Deciding program properties via complete abstractions on bounded domains [⋆]

Roberto Bruni[1][0000−0002−7771−4154], Roberta Gori[1][0000−0002−7424−9576], and Nicolas Manini[123][0000−0002−7561−3763]

[1] Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, Pisa, Italy, {roberto.bruni,roberta.gori}@unipi.it.
[2] IMDEA Software Institute, Madrid, Spain, nicolas.manini@imdea.org.
[3] Universidad Politécnica de Madrid, Madrid, Spain

Abstract. Abstract interpretation provides an over-approximation of program behaviours that is used to prove the absence of bugs. When the computed approximation in the chosen abstract domain is as precise as possible, we say the analysis is complete and false alarms cannot arise. Unfortunately for any non trivial abstract domain there is some program whose analysis is incomplete. In this paper we want to characterize the classes of complete programs on some non-trivial abstract domains for studying their expressiveness. To this aim we introduce the notion of *bounded domains* for posets with ascending chains of bounded length only. We show that any complete program on bounded domains can be rewritten in an equivalent canonical form without nontrivial loops. This result proves that program termination on the class of complete programs on bounded domain is decidable. Moreover, semantic equivalence between programs in the above class can be reduced to determining the equivalence of a set of guarded statements. We show how our approach can be applied to a quite large class of programs. Indeed, abstract domains defined on Boolean abstractions that are complete for the same functions can be composed by preserving boundedness and completeness also w.r.t. any expressible guard. This suggests that new complete bounded abstract domains can be tailored on the guards and functions appearing in the program. Their existence is sufficient to prove decidability of termination and program equivalence for such programs.

Keywords: Abstract Interpretation, Complete abstraction, Bounded domains, Program transformation, Termination, Program equivalence

1 Introduction

The current spread of software-driven computing devices and the fact that our daily activities and lives are dependent on them makes program verification

[⋆] Research supported by MIUR PRIN Project 201784YSZ5 ASPRA–Analysis of Program Analyses. Partially funded by RYC-2016-20281, by ESF Investing in your future and by Madrid regional government as part of the program S2018/TCS-4339 co-funded by EIE Funds of the European Union.

extremely important to prevent crashes that may involve millions of users (see, for example, [12,2,18,23,10]). Formal methods and static analysis techniques [19,21] are a useful tool to verify program properties before deployment and to gain confidence on programs behaviour without running the actual code. Unfortunately the founding fathers of Computer Science had well established the limits of such approaches, by showing that all interesting problems about Turing equivalent programming languages are undecidable, like program termination and extensional equivalence [20,22], so that the pretension of devising universal analysis procedures that works fine for any program is deemed to fail. Intensional analysis is more subtle, because it takes into account *how a program is written* and not just *what a program computes.*

Abstract Interpretation [7,16,21,5] is an intensional, sound-by-construction static analysis method whose precision depends very much on the way in which the program is coded. The basic idea of Abstract Interpretation is to execute the program over an abstract domain that over-approximate the concrete program semantics. In this sense, each set of concrete stores is approximated by its least superset available in the abstract domain. For example, if one is interested in sign analysis, the abstract domain can be the finite set $\{\varnothing, \mathbb{Z}_{<0}, \mathbb{Z}_{\geqslant 0}, \mathbb{Z}\}$ such that the empty set is approximated by $\varnothing$, any set of negative values, like $\{-4, -2\}$, is approximated by $\mathbb{Z}_{<0}$, any set of non-negative values by $\mathbb{Z}_{\geqslant 0}$ and any other (non-empty) set by $\mathbb{Z}$. The symbolic execution of the program on the abstract domain is performed by a so-called abstract interpreter that may loose precision because it operates on abstract elements only. The abstract interpreter is sound-by-construction in the sense that it is guaranteed to return an over-approximation of the concrete result. Completeness of the abstract interpreter would ensure that the abstract result is the least representative available in the abstract domain of the concrete result, that is the abstract result is as much precise as possible. Recent work has shown that only trivial abstract domains can be complete for all programs of a Turing equivalent language [3,14]. However, if the abstract analysis is complete for all primitives appearing in a program then we can conclude that, for that particular program, the analysis is also complete-by-construction. Consequently, if we consider the sublanguage consisting of all programs composed by complete primitives, then Abstract Interpretation gives us an analysis framework that is sound-and-complete by construction.

Contribution. In this paper, we investigate the connections between completeness in Abstract Interpretation and decidability of program termination and (extensional) equivalence. The idea is to fix some constraints over the abstract domain that guarantees the decidability of relevant properties for any program for which the analysis is complete-by-construction. The notion we put forward is that of *bounded abstract domain* (see Definition 13), where the termination of the abstract interpreter is always guaranteed. Note that, in the general case, termination of the abstract interpreter does not imply termination of the concrete program.

As a first result we show that for programs that are complete-by-construction on a bounded abstract domain termination is decidable. This is obtained by showing that each such program can be rewritten in an equivalent form by unrolling each loop a finite number of times, possibly ending up in a trivial loop. Since the equivalent form can only contain trivial loops, which are immediate to detect, it follows that program termination is decidable.

As a second main result, we show a convenient way of attacking program equivalence for programs that are complete-by-construction on a bounded abstract domain. This can be done by unrolling each program as specified above and by then rewriting the code in a so-called *reduced select normal form* (see Definition 32) that is essentially a series of nested if-then-else structures whose basic commands are assignments and trivial loops. Finally, we give a procedure to decide whether two programs in reduced select normal form are equivalent or not by reducing the problem to the validity of a set of guarded statements defined using the primitives appearing in the programs only.

To support the applicability of our approach we prove how abstract domains defined on Boolean abstractions complete for the same functions can be composed to obtain new bounded domains complete for the same functions and for any guard expressible in one of the original domain. By composing different domains each one complete for a different guard of the program we may end up in designing a new abstract domain complete for *any* guard appearing in the program.

Structure of the paper: In Section 2 we introduce the notation and recall the basic concepts of Abstract Interpretation. The notion of bounded abstract domain is introduced in Section 3, together with some results on their composition. In Section 4 we prove that any program whose analysis is complete on a bounded domain can be transformed in an equivalent one for which termination is decidable. We conclude Section 4 by discussing the applicability of the approach when Boolean abstractions are used. Section 5 shows that equivalence between complete programs on abstract bounded domain is decidable. Finally, Section 6 draws some conclusions and discusses future work. All technical proofs are collected in the Appendix for reviewers' convenience.

2 Background

2.1 Notation

We let $\mathbb{N}$ be the set of natural numbers, $\mathbb{Z}$ the set of integers and $\mathbb{B}$ the set of Booleans and write $X \cup Y$ for the union of X and Y, $X \cap Y$ for their intersection, $X \backslash Y$ for their difference, $X \times Y$ for their cartesian product, and X^n for the cartesian product of X with itself n times. The powerset of X is denoted by $\mathcal{P}(X)$. Set inclusion is denoted as $X \subseteq Y$ and strict inclusion as $X \subset Y$.

The identity function over a set X is written $id_X : X \to X$ and we omit the subscript when it is clear from the context. The composition of two functions $f : X \to Y$ and $g : Y \to Z$ is denoted by $g \circ f : X \to Z$ or more concisely by gf. We also define the iterated application of a function $f : X \to X$ as $f^0 \overset{\text{def}}{=} id_X$ and

$f^n \overset{\text{def}}{=} f \circ f^{n-1}$. Abusing the notation, we extend function application to denote its lifting to sets of elements $f(X) \overset{\text{def}}{=} \{f(x) \mid x \in X\}$. Tuples will be denoted by $\tilde{x} = \langle x_1, \ldots, x_n \rangle \in X^n$, however, by overloading the notation $\tilde{x}$ will also denote the set $\{x_1, \ldots, x_n\}$ when no ambiguity arises, moreover, let $\tilde{x}' \in X^m$ we denote as $\tilde{x} +\!\!+ \tilde{x}'$ the concatenation $\langle x_1, \ldots, x_n, x'_1, \ldots, x'_m \rangle$. $\tilde{X} \cap Y$ indicates the tuple $\langle X_1 \cap Y, \ldots, X_n \cap Y \rangle$ when each X_i is a set itself.

We formally define a partitioning of a set U where each partition does not need to be nonempty.

Definition 1 (Partitioning). *Given $n \in \mathbb{N}$, we say that $P = \{P_1, \ldots, P_n\}$ is a partitioning of a set U iff $U = \bigcup_{i=1}^{n} P_i$ and $P_i \cap P_j = \varnothing$ for $i \neq j$*

We will refer to complete lattices as $\mathcal{C} = \langle C, \leq_C, \vee_C, \wedge_C, \top_C, \bot_C \rangle$ where $\vee_C, \wedge_C$ are the lub and glb respectively and $\top_C, \bot_C$ are the top and bottom elements. When clear from the context the subscripts will be omitted. We define an order on functions $f, g : C \to D$ between lattices, denoted by $f \preceq g$, iff for all $c \in C$ it holds that $f(c) \leq_D g(c)$.

We say that a function f between posets is monotone if it is order preserving. The function f is called additive if it is lub preserving and co-additive if it preserves glbs. Moreover, we say that a mapping $f : X \to X$ on a poset is extensive (or reductive) iff for all x it holds that $x \leq f(x)$ (resp. $f(x) \leq x$). We also denote with $\text{lfp}\,(f)$ the least fixpoint of f (w.r.t. $\leq$) when it exists.

2.2 Abstract Interpretation

Abstract interpretation [7] is based on the notion of Galois connections/insertions. We recall the basic concepts here, but see [5,7,9,6] for further details.

