Analyzing Program Analyses

Roberto Giacobazzi
Dipartimento di Informatica
University of Verona, Italy
& Irdeto Canada

Francesco Logozzo
Microsoft Research
Redmond, WA, USA

Francesco Ranzato
Dipartimento di Matematica
University of Padova, Italy

Dedicated to Radhia Cousot

Abstract

We want to prove that a static analysis of a given program is complete, namely, no imprecision arises when asking some query on the program behavior in the concrete (*i.e.*, for its concrete semantics) or in the abstract (*i.e.*, for its abstract interpretation). Completeness proofs are therefore useful to assign confidence to alarms raised by static analyses. We introduce the completeness class of an abstraction as the set of all programs for which the abstraction is complete. Our first result shows that for any nontrivial abstraction, its completeness class is not recursively enumerable. We then introduce a stratified deductive system $\vdash_A$ to prove the completeness of program analyses over an abstract domain A. We prove the soundness of the deductive system. We observe that the only sources of incompleteness are assignments and Boolean tests — unlikely a common belief in static analysis, joins do not induce incompleteness. The first layer of this proof system is generic, abstraction-agnostic, and it deals with the standard constructs for program composition, that is, sequential composition, branching and guarded iteration. The second layer is instead abstraction-specific: the designer of an abstract domain A provides conditions for completeness in A of assignments and Boolean tests which have to be checked by a suitable static analysis or assumed in the completeness proof as hypotheses. We instantiate the second layer of this proof system first with a generic nonrelational abstraction in order to provide a sound rule for the completeness of assignments. Orthogonally, we instantiate it to the numerical abstract domains of Intervals and Octagons, providing necessary and sufficient conditions for the completeness of their Boolean tests and of assignments for Octagons.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs, formal methods; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—program analysis;

General Terms Languages.

Keywords Abstract interpretation; abstract domain.

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676987

1. Introduction

Consider the typical creative process faced by Alice, the designer of a new static analysis $\mathcal{A}$. Alice starts with a set of "representative" programs and properties to be checked, *e.g.*, the absence of certain runtime errors or of infinite computations. She designs the static analysis—equivalently the abstract domain—that best fits her initial dataset. Next, she will try the analysis $\mathcal{A}$ on a new set of programs, validate the results, and carefully refine the abstract domain or improve the efficiency as needed. When $\mathcal{A}$ becomes precise and fast enough it is shipped to external users.

A major part of the design of $\mathcal{A}$ is the tuning of the cost/precision trade-off—the analysis should deliver the expected answer on a set $\mathbb{T}$ of test programs without asking too many computational (time, space) resources. To achieve it, Alice gives up some of the precision, *i.e.*, she consciously introduces incompleteness into $\mathcal{A}$. The rationale for this design choice is that $\mathbb{T}$ represents real-world programs and as such she needs $\mathcal{A}$ to be complete for them. We denote this set of programs with $\mathbb{C}(A)$, where A is the abstraction, or a set of abstractions, employed by the static analysis $\mathcal{A}$. On the other hand, the incompleteness of the analysis will only manifest in less common or pathological programs.

Ideally, given a program P, Alice wants to know whether P is in $\mathbb{C}(A)$ or not. In fact, if $P \in \mathbb{C}(A)$ then the analysis will be precise and it will answer exactly to any query q on the behaviour of P which is expressible in the language of A. As a consequence, any alarm raised by the static analysis $\mathcal{A}$ on P will be a true alarm, and $\mathcal{A}$ can then be used as an effective program verifier for P.

The Problem. We are interested in proving that $P \in \mathbb{C}(A)$. For instance, let us consider the basic abstract domain $\mathsf{Null} = \{\bot, \mathit{Null}, \mathit{NotNull}, \top\}$ for nullness analysis of variables [1]. The non-null analysis will infer that, for the (intentionally very simple) program NN in Figure 1, x is not null after the conditional—if the true branch is taken, x is null but it is assigned a freshly allocated memory location; otherwise x maintains its non-null value. Intuitively, $NN \in \mathbb{C}(\mathsf{Null})$: both the Boolean guard $x = \mathit{null}$ and its negation $x \neq \mathit{null}$ are exactly representable in Null; **new** is complete w.r.t. nullness: we do not know what happens to the heap (abstraction), but we know for sure it returns a non-null value (complete abstraction); **skip** does not affect the current state; and, eventually, the join does not introduce imprecision.

Next, let us consider the program *Dec* in Figure 2. Let us assume to analyze it using the abstract domain of intervals, Int [3], *without* loop unrolling and widening—in this case, the number of abstract iterations is finite. The analysis will report that $x \in [0, 0]$ at the loop exit, which is the most precise answer. Why is this the case? At first, one may argue that since the inferred invariant is a singleton it is necessarily the most precise one. However, this way of reasoning is incorrect in general. For instance, a program point may be unreachable, so that the concrete set of states at that point is

```
if(x = null)
    x := new object();
else skip;
// query: x ≠ null?
```

Figure 1. The program *NN*: an example where the nullness abstraction is complete.

```
x := 9;
while(x > 0)
   x := x − 1;
// query: x = 0?
```

Figure 2. The program *Dec*: an example where intervals are complete, but the proof is not trivial.

```
x := 9;
while(x > 0)
   x := x − 2;
// query: x = −1?
```

Figure 3. The program Dec_2: an example where intervals are incomplete, and a naif proof system may deduce the opposite.

$\varnothing$, and for any abstraction A every abstract value different from the bottom value $\bot_A$ will be correct yet incomplete. Also, one could reason as follows: the assignment $x := 9$ with a constant is precise with intervals; both Boolean guards $x > 0$ and $x \leq 0$ can be exactly represented with intervals, resp. by $[1, +\infty)$ and $(-\infty, 0]$; the decrement operation $x := x - 1$ on intervals is precise, *e.g.*, $[3, 9] - [1, 1] = [2, 8]$. Therefore, one may conclude that this is why the analysis of *Dec* with intervals is complete. However, this way of reasoning is wrong. Consider the program Dec_2 in Figure 3. In its concrete semantics, at the loop exit point we have that $x \in [-1, -1]$, whereas in its analysis with Int, the *most precise* invariant we can obtain, *even without widenings*, is $x \in [-1, 0]$. Therefore, $Dec_2 \notin \mathbb{C}(\mathsf{Int})$. By contrast, here, the same (intuitive) argument as for the *Dec* program still holds: assignment $x := 9$ is precise with Int; both $x > 0$ and $x \leq 0$ can be exactly represented with Int; the decrement $x := x - 2$ on intervals is precise, *e.g.*, $[3, 9] - [2, 2] = [1, 7]$.

Where is the problem with the above reasoning? It turns out that, even if the guard $x \leq 0$ can be exactly represented with intervals, its transfer function $[\![x \leq 0]\!] : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ is not complete with respect to Int. To give an example, let us consider the set of concrete values $S = \{-1, 1, 3, 5, 7, 9\}$, which is the set of values of x computed by the while-loop of Dec_2. Hence, if the input to $[\![x \leq 0]\!]$ is S then we obtain in Int:

$$\begin{aligned}\mathsf{Int}([\![x \leq 0]\!]S) &= \mathsf{Int}(\{x \mid x \leq 0\} \cap S)\\ &= \mathsf{Int}(\{-1\}) = [-1, -1].\end{aligned}$$

On the other hand, if the input to $[\![x \leq 0]\!]$ is instead the abstraction $\mathsf{Int}(S)$ then we have:

$$\begin{aligned}\mathsf{Int}([\![x \leq 0]\!]\mathsf{Int}(S)) &= \mathsf{Int}([\![x \leq 0]\!][-1, 9])\\ &= \mathsf{Int}([-1, 0]) = [-1, 0],\end{aligned}$$

that is, the transfer function $[\![x \leq 0]\!]$ is not complete on Int. Therefore, we cannot use mere syntactic arguments to prove whether $P \in \mathbb{C}(A)$ because they may lead to wrong conclusions.

Our Solution. As a first step, one may think to design a deductive system for proving that a program P is complete for an abstraction A which requires that all the assignments and Boolean guards of P are complete for A. Such a deductive system will be sound, but definitely too strong. For instance, we cannot use it to prove that the analysis of very simple programs as *Dec* in Figure 2 is complete—this deductive system can deduce completeness of *NN* in Figure 1, though.

