \in \text{Prog}, V \subseteq \text{Var} \) and \( S \subseteq I^n \). 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 \( \text{Congr} \in \text{Abs}(\varphi(Z)) \) [Granger 1989] defined as \( \text{Congr} \overset{def}{=} \{aZ + b \mid a \in \mathbb{N}, b \in \mathbb{Z} \} \cup \{-\text{Congr}\} \), contains abstract elements having the form \( aZ + b \) such that \( \gamma_{\text{Congr}}(aZ + b) \overset{def}{=} \{ak + b \mid k \in \mathbb{Z} \} \) and \( \alpha_{\text{Congr}}(S) \overset{def}{=} \backslash_{\text{Congr}}(0Z + c) \). The parity domain Parity is a special case of this domain where \( a = 2 \). Congr contains convex properties, e.g. \( \gamma_{\text{Congr}}(1Z + 0) = \mathbb{Z} \) or singletons \( \gamma_{\text{Congr}}(0Z + b) = \{b\} \), and non-convex properties. For example, Congr is \( \{x\} \)-convex for the assignment \( x \leftarrow 1 \) regardless of the input \( S \subseteq \mathbb{Z} \) since \( \gamma_{\text{Congr}}(\alpha_{\text{Congr}}([x := 1]S))(x) = \{1\} \) 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 \( \gamma_{\text{Congr}}(\alpha_{\text{Congr}}([x := x + 2 + 1]2, 3)) = \gamma_{\text{Congr}}(2Z + 1) = \{1, 3, 5, \ldots \} \) 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 \( P \in \text{Prog} \) is said to be \( V \)-complete-analyzable for the non-relational abstraction \( \mathcal{A} \in \text{Abs}(\varphi(I^n)) \), variables in \( V \subseteq \text{Var} \) and input \( S \subseteq I^n \) 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 \( \text{Comple}^{\mathcal{A}}(P, S, V) \) 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 \([P]_{\text{Sign}}\) and \([\text{ReLU}]_{\text{Int}}\) are, respectively, \( \{y\} \)- and \( \{x\} \)-complete for all inputs. The following result is a straightforward consequence of Theorem 2.6.

Lemma 4.8. \( \text{Comple}^{\mathcal{A}}(P, S, V) \iff [P]_\mathcal{A} \text{ is } V \text{-complete at } S. \)
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 $\llbracket P \rrbracket^\mathcal{A}_\mathcal{A}$ 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 $P \in \text{Prog}$. If one of the following holds:

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

(ii) $V \cap \text{Var}(P) = \emptyset$;

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

then $\text{Compl}^\mathcal{A}(P, S, V)$ 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 $[\ ]$.

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

(iii) Suppose $S = \gamma\mathcal{A}(\alpha\mathcal{A}(S))$. Then, for every $\mathcal{A}$ and $P$, $\alpha\mathcal{A}(\llbracket P \rrbracket S) = \alpha\mathcal{A}(\llbracket P \rrbracket S_{\alpha\mathcal{A}(\mathcal{A}(S))}) = \llbracket P \rrbracket^\mathcal{A}_\mathcal{A}\alpha\mathcal{A}(S)$, i.e., the BCA $\llbracket P \rrbracket^\mathcal{A}_\mathcal{A}$ is complete at $S$ and therefore $V$-complete for any $V \subseteq \text{Var}$. By Lemma 4.8, this implies that $P$ is $V$-complete-analyzable. $\square$

We are interested in studying the non-trivial cases, namely, when the abstract domain $\mathcal{A}$ differs from the concrete domain $\varphi(I^n)$, 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 $P \in \text{Prog}$, $V \subseteq \text{Var}$, $\mathcal{A} \in \text{Abs}(\varphi(I^n))$ non-relational, and assume the following:

(1) $S$ is $V$-bounded,

(2) $\mathcal{A}$ is $\text{Int}$-abstractable, and

(3) $\mathcal{A}$ is $V$-convex at $S$ for $P$.