Given two complete lattices $\mathcal{C}$ and $\mathcal{A}$, a pair of functions $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{C}$ forms a Galois connection (GC) iff for all $a \in A, c \in C$:

$$\alpha(c) \leq_A a \iff c \leq_C \gamma(a)$$

holds. The two domains $\mathcal{C}$ and $\mathcal{A}$ are called the concrete and the abstract domain, respectively. α is the abstraction map while γ is the concretization map.

The elements of the abstract domain are usually denoted by using the symbol $\sharp$, as $S^\sharp$. As some relevant properties: $\gamma\alpha$ is extensive and $\alpha\gamma$ is reductive, both α and γ are monotone, and α is additive, while γ is co-additive.

Definition 2 (Galois Insertion). *A Galois connection where $\alpha\gamma = id_A$ is called a Galois Insertion (GI), in this case α is onto and γ is one-to-one.*

An abstract domain $\mathcal{A}$ is said to be strict when $\gamma(\bot_A) = \bot$. In a GI the property $\gamma(S^\sharp) = \bot \iff S^\sharp = \bot_A$ also holds. From now on we consider GIs on strict abstract domains (unless otherwise specified).

Some elements of the concrete domain can be approximated without any loss of informations: we call them expressible values.

Definition 3 (Expressible Value). *We say that a concrete element $c \in \mathcal{C}$ is expressible in $\mathcal{A}$ when $\gamma\alpha(c) = c$. When instead $c \prec \gamma\alpha(c)$ we say that c is strictly approximated in $\mathcal{A}$.*

Also functions need to be approximated on abstract domains.

Definition 4 (Correct Approximation). *Given a concrete function $f : C \to C$, we say that $f^\sharp : A \to A$ is a correct approximation of f iff $\alpha f \preceq f^\sharp \alpha$.*

It is known that if $f^\sharp$ is a correct approximation of f then we also have fixpoint correctness when least fixpoints exist, i.e., $\alpha(\mathrm{lfp}\,(f)) \preceq \mathrm{lfp}\,\left(f^\sharp\right)$ holds.

Between all abstract functions that approximate a concrete one we can define the most precise one.

Definition 5 (Best Correct Approximation). *We define the best correct approximation (BCA) of a concrete function f as $f^A \overset{\text{def}}{=} \alpha f \gamma$.*

Such function is called best correct approximation because it holds $f^A \preceq f^\sharp$ for any other correct approximation $f^\sharp$ of f.

Definition 6 (Complete approximation). *A correct approximation $f^\sharp$ is complete iff $\alpha f = f^\sharp \alpha$ holds.*

Analogously to soundness, completeness transfers to fixpoints, meaning that if $f^\sharp$ is complete for f then fixpoint completeness $\alpha(\mathrm{lfp}\,(f)) = \mathrm{lfp}\,\left(f^\sharp\right)$ holds.

An abstract domain is said to be complete for f if there exists a complete approximation for f in that domain. A known result is that a complete abstraction exists iff $\alpha f = \alpha f \gamma \alpha$, or equivalently $\gamma \alpha f = \gamma f^A \alpha$.

We use $\mathbb{C}^A(f)$ to indicate that f admits a complete approximation in A (the abstraction domain will be omitted when clear from the context), this notation naturally extends to sets of functions F in the sense that we write $\mathbb{C}^A(F)$ when all the functions in F admit a complete approximation in A.

Abstract domains can be finite or infinite with some desiderable properties that ensure the termination of the abstract semantics computation.

Definition 7 (ACC Poset). *A poset is ACC (satisfies the Ascending Chain condition) if it has no infinite strictly increasing chain.*

Any analysis through abstract interpretation over an ACC domain is guaranteed to terminate, since by definition it follows that any fixpoint computation will converge in a finite number of steps.

2.3 Programs

Syntax. We consider the usual definitions for Boolean and integer expressions, where, for simplicity we omit expressions that can generate runtime errors, like division by zero. At the level of the concrete collecting semantics, runtime errors could be handled either with the introduction of distinguished elements in the domain or by using the bottom element (the empty set of results). In the former case, runtime errors are distinguished from divergence and must be propagated ad hoc in the semantic definitions, while in the latter case they are just handled as absence of result. We let:

$$\begin{aligned} AExp \ni a &::= v \in \mathbb{Z} \mid x \in Var \mid a + a \mid a - a \mid a * a \mid a \div k \\ BExp \ni b &::= \mathbf{tt} \mid \mathbf{ff} \mid a = a \mid a > a \mid b \wedge b \mid \neg b. \end{aligned}$$

where *Var* is a denumerable set of program variables and $k \in \mathbb{Z}$ is different from 0. We will introduce some syntax sugar whenever required to keep the notation short by writing e.g. $x \leqslant y$ instead of $\neg(x > y)$ or $(x \vee y)$ instead of $\neg(\neg x \wedge \neg y)$.

Moreover, we define the syntactic substitution of all the occurrences of a variable x with an expression a' inside the expression a, denoted by $a[a'/x]$, as:

$$v[a'/x] \overset{\text{def}}{=} v, \qquad y[a'/x] \overset{\text{def}}{=} \begin{cases} a' & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

$$(a_1\ op\ a_2)[a'/x] \overset{\text{def}}{=} a_1[a'/x]\ op\ a_2[a'/x], \qquad \text{for } op \in \{+, -, *\}$$

$$(a \div k)[a'/x] \overset{\text{def}}{=} a[a'/x] \div k.$$

Such definition extends naturally to Boolean expressions in *BExp*.

Given any subset of arithmetic expressions $A \subseteq AExp$ and of Boolean expressions $B \subseteq BExp$, we define two sets of programs: $Imp(A, B)$ the set of imperative programs on A and B , and $Imp^-(A, B)$ a set of programs using only trivial loops of the form **while tt do skip**, for which we use the shorthand $\mathbf{w}_\perp$.

The set of programs $Imp(A, B)$ and $Imp^-(A, B)$ are generated by the following grammars, where $a \in A$ and $b \in B$:

$$\begin{aligned} Imp \ni c &::= \mathbf{skip} \mid x := a \mid c; c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ \mathbf{do}\ c \\ Imp^- \ni c &::= \mathbf{skip} \mid x := a \mid c; c \mid \mathbf{if}\ b\ \mathbf{then}\ c\ \mathbf{else}\ c \mid \mathbf{w}_\perp \end{aligned}$$

The two sets A and B will be omitted when clear by the context.

Concrete semantics. In order to define the semantics of an imperative program, we consider a store $\sigma \in \Sigma$ as a function from $V \subseteq Var$ to integers, that is, $\Sigma \overset{\text{def}}{=} V \to \mathbb{Z}$. We define the semantics for integer expressions $(\!|\cdot|\!) : AExp \times \Sigma \to \mathbb{Z}$ as:

$$\begin{aligned} (\!|v|\!)\sigma &\overset{\text{def}}{=} v & (\!|x|\!)\sigma &\overset{\text{def}}{=} \sigma(x) \\ (\!|a_1 + a_2|\!)\sigma &\overset{\text{def}}{=} (\!|a_1|\!)\sigma \oplus (\!|a_2|\!)\sigma & (\!|a_1 - a_2|\!)\sigma &\overset{\text{def}}{=} (\!|a_1|\!)\sigma \ominus (\!|a_2|\!)\sigma \\ (\!|a_1 * a_2|\!)\sigma &\overset{\text{def}}{=} (\!|a_1|\!)\sigma \circledast (\!|a_2|\!)\sigma & (\!|a \div k|\!)\sigma &\overset{\text{def}}{=} (\!|a|\!)\sigma \mathbin{\bigcirc\!\!\!\!\div} k \end{aligned}$$

where $\oplus, \ominus, \circledast$ and $\mathbin{\bigcirc\!\!\!\!\div}$ are the usual mathematical operations. Analogously we define the semantic of Boolean expressions $(\!|\cdot|\!) : BExp \times \Sigma \to \mathbb{B}$ corresponding to the usual comparison and logical operators $=, >, \wedge, \neg$.

We define the concrete collecting semantics by extending the previous semantics to sets of stores. Let $\mathbb{S} \overset{\text{def}}{=} \mathcal{P}(\Sigma)$, $[\![\cdot]\!] : AExp \times \mathbb{S} \to \mathcal{P}(\mathbb{Z})$ and $[\![\cdot]\!] : BExp \times \mathbb{S} \to \mathbb{S}$ where $[\![a]\!]S \overset{\text{def}}{=} \{(\!|a|\!)\sigma \mid \sigma \in S\}$ and $[\![b]\!]S \overset{\text{def}}{=} \{\sigma \in S \mid (\!|b|\!)\sigma = \mathbf{tt}\}$. The concrete collecting semantics for programs in *Imp* (and *Imp*$^-$) is defined as follows:

$$\begin{aligned}
[\![x := a]\!]S &\overset{\text{def}}{=} \{\sigma[x \mapsto (\!|a|\!)\sigma] \mid \sigma \in S\}\\
[\![\mathbf{skip}]\!]S &\overset{\text{def}}{=} S\\
[\![c_1; c_2]\!]S &\overset{\text{def}}{=} [\![c_2]\!][\![c_1]\!]S\\
[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]S &\overset{\text{def}}{=} [\![c_1]\!][\![b]\!]S \cup [\![c_2]\!][\![\neg b]\!]S\\
[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]S &\overset{\text{def}}{=} [\![\neg b]\!]\mathrm{lfp}\left(\Gamma_S^{b,c}\right)
\end{aligned}$$

where $\Gamma_S^{b,c} \overset{\text{def}}{=} \lambda X.S \cup [\![c]\!][\![b]\!]X$. We also denote with b the set $[\![b]\!]\Sigma$ of all stores satisfying b. By this convention, abusing the notation, $[\![b]\!]S = b \cap S$.