Instead, we propose a layered proof system for completeness. Roughly, the layered proof system pushes the completeness proof of the analysis from the program statements up to the assignments and Boolean guards, and then it may use a further analysis to prove their completeness. For instance, in the example of Figure 2, the analysis of the program *Dec* is complete if both the initialization and the while loop are complete. The analysis of the initialization $x := 9$ is trivially complete for intervals. The analysis of the while loop is complete if the analysis of the Boolean guard, of its negation, and of the body are. With a little bit of effort, we can prove that the analysis of the body $x := x - 1$ is complete for intervals. The problem comes with the Boolean guards $x > 0$ and $x \leq 0$. We observed before that, in general, intervals are not complete for those guards. We will show that incompleteness happens when there is a "hole" in the concrete set of values *and* the Boolean guard discerns among those: in fact, in our counterexample above, S has a hole between -1 and 1 and the guard mentions 0. In the program *Dec*, there is no such a hole, because the variable x assumes, in the concrete, all the values between 0 and 9. Thus, the analysis of the Boolean tests of *Dec* with intervals is complete. We can use a static analysis to prove this, and to conclude that the interval analysis of *Dec* is complete.

```
x := 9; y := 0;
while(x > 0)
   {x := x − 1; y := y + 1}
// query: y = 9?
```

Figure 4. The program *DecInc*: an example where Intervals are incomplete whereas Octagons are complete.

We should be careful in defining what "holes" are. Consider the program *DecInc* in Figure 4. Here, both x and y uniformly assume all the values between 0 and 9 and at the end of the loop we have that $x = 0$ and $y = 9$. However, the analysis of *DecInc* with intervals will determine at best that $y \in [0, +\infty)$. Once again, the source of incompleteness is the Boolean guard. In fact, the transfer function of $x \leq 0$, which is the negation of the guard $x > 0$, is not complete for intervals with the set of input pairs (x, y) of concrete values $S = \{(9, 0), (8, 1), \ldots, (1, 8), (0, 9)\}$. It is worth remarking that this incompleteness does not originate from a perceived imprecision in the abstract join of intervals: in fact, the abstract lub $\sqcup_A$ of any abstraction A is *always* complete, meaning that $\alpha(X \cup Y) = \alpha(X) \sqcup_A \alpha(Y)$ always holds—since abstraction functions in Galois connections always preserve joins [3]. On the other hand, using the abstract domain of Octagons [20], the guard $x > 0$ and its negation $x \leq 0$ are instead complete for the input set S.

A further issue arises with assignments. For relational abstractions as Octagons, it turns out that assignments like $z := x + y$ and $x := 2 * y$ are incomplete even if $x + y$ and $2 * y$ are complete considered as numerical expressions. Indeed, we will show that the only assignments which are complete for Octagons have the following shape: $x := a * y + k$ and $x := a * x + k$ for $a \in \{-1, 0, 1\}$ and $k \in \mathbb{Z}$.

Given an abstraction, it is therefore natural to ask: (i) which conditions on assignments/Boolean expressions and their input concrete states ensure completeness of the analysis; and (ii) how to

statically check these conditions so that the we can prove that the analysis is complete. We will address these issues in the article.

Vision. This work originates from our experience in building real world-static analyses. We want to be able to prove that the static analysis on a given abstraction in some cases will infer an invariant which is complete, and therefore use this information for alarm ranking: alarms originating from analyses that cannot be proved complete should be ranked lower. Therefore, we advocate a new methodology for designing static analyses. In addition to the "main" static analysis for property checking, Alice also provides one or more completeness conditions, to be used to validate the completeness of the analysis for a given program. Our goal is ambitious because, as we will see in the paper, a proof of completeness relies upon the determination of the best correct approximation for all the statements occurring in the program. While this requirement can be bypassed for the fundamental constructs for program composition—that is, sequential composition, branching and guarded iteration, for the latter two by requiring completeness of their Boolean guards—for generic assignments this is still an open problem, as there exist (relational) abstract domains where the best correct approximation of the assignment for certain expressions is not known yet.

Results. Firstly, we recall the necessary basic notions of abstract interpretation and completeness of abstract domains (Section 2), and we define the syntax, concrete semantics and abstract semantics for a basic imperative language (Section 3). We introduce the notion of completeness class for an abstraction A as the set of programs for which the analysis over A is complete; we show that for all nontrivial abstractions the corresponding completeness class is infinite, nontrivial, nonextensional, and in particular it is a productive set, namely it shares the same structure of the enumerations of true sentences in first order arithmetics, therefore inheriting Gödel's incompleteness (Section 4). We provide the core proof system for completeness; we prove that it is sound for all nonrelational abstract domains and argue that completeness of Boolean guards can be regarded as the hardest part in these completeness proofs (Section 5). We then analyze the soundness of the proof system for relational abstractions, in particular for the case of Octagons. We introduce a further layer in the proof system to prove completeness of assignments and Boolean guards; we instantiate it with the Interval and Octagon abstractions; we characterize the assignments which are complete for Octagons; we provide necessary and sufficient conditions for the completeness of Intervals and Octagons when handling Boolean guards (Section 6). We conclude by comparing our work to previous one (Section 7) and by discussing other possible applications of completeness proofs (Section 8).

2. Background

Orders. Given a function $f : X \to Y$ and a subset $S \subseteq X$ then $f(S) \triangleq \{f(s) \in Y \mid s \in S\}$ denotes the image of f on S. A complete lattice C w.r.t. an ordering relation $\leq$ is denoted by $\langle C, \leq\rangle$. The pointwise ordering relation $\sqsubseteq$ between two functions $f, g : X \to C$ whose co-domain is a complete lattice C, is defined by $f \sqsubseteq g$ if for any $x \in X$, $f(x) \leq_C g(x)$. A function $f : C \to D$ between complete lattices is additive (co-additive) when f preserves arbitrary lub's (glb's), while it is continuous when f preserves lub's of chains. Given a function $f : C \to C$ on a complete lattice C, $\mathsf{lfp}(f)$ and $\mathsf{gfp}(f)$ denote, respectively, the least and greatest fixpoints of f, when they exist. Recall that least and greatest fixpoints always exist for monotone functions.

Abstract Domains. In standard abstract interpretation [3, 4], abstract domains, also called abstractions, are specified by Galois connections/insertions (GCs/GIs for short). Concrete and abstract domains, $\langle C, \leq_C\rangle$ and $\langle A, \leq_A\rangle$, are assumed to be complete lattices which are related by abstraction and concretization maps $\alpha : C \to A$ and $\gamma : A \to C$ that give rise to a GC (α, C, A, γ), that is, for all $a \in A$ and $c \in C$, $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. A GC is a GI when $\alpha \circ \gamma = \lambda x.x$. We use $\mathrm{Abs}(C)$ to denote all the possible abstractions of C: $A \in \mathrm{Abs}(C)$ means that A is an abstract domain of C specified by some GC/GI, while the notations $A_\alpha \in \mathrm{Abs}(C)$ and $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ are used to emphasize the underlying abstraction and concretization functions α and γ.

If $\alpha : C \to A$ is additive then we obtain a GC (α, C, A, α^+) by considering its right-adjoint $\alpha^+ \triangleq \lambda a.\ \vee_C \{c \in C \mid \alpha(c) \leq_A a\}$. Dually, if $\gamma : C \to A$ is co-additive then (γ^-, C, A, γ) is a GC where $\gamma^- \triangleq \lambda c.\ \wedge_A \{a \in A \mid c \leq_C \gamma(a)\}$ is the left-adjoint of γ. We recall well known properties of a GC (α, C, A, γ): (1) α is additive; (2) γ is co-additive; (3) $\gamma \circ \alpha : C \to C$ is a closure operator, *i.e.*, it is monotone, idempotent, and increasing (*i.e.*, $x \leq \gamma(\alpha(x))$); (4) if $\mu : C \to C$ is a closure operator then $(\mu, C, \mu(C), id)$ is a GI.

If $A_1, A_2 \in \mathrm{Abs}(C)$ then A_1 is equivalent to A_2, denoted $A_1 \cong A_2$, when $\gamma_{A_1}(A_1) = \gamma_{A_2}(A_2)$. The quotient $\mathrm{Abs}(C)_{/\cong}$ is called the lattice of abstractions [4] because it turns out to be a complete lattice w.r.t. the relative precision ordering $\sqsubseteq$: $A_1 \sqsubseteq A_2$ iff for any $c \in C$, $\gamma_{A_1}(\alpha_{A_1}(c)) \leq_C \gamma_{A_2}(\alpha_{A_2}(c))$. $A_1 \sqsubseteq A_2$ means that A_1 is a more precise abstraction of the concrete domain C than A_2, or, equivalently, that A_2 abstracts A_1. A lub $\sqcup_i A_i$ in the lattice of abstractions is therefore the most precise domain in $\mathrm{Abs}(C)_{/\cong}$ which abstracts all A_i's. The glb $\sqcap_i A_i$ is the most abstract (*i.e.*, less precise) domain in $\mathrm{Abs}(C)_{/\cong}$ which is more precise than all A_i's, and is also called reduced product of the A_i's.