Then, the following implication holds:

$$\text{Mon}(P, \gamma\mathcal{A}(\alpha\mathcal{A}(S)), V) \Rightarrow \text{Compl}^\mathcal{A}(P, S, V)$$

**Proof.** We prove the implication by showing that, by the four assumptions (1),(2),(3) and monotonicity, the BCA $\llbracket P \rrbracket^\mathcal{A}_\mathcal{A}$ 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 $\mathcal{A} \neq \varphi(I^n)$, $V \cap \text{Var}(P) \neq \emptyset$ and $S \subset \gamma\mathcal{A}(\alpha\mathcal{A}(S))$ hold. Let $\tilde{V} = V \cap \text{Var}(P)$. 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 $\llbracket P \rrbracket \min^{\tilde{V}} S$, $\llbracket P \rrbracket \max^{\tilde{V}} S \neq \emptyset$;

2) $\mathcal{A}$ can exactly represents intervals;

3) for all $x \in \tilde{V}$, the set $(\gamma\mathcal{A}(\alpha\mathcal{A}(S)))(x)$ 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) $\forall \sigma_1, \sigma_2 \in \gamma\mathcal{A}(\alpha\mathcal{A}(S)) \cap \mathcal{D}_\mathcal{P}$. $(\sigma_1(x) \leq \sigma_2(x)) \Rightarrow (\llbracket P \rrbracket \sigma_1(x) \leq (\llbracket P \rrbracket \sigma_2(x))$ or

ii) $\forall \sigma_1, \sigma_2 \in \gamma\mathcal{A}(\alpha\mathcal{A}(S)) \cap \mathcal{D}_\mathcal{P}$. $(\sigma_1(x) \leq \sigma_2(x)) \Rightarrow (\llbracket P \rrbracket \sigma_1(x) \geq (\llbracket P \rrbracket \sigma_2(x)$;

5) $\text{Compl}^\mathcal{A}(P, S, V)$ does not hold, namely, $\exists x \in \tilde{V}$. $\gamma\mathcal{A}(\llbracket P \rrbracket S) < \llbracket x \rrbracket^\mathcal{A}_\mathcal{A}\alpha\mathcal{A}(S)$.

By (2), we know that all the spurious elements added by $\mathcal{A}$ are contained in the Int abstraction, namely, for all $I \in \varphi(I^n)$, $\gamma\mathcal{A}(\alpha\mathcal{A}(I)) \subseteq \gamma\text{Int}(\alpha\text{Int}(I))$. 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, $\min^{\tilde{V}} S \gamma\mathcal{A}(\alpha\mathcal{A}(S))$ and $\max^{\tilde{V}} S \gamma\mathcal{A}(\alpha\mathcal{A}(S))$. 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:

\[ \min \bar{\nu} S = \bar{\nu} \min \bar{\nu} \forall A(\alpha_A(S)) \Rightarrow \]
\[ [P] \min \bar{\nu} S = \bar{\nu} [P] \min \bar{\nu} \forall A(\alpha_A(S)) \Rightarrow \text{[by (4)]} \]
\[ (a) \min \bar{\nu} [P] S = \bar{\nu} \min \bar{\nu} [P] \forall A(\alpha_A(S)) \leq \bar{\nu} \text{[by (1)]} \]
\[ \max \bar{\nu} S = \bar{\nu} \max \bar{\nu} \forall A(\alpha_A(S)) \Rightarrow \]
\[ [P] \max \bar{\nu} S = \bar{\nu} [P] \max \bar{\nu} \forall A(\alpha_A(S)) \Rightarrow \text{[by (4)]} \]
\[ \max \bar{\nu} [P] S = \bar{\nu} \max \bar{\nu} [P] \forall A(\alpha_A(S)) \Rightarrow \text{[by (a),(2)]} \]
\[ \forall x \in \bar{\nu}. ((\min^{(x)}[P] S)(x), (\max^{(x)}[P] S)(x)) = \]
\[ ((\min^{(x)}[P] \forall A(\alpha_A(S)))(x), (\max^{(x)}[P] \forall A(\alpha_A(S)))(x)) \Rightarrow \text{[by (3)]} \]
\[ \forall x \in \bar{\nu}, \forall A(\alpha_A([P] S)) =^{(x)} \forall A([P] \forall A(S)), \text{i.e., Compl}_A(P, S, V) \]
\[ \text{holds, contradicting (5).} \]

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-monotonocity of program P over the inputs \( \forall A(\alpha_A(S)) \) is a sufficient condition to ensure the V-complete-analyzability of program P at S over the abstract domain \( A \). This result reveals a connection between the extensional property of V-monotonocity in programs and the completeness property of an abstract interpreter \( [\cdot]_A \): 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 \( [\cdot]_A \) 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 implement three well known activation functions:

\[ \text{Signum} \overset{\text{def}}{=} \begin{cases} \text{if } x < 0 & \text{then } x := -1 \\ \text{else if } x > 0 & \text{then } x := 1 \\ \text{else } x := 0 \end{cases} \]
\[ \text{ReLU} \overset{\text{def}}{=} \begin{cases} \text{if } x \leq 0 & \text{then } x := 0 \\ \text{else } x := x \end{cases} \]
\[ \text{SiL} \overset{\text{def}}{=} \frac{x}{1 + e^{-x}} \]