Abstract semantics. By considering A as an abstract domain for $\mathbb{S}$, we can define the abstract collecting semantics as follows. For integer and Boolean expressions, consider the best correct approximations $[\![a]\!]_A^\sharp \overset{\text{def}}{=} [\![a]\!]^A$ and $[\![b]\!]_A^\sharp \overset{\text{def}}{=} [\![b]\!]^A$. The semantics for *Imp* and *Imp*$^-$ is defined as follows:

$$\begin{aligned}
[\![x := a]\!]_A^\sharp S^\sharp &\overset{\text{def}}{=} \alpha[\![x := a]\!]\gamma S^\sharp\\
[\![\mathbf{skip}]\!]_A^\sharp S^\sharp &\overset{\text{def}}{=} S^\sharp\\
[\![c_1; c_2]\!]_A^\sharp S^\sharp &\overset{\text{def}}{=} [\![c_2]\!]_A^\sharp[\![c_1]\!]_A^\sharp S^\sharp\\
[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]_A^\sharp S^\sharp &\overset{\text{def}}{=} [\![c_1]\!]_A^\sharp[\![b]\!]_A^\sharp S^\sharp \vee_A [\![c_2]\!]_A^\sharp[\![\neg b]\!]_A^\sharp S^\sharp\\
[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]_A^\sharp S^\sharp &\overset{\text{def}}{=} [\![\neg b]\!]_A^\sharp\mathrm{lfp}\left(\mathbb{A}_{S^\sharp}^{b,c}\right)
\end{aligned}$$

where $\mathbb{A}_{S^\sharp}^{b,c} \overset{\text{def}}{=} \lambda X^\sharp.S^\sharp \vee_A [\![c]\!]_A^\sharp[\![b]\!]_A^\sharp X^\sharp$. For our following applications we need to observe the following straightforward property, which is a consequence of [14].

Lemma 8. *If all assignments in A and all guards in B are complete on $\mathcal{A}$, then any program in Imp(A, B) (and in Imp$^-(A, B)$) is complete on $\mathcal{A}$.*

Note that for any $X \in \mathbb{S}, X^\sharp \in A$, we have $[\![\mathbf{w}_\perp]\!]X = \varnothing$ and $[\![\mathbf{w}_\perp]\!]_A^\sharp X^\sharp = \perp$.

The concrete semantics is additive and, moreover, when A and B are sets of respectively complete assignments and guards, then for any $c \in Imp^-(A, B)$, $[\![c]\!]_A^\sharp$ is also additive.

In the paper we will exploit Boolean abstraction domains [1]. They are defined by mapping concrete elements into sets of bitvectors as follows (we use $\sigma \models p$ for a given predicate p and a concrete state σ to denote that p holds in σ):

Definition 9 (Boolean abstraction). *Given a set of Boolean predicates $\mathcal{P} = \{p_1, \dots, p_n\}$ defined over concrete states, we define the associated Boolean abstraction on the abstract domain* $\mathrm{Bool}(\mathcal{P}) \overset{\text{def}}{=} \langle \mathcal{P}(\{0,1\}^n), \subseteq, \cup, \cap, \{0,1\}^n, \varnothing \rangle$ *via* *the following abstraction/concretization maps, where* $1\cdot p_i \overset{\text{def}}{=} p_i$ *and* $0\cdot p_i \overset{\text{def}}{=} \neg p_i$*:*

$$\alpha_{\mathcal{P}}(S) \overset{\text{def}}{=} \{\langle v_1,\ldots,v_n\rangle \mid S \cap \{\sigma \mid \sigma \models v_1\cdot p_1 \wedge \cdots \wedge v_n \cdot p_n\} \neq \varnothing\}$$
$$\gamma_{\mathcal{P}}(S^\sharp) \overset{\text{def}}{=} \big\{\sigma \mid \exists\langle v_1,\ldots,v_n\rangle \in S^\sharp .\, \sigma \models v_1\cdot p_1 \wedge \cdots \wedge v_n\cdot p_n\big\}$$

2.4 Conditions for Completeness of Guards

The only abstract domains that are complete for all programs in any Turing complete programming language are the trivial ones[4] (see [14,3]). In [14] the authors further observed that the completeness of (the semantic functions associated with) assignments and Boolean guards occurring in a program is a sufficient condition to guarantee the completeness of the whole program (see Lemma 8 above). While the completeness of assignments has been extensively studied (e.g., the completeness conditions for assignments in major numerical domains such as intervals, congruences, octagons and affine relations have been fully settled [14,16], while the case of Boolean guards is more troublesome and has been studied in [4], from which we report below the main results we exploit here. Formally, completeness of guards is defined as follows:

Definition 10 (Complete Guard). *We say that a guard b is complete (in short $\mathbb{C}(b)$) to indicate that the filtering functions for both b and $\neg b$ are complete, that is, letting $F_b \overset{\text{def}}{=} \{\lambda X \in \mathcal{S} .\, b \cap X, \lambda X \in \mathcal{S} .\, \neg b \cap X\}$, then $\mathbb{C}(b) \iff \mathbb{C}(F_b)$.*

Both b and $\neg b$ being expressible is a necessary condition for $\mathbb{C}(b)$ to hold. Moreover:

Theorem 11 (cf. [4]). *If b and $\neg b$ are expressible in A, then:*

$$\begin{aligned}\mathbb{C}(b) &\iff \forall S \in \mathbb{S} .\, (\, \alpha(S \cap b) = \alpha(S) \wedge_A \alpha(b) \quad \wedge \quad \alpha(S \cap \neg b) = \alpha(S) \wedge_A \alpha(\neg b)\,)\\ &\iff \forall S_1^\sharp, S_2^\sharp \in A .\, \Big(S_1^\sharp \leq \alpha(b) \wedge S_2^\sharp \leq \alpha(\neg b) \implies \gamma(S_1^\sharp \vee_A S_2^\sharp) = \gamma(S_1^\sharp) \cup \gamma(S_2^\sharp)\Big)\end{aligned}$$

Theorem 11 offers a convenient way to check guard completeness: it is necessary and sufficient to check that the join of every two points under b and $\neg b$ respectively is expressible in the domain. Theorem 11 also gives a way to compute the completeness closure w.r.t. to a guards b, by enforcing the presence in the abstract domain of the elements b and $\neg b$ together with the (concrete) join of every two (abstract) points under b and $\neg b$.

3 Bounded Domains

We first introduce the notion of bounded (abstract) domain in order to characterize the class of programs that we will manipulate in order to remove any nontrivial loop.

[4] Namely, the identical abstraction, making abstract and concrete semantics the same, and the top abstraction, making all programs equivalent by abstract semantics.

Definition 12 (k-ACC Poset). *A poset is k-ACC iff all ascending chain lengths are bound by a value $k \in \mathbb{N}$.*

Definition 13 (Bounded domain). *A poset is said to be bounded if there exists some value k for which it is k-ACC.*

Whenever a complete abstract interpretation can be conducted on a bounded domain, then we can exploit the parameter k which gives us an upper bound to the number of iterations required to compute any abstract fixpoint.

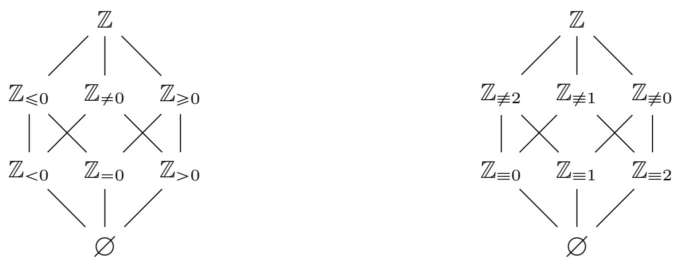
Focusing on the chain of iterates produced when computing $\operatorname{lfp}\left(\mathbb{A}^{b,c}_{S^\sharp}\right)$ we observe that if our abstract domain is bounded, then it is (k+1)-ACC for some k, thus it holds that the produced chain contains no more than $k+1$ distinct values, meaning that the fixpoint computation converges in no more than $k+1$ steps, that is $\operatorname{lfp}(\mathbb{A}) = \mathbb{A}^{(k)}(\bot_A)$.

An interesting result about complete abstractions is the following:

Lemma 14. *Let $\mathcal{A}$ be a strict domain for which $[\![c_1]\!]$ and $[\![c_2]\!]$ are complete. If $[\![c_1]\!]^\sharp_A = [\![c_2]\!]^\sharp_A$ it holds that:*

$$[\![c_1]\!]S = \bot \iff [\![c_2]\!]S = \bot$$

We present two well-known abstract domains *Sign* and *Mod3* in Figure 1 which are both bounded and will be used in the upcoming examples.



(a) *Sign* domain. (b) *Mod3* domain.

Fig. 1: Abstract domains

Note that we use the symbol $\equiv_3$, or $\equiv$ when no ambiguity arises, to identify modulo 3 congruences.