Intervals. The interval abstraction was introduced by Cousot and Cousot [2] and it is still a widely used nonrelational abstraction since it is efficient and yet able to give useful information to prove, *e.g.*, the absence of arithmetic overflows or out-of-bounds array accesses. Let $\mathbb{Z}^\star \triangleq \mathbb{Z} \cup \{-\infty, +\infty\}$ and assume that the standard ordering $\leq$ on $\mathbb{Z}$ is extended to $\mathbb{Z}^\star$ in the usual way. Hence:

$$\mathsf{Int} \triangleq \{[a, b] \mid a, b \in \mathbb{Z}^\star,\ a \leq b\} \cup \{\bot\}$$

endowed with the standard ordering $\leq_{\mathsf{Int}}$ induced by interval containement gives rise to a complete lattice, where $\bot$ is the bottom element and $[-\infty, +\infty]$ is the top element. Then, consider the function $min : \wp(\mathbb{Z}) \to \mathbb{Z}^\star$ defined as follows:

$$min(X) \triangleq \begin{cases} x & \text{if } \exists x \in X. \forall y \in X.\ x \leq y \\ -\infty & \text{otherwise} \end{cases}$$

while $max : \wp(\mathbb{Z}) \to \mathbb{Z}^\star$ is dually defined. The abstraction map $\alpha : \wp(\mathbb{Z}) \to \mathsf{Int}$ defined by:

$$\alpha(X) \triangleq \begin{cases} \bot & \text{if } X = \varnothing \\ [min(X), max(X)] & \text{if } X \neq \varnothing \end{cases}$$

preserves arbitrary unions in $\wp(\mathbb{Z})$ and therefore gives rise to a GI, *i.e.*, $\mathsf{Int}_\alpha \in \mathrm{Abs}(\langle \wp(\mathbb{Z}), \subseteq\rangle)$.

It is straightforward to define the nonrelational lift of intervals to n-dimensions, with $n > 1$. If $\vec{x} \in \mathbb{Z}^n$ is a n-dimensional vector of integers and $i \in [1, n]$ then $\vec{x}_i \in \mathbb{Z}$ denotes the i-th component of $\vec{x}$. In turn, if $X \in \wp(\mathbb{Z}^n)$ is a set of vectors then $X_i \triangleq \{\vec{x}_i \in \mathbb{Z} \mid \vec{x} \in \mathbb{Z}\}$ denotes the i-th projection of X. Then, the n-dimensional interval abstraction is $\mathsf{Int}^n \triangleq \bigtimes_1^n \mathsf{Int}$, endowed with the component-wise ordering of Int. The following component-wise abstraction map $\alpha : \wp(\mathbb{Z}^n) \to \mathsf{Int}^n$ defined as $\alpha(X) \triangleq \langle \alpha_{\mathsf{Int}}(X_1), \ldots, \alpha_{\mathsf{Int}}(X_1)\rangle$ yields an abstract domain $\mathsf{Int}^n_\alpha \in \mathrm{Abs}(\langle \wp(\mathbb{Z}^n), \subseteq\rangle)$.

Octagons. The octagon abstract domain [19, 20] is a relational refinement of the interval abstraction which is able to represent vari-

able relations of the form $\pm x \pm y \leq k$ while keeping an acceptable efficiency by exploiting a representation based on a modification of so-called difference bound matrices. The terminology "octagon" comes from the fact that in two dimensions an abstract element is a polyhedron with at most eight sides. Given $n \geq 1$, a difference bound matrix (DBM) $\boldsymbol{m} \in \mathrm{DBM}_n$ is a $n \times n$ square matrix having entries $\boldsymbol{m}_{i,j} \in \mathbb{Z} \cup \{+\infty\}$. Then, any $\boldsymbol{m} \in \mathrm{DBM}_{2n}$ induces, according to the canonical representation described in [20], a n-dimensional octagon $\mathsf{oct}_{\boldsymbol{m}} \in \wp(\mathbb{Z}^n)$ defined as follows:

$$\begin{aligned}\mathsf{oct}_{\boldsymbol{m}} \triangleq \big\{\langle x_1, \ldots, x_n\rangle \in \mathbb{Z}^n \mid \forall i, j \in [1, n].\\ x_i - x_j \leq \boldsymbol{m}_{2i-1,2j-1},\ x_i + x_j \leq \boldsymbol{m}_{2i-1,2j},\\ -x_i + x_j \leq \boldsymbol{m}_{2i,2j-1},\ -x_i - x_j \leq \boldsymbol{m}_{2i,2j}\big\}.\end{aligned}$$

Observe that interval bounds like $x_i \geq a$ and $x_i \leq b$ can be encoded, respectively, by $-x_i - x_i \leq -2a$ and $x_i + x_i \leq 2b$, so that a n-dimensional interval is a particular octagon. Let us also recall that DBM representations are not unique, so that it can happen that for different DBMs $\boldsymbol{m} \neq \boldsymbol{m}'$, we have that $\mathsf{oct}_{\boldsymbol{m}} = \mathsf{oct}_{\boldsymbol{m}'}$. The n-dimensional octagon abstract domain is defined to be:

$$\mathsf{Oct}^n \triangleq \{\mathsf{oct}_{\boldsymbol{m}} \in \wp(\mathbb{Z}^n) \mid \boldsymbol{m} \in \mathrm{DBM}_{2n}\}$$

and turns out to be a complete lattice wr.t. the subset relation. Observe that octagons are closed under arbitrary set intersections but not under set unions, so that $\langle \mathsf{Oct}^n, \subseteq\rangle$ is a complete lattice which is not a join sublattice of $\wp(\mathbb{Z}^n)$. $\alpha : \wp(\mathbb{Z}^n) \to \mathsf{Oct}^n$ is the abstraction given by $\alpha(X) \triangleq \bigcap\{O \in \mathsf{Oct}^n \mid X \subseteq O\}$. It is well defined and, being a closure operator, it is additive, so that it yields a GI, namely, we have that $\mathsf{Oct}^n_\alpha \in \mathrm{Abs}(\langle \wp(\mathbb{Z}^n), \subseteq\rangle)$. Clearly, octagons are a refinement of intervals, *i.e.*, $\mathsf{Oct}^n \sqsubseteq \mathsf{Int}^n$.

Correctness. Let $f : C \to C$ be some concrete monotone function—to keep notation simple, we consider 1-ary functions—and let $f^\sharp : A \to A$ be a corresponding monotone abstract function defined on some abstraction $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. Then, $f^\sharp$ is a correct (or sound) abstract interpretation of f on A when $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ holds. If $f^\sharp$ is correct for f then we also have fixpoint correctness, that is, $\alpha(\mathsf{lfp}(f)) \leq_A \mathsf{lfp}(f^\sharp)$. The abstract function

$$f^\alpha \triangleq \alpha \circ f \circ \gamma : A \to A$$

is called the best correct approximation of f on A, because any abstract function $f^\sharp$ is correct iff $f^\alpha \sqsubseteq f^\sharp$. Hence, f^α plays the role of the best possible approximation of f on the abstraction A.

Completeness. An abstract function $f^\sharp$ is a complete abstract interpretation of f on A when $\alpha \circ f = f^\sharp \circ \alpha$ holds [4]. When $f^\sharp$ is an abstract transfer function on the abstraction A used by a static analysis, completeness intuitively encodes the greatest precision for $f^\sharp$, meaning that the abstract behaviour of $f^\sharp$ on A exactly matches the abstraction in A of the concrete behaviour of f. If $f^\sharp$ is complete for f then we have fixpoint completeness (also called fixpoint transfer): $\alpha(\mathsf{lfp}(f)) = \mathsf{lfp}(f^\sharp)$. It turns out that completeness $\alpha \circ f = f^\sharp \circ \alpha$ holds iff

$$\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha.$$

Thus, the possibility of defining a complete approximation $f^\sharp$ of f on some $A \in \mathrm{Abs}(C)$ only depends on the concrete function f and on the abstraction A [13]. We will use both A is complete for f and f is complete on A to refer to the completeness equation $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$.