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

\[ \text{We assume that SiL is an implementation in Prog using only variable } x. \]
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 $[\text{Signum}]^\alpha_{\text{Int}}$ and $[\text{ReLU}]^\alpha_{\text{Int}}$ 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 $\alpha_{\text{Int}}([\text{SiL} \{\{-4, 0\}\}) \approx [-0.07, 0]$, while the BCA on Int outputs $[\text{SiL}]^\alpha_{\text{Int}}(\{-4, 0\}) \approx [-0.28, 0]$. Note that $[\text{SiL}]^\alpha_{\text{Int}}$ leads to a false-alarm since $[\text{SiL}]^\alpha_{\text{Int}}(\{-4, 0\}) \not\subseteq \text{Int} [-0.1, 0]$ even if the specification is satisfied: $\alpha_{\text{Int}}([\text{SiL} \{-4, 0\}\}) \subseteq \text{Int} [-0.1, 0]$. This result excludes the possibility to define a complete analysis $[\text{SiL}]^\alpha_{\text{Int}}$ over $\{ -4, 0 \}$. Nevertheless, if we consider an input set where SiL is monotone, e.g. $\{0, 4\}$, we have completeness: $\alpha_{\text{Int}}([\text{SiL} \{0, 4\}\}) \approx [0, 3.93] \approx [\text{SiL}]^\alpha_{\text{Int}}(\{0, 4\})$, 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 $P : \text{if } x \neq 3 \text{ then } x := x \text{ else } x := x - 2$. 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 Int $\in \text{Abs}(\varphi(\mathbb{Z}))$ abstraction. Clearly, Int and $\{1, 5\}$ satisfy the assumptions of Theorem 4.10. However, $\alpha_{\text{Int}}([P]_{\text{Int}}(\{1, 5\}) = [1, 5] = [P]_{\text{Int}}^\alpha_{\text{Int}}(\{1, 5\})$, namely $[P]_{\text{Int}}^\alpha_{\text{Int}}$ 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 \(\mathcal{A} \not\leq_{\text{Abs}(\mathcal{P}(\nu))} \text{Int}\), \(\mathcal{A}\) could add new maximum and minimum values for a variable \(x \in V\) at \(y_{\mathcal{A}}(\alpha_{\mathcal{A}}(S))\) 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:

\[
P : \begin{cases} \text{if } x + y - 8 \geq 0 \text{ then } x := 1; \ y := 1 \ \\ \text{else } x := -1; \ y := -1 \end{cases}
\]

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

\[
\left[\begin{array}{ll}
\alpha_{\text{Int}}(\left[\begin{array}{l}
P\end{array}\right]\{S\}) & = \alpha_{\text{Int}}(\{(1, 1)\}) = ([1, 1], [1, 1]) \\
\left[\begin{array}{l}
P\end{array}\right]_{\text{Int}}^{\alpha} \alpha_{\text{Int}}(S) & = \alpha_{\text{Int}}(\left[\begin{array}{l}
P\end{array}\right]\{(4, 4), (3, 5), (3, 4), (4, 5)\}) \\
& = \alpha_{\text{Int}}(\{(1, 1), (-1, -1)\}) = ([1, 1], [-1, 1]).
\end{array}\right.
\]

By \((1, 1), [1, 1]) <_{\text{Int}} ((-1, 1), [-1, 1]),\) we can deduce the \((x, y)\)-incompleteness of \(\left[\begin{array}{l}
P\end{array}\right]_{\mathcal{A}}\) at \(\{(4, 4), (3, 5)\}\). In this example, the minimal point \((3, 4)\) added by \(\text{Int}\) caused the incompleteness.

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

\[
P' : \begin{cases} \text{if } x + y - 7 \geq 0 \text{ then } x := 1; \ y := 1 \ \\ \text{else } x := -1; \ y := -1 \end{cases}
\]

We analyze this new program over the \(\text{Sign} \in \text{Abs}(\mathbb{Z})\) abstraction with input \(S' = \{(3, 4), (4, 5)\}\). Clearly, \(\text{Sign}\) is not Int-abstractable as for instance \(y_{\text{sign}}(\alpha_{\text{sign}}(\{1, 4\})) = \{0, 1, 2, 3, 4, 5, \ldots\} \not\subseteq \{1, 2, 3, 4\}\). Here, even if \(S'\) \((x, y)\)-bounded, \(\text{Sign}\) 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:

\[
\left[\begin{array}{l}
\alpha_{\text{sign}}(\left[\begin{array}{l}
P'
\end{array}\right]\{S'\}) & = \alpha_{\text{sign}}(\{(1, 1)\}) = (+, +) \\
\left[\begin{array}{l}
P'
\end{array}\right]_{\text{Sign}}^{\alpha} \alpha_{\text{sign}}(S') & = \alpha_{\text{sign}}(\left[\begin{array}{l}
P'
\end{array}\right]\{(Z, \geq 0), (Z, \geq 0)\}) = (Z, Z)
\end{array}\right.
\]