The abstraction function for the *Sign* domain is defined by mapping each set X of concrete values based on the sign of its elements, let us define an auxiliary function $sgn(x) : \mathbb{Z} \to \{\mathbb{Z}_{<0}, \mathbb{Z}_{=0}, \mathbb{Z}_{>0}\}$ which maps concrete values based on their sign (and zero in $\mathbb{Z}_{=0}$), the abstraction map is then defined as:

$$\alpha_{Sign}(S) \stackrel{\text{def}}{=} \bigvee_{x \in S} sgn(x)$$

The concretization map is then defined intuitively over $\mathbb{Z}_{<0}$ as

$$\gamma_{Sign}(\mathbb{Z}_{<0}) \overset{\text{def}}{=} \{x \in \mathbb{Z} \mid x < 0\}$$

and following a similar approach for all the other abstract values.

Both the abstract and the concretization maps for the *Mod3* domain are defined in a similar fashion, by mapping every concrete value based on its modulo 3 reminder as classically defined.

Multiplication is a complete operation in both domains, while addition and difference are complete in *Mod3* only.

We now show that Boolean abstractions give rise to bounded domains which can be composed via predicate union while preserving functional completeness.

Lemma 15. *Let* $\mathcal{P} \overset{\text{def}}{=} \{p_1, \ldots, p_n\}$*,* $\mathcal{Q} \overset{\text{def}}{=} \{q_1, \ldots, q_m\}$ *be sets of predicates, and let* $P \overset{\text{def}}{=} \mathrm{Bool}(\mathcal{P})$ *and* $Q \overset{\text{def}}{=} \mathrm{Bool}(\mathcal{Q})$ *be the Boolean abstraction domains built over the two predicate sets, respectively, and let* $f : \Sigma \to \Sigma$ *be a complete function over both* P *and* Q*. Then, the Boolean abstraction domain* $D \overset{\text{def}}{=} \mathrm{Bool}(\mathcal{P} \cup \mathcal{Q})$ *built over the set of predicates* $\mathcal{P} \cup \mathcal{Q}$ *is such that:*

1. D *is bounded*
2. *The predicate filter for every predicate in* $\mathcal{P} \cup \mathcal{Q}$ *is complete*
3. f *is complete over* D

We also note that the class of bounded domains is closed under reduced product [8, Section 10.1] (which also preserves completeness for functions which are complete on both domains). Moreover, computing the completeness closure w.r.t. to guards as per Theorem 11 preserves boundedness, too.

It is worth noting that using Cartesian predicate abstraction [1] instead of Boolean abstraction would not offer the same guarantees about completeness for predicate filters. Indeed Lemma 15 requires the presence of the disjunction of predicate filters, which is in general missing in the Cartesian predicate abstraction.

4 Program Termination

In this section we explore the connections between complete abstractions in bounded domains and program termination on a given input. Formally, given a command $c \in Imp(A, B)$ and an input set S, the termination problem corresponds to deciding whether $[\![c]\!]S = \bot$ or not, i.e., we want to establish if there is some input in S where c terminates or not.[5] We show that the bound on the length of any ascending chain in the abstract domain can be used to infer the largest number of times each loop must be unrolled. This allows us to define a program transformation that replaces each loop with its bounded unrolling

[5] Note that this is different from establishing termination for all input in S, which should be addressed separately.

while preserving the concrete collecting semantics. While the original program belongs to *Imp*(A, B), the transformed program will belong to *Imp*$^-(A, B)$, that is the only loops have the form $\mathbf{w}_\perp$, which is the only source of divergence. As a main result, termination is thus decidable for any complete program (and any input).

The first observation is that in any $(k+1)$-ACC domain and for any *Imp*(A, B) program **while** b **do** c we have that, for all $S^\sharp$,

$$[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]^\sharp_A S^\sharp = [\![\mathbf{if}^{(k-1)}_{b,c}]\!]^\sharp_A S^\sharp \tag{1}$$

where $\mathbf{if}^{(k-1)}_{b,c}$ is the *Imp*$^-(A, B)$ command inductively defined as:

$$\mathbf{if}^{(0)}_{b,c} \overset{\text{def}}{=} \mathbf{if}\ b\ \mathbf{then}\ \mathbf{w}_\perp\ \mathbf{else\ skip}$$
$$\mathbf{if}^{(n+1)}_{b,c} \overset{\text{def}}{=} \mathbf{if}\ b\ \mathbf{then}\ \left(c; \mathbf{if}^{(n)}_{b,c}\right)\ \mathbf{else\ skip}$$

To see this, we exploit the equality

$$\mathbb{A}^{(k+1)}(\perp_A) = \left(\bigvee_{i=0}^{k} ([\![c]\!]^\sharp_A [\![b]\!]^\sharp_A)^{(i)} S^\sharp\right) \vee_A \perp_A \tag{2}$$

which can be immediately proved by induction on k. Then, the equality (1) can be proved as follows:

$$\begin{aligned}
&[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]^\sharp_A S^\sharp = [\![\neg b]\!]^\sharp_A \mathrm{lfp}\,(\mathbb{A}) = && \{\text{Hypothesis}\}\\
&[\![\neg b]\!]^\sharp_A \mathbb{A}^{(k)}(\perp_A) = && \{\text{Equation 2}\}\\
&[\![\neg b]\!]^\sharp_A \left(\left(\bigvee_{i=0}^{k-1} ([\![c]\!]^\sharp_A [\![b]\!]^\sharp_A)^{(i)} S^\sharp\right) \vee_A \perp_A\right) = && \{\text{Additivity of } [\![\neg b]\!]^\sharp_A\}\\
&\left(\bigvee_{i=0}^{k-1} [\![\neg b]\!]^\sharp_A ([\![c]\!]^\sharp_A [\![b]\!]^\sharp_A)^{(i)} S^\sharp\right) \vee_A [\![\neg b]\!]^\sharp_A \perp_A = && \{\text{Definition of } [\![\mathbf{skip}]\!]^\sharp_A,\ [\![\neg b]\!]^\sharp_A\}\\
&\left(\bigvee_{i=0}^{k-1} [\![\mathbf{skip}]\!]^\sharp_A [\![\neg b]\!]^\sharp_A ([\![c]\!]^\sharp_A [\![b]\!]^\sharp_A)^{(i)} S^\sharp\right) \vee_A \perp_A = && \{\text{By induction on } k\}\\
&[\![\mathbf{if}^{(k-1)}_{b,c}]\!]^\sharp_A S^\sharp
\end{aligned}$$

This process of "unrolling" **while** loops introduces a sequence of nested **if-else** commands of depth k; unrolling a **while** loop having $d - 1$ nested loops inside produces a program having a total of k^d **if-else** commands. Equation 1 proves that the transformed program will exhibit an equivalent behaviour as the original one on the abstract domain $\mathcal{A}$ (for any abstract input).

Next we exploit the notion of complete abstraction. Assuming that the set A contains only complete assignments and B only complete guards on the abstract domain $\mathcal{A}$, we have that every program in *Imp*(A, B) is complete as well as its transformed version in $\mathit{Imp}^{-}(A, B)$, because the transformation does not introduce any new guard or assignment (see Lemma 8). By Lemma 14 and Equation (1), we conclude that, from a divergence perspective, the **while** command is equivalent to its transformed version in **if-else** form, that is, the concrete semantics of the first one diverges if and only the concrete semantics of the second does.

Theorem 16 (Termination). *Let A contain only complete assignments and B only complete guards on the abstract domain $\mathcal{A}$. For any guard $b \in B$ and any command $c \in \mathit{Imp}(A, B)$ we have*

$$[\![\textbf{while } b \textbf{ do } c]\!]S = \bot \iff [\![\textbf{if}^{(k-1)}_{b,c}]\!]S = \bot \tag{3}$$

In fact, a much stronger result can be obtained, namely that the concrete collecting semantics of the program and its transformation coincide.

Theorem 17 (Unrolling). *Let A contains only complete assignments and B only complete guards on the abstract domain $\mathcal{A}$. For any guard $b \in B$ and any command $c \in \mathit{Imp}(A, B)$ we have*

$$[\![\textbf{while } b \textbf{ do } c]\!] = [\![\textbf{if}^{(k-1)}_{b,c}]\!] \tag{4}$$

Proof. By applying $k-1$ expansions

$$[\![\textbf{while } b \textbf{ do } c]\!] = [\![\textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}]\!]$$

we get an equivalent command which is identical to $\textbf{if}^{(n)}_{b,c}$ except for the $\textbf{if}^{(0)}_{b,c}$ element which is replaced by **while** b **do** c. Morever, by Equation (3), we obtain the thesis. □

This result lets us conclude that:

Corollary 18. *For any complete program $c \in \mathit{Imp}(A, B)$ on a bounded strict abstract domain there exists an $\mathit{Imp}^{-}(A, B)$ program which is equivalent under the concrete semantics.*

The above procedure also gives us a constructive way to obtain such an equivalent program that will also be complete.

4.1 Deciding Program Termination

We now use our results to solve the program termination problem, which consists of, given a program c and an input σ, determinining if $[\![c]\!]\{\sigma\} = \varnothing$.

Let us consider the command $c' \in \mathit{Imp}^{-}$ obtained by the previous transformation of c. The result builds on the fact that for any Imp^{-} program, termination is decidable since it can only involve trivial loops $\mathbf{w}_\bot$, i.e., we can safely state that any nonterminating computation will reach some $\mathbf{w}_\bot$ in a finite number of steps.

In fact, for any input, we can safely execute the semantics of the equivalent *Imp*$^-$ program c' and as soon as we enter any loop we can safely conclude that the program diverges on such input.

On the other hand, executing c' on any terminating input will never enter any loop, since that would lead to divergence. Observing that the number of executed steps in absence of any loop is bounded by the program length (since no program line can be executed more than once) concludes that termination will be decided in a finite number of steps.