The problem of making abstract domains complete has been addressed in [13]. A constructive characterization of the most abstract refinement, called complete shell, and of the most concrete simplification, called complete core, of any abstract domain $A \in \mathrm{Abs}(C)$, making it complete for a given continuous function $f : C \to C$, is given as a fixpoint solution of an abstract domain equation derived from f and A. Let us recall this definition for the case of the complete shell refinement. This is the abstract domain refinement $\mathrm{Shell}_f : \mathrm{Abs}(C) \to \mathrm{Abs}(C)$ defined as:

$$\mathrm{Shell}_f(A) \triangleq \mathsf{gfp}(\lambda X.\, A \sqcap R_f(X))$$

where, for any domain $X_{\alpha,\gamma} \in \mathrm{Abs}(C)$, $R_f(X)$ is the most abstract domain which contains the maximal inverse image of f on X, namely, $R_f(X) \supseteq \bigcup_{y \in X} \max(\{c \in C \mid f(c) \leq \gamma(y)\})$. As a consequence, the following characterization of complete abstract domains holds: the abstraction A is complete for f iff for any $y \in A$, $\alpha(\max(\{c \in C \mid f(c) \leq \gamma(y)\})) \in A$.

3. Language and Abstract Semantics

Syntax. We consider a basic deterministic while-language Imp with arithmetic and Boolean expressions, as defined, e.g., in [25], whose syntax is as follows:

$$\begin{aligned}\mathrm{AExp} \ni a &::= v \in \mathbb{Z} \mid x \in \mathit{Var} \mid a + a \mid a - a \mid a * a\\ \mathrm{BExp} \ni b &::= \mathbf{t} \mid \mathbf{f} \mid a = a \mid a > a \mid b \wedge b \mid \neg b\\ \mathsf{Imp} \ni C &::= \textbf{skip} \mid x := a \mid C; C \mid \textbf{if } b \textbf{ then } C \mid \textbf{while } b \textbf{ do } C\end{aligned}$$

Programs are commands in Imp. The set of variables occurring in some syntactic object is $\mathit{svars}(s) \subseteq \mathit{Var}$.

Concrete Semantics. We let $\mathbb{S} \triangleq \mathit{Var} \to \mathbb{Z}$ denote the set of program stores. We will often represent a store $\rho \in \mathbb{S}$ as a tuple $\langle x_1/v_{x_1}, \ldots, x_n/v_{x_n}\rangle$.

The semantics of arithmetic expressions is the function $(\!|a|\!) : \mathbb{S} \to \mathbb{Z}$ defined as usual. Similarly, the semantics of Boolean predicates is the function $(\!|b|\!) : \mathbb{S} \to \{\mathbf{t}, \mathbf{f}\}$.

The collecting semantics of an arithmetic expression a is the function $[\![a]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{Z})$ defined as $[\![a]\!]S \triangleq \{(\!|a|\!)\rho \mid \rho \in S\}$. Similarly, the collecting semantics of a predicate b is the function $[\![b]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ defined as $[\![b]\!]S \triangleq \{\rho \in S \mid (\!|b|\!)\rho = \mathbf{t}\}$. Intuitively, $[\![b]\!]S$ filters the concrete states of S which make b true.

The collecting denotational semantics of a command C is the function $[\![C]\!] : \wp(\mathbb{S}) \to \wp(\mathbb{S})$ defined as

$$\begin{aligned}[\![\textbf{skip}]\!]S &\triangleq S\\ [\![x := a]\!]S &\triangleq \{\rho[x \mapsto (\!|a|\!)\rho] \mid \rho \in S\}\\ [\![C_1; C_2]\!]S &\triangleq [\![C_2]\!]([\![C_1]\!]S)\\ [\![\textbf{if } b \textbf{ then } C]\!]S &\triangleq [\![C]\!][\![b]\!]S \cup [\![\neg b]\!]S\\ [\![\textbf{while } b \textbf{ do } C]\!]S &\triangleq [\![\neg b]\!]\big(\mathsf{lfp}(\lambda T.\, S \cup [\![C]\!][\![b]\!]T)\big).\end{aligned}$$

Please observe that nontermination is encoded by the empty set: A program $C \in \mathsf{Imp}$ does not terminate on an input $S \in \wp(\mathbb{S})$ when $[\![C]\!]S = \varnothing$. In the following, we make the hypothesis that the inputs to semantic functions, *i.e.*, the sets of stores in $\wp(\mathbb{S})$ are recursively enumerable sets.

Store Abstractions. A store abstraction is specified by a GC $A_{\alpha,\gamma} \in \mathrm{Abs}(\langle \wp(\mathbb{S}), \subseteq\rangle)$ on sets of stores. Variable projection (a.k.a. existential abstraction) of a set of stores on a set of variables $V \subseteq \mathit{Var}$ is defined by the following mapping $\exists_V : \wp(\mathbb{S}) \to \wp(\mathbb{S})$:

$$\exists_V(S) \triangleq \{\rho \in \mathbb{S} \mid \exists \eta \in S. \forall y \notin V.\, \rho(y) = \eta(y)\}.$$

Variable projection on a single variable $x \in \mathit{Var}$ is denoted by $\exists_x$. We will consider store abstractions with a complete variable projection, namely, an abstract domain $A \in \mathrm{Abs}(\wp(\mathbb{S}))$ such that for any $V \subseteq \mathit{Var}$, $\alpha \circ \exists_V = \alpha \circ \exists_V \circ \gamma \circ \alpha$. Completeness of variable projection is satisfied by virtually any store abstraction used in practice, *e.g.*, this property holds for all known nonrelational and relational numerical store abstractions like Sign, Int, Oct, congruences [14], Karr's domain of linear equalities [16], etc. Any store

abstraction A (which is complete for variable renaming—the easy formalization of variable renaming is omitted here) can be also viewed as an abstraction of sets of integer values in $\langle \wp(\mathbb{Z}), \subseteq \rangle$ by considering stores that "focus" on a single variable (call it y): this is determined by the abstraction map $\alpha^{\mathbb{Z}} : \wp(\mathbb{Z}) \to A$ defined as $\alpha^{\mathbb{Z}}(Z) \triangleq \alpha(\{\rho \in \mathbb{S} \mid \rho(y) \in Z\})$. Hence, in the following, the store abstraction A is also viewed and used as an abstraction of sets of integers.

Abstract Semantics. We define the best correct abstract semantics for a generic store abstraction $A_{\alpha,\gamma} \in \mathrm{Abs}(\langle \wp(\mathbb{S}), \subseteq \rangle)$. We assume that $\sqcup$ is the join in A.

The best correct abstract semantics $[\![a]\!]^\alpha : A \to A$ for an arithmetic expression $a \in \mathrm{AExp}$ is $[\![a]\!]^\alpha S^\sharp \triangleq \alpha([\![a]\!]\gamma(S^\sharp))$. It is worth noting that, in general, the best correct abstract semantics is not the standard compositional definition $[\![\cdot]\!]^{\alpha_c}$ of the abstract semantics of expressions. For instance, the compositional definition of a binary integer operation op is:

$$[\![a_1\, op\, a_2]\!]^{\alpha_c} S^\sharp \triangleq [\![a_1]\!]^{\alpha_c} S^\sharp\ op^\alpha\ [\![a_2]\!]^{\alpha_c} S^\sharp.$$

When instantiating such a definition for the intervals Int, it turns out that:

$$\begin{aligned}[\![x + x * (-1)]\!]^{\mathsf{Int}}\langle x/[1,2]\rangle &= [0,0] \sqsubset \\ [\![x + x * (-1)]\!]^{\mathsf{Int}_c}\langle x/[1,2]\rangle &= [1,2] + ([1,2] * [-1,-1]) \\ &= [-1,1]\end{aligned}$$

The best correct abstract semantics for a predicate $b \in \mathrm{BExp}$ on A is the abstract function $[\![b]\!]^\alpha : A \to A$ defined as

$$[\![b]\!]^\alpha S^\sharp \triangleq \alpha([\![b]\!]\gamma(S^\sharp)).$$

We derive the best correct abstract semantics $[\![C]\!]^\alpha : A \to A$ of a program $C \in \mathsf{Imp}$ as the best correct approximation of its concrete semantics. We exploit two well-known facts: (i) the composition of two complete functions is complete; and, (ii) in a GC, the abstract join is always the best correct approximation of the concrete join. Let $S^\sharp \in A$ be an abstract store:

$$\begin{aligned}[\![\mathbf{skip}]\!]^\alpha S^\sharp &\triangleq S^\sharp \\ [\![x := a]\!]^\alpha S^\sharp &\triangleq \alpha(\{\rho[x \mapsto (\!|a|\!)\rho] \mid \rho \in \gamma(S^\sharp)\}) \\ [\![C_1; C_2]\!]^\alpha S^\sharp &\triangleq [\![C_2]\!]^\alpha([\![C_1]\!]^\alpha S^\sharp) \\ [\![\mathbf{if}\ b\ \mathbf{then}\ C]\!]^\alpha S^\sharp &\triangleq [\![\neg b]\!]^\alpha S^\sharp \sqcup [\![C]\!]^\alpha([\![b]\!]^\alpha S^\sharp) \\ [\![\mathbf{while}\ b\ \mathbf{do}\ C]\!]^\alpha S^\sharp &\triangleq [\![\neg b]\!]^\alpha\big(\mathsf{lfp}(\lambda X^\sharp.\, S^\sharp \sqcup [\![C]\!]^\alpha [\![b]\!]^\alpha X^\sharp)\big)\end{aligned}$$