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 \(P \sqcap 1 \in \text{Abs}(\phi(\mathbb{Z}))\) which is the reduced product of Parity and Int abstract domains. \(P \sqcap 1\) represents properties which are the intersection between intervals and even or odd numbers. For example, \([0, 10]_{\text{even}}\) 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
properties, such as \([0, 10]_{\text{even}}\). However, \(P \cap 1\) is \(\text{Int}\)-abstractable because it can represent exactly all the intervals. Let us consider the following program:

\[
Q : \text{ if } x \mod 2 = 0 \lor x = 3 \text{ then skip else } x := x + 1
\]

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

\[
\alpha_{\text{Pr}}(\llbracket Q \rrbracket(\{2, 5\})) = \alpha_{\text{Pr}}(\{2, 6\}) = [2, 6]_{\text{even}}\\
\llbracket Q \rrbracket_{\text{Pr}} \alpha_{\text{Pr}}(\{2, 5\}) = \alpha_{\text{Pr}}(\{2, 3, 4, 6\}) = [2, 6]
\]

we can conclude that \(P \cap 1\) is not \(\{x\}\)-convex at \(\{2, 5\}\), since \([2, 6]_{\text{even}}\) has many holes, namely, all the odd numbers in \([2, 6]\). \(Q\) is monotone over \(\gamma_{\text{Pr}}(\alpha_{\text{Pr}}(\{2, 5\}))\) but \(\text{Compl}_{\text{Pr}}(Q, \{2, 5\}, \{x\})\) does not hold as \([P]_{\text{Pr}}^\alpha\) is not complete at \(\{2, 5\}: [2, 6]_{\text{even}} \prec_{\text{Pr}} [2, 6]\).

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 \(\text{Zone}, \text{Oct} \in \text{Abs}(\varphi(\mathbb{Z}^n))\) which are able to express constraints with two variables: \(x - y \leq k\) for \(\text{Zone}\), while \(\pm x \pm y \leq k\) for \(\text{Oct}\) 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 \(\text{Var}(P) = \{x, y\}\). Firstly, note that the 2-dimensional abstraction shape of a relational \(\text{Int}\)-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 \(\text{Int}\), and, by assuming monotonicity, \(\gamma_\mathcal{A}(\alpha_\mathcal{A}([P]_\mathcal{S}))\) and \(\gamma_\mathcal{A}(\alpha_\mathcal{A}(\alpha_\mathcal{A}(S)))\) 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 \(\text{Zone}\in\text{Abs}(\varphi(\mathbb{Z}^2))\) 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 $\gamma_{\text{Zone}}(\alpha_{\text{Zone}}(S))$ 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 $\gamma_{\text{Zone}}(\alpha_{\text{Zone}}([P]S))$. 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 $\gamma_{\text{Zone}}(\alpha_{\text{Zone}}(S))$ since $[P]b \leq [P]e \leq [P]c$. However, the shape outlined by $\gamma_{\text{Zone}}(\alpha_{\text{Zone}}([P]S))$ 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 $\gamma_{\text{Zone}}(\alpha_{\text{Zone}}(S))$, $[P]_{\text{Zone}}^\alpha$ 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 [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. [2022]) 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. 2022; 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 monotonicity. 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 [Ascarì 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 $b \in Lin^*_S$ 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 $\forall \sigma \in S. \exists \sigma' \in S. \sigma < \sigma' \lor \sigma > \sigma'$), 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].

Proc. ACM Program. Lang., Vol. 8, No. POPL, Article 55. Publication date: January 2024.
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


Monotonicity and the Precision of Program Analysis

55:31


The formal semantics of programming languages - an introduction


Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal Security Analysis of Neural


Commun. ACM


Caterina Urban and Antoine Miné. 2014b. A Decision Tree Abstract Domain for Proving Conditional Termination. In Programming

Introduction to real analysis


Proc. ACM Program. Lang., Vol. 8, No. POPL, Article 55. Publication date: January 2024.
Received 2023-07-11; accepted 2023-11-07