Putting it all together:

Theorem 19 (Deciding termination). *Let $c \in Imp(A, B)$ be any program which admits a complete approximation in a bounded strict abstract domain, then program termination of c is decidable for any input σ.*

This gives some interesting insight in characterizing the expressiveness of the class of programs for which such an analysis is effective, since classical results such as Rice's Theorem and the undecidability of the halting problem state that program termination is, in general, undecidable. This result can also be applied in a different way: given a program c for which we want to investigate termination, we aim at finding a bounded abstract domain in which all of the guards and assignments appearing in c are complete. By exhibiting such a domain we are able to conclude that termination is decidable for c.

We now show an example to give an idea of the manipulations occurring during the proposed program transformation.

Example 20. Consider the program w_1 defined as follows:

$$w_1 \overset{\text{def}}{=} \textbf{while } (x \not\equiv_3 0) \textbf{ do } (x := 2 * x)$$

where $x \not\equiv_3 0$ is a shorthand for $\neg(x - 3 * (x \div 3) = 0)$. The program w_1 does not contain any other **while** loops inside its body and admits a complete approximation in the domain *Mod3*, which is 4-ACC thus we conclude that termination is decidable on w_1, and its equivalent form is:

$$[\![w_1]\!] = [\![\textbf{if}^{(2)}_{(x \not\equiv_3 0),(x:=2*x)}]\!]$$

that is, $\textbf{if}^{(2)}_{(x \not\equiv_3 0),(x:=2*x)} =$

```
if $x \not\equiv_3 0$ then
    x := 2 * x;
    if $x \not\equiv_3 0$ then
        x := 2 * x;
        if $x \not\equiv_3 0$ then
            while true do skip
        else skip
    else skip
else skip
```

For example, it is now immediate to check that if initially $x = 1$, then we multiply x by 2 twice and we reach the innermost **if** with $x = 4$, thus entering the trivial non-terminating loop. Similarly, for $x = 5$ we reach the innermost **if** with $x = 20$ and detect divergence.

4.2 Exploiting Boolean Abstractions

The decidability result presented in Theorem 19 (but the same considerations will also hold for Theorem 35) can be applied whenever we can prove the existence of a bounded domain satisfying the required hypotheses. Lemma 15 suggests a strong approach to proving the existence of such a domain. The idea is to tailor some boolean abstraction domain to each fragment of the program, possibly using different guards, but complete w.r.t. the same kinds of assignments, and then derive the existence of a complete bounded abstract domain for the whole program from Lemma 15.

As a notable example, we observe that domains built over congruences modulo some given number, like *Mod3*, are complete w.r.t. sum, difference and product. They are also complete w.r.t. all the guards testing the remainder of the division modulo the given number. As they are all boolean abstractions, it follows that both termination and program equivalence are decidable for programs in which all guards test for congruences, and assignments apply arithmetic operations.

5 Program Equivalence

In this section we address the problem of checking program equivalence, which can formally be stated as follows: given two programs $c_1, c_2 \in \mathit{Imp}(A, B)$ we want to decide whether $[\![c_1]\!] = [\![c_2]\!]$ or not. Thanks to the results in Section 4, we define here a program transformation that produces a so-called reduced select normal form, such that program equivalence reduces to decide the validity of a set of guarded statements. The technique presented here applies to deterministic programs as the ones in *Imp*. Its extension to more general analysis frameworks where nondeterministic languages are also considered may not be trivial and needs further investigation.

First, we introduce an intermediate syntax defining a **select** command which constitutes a generalization of **if-else** as a n-way conditional.

Definition 21 (select). *Given two vectors $\tilde{b} \in BExp^n, \tilde{c} \in (Imp^-)^n$ such that $\tilde{b}$ forms a partitioning of Σ, we call n the branching factor of the* **select** *construct with a semantics defined as:*

$$[\![\mathbf{select}(\tilde{b} : \tilde{c})]\!]S \overset{\text{def}}{=} \bigcup_{i=1}^{n} [\![c_i]\!][\![b_i]\!]S$$

This can be seen as a generalized multi-way **if** command, like Dijkstra's guarded statements, and can be expressed as a sequence of nested **if-else** by following a nested structure of the form **if** b_1 **then** c_1 **else** (**if** b_2 **then** c_2 **else** $\ldots$). Since $\tilde{b}$ forms a partitioning of Σ, the order in which the various disjoint cases are nested is not important: semantic equivalence holds under any arbitrary permutation applied to the entries of both $\tilde{b}$ and $\tilde{c}$. We note that as a special case:

$$[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!] = [\![\textbf{select}(\langle b, \neg b\rangle : \langle c_1, c_2\rangle)]\!].$$

By this observation we can define a new auxiliary grammar:

$$\mathit{Select} \ni c \ ::= \ \textbf{skip} \ \mid \ x := a \ \mid \ c; c \ \mid \ \textbf{select}(\tilde{b} : \tilde{c}) \ \mid \ \mathbf{w}_\perp$$

In the following we refer to **skip**, $\mathbf{w}_\perp$ and assignments as *basic commands.* We will also use the notation $\mathit{Select}(A, B)$ to explicitly indicate the sets of expressions and guards used to construct the *Select* commands, as we did for *Imp* and Imp^-. We will show that every Imp^- program can be translated in an equivalent *Select* one (and every *Select* program can be translated into an Imp^- one by applying the above definition of **select** as nested **if** statements).

The program transformation is defined in two phases: first we transform the Imp^- program in a so-called select normal form (see Definition 29) that consists of at most one **select** statement and then we compress series of assignments into a single one (called reduced select normal form, see Definition 32). The transformation to select normal form requires the ability to invert the order in which assignments and guards are applied, so to move all guards upfront. The next section on backward computation introduces the main concepts and notation exploited in the reduction to normal form. Finally, in Section 5.4 it is explained how to compare two programs in reduced select normal forms.

5.1 Backward Computation

In order to manipulate the program structure obtained in the previous section we are going to introduce a concept of inverse semantics for Imp^- commands, which is a function mapping a command c and a set of states S to all possible states for which the execution of the semantics of c may lead to some state in S, also called the *weakest liberal precondition* [11].

Definition 22 (Inverse Concrete Semantics). *We define* $X = [\![c]\!]^{-1}S$ *as the largest set* X *such that* $[\![c]\!]X \subseteq S$*, this can be computed as:*

$$\begin{aligned}
[\![x := a]\!]^{-1}S &\overset{\text{def}}{=} \left\{\sigma' \;\middle|\; \sigma \in S,\, \sigma' \in (\!|a|\!)^{-1}\sigma(x)\right\}\\
[\![\textbf{skip}]\!]^{-1}S &\overset{\text{def}}{=} S\\
[\![c_1; c_2]\!]^{-1}S &\overset{\text{def}}{=} [\![c_1]\!]^{-1}[\![c_2]\!]^{-1}S\\
[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]^{-1}S &\overset{\text{def}}{=} [\![b]\!][\![c_1]\!]^{-1}S \cup [\![\neg b]\!][\![c_2]\!]^{-1}S\\
[\![\mathbf{w}_\perp]\!]^{-1}S &\overset{\text{def}}{=} \Sigma
\end{aligned}$$

where $(\!|a|\!)^{-1}(x) = \{\sigma \mid (\!|a|\!)\sigma = x\}$.

In general $\llbracket\cdot\rrbracket^{-1}$ is not the inverse function (in the mathematical sense) of the concrete semantics, this can be observed for example as:

$$\begin{array}{rl} & \llbracket \textbf{if } x = 0 \textbf{ then } (x := x + 1) \textbf{ else } (x := x + 2)\rrbracket\{0\} = \{1\} \\ \text{but} & \llbracket \textbf{if } x = 0 \textbf{ then } (x := x + 1) \textbf{ else } (x := x + 2)\rrbracket^{-1}\{1\} = \{0, -1\} \neq \{0\} \end{array}$$

This is due to the fact that we can loose some information related to the previous state at each conditional branching. We also note that the function $(\!|a|\!)^{-1}$ in the definition maps each post-value to a set of possible pre-states, this is needed in cases such as that of constant assignment, since we loose any information about the previous value of x after we assign a constant value to it:

$$\llbracket x := 0\rrbracket^{-1}\{0\} = \top$$

Moreover, let us notice that the inverse semantics could give us a set of values smaller (in cardinality) than the input, as for:

$$\llbracket x := 0\rrbracket^{-1}\{1\} = \bot$$

In general, it holds for all Imp^- commands c that $\llbracket c\rrbracket^{-1}$ is additive, implying that it is also monotone, that $\llbracket c\rrbracket^{-1}\llbracket c\rrbracket$ is extensive and dually $\llbracket c\rrbracket\llbracket c\rrbracket^{-1}$ is reductive.

We now observe that, in general, $\llbracket c\rrbracket\llbracket c\rrbracket^{-1}S \neq S$ whenever there exists some $x \in S$ such that x is not reachable through $\llbracket c\rrbracket$ from any input, then $x \notin \llbracket c\rrbracket\llbracket c\rrbracket^{-1}S$, for example:

$$\llbracket x := 0\rrbracket\llbracket x := 0\rrbracket^{-1}\{0, 1\} = \llbracket x := 0\rrbracket\top = \{0\}.$$

The following result follows from the literature.

Lemma 23 (Adjointness). *For any $c \in Imp^-$ and $X, S \subseteq \Sigma$ it holds*

$$X \subseteq \llbracket c\rrbracket^{-1}S \iff \llbracket c\rrbracket X \subseteq S$$

We now show an important result that arises whenever we apply $\llbracket c\rrbracket^{-1}$ to sets which constitute a partitioning. In our case we will apply this result to the partitioning $\{b, \neg b\}$ whenever b is a valid guard.