Let us comment on the definition: (i) The best correct approximation of an assignment $[\![x := a]\!]^\alpha$ does not rely on the best correct abstract semantics $[\![a]\!]^\alpha$ of the arithmetic expression a; (ii) In the least fixpoint definition for $[\![\mathbf{while}\ b\ \mathbf{do}\ C]\!]^\alpha$, the abstract function $\lambda X^\sharp.\, S^\sharp \sqcup [\![C]\!]^\alpha [\![b]\!]^\alpha X^\sharp$ turns out to be the best correct approximation on A of the concrete function $\lambda T.\, S \cup [\![C]\!][\![b]\!]T$; (iii) In the abstract function $\lambda X^\sharp.\, S^\sharp \sqcup [\![C]\!]^\alpha [\![b]\!]^\alpha X^\sharp$, practical static analyzers replace the abstract lub $\sqcup$ with a widening operator ∇ to accelerate or force the convergence of the iterations in the abstract domain A [3], thus losing the property of having the best correct abstract semantics of programs.

Complete Abstractions. A store abstraction $A_\alpha \in \mathrm{Abs}(\wp(\mathbb{S}))$ is complete for an arithmetic expression $a \in \mathrm{AExp}$, a Boolean test $b \in \mathrm{BExp}$ and a program $C \in \mathsf{Imp}$, when for any set of stores $S \in \wp(\mathbb{S})$:

$$\begin{aligned}\alpha([\![a]\!]S) &= [\![a]\!]^\alpha \alpha(S), \\ \alpha([\![b]\!]S) &= [\![b]\!]^\alpha \alpha(S), \\ \alpha([\![C]\!]S) &= [\![C]\!]^\alpha \alpha(S).\end{aligned}$$

4. The Class of Complete Abstractions

Completeness, Queries and Alarms. The goal of a static analysis/verification tool is to soundly answer some questions on the dynamic (concrete) execution of programs. For instance, common queries to static analysis tools are: "*Is this variable not-null?*", "*Is this variable non-negative?*", "*Does this loop ever terminate?*". The first two are examples of safety properties, the third of a liveness property. Here, we focus on safety properties. Given a program P, a set of input stores $I \subseteq \mathbb{S}$ and a store query (*i.e.*, a store predicate) q, we are interested to know whether the final states of P satisfy the query q, *i.e.*, whether the following formula holds:

$$\forall \rho \in \mathbb{S}\,.\, \rho \in [\![P]\!]I \Rightarrow q(\rho) = \mathbf{true},$$

or, equivalently, by denoting with $[\![q]\!] \subseteq \mathbb{S}$ the stores satisfying q, whether $[\![P]\!]I \subseteq [\![q]\!]$. Obviously, we cannot decide this for each possible query. Therefore, static analyses over-approximate the collecting semantics $[\![P]\!]$ and under-approximate the query $[\![q]\!]$. In general, the abstractions used for these two approximations may be different—for instance, this happens when a numerical abstract domain (like intervals or octagons) is used to infer variable bounds and a SMT solver is used to check the absence of buffer overruns. We are interested in the case where there is no under-approximation of the query, that is, we require: (i) the two abstract domains to coincide; and (ii) the query q to be exactly represented in this abstract domain, namely, $\gamma(\alpha([\![q]\!])) = [\![q]\!]$. For instance, a query like $x \geq y \wedge y \geq 0$ is exactly represented with octagons but not with intervals—the best approximation with intervals being $x \geq 0 \wedge y \geq 0$.

The key observation is that if a program query q is exactly representable in a store abstraction $A \in \mathrm{Abs}(\wp(\mathbb{S}))$ and the abstract semantics on A for a program P is complete, then answering the query q in the abstract is the same as answering q in the concrete, *i.e.*, no imprecision in answering q for P is introduced by the abstraction A. Thus, completeness of a static analysis of P implies that no false alarm can arise for queries which are representable in A. This is summarized by the following lemma, which comes straight from the definitions. If $[\![q]\!]^\alpha \triangleq \alpha([\![q]\!]) \in A$ is the abstraction of $[\![q]\!]$, q is *representable in* A when $[\![q]\!] = \gamma([\![q]\!]^\alpha)$.

Lemma 4.1. *If $A_\alpha \in \mathrm{Abs}(\wp(\mathbb{S}))$ is complete for P and q is representable in A then, for any $I \in \wp(\mathbb{S})$*

$$[\![P]\!]I \subseteq [\![q]\!] \Leftrightarrow [\![P]\!]^\alpha \alpha(I) \leq_A [\![q]\!]^\alpha.$$

Proof. Let $A_{\alpha,\gamma} \in \mathrm{Abs}(\wp(\mathbb{S}))$ be complete for P and q be representable in A.
($\Rightarrow$) By monotonicity of α, from $[\![P]\!]I \subseteq [\![q]\!]$ we obtain, by using the completeness and representability hypotheses, $\alpha([\![P]\!]I) = [\![P]\!]^\alpha \alpha(I) \leq_A \alpha([\![q]\!]) = \alpha(\gamma([\![q]\!]^\alpha)) \leq [\![q]\!]^\alpha$.
($\Leftarrow$) Since $[\![P]\!]^\alpha$ is always correct we have that $[\![P]\!]^\alpha \alpha(I) \leq_A [\![q]\!]^\alpha \Rightarrow \alpha([\![P]\!]I) \leq_A \alpha([\![q]\!])$. Thus, by applying γ to both sides of the inequality, we obtain $[\![P]\!]I \subseteq \gamma(\alpha([\![P]\!]I)) \subseteq \gamma(\alpha([\![q]\!])) = [\![q]\!]$. □

Note that the proof of $[\![P]\!]^\alpha \alpha(I) \leq_A [\![q]\!]^\alpha \Rightarrow [\![P]\!]I \subseteq [\![q]\!]$ does not make use of the completeness hypothesis, since this implication is a straight consequence of the soundness of the best correct approximation $[\![P]\!]^\alpha$. On the other hand, completeness is crucial for proving the reverse implication. Completeness can be therefore read as follows: if a complete static analysis of P raises an alarm in answering a representable query q then this alarm is surely real:

$$[\![P]\!]^\alpha \alpha(I) \not\leq_A [\![q]\!]^\alpha \Rightarrow [\![P]\!]I \not\subseteq [\![q]\!].$$

Example 4.2. Consider the program *DecInc* in Figure 4 and the final query $q \equiv y < 9$, which is representable by the intervals $\langle x \in (-\infty, +\infty), y \in (-\infty, 8]\rangle$, and therefore by an octagon as well.

The static analysis of *DecInc* on Int raises an alarm for q, meaning that $[\![\mathit{DecInc}]\!]^{\mathsf{Int}}\top_{\mathsf{Int}} = \langle x \in [0,0], y \in [0,+\infty)\rangle \not\leq_{\mathsf{Int}} [\![y < 9]\!]$. However, since Int is not complete for *DecInc*, Lemma 4.1 does not allow us to conclude that this is a false alarm. On the other hand, the static analysis on Oct also raises an alarm for q, because $[\![\mathit{DecInc}]\!]^{\mathsf{Oct}}\top_{\mathsf{Oct}} = \langle x \in [0,0], y \in [9,9]\rangle \not\leq_{\mathsf{Oct}} [\![y < 9]\!]$, and in this case Lemma 4.1 tells us that this is a real alarm. □

Classes of Completeness. From Lemma 4.1 it is therefore natural to reason on the precision of an abstraction A in terms of the set of programs that are complete for the program properties represented by A. This is the class of programs which represents at best (and uniquely) the potential of an abstract domain in answering questions on their behavior. We therefore introduce the notion of completeness class as a mapping from abstract domains to sets of programs: those for which the abstraction is complete. An analogous completeness class is defined for arithmetic expressions and Boolean predicates.