Lemma 24. *For any $c \in Imp^-$ and any partitioning $P = \{P_1, \ldots, P_n\}$ of Σ it holds that:*

$$\Sigma = \bigcup_{i=1}^{n} \llbracket c\rrbracket^{-1}P_i$$

Lemma 25. *For any Imp^- command c and any partitioning $P = \{P_1, \ldots, P_n\}$ of Σ it holds that:*

$$\llbracket c\rrbracket\{\sigma\} = \bot \iff \sigma \in \left(\llbracket c\rrbracket^{-1}P_i \cap \llbracket c\rrbracket^{-1}P_j\right) \qquad \textit{for any } i \neq j$$

We now introduce $b^{-c} \overset{\text{def}}{=} \llbracket c\rrbracket^{-1}b$ as a shorthand which we will use in the upcoming sections.

Another key result which can be obtained by using the inverse semantics enables us to swap the order of a command execution and a filtering as follows:

Lemma 26. *For any guard b and command c:*

$$[\![b]\!][\![c]\!]S = [\![c]\!]\left([\![c]\!]^{-1}b \cap S\right)$$

that is, in a more succinct notation: $[\![b]\!][\![c]\!]S = [\![c]\!][\![b^{-c}]\!]S$.

We now address the problem of guaranteeing that the process of applying the inverse semantics of c to any guard b produces some b^{-c} which is contained in our language $Imp^{-}(A, B)$. In the upcoming sections we will show that the only commands for which we will need to apply Lemma 26 are those where c is a basic command. Of these three cases, **skip** is trivial and does not need any manipulation, since $[\![b]\!][\![\mathbf{skip}]\!] = [\![\mathbf{skip}]\!][\![b]\!] = [\![b]\!]$, and the same holds for $[\![b]\!][\![\mathbf{w}_\bot]\!] = [\![\mathbf{w}_\bot]\!][\![b]\!] = [\![\mathbf{w}_\bot]\!]$.

The case for assignment can be resolved by applying the following property.

Theorem 27. *For any $a \in AExp$ and $b \in BExp$ it holds:*

$$[\![b]\!][\![x := a]\!] = [\![x := a]\!][\![b[a/x]]\!]$$

The previous theorem together with the previous observations allow us to conclude the main result of this section.

Corollary 28. *If the set of guards B is closed under syntactical substitution, in the sense that for any $a \in A$ and $b \in B$ we have $b[a/x] \in B$, then for any $c \in Imp^{-}$ and $b \in B$, there exists some $b' \in B$ such that $[\![b]\!][\![c]\!] = [\![c]\!][\![b']\!]$.*

5.2 Select Normal Form

Next, we define select normal form and prove that any *Select* command can be put in such format by a semantic-preserving transformation.

Definition 29 (Select normal form, SNF). *We say that a program $c \in Select$ is in normal form (in short, SNF) if either:*

- *c is $\mathbf{w}_\bot$;*
- *c is a sequential composition of* **skip***s and assignments;*
- *c is in the form* $\mathbf{select}(\tilde{b} : \tilde{c})$ *and every c_i is $\mathbf{w}_\bot$ or a sequential composition of* **skip***s and assignments.*

We also use $\tilde{c}; c$ as a shorthand for the vector $\langle (c_1; c), \ldots, (c_n; c)\rangle$ obtained by post-composing c to every command c_i in a sequential way, and the same goes for $c; \tilde{c}$ using pre-composition. We now introduce some rewriting rules involving **select** which are helpful in manipulating *Select* programs:

Post-composition with arbitrary c: The case $[\![\mathbf{select}(\tilde{b} : \tilde{c}); c]\!]$ can be rewritten as $[\![\mathbf{select}(\tilde{b} : \tilde{c}; c)]\!]$, post-composing c to every c_i sequentially; the equality holds in a straightforward way by expanding the definition and applying additivity.

Pre-composition with $\mathbf{w}_\perp$ *and* **skip***:* These two cases are trivial, since:

$$\llbracket \mathbf{skip}; \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket = \llbracket \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket$$
$$\llbracket \mathbf{w}_\perp; \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket = \llbracket \mathbf{w}_\perp \rrbracket$$

Pre-composition with an assignment: In the case $\llbracket x := a; \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket$ we can safely observe that $x := a$ is always terminating, thus by expanding the definitions:

$$\llbracket x := a; \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket S = \bigcup_{i=1}^{n} \llbracket c_i \rrbracket \llbracket b_i \rrbracket \llbracket x := a \rrbracket S = \bigcup_{i=1}^{n} \llbracket c_i \rrbracket \llbracket x := a \rrbracket \llbracket b_i^{-x:=a} \rrbracket S$$

where the fact that the set of states on which $x := a$ diverges is empty ensures that $\tilde{b}^{-x:=a} \stackrel{\text{def}}{=} \llbracket x := a \rrbracket^{-1} \tilde{b} = \langle b_1^{-x:=a}, \ldots, b_n^{-x:=a} \rangle$ forms a partitioning by means of Lemmas 24 and 25, thus $\llbracket x := a; \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket = \llbracket \mathbf{select}(\tilde{b}^{-x:=a} : c; \tilde{c}) \rrbracket$.

Nested **select** *commands:* Let us consider the case $\mathbf{select}(\tilde{b} : \tilde{c})$ where some $c_i = \mathbf{select}(\tilde{b}' : \tilde{c}')$, with $|\tilde{b}| = n$ and $|\tilde{b}'| = m$, we take $i = 1$ (without loss of generality, since the semantics is preserved under permutation of the indexes) and by expanding the definition we get:

$$\llbracket \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket S = \llbracket \mathbf{select}(\tilde{b}' : \tilde{c}') \rrbracket \llbracket b_1 \rrbracket S \cup \bigcup_{i=2}^{n} \llbracket c_i \rrbracket \llbracket b_i \rrbracket S$$

and expanding the isolated term: $\llbracket \mathbf{select}(\tilde{b}' : \tilde{c}') \rrbracket \llbracket b_1 \rrbracket S = \bigcup_{j=1}^{m} \llbracket c'_j \rrbracket \llbracket b'_j \rrbracket \llbracket b_1 \rrbracket S$.

Since $\tilde{b}'$ is a partitioning of Σ, then $\tilde{b}' \cap b_1$ is a partitioning of b_1, thus $\tilde{b}'' = \langle b'_1 \cap b_1, \ldots, b'_m \cap b_1, b_2, \ldots, b_n \rangle$ is a partitioning of Σ and defining $\tilde{c}'' = \langle c'_1, \ldots, c'_m, c_2, \ldots, c_n \rangle$ gives the equality $\llbracket \mathbf{select}(\tilde{b} : \tilde{c}) \rrbracket = \llbracket \mathbf{select}(\tilde{b}'' : \tilde{c}'') \rrbracket$ which has one less **select** command and $|\tilde{b}''| = n + m - 1$.

Sequence of **select** *commands:* We now consider the case where for some $\tilde{b}, \tilde{b}', \tilde{c}, \tilde{c}'$ s.t. $|\tilde{b}| = n$ and $|\tilde{b}'| = m$ we have a sequential composition of $\mathbf{select}(\tilde{b} : \tilde{c})$ and $\mathbf{select}(\tilde{b}' : \tilde{c}')$. We first give an intuitive reasoning for this case: we can expand this term by applying the post-composition rule and we get a new command of the form $\mathbf{select}(\tilde{b} : (\tilde{c}; \mathbf{select}(\tilde{b}' : \tilde{c}')))$ and by (recursively) applying these rules we can obtain a new command such that $\llbracket \mathbf{select}(\tilde{b}''_i : \tilde{c}''_i) \rrbracket = \llbracket c_i; \mathbf{select}(\tilde{b}' : \tilde{c}') \rrbracket$ for each $i = 1 \ldots n$, thus allowing us to apply the rule for nested **select**s to each of the n branches, successfully producing a single **select** command.

We now consider the case where $\mathbf{select}(\tilde{b} : \tilde{c})$ is in normal form, in order to get an explicit formula to rewrite these terms we observe that:

$$\llbracket \mathbf{select}(\tilde{b} : \tilde{c}); \mathbf{select}(\tilde{b}' : \tilde{c}') \rrbracket S = \bigcup_{j=1}^{m} \llbracket c'_j \rrbracket \llbracket b'_j \rrbracket \bigcup_{i=1}^{n} \llbracket c_i \rrbracket \llbracket b_i \rrbracket S$$

which by additivity of $\llbracket \cdot \rrbracket$ can be rewritten as $\bigcup_{j=1}^{m} \bigcup_{i=1}^{n} \llbracket c'_j \rrbracket \llbracket b'_j \rrbracket \llbracket c_i \rrbracket \llbracket b_i \rrbracket S$.

Since we are in normal form, then the vector $\tilde{c}$ does not contain any **select** command and each of its entries is either $\mathbf{w}_\perp$ or a composition of assignments and **skip**s.

We now consider the case where $c_i = \mathbf{w}_\perp$ and we notice that the corresponding terms are $\bigcup_{j=1}^{m} [\![c'_j]\!][\![b'_j]\!][\![\mathbf{w}_\perp]\!][\![b_i]\!]S = \bigcup_{j=1}^{m} [\![\mathbf{w}_\perp]\!][\![b_i]\!]S = [\![\mathbf{w}_\perp]\!][\![b_i]\!]S$ by definition of $\mathbf{w}_\perp$.