Definition 4.3 (Completeness Class)**.** Given $A_\alpha \in \mathrm{Abs}(\wp(\mathbb{S}))$, its *completeness class* $\mathbb{C}(A)$ (for commands) is defined as:

$$\mathbb{C}(A) \triangleq \{P \in \mathsf{Imp} \mid \alpha([\![P]\!]) = [\![P]\!]^\alpha\}.$$

Similarly, the completeness classes of A for arithmetic, Boolean, and generic expressions are:

$$\mathbb{A}(A) \triangleq \{a \in \mathrm{AExp} \mid \alpha([\![a]\!]) = [\![a]\!]^\alpha\}$$
$$\mathbb{B}(A) \triangleq \{b \in \mathrm{BExp} \mid \alpha([\![b]\!]) = [\![b]\!]^\alpha\}. \qquad \square$$

Roughly, the completeness class $\mathbb{C}(A)$ is defined to be the set of all programs whose static analysis on a given abstraction A will never produce false alarms for queries representable in A. This is therefore a property of programs with respect to a fixed abstraction. It is worth noting that this property is infinite and non-extensional (cf. [23]). It is infinite because for any abstract domain A whose abstraction function is computable we have that $|\mathbb{C}(A)| = \omega$. This is shown by a straightforward padding argument by observing that **skip** $\in \mathbb{C}(A)$ for any A and because sequential composition of complete commands is still complete. It is also non-extensional, *i.e.*, it is not an index set for partial recursive functions, because there always exist programs P and Q such that: P is complete for A, $[\![P]\!] = [\![Q]\!]$, and Q is not complete for A. This phenomenon is known in static analysis where semantics preserving program transformations may lose precision of analyses (e.g., see [12, 17]).

Example 4.4. Consider the abstract domain for sign analysis of integer variables $\mathsf{Sign} = \{\mathbb{Z}, -, 0, +, \varnothing\}$, which is a straightforward abstraction of $\langle\wp(\mathbb{Z}), \subseteq\rangle$. We consider the standard nonrelational lifting of Sign to a store abstraction in $\mathrm{Abs}(\langle\wp(\mathbb{S}), \subseteq\rangle)$. Consider the following two programs:

$$P \equiv y := 2; z := 3; x := y * z$$
$$Q \equiv y := 2; z := 3; x := y * (z - 1) + y$$

Obviously, we have that $[\![P]\!] = [\![Q]\!]$. However, since Sign is complete for multiplication and incomplete for addition, it turns out that

$$[\![x := y * z]\!]^{\mathsf{Sign}}\langle x/\mathbb{Z},\ y/+, z/+\rangle = \langle x/+, y/+, z/+\rangle$$
$$[\![x := y * (z-1) + y]\!]^{\mathsf{Sign}}\langle x/\mathbb{Z},\ y/+, z/+\rangle = \langle x/\mathbb{Z}, y/+, z/+\rangle$$

so that

$$[\![P]\!]^{\mathsf{Sign}}\varnothing = \langle x/+, y/+, z/+\rangle, \quad [\![Q]\!]^{\mathsf{Sign}}\varnothing = \langle x/\mathbb{Z}, y/+, z/+\rangle.$$

Thus, $P \in \mathbb{C}(\mathsf{Sign})$ while $Q \notin \mathbb{C}(\mathsf{Sign})$. □

It turns out that the relative precision of abstract domains—encoded by the ordering $\sqsubseteq$ on the lattice of abstractions—and the corresponding classes of completeness are not related. In particular, it may well happen that for an abstraction A which is complete for a given P, a generic refinement A^{ref} of A, *i.e.*, $A^{\mathrm{ref}} \sqsubseteq A$, turns out to be instead incomplete for the same program P. This phenomenon is well known in static program analysis and it corresponds to the fact that coarse abstractions may induce a complete static analysis for some program where more precise ones instead fail for that same program. In the following, we assume that the set $\mathbb{S}$ of all stores is infinite. If we consider a *trivial store abstraction*, *i.e.*, there is no abstraction at all or any set of stores is abstracted into a single (top) value corresponding to the "don't know" answer, then the corresponding completeness class turns out to be the whole set of programs Imp. Let $id \in \mathrm{Abs}(\wp(\mathbb{S}))$ denote the trivial identical store abstraction (*i.e.*, for any S, $id(S) = S$) and $\top_{\mathbb{S}} \in \mathrm{Abs}(\wp(\mathbb{S}))$ denote the trivial "don't know" store abstraction (*i.e.*, for any S, $\top_{\mathbb{S}}(S) = \mathbb{S}$). Also, let $\wp^{\mathrm{re}}(\mathbb{S})$ denote the set of all recursively enumerable subsets of $\mathbb{S}$. Let us point out that any static program analysis always relies on recursive, namely decidable, abstractions $A_\alpha \in \mathrm{Abs}(\wp(\mathbb{S}))$, meaning that: (i) for any store $\rho \in \mathbb{S}$, $\alpha(\{\rho\})$ is computable; and (ii) for any $S \in \wp^{\mathrm{re}}(\mathbb{S})$ and $\rho \in \mathbb{S}$, $\rho \in^? \alpha(S)$ is decidable.

Theorem 4.5. *If $A \in \mathrm{Abs}(\wp(\mathbb{S}))$ is recursive then $\mathbb{C}(A) = \mathsf{Imp}$ iff $A \in \{id, \top_{\mathbb{S}}\}$.*

Proof. The trivial abstractions id and $\top_{\mathbb{S}}$ are complete for any program, so that if $A = id$ or $A = \top_{\mathbb{S}}$ then $\mathbb{C}(A) = \mathsf{Imp}$. Assume now that $\mathbb{C}(A) = \mathsf{Imp}$ and $A \neq id$ and $A \neq \top_{\mathbb{S}}$. Hence, there exists some $S \in \wp^{\mathrm{re}}(\mathbb{S})$ such that $S \subsetneq \gamma(\alpha(S)) \subsetneq \mathbb{S}$. Since A is a decidable abstraction, we have that S and $\gamma(\alpha(S))$ are recursively enumerable. Hence, there exist programs P_S and $P_{\alpha(S)}$ on a single integer variable x such that $[\![P_S]\!]\mathbb{S} = S$ and $[\![P_{\alpha(S)}]\!]\mathbb{S} = \gamma(\alpha(S))$. We pick some $b \in \mathbb{S} \smallsetminus \gamma(\alpha(S))$ and some $c \in \gamma(\alpha(S)) \smallsetminus S$. Then, let us consider the program Q_{bc} associated with the following partial recursive function $\psi_{bc} : \mathbb{S} \to \mathbb{S}$:

$$\psi_{bc}(x) \triangleq \begin{cases} x & \text{if } x \in S \\ b & \text{if } x = c \\ \text{undefined} & \text{otherwise} \end{cases}$$

Observe that if $x \notin S$ then $[\![Q_{bc}]\!]\{x\} \neq \{x\}$, since $c \neq b$. Then, it turns out that $\alpha([\![Q_{bc}]\!]S) = \alpha(S)$ and $\alpha([\![Q_{bc}]\!]\gamma(\alpha(S))) = \alpha(S \cup \{b\})$. Moreover, since $S \subsetneq \gamma(\alpha(S)) \subseteq \gamma(\alpha(S \cup \{b\}))$, by GC, we obtain that $\alpha(S) \subsetneq \alpha(S \cup \{b\})$. Thus, we have shown that there exists a set of stores $S \in \wp^{\mathrm{re}}(\mathbb{S})$ such that:

$$\alpha([\![Q_{bc}]\!]S) \subsetneq \alpha([\![Q_{bc}]\!]\gamma(\alpha(S))) \leq [\![Q_{bc}]\!]^\alpha\alpha(S).$$

This means[1] that $Q_{bc} \notin \mathbb{C}(A)$ which contradicts the hypothesis that $\mathbb{C}(A) = \mathsf{Imp}$. □

Informally, the result above states that for all nontrivial abstractions there exists a program for which the abstraction is incomplete. Moreover, by a straightforward padding argument, any of these programs (e.g., Q_{bc} in the proof of Theorem 4.5) can be extended to an infinite set of programs for which the abstraction is incomplete. This means that any nontrivial abstraction has an infinite set of programs for which it is incomplete. We show that $\mathbb{C}(A)$ and its complement $\overline{\mathbb{C}(A)}$, for any nontrivial abstraction A, are *productive* sets. Recall that a set S is productive if there exists a general effective method (*i.e.*, a total recursive function) which enables us to find, for any recursively enumerable subset $Y \subseteq X$, an element

[1] Note that, for generic recursively enumerable sets S and $\gamma(\alpha(S))$ such that $S \subsetneq \gamma(\alpha(S))$, the set $\gamma(\alpha(S)) \smallsetminus S$ may not be recursively enumerable. This means that (due to the $c \in \gamma(\alpha(S)) \smallsetminus S$) the program Q_{bc} exists but we may not have a constructive computable way for building it. The mere existence of Q_{bc} as a mathematical object is enough for the purpose of this proof.

$x \in X \smallsetminus Y$ (see [10, 23]). It follows that no productive set can be recursively enumerable. This proves that, for a nontrivial abstraction A, both $\mathbb{C}(A)$ and its complement $\overline{\mathbb{C}(A)}$ are intrinsically nonrecursively enumerable sets having a structure which is similar to the set of Gödel numbers of true sentences in first-order arithmetics.

Theorem 4.6. *If $A \in \mathrm{Abs}(\wp(\mathbb{S}))$ is nontrivial and recursive then $\mathbb{C}(A)$ and $\overline{\mathbb{C}(A)}$ are productive sets.*

Proof. In the following, without loss of generality, we assume that programs in Imp have a single variable ranging over $\mathbb{N}$, so that $\mathbb{S} = \mathbb{N}$. If P is a program and $S \in \wp(\mathbb{S})$, we denote by $[\![P]\!](S)^{\downarrow \leq n}$ the fact that P terminates with any input in S in less than n steps. Let $\mathfrak{g} : \mathsf{Imp} \to \mathbb{N}$ be an enumeration of programs. This induces a corresponding enumeration $\mathfrak{g} : \wp^{\mathrm{re}}(\mathbb{S}) \to \mathbb{N}$ of recursively enumerable sets. Consider the representation in Imp of the halting problem of Turing machines (cf. [23]):

$$K \triangleq \{P \in \mathsf{Imp} \mid \exists n \in \mathbb{N}. [\![P]\!](\{\mathfrak{g}(P)\})^{\downarrow \leq n}\}.$$

We first prove that $\mathbb{C}(A)$ is productive. The proof relies on a many-to-one reduction of $\overline{K}$ to $\mathbb{C}(A)$, denoted by $\overline{K} \preceq_m \mathbb{C}(A)$, and means that there exists a total recursive function $f : \mathsf{Imp} \to \mathsf{Imp}$ such that $x \in \overline{K}$ iff $f(x) \in \mathbb{C}(A)$.

Let $A_{\alpha,\gamma} \in \mathrm{Abs}(\wp(\mathbb{S}))$ be recursive and nontrivial. As in Theorem 4.5, there exists $S \in \wp^{\mathrm{re}}(\mathbb{S})$ such that $S \subsetneq \gamma(\alpha(S)) \subsetneq \mathbb{S}$. Let $b \in \mathbb{S} \smallsetminus \gamma(\alpha(S))$, $c \in \gamma(\alpha(S)) \smallsetminus S$, and consider the program Q_{bc} as in the proof of Theorem 4.5, where we proved that $Q_{bc} \notin \mathbb{C}(A)$. Since α is recursive, there exists $P^{\top} \in \mathsf{Imp}$ such that $\alpha([\![P^{\top}]\!]\varnothing) = \mathbb{S}$. By monotonicity of α, for any $X \in \wp(\mathbb{S})$ we have that $\alpha([\![P^{\top}]\!]X) = \mathbb{S}$. Clearly, $P^{\top} \in \mathbb{C}(A)$ holds. Consider the partial recursive function $\psi : \mathsf{Imp} \times \wp^{\mathrm{re}}(\mathbb{S}) \to \wp^{\mathrm{re}}(\mathbb{S})$:

$$\psi(P, X) \triangleq \begin{cases} [\![Q_{bc}]\!](X) & \text{if } [\![P]\!](\{\mathfrak{g}(P)\})^{\downarrow \leq \mathfrak{g}(X)} \\ [\![P^{\top}]\!](X) & \text{otherwise} \end{cases}$$

Since ψ is partial recursive, there exists a program $R \in \mathsf{Imp}$ such that $[\![R]\!](P, X) = \psi(P, X)$. By the s-m-n Theorem, there exists a total recursive function $f : \mathsf{Imp} \times \mathsf{Imp} \to \mathsf{Imp}$ such that for any $P \in \mathsf{Imp}$ and $X \in \wp^{\mathrm{re}}(\mathbb{S})$ we have that $[\![f(R, P)]\!](X) = [\![R]\!](P, X) = \psi(P, X)$. Consider a generic program $P \in \mathsf{Imp}$. If $P \in K$ then there exists $n \in \mathbb{N}$ such that $[\![P]\!](\{\mathfrak{g}(P)\})^{\downarrow \leq n}$. Let $\mathcal{I}^{\alpha}_{Q_{bc}}$ denote the set of all sets of stores for which A_{α} is not complete for Q_{bc}:

$$\mathcal{I}^{\alpha}_{Q_{bc}} \triangleq \{X \in \wp^{\mathrm{re}}(\mathbb{S}) \mid \alpha([\![Q_{bc}]\!]X) \neq [\![Q_{bc}]\!]^{\alpha}\alpha(X)\}.$$

It is easy to see that $\mathcal{I}^{\alpha}_{Q_{bc}}$ is infinite. In particular, as shown in the proof of Theorem 4.5, $S \in \mathcal{I}^{\alpha}_{Q_{bc}}$. It is also easy to derive that for any $x \in \mathbb{S}$ such that $x \neq c$ we have that $S \cup \{x\} \in \mathcal{I}^{\alpha}_{Q_{bc}}$. Therefore, since $|\,\mathbb{S}\,| = \omega$, we also have that $|\mathcal{I}^{\alpha}_{Q_{bc}}| = \omega$. This implies that there exists $X \in \mathcal{I}^{\alpha}_{Q_{bc}}$ such that $n \leq \mathfrak{g}(X)$ and therefore $[\![P]\!](\{\mathfrak{g}(P)\})^{\downarrow \leq \mathfrak{g}(X)}$. Hence, $[\![f(R, P)]\!](Y) = [\![Q_{bc}]\!](Y)$ for some $Y \in \mathcal{I}^{\alpha}_{Q_{bc}} \subset \wp(\mathbb{S})$. Therefore, $f(R, P) \notin \mathbb{C}(A)$. If $P \notin K$ then, for any $n \in \mathbb{N}$, we have that $[\![P]\!](\{\mathfrak{g}(P)\})$ does not converge in less than n steps. Therefore, for any $X \in \wp^{\mathrm{re}}(\mathbb{S})$ we have that $\psi(P, X) = [\![P^{\top}]\!]X$ and therefore $[\![f(R, P)]\!] = [\![P^{\top}]\!]$, which implies that $f(R, P) \in \mathbb{C}(A)$.

Because $\lambda P \in \mathsf{Imp}.\, f(R, P)$ above is total recursive, we have that $K \preceq_m \overline{\mathbb{C}(A)}$, which is equivalent to $\overline{K} \preceq_m \mathbb{C}(A)$. This proves that $\mathbb{C}(A)$ is productive (see Theorem VII §7.5 in [23] and Theorem 4.5 in [24]). The proof that also $\overline{\mathbb{C}(A)}$ is productive is analogous and it is based on a many-to-one reduction $K \preceq_m \mathbb{C}(A)$ which exploits the partial recursive function:

$$\psi'(P, X) \triangleq \begin{cases} [\![P^{\top}]\!](X) & \text{if } [\![P]\!](\{\mathfrak{g}(P)\})^{\downarrow \leq \mathfrak{g}(X)} \\ [\![Q_{bc}]\!](X) & \text{otherwise} \end{cases}$$

□

As a consequence, neither $\mathbb{C}(A)$ nor its complement $\overline{\mathbb{C}(A)}$ turn out to be recursively enumerable sets. Let us notice that the proof of the previous theorem provides a further insight into the structure of $\mathbb{C}(A)$. Given a nontrivial and recursive abstraction $A \in \mathrm{Abs}(\wp(\mathbb{S}))$, it is always possible to systematically and effectively transform any algorithmic procedure attempting to enumerate $\mathbb{C}(A)$ into a program P such that $P \notin \mathbb{C}(A)$. The same holds for $\overline{\mathbb{C}(A)}$. This can be read as follows: automating the proof that an abstraction is complete or incomplete for a given program—*i.e.*, a static program analysis can produce or cannot produce false alarms—is impossible. The completeness class of an abstraction is therefore a nontrivial property of programs for which no recursively enumerable procedure may exist which is able to enumerate all of its elements. In the following, we provide recursively enumerable under-approximations of $\mathbb{C}(A)$, namely systematic, and possibly automatic, proof systems such that if the proof succeeds for some program P and nontrivial abstraction A, then $P \in \mathbb{C}(A)$.