When considering any other $c_i \neq c_j$, then c_i converges for any input since it consist of a composition of assignments and **skip**s, thus the corresponding term can be rewritten as

$$\bigcup_{j=1}^{m} [\![c'_j]\!][\![b'_j]\!][\![c_i]\!][\![b_i]\!]S = \bigcup_{j=1}^{m} [\![c'_j]\!][\![c_i]\!][\![b'^{-c_i}_j]\!][\![b_i]\!]S$$

Since c_i is always terminating, by Lemmas 24–25 the sets $b'^{-c_i}_j$ form a partitioning of Σ. Thus we conclude that the sets $b'^{-c_i}_j \cap b_i$ form a partitioning of b_i.

The assumption we made on $\mathbf{select}(\tilde{b} : \tilde{c})$ being in normal form can always be achieved, since by these rules we can always rewrite in normal form the innermost **select** constructs first and proceed our way merging them with the outer ones (a more detailed proof of how we can reduce every *Select* program to normal form is given in Lemma 30). We thus conclude that every composition of two **select** commands can be substituted with a single **select** command having a branching factor less or equal than nm (equality holds when no $\mathbf{w}_\perp$ appear).

Successive applications of the above rewriting rules give us an effective way to reduce every *Select* program to a normal form, in fact:

Lemma 30. *Every Select command c can be reduced in normal form using the above rules.*

We note that the reduction procedure is guaranteed to terminate, therefore giving an effective procedure to obtain a SNF. We also introduce some auxiliary rules which are not necessary in order to reach a normal form but which could help in simplifying some program structures:

Select branch pruning: If we have a command of the form $\mathbf{select}(\tilde{b} : \tilde{c})$ such that there exists b_i for which $[\![b_i]\!] = [\![\mathbf{ff}]\!]$, then we can drop the corresponding branch by removing both b_i and c_i from $\tilde{b}$ and $\tilde{c}$.

Select branch merging: If we have a command of the form $\mathbf{select}(\tilde{b} : \tilde{c})$ such that there exist two indexes $i \neq j$ and $[\![c_i]\!] = [\![c_j]\!]$ we can safely merge the two branches by removing b_j and c_j from $\tilde{b}$ and $\tilde{c}$ respectively and updating $b_i = b_i \vee b_j$.

Select removal: This rule is dual to the select introduction one: every command of the form $\mathbf{select}(\langle b \rangle : \langle c \rangle)$ (thus having branching factor 1) can be rewritten by removing the **select** construct as $[\![c]\!]$; this follows directly observing that since $\{b\}$ forms a partitioning then $[\![b]\!] = [\![\mathbf{tt}]\!]$ and by expanding the definition.

Newly introduced guards: We now examine the new guards which are introduced by the aforementioned manipulations, let $a \in A$ be an arithmetic expression and $b, b_1, b_2 \in B$ be guards in the program we are rewriting, then the newly introduced guards will be of the forms:

$b[a/x]$: If we are applying either the rule for pre-composition with an assignment or that for a sequence of **select** commands;
$b_1 \wedge b_2$: If we are applying the rule for nested **select** commands.
$b_1 \vee b_2$: If we are applying the **select** branch merging rule, this guard can be rewritten as $\neg(\neg b_1 \wedge \neg b_2)$ by means of De Morgan.
tt: If we are applying the **select** introduction rule.

We now observe that main rules (that is, the non-auxiliary ones) only introduce new guards in the form of $b[a/x]$ or $b_1 \wedge b_2$ and we observe that:

Lemma 31. *If $\mathbb{C}(b_1), \mathbb{C}(b_2)$, then the filtering function for $b_1 \wedge b_2$ is also complete:*

This lets us conclude that, under the hypothesis:

$$\mathbb{C}(a) \wedge \mathbb{C}(b) \implies \mathbb{C}(b[a/x]) \tag{5}$$

the rewriting process we defined to reduce every *Select* program into normal form produces new guards by preserving completeness of their filtering functions. Moreover, if B is closed under syntactical substitution for every $a \in A$ to x and forms a Boolean algebra (i.e. is closed under $\wedge$, $\vee$ and $\neg$), then for every $c \in \mathit{Select}(A, B)$ its rewritten normal form c' is such that $c' \in \mathit{Select}(A, B)$.

5.3 Normal Form Scaling in Combined Domains

In order to discuss how the normal form may scale when different abstract domains are combined, we consider the following program p, whose conditions of termination are not easy to detect.

```
while x $\not\equiv_2$ 0:
        x := 5 * x
        while x $\not\equiv_3$ 0:
                x := 2 * x + 1
```

We can observe that each assignment is complete w.r.t. modulo k congruences and this allows us to build a complete bounded domain following the approach of Lemma 15. Given the guards in the program p, the idea is to consider the sets of predicates $M_2 \overset{\text{def}}{=} \{``x \equiv_2 1"\}$ and $M_3 \overset{\text{def}}{=} \{``x \equiv_3 1", ``x \equiv_3 2"\}$ so that the predicates in M_2 ensure completeness of the outer while-guard and those in M_3 ensure completeness for the inner while-guard. Note that the Boolean domain $\mathrm{Bool}(M_3)$ has 16 elements (it is a powerset of four 2-bitvectors) but only 8 elements are relevant, because the 2-bitvector associated with "$x \equiv_3 1$" and "$x \equiv_3 2$" corresponds to false. In fact $\mathrm{Bool}(M_3)$ is equivalent to $\mathit{Mod}3$ and its ascending chains have at most 4 elements. For similar reasons, the resulting bounded abstract domain $\text{Bool}(M_2 \cup M_3)$ is $7 - ACC$, since it is defined as the powerset over the set of 3-bitvectors corresponding to value assignments for each of the predicates in M_2 and M_3.

The result of this paper assures us that we can detect the inputs for which p terminates by investigating its SNF form obtained considering $k = 7$. However, computing the SNF form of p leads to select form with a quite high branching factor. Even if we are interested in characterizing the diverging executions only, the computed SNF will contain several thousands of diverging branches (assuming that the select branch pruning rule is never applied). Also the size of the guards corresponding to such branches will grow rapidly due to the subsequent syntactical substitutions. For example, for program p there will be one diverging branch whose guard is semantically equivalent to "$x \equiv_2 1 \wedge x \equiv_3 0$", but its syntactical expression is more complex. This poses a challenge to gaining useful insight on the program behavior by analyzing the SNF.

We can observe, however, that even if the different branches of the SNF contain syntactically different guards, the number of such guards that are semantically distinct ones is limited by the number of elements in the abstract domain (which in the case of the example is at most 32). This allows us to conclude that many guards appearing in the SNF will be semantically equivalent. Moreover, since the guards in any select command are mutually exclusive, we can be sure that all such redundant guards are indeed semantically equivalent to false. Of course, the problem to detect such false guards must be entrusted to a SMT solver that should support an effective SNF reduction tool implementation. This would allows us to maintain a concise select structure during the rewriting process.

5.4 Deciding Program Equivalence

The problem of deciding semantic equivalence is defined as, given two programs c_1 and c_2, determining whether $[\![c_1]\!] = [\![c_2]\!]$, that is, the two diverge on the same set of inputs and for every converging input, they give the same result.

We now present the main idea to solving program equivalence for programs containing a single variable x. This approach can be straightforwardly generalized to multiple variables by extending our language with a notion of multi-assignments (i.e. every assignment is defined by a tuple of variable-expression pairs and its semantics executes every variable assignment at the same time), but we prefer to keep the notation simpler for the sake of exposition.

The notion of a reduced normal form is as follows.

Definition 32 (Reduced select normal form (RSNF)). *We say that a program $c \in Select$ is in reduced select normal form (in short RSNF) if either:*

- *c is a basic command (that is either* $\mathbf{w}_\perp$, **skip** *or an assignment);*
- *c has the form* $\mathbf{select}(\tilde{b} : \tilde{c})$ *where every c_i is a basic command.*

We first observe that *Select* programs in this form do not allow for arbitrary sequences of assignments and **skip** to occur either inside (or outside) any select branch, but every sequence $c = c_1; c_2; \ldots; c_3$ of said commands can always be reduced into either one single **skip** or one single assignment as follows:

- If for every i it holds $c_i = \textbf{skip}$, then $[\![c]\!] = [\![\textbf{skip}]\!]$
- Otherwise, we remove every c_i for which $c_i = \textbf{skip}$ and merge the remaining assignments observing that:

$$[\![x := a_1; x := a_2]\!] = [\![x := a_2[a_1/x]]\!] \tag{6}$$

  which follows directly from the definition.

We also note that the process of merging two complete assignments preserves completeness.

Lemma 33. $\mathbb{C}([\![x := a_1]\!]) \wedge \mathbb{C}([\![x := a_2]\!]) \Longrightarrow \mathbb{C}([\![x := a_2[a_1/x]]\!])$.

These observations let us conclude that any SNF program can be easily transformed into RSNF by collapsing every sequence of assignments (and **skip**s) into one single command, and the procedure is guaranteed to terminate.

More in detail, given any program $c \in \mathit{Select}(A, B)$ we can compute its SNF $c' \in \mathit{Select}(A, B^*)$ where B^* is the closure of B under $\wedge$ and substitution $b[a/x]$ for $a \in A$, then we can rewrite c' as some RSNF $c'' \in \mathit{Select}(A^*, B^*)$ where A^* is the closure of A under substitution $a[a'/x]$.

When considering two RSNF programs, proving their semantic equivalence can be done by observing that (RSNF programs not containing any **select** construct can be checked as if they contained a single branch):

Lemma 34. *Given two RSNF programs $c = \textbf{select}(\tilde{b} : \tilde{c})$ and $c' = \textbf{select}(\tilde{b}' : \tilde{c}')$ such that $|\tilde{b}| = n$ and $|\tilde{b}'| = m$, then semantic equivalence between c and c' holds iff every formula in the set $E = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \mathcal{E}(i, j)$ is valid, where $\mathcal{E}(i, j)$ is defined according to:*

- *If $c_i = c'_j = \mathbf{w}_\perp$, then $\mathcal{E}(i, j) = \varnothing$*
- *If $c_i \neq c'_j$ and $\mathbf{w}_\perp \in \{c_i, c_j\}$ then $\mathcal{E}(i, j) = \{\neg(b_i \wedge b'_j)\}$*
- *If $c_i = c'_j = \textbf{skip}$, then $\mathcal{E}(i, j) = \varnothing$*
- *If $\{c_i, c_j\} = \{\textbf{skip}, x := a\}$ then $\mathcal{E}(i, j) = \{b_i \wedge b'_j \implies a = x\}$*
- *If $\{c_i, c_j\} = \{x := a, x := a'\}$ then $\mathcal{E}(i, j) = \{b_i \wedge b'_j \implies a = a'\}$*

We can now make use of our previous results to conclude that:

Theorem 35. *Let $c_1 \in Imp(A_1, B_1)$, $c_2 \in Imp(A_2, B_2)$ be any two single-variable programs admitting complete approximation in some (possibly different) bounded strict abstract domains, then the problem of deciding semantic equivalence between c_1 and c_2 can be reduced to that of determining the validity of a set of formulas built by using only Boolean and arithmetic expressions contained in the closures (by substitution) A_1^*, A_2^*, B_1^* and B_2^*.*

Proof. By applying the program transformation defined in Section 4, both c_1 and c_2 can be reduced to some $c'_1 \in \mathit{Imp}^-(A_1, B_1)$, $c'_2 \in \mathit{Imp}^-(A_2, B_2)$. Those two programs can then be expressed as *Select* commands and reduced to SNF by means of Lemma 30 and further reduced to RSNF as discussed in 5.4. Applying Lemma 34 completes the proof. □

In a similar way to what we proposed for Theorem 19 we can apply the result given in Theorem 35 whenever we want to investigate the decidability of semantic equivalence between programs: if we are able to exhibit a domain for each program in which all the guards and assignments are complete, then we have successfully proven that their equivalence is reducible to checking a set of guarded statements, which can be done, e.g., by exploiting SMT solvers like Z3 [17].

Example We consider the following *Imp* program:

$$w \overset{\text{def}}{=} x := -1 * x; \textbf{while } x < 0 \textbf{ do } x := 2 * x$$

In order to reduce w to RSNF we first transform w is SNF. In fact, since it is complete in *Sign* we can find an equivalent Imp^- which can be translated into a select program as:

```
x := −1 * x;
select (
x < 0:  x := 2 * x;
        select (
        x < 0:  x := 2 * x;
                select (
                x < 0:  w⊥,
                x ⩾ 0:  skip )
        x ⩾ 0:  skip )
x ⩾ 0:  skip )
```

we can now reduce this program to RSNF and obtain:

```
select (
x > 0:  w⊥,
x ⩽ 0:  x := −1*x )
```

Now, by taking another equivalent program such as:

```
if x ≠ 0 then
    x := x + 1;
    if x < 1 then
        x := −1 * x + 1;
    else
        while x ⩽ 0 do
                x := x − 2;
else skip
```

which gets reduced to the following RSNF:

```
select (
¬(x ⩽ 0): w⊥ ,
x = 0:  skip ,
x < 0:  x := −1∗(x+1)+1 )
```

we can reduce the problem of determining their semantic equivalence to that of proving the validity of the following set of guarded statements:

$$E = \left\{ \begin{array}{l} \neg((x > 0) \wedge (x = 0)) \\ \neg((x > 0) \wedge (x < 0)) \\ \neg((x \leqslant 0) \wedge \neg(x \leqslant 0)) \\ (x \leqslant 0) \wedge (x = 0) \implies (-1 * x) = x \\ (x \leqslant 0) \wedge (x < 0) \implies (-1 * x) = (-1 * (x + 1) + 1) \end{array} \right\}$$

Since they are all tautologies, the two programs are equivalent.

The same considerations of Section 4.2 about the applicability of the method based on Boolean abstractions for deciding termination are straightforwardly extended to the case of program equivalence.

6 Conclusions

We have investigated the relationship between completeness in Abstract Interpretation and expressiveness of programs, showing that several important properties become decidable for the class of complete programs in certain domains. In particular, we have given a notion of bounded domain and we have studied classes of programs that are parametric on sets of guards and assignments whose abstract semantics is complete on such domains.

In order to study the expressiveness of this class, we have considered two well-known problems: program termination and semantic equivalence, which are of course not decidable in the general case. Our findings seem interesting: as a first result we have shown that under the above hypotheses the termination problem becomes decidable for complete programs. This, of course, severely limits the expressiveness of our class of programs. Then, we defined an intermediate *Select* syntax and a notion of inverse semantics in order to derive a set of rewriting rules for *Select* programs. Applying such rules gives an effective way to express every program from our target class in a canonical form that highlights the program semantics. By further program transformations to the so-called reduced select normal form we are also able to rephrase the problem of deciding semantic equivalence to that of proving the validity of a set of formulas constructed using the original guards and assignments (along with their composition as needed by normalization), giving an effective procedure to solve the semantic equivalence problem. We have developed a proof-of-concept Haskell implementation that has been used to check the program transformations reported in the examples. The tool takes an input program and the bound k of the abstract domain and transforms it in (reduced) select normal form. Note that completeness has to be checked beforehand, as the tool just assumes the existence of the bounded abstract domain.

We have also investigated the applicability of our approach by proposing a method to compose Boolean abstractions, each one designed for being complete for all functions and for some guards appearing in the program. The proposed approach is structural, in the sense that it builds on the functions and guards used in programs, for which suitable complete bounded domains must be detected. Here the main limitation is therefore the completeness requirement: although abstract domains can always be refined to achieve completeness for a given set of functions [13], it is often the case that this process leads to the whole (unbounded) concrete domain. On the other hand, once a library of bounded domains is available, Boolean abstractions could be used to compose them and make the technique applicable to larger sets of programs.

The process described in this work focused on an imperative language with standard single-variable assignments, and reduction to normal form has been defined for single-variable programs only. Considering a more general notion of multi-assignments $\tilde{x} := \tilde{a}$ (i.e. where multiple variables are assigned simultaneously to corresponding expressions) gives a direct generalization of our approach to programs with more than one variable (observing that every assignment is a trivial case of multi-assignment). The select normal form we used can be seen as a star-free fragment of Kleene Algebra with Test (KAT) [15]. In this sense, it is worth pushing the analogy even further and consider the full KAT instead of *Imp* as a reference language, finding suitable conditions under which star expressions can be equivalently iterated only a bounded number of times.

We think that further studies could be conducted on several aspects, such as investigating whether weakening the constraint over boundedness of the domain (that is, when considering ACC domains with finite but not bounded chains) makes program termination undecidable.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking C programs. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2031, pp. 268–283. Springer (2001). https://doi.org/10.1007/3-540-45319-9_19
2. Barringer, H.: A survey of verification techniques for parallel programs. Springer-Verlag (1985)
3. Bruni, R., Giacobazzi, R., Gori, R., Garcia-Contreras, I., Pavlovic, D.: Abstract extensionality: On the properties of incomplete abstract interpretations. Proceedings of the ACM on Programming Languages **4**(POPL), 1–28 (2019)
4. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: Proceedings of LICS 2021, 36th Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 1–13. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470608, distinguished paper

5. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
6. Cousot, P.: Abstract interpretation based formal methods and future challenges. In: Informatics. pp. 138–156. Springer (2001)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252 (1977)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 269–282 (1979)
9. Cousot, P., Cousot, R.: Basic concepts of abstract interpretation. In: Building the Information Society, pp. 359–366. Springer (2004)
10. Das, M., Lerner, S., Seigle, M.: Esp: Path-sensitive program verification in polynomial time. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. pp. 57–68 (2002)
11. Dijkstra, E.W.: A discipline of programming. Series in automatic computation, Prentice-Hall (1976)
12. D'silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**(7), 1165–1178 (2008)
13. Giacobazzi, R., Ranzato, F., Scozzari., F.: Making abstract interpretation complete. Journal of the ACM **47**(2), 361–416 (March 2000). https://doi.org/10.1145/333979.333989, `https://doi.org/10.1145/333979.333989`
14. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. ACM SIGPLAN Notices **50**(1), 261–273 (2015)
15. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (May 1997). https://doi.org/10.1145/256167.256195, `https://doi.org/10.1145/256167.256195`
16. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Foundations and Trends in Programming Languages **4**(3-4), 120–372 (2017)
17. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Nelson, C.G.: Techniques for program verification. Stanford University (1980)
19. Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis. Springer (2010). https://doi.org/10.1007/978-3-662-03811-6, `https://doi.org/10.1007/978-3-662-03811-6`
20. Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society **74**(2), 358–366 (1953)
21. Rival, X., Yi, K.: Introduction to Static Analysis – An Abstract Interpretation Perspective. MIT Press (2020)
22. Turing, A.M.: On computable numbers, with an application to the entscheidungsproblem. A correction. Proceedings of the London Mathematical Society **2**(1), 544–546 (1938)

23. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the First Symposium on Logic in Computer Science. pp. 322–331. IEEE Computer Society (1986)