5. Proving Completeness of Programs

Given a store abstraction $A \in \mathrm{Abs}(\wp(\mathbb{S}))$, our goal is to prove whether $P \in \mathbb{C}(A)$. To this aim, we design a layered proof system parametric on the abstraction A. The first layer is a compositional proof system $\vdash_A$ for a generic abstraction A. It deals with the fundamental constructs for program composition: sequential composition, branching, and guarded iteration. The second layer is instead domain-specific. It deals with the completeness of the assignments and of the Boolean guards — The handling of the assignments depends on whether the store abstraction is relational or not.

Let us remark that the abstract domain A here is fixed. An approach based on complete shell refinements (or complete core simplifications) of the abstraction A, as recalled in Section 2, would therefore be orthogonal. A complete refinement of A for all the assignments and Boolean guards (and their negations) occurring in P would generate an abstract domain which is complete for all the programs that can be constructed by combining through sequential compositions, conditionals and loops, and for all the possible sets of input stores—in most practical cases, the abstraction A would likely be refined to the concrete domain.

5.1 The Core Proof System

We report in Figure 5 the core proof system $\vdash_A$ for proving the completeness w.r.t. a generic store abstraction $A_{\alpha,\gamma} \in \mathrm{Abs}(\wp(\mathbb{S}))$. This deductive system is fully compositional on the program's syntax. A no-op command is trivially complete for all abstractions ([**skip**]). The rule [**seq**] states that the sequential composition of two complete commands is complete, too. It is worth remarking that the composition of the best correct approximations of two functions f and g, in general, is not the best correct approximation of their composition $f \circ g$, while this indeed holds when these best correct approximations of f and g are indeed complete. The rule [**if**] asserts that a conditional statement is complete when its body, its Boolean guard *together with* its negation are complete. Loop statements are handled by the rule [**while**]: Iteration is complete if its body is complete as well as the loop guard and its negation. It is worth remarking that the rules for branching and loop commands imply that the abstract join operation *never* injects incompleteness in the analysis, unlike a common belief in static program analysis. Let us show that these rules are sound for proving completeness in A.

Theorem 5.1. *For any $A \in \mathrm{Abs}(\wp(\mathbb{S}))$ the rules in $\vdash_A$ are sound.*

Proof. [**skip**]: **skip** $\in \mathbb{C}(A)$ always holds.

$$\frac{}{\vdash_A \textbf{skip}}\ [\textbf{skip}] \qquad \frac{\vdash_A P \quad \vdash_A Q}{\vdash_A P;Q}\ [\textbf{seq}] \qquad \frac{\vdash_A C \quad b \in \mathbb{B}(A) \quad \neg b \in \mathbb{B}(A)}{\vdash_A \textbf{if } b \textbf{ then } C}\ [\textbf{if}] \qquad \frac{\vdash_A C \quad b \in \mathbb{B}(A) \quad \neg b \in \mathbb{B}(A)}{\vdash_A \textbf{while } b \textbf{ do } C}\ [\textbf{while}]$$

Figure 5. The core proof system $\vdash_A$.

[**seq**]: If A is complete for C_1 and C_2 then A is complete for $C_1; C_2$:

$$\begin{aligned}\alpha([\![C_1;C_2]\!]S) &= \alpha([\![C_2]\!]([\![C_1]\!]S))\\ &= [\![C_2]\!]^\alpha\alpha([\![C_1]\!]S))\\ &= [\![C_2]\!]^\alpha[\![C_1]\!]^\alpha\alpha(S)\\ &= [\![C_1;C_2]\!]^\alpha\alpha(S).\end{aligned}$$

[**if**]: If A is complete for b, $\neg b$ and C then A is complete for **if** b **then** C:

$$\begin{aligned}\alpha([\![\textbf{if } b \textbf{ then } C]\!]S) &= \alpha([\![C]\!][\![b]\!]S \cup [\![\neg b]\!]S)\\ &= \alpha([\![C]\!][\![b]\!]S) \sqcup \alpha([\![\neg b]\!]S)\\ &= [\![C]\!]^\alpha\alpha([\![b]\!]S) \sqcup [\![\neg b]\!]^\alpha\alpha(S)\\ &= [\![C]\!]^\alpha[\![b]\!]^\alpha\alpha(S) \sqcup [\![\neg b]\!]^\alpha\alpha(S)\\ &= [\![\textbf{if } b \textbf{ then } C]\!]^\alpha\alpha(S).\end{aligned}$$

[**while**]: If A is complete for b, $\neg b$ and C then A is complete for **while** b **do** C. We first show that

$$\alpha \circ (\lambda T.S \cup [\![C]\!][\![b]\!]T) = (\lambda X^\sharp.\alpha(S) \sqcup [\![C]\!]^\alpha[\![b]\!]^\alpha X^\sharp) \circ \alpha$$

For any $T \in \wp(\mathbb{S})$, we have that:

$$\begin{aligned}\alpha(S \cup [\![C]\!][\![b]\!]T) &= \alpha(S) \sqcup \alpha([\![C]\!][\![b]\!]T)\\ &= \alpha(S) \sqcup [\![C]\!]^\alpha\alpha([\![b]\!]T)\\ &= \alpha(S) \sqcup [\![C]\!]^\alpha[\![b]\!]^\alpha\alpha(T),\end{aligned}$$

so that

$$\alpha \circ (\lambda T.S \cup [\![C]\!][\![b]\!]) = (\lambda X^\sharp.\alpha(S) \sqcup [\![C]\!]^\alpha[\![b]\!]^\alpha X^\sharp) \circ \alpha.$$

Hence, by fixpoint transfer (cf. Section 2):

$$\alpha(\mathsf{lfp}(\lambda T.S \cup [\![C]\!][\![b]\!]T)) = \mathsf{lfp}(\lambda X^\sharp.\alpha(S) \sqcup [\![C]\!]^\alpha[\![b]\!]^\alpha X^\sharp).$$

We therefore obtain:

$$\begin{aligned}\alpha([\![\textbf{while } b \textbf{ do } C]\!]S) &= \alpha([\![\neg b]\!](\mathsf{lfp}(\lambda T.S \cup [\![C]\!][\![b]\!]T)))\\ &= [\![\neg b]\!]^\alpha\alpha(\mathsf{lfp}(\lambda T.S \cup [\![C]\!][\![b]\!]T))\\ &= [\![\neg b]\!]^\alpha\mathsf{lfp}(\lambda X^\sharp.\alpha(S) \sqcup [\![C]\!]^\alpha[\![b]\!]^\alpha X^\sharp)\\ &= [\![\textbf{while } b \textbf{ do } C]\!]^\alpha\alpha(S).\end{aligned}$$ □

Let us point out that for a Boolean guard b occuring in a program P, the rules of $\vdash_A$ require both b and $\neg b$ to be complete on A for any possible set of input stores. It is therefore important to remark that Boolean guards are a major source of incompleteness, even in seemingly innocuous cases, as shown by the following example:

Example 5.2. Let us consider the loop guard in the program *Dec* in Figure 2. Assume that the body of the while loop $x := x - 1$ is complete for the interval abstraction Int—this will be formally proved later on. The proof of completeness for *Dec* in $\vdash_{\mathsf{Int}}$ would need the hypotheses $x > 0 \in^? \mathbb{B}(\mathsf{Int})$ and $x \leq 0 \in^? \mathbb{B}(\mathsf{Int})$ stating the completeness of the guards of *Dec*. However, is is not true, in general, that intervals are complete for a guard like $x > 0$, even if $x > 0$ is exactly representable in Int. In fact, we have that:

$$\begin{aligned}\alpha_{\mathsf{Int}}([\![x > 0]\!]\{0,2,3\}) &= \alpha_{\mathsf{Int}}(\{2,3\}) = [2,3]\\ &\sqsubset [\![x > 0]\!]^{\alpha_{\mathsf{Int}}}\alpha_{\mathsf{Int}}(\{0,2,3\})\\ &= [\![x > 0]\!]^{\alpha_{\mathsf{Int}}}[0,3] = [1,3].\end{aligned}$$

Therefore, $x > 0 \notin \mathbb{B}(\mathsf{Int})$. A similar counterexample may show that $x \leq 0 \notin \mathbb{B}(\mathsf{Int})$.

Let us also observe that even a simple equality test $[\![x = y]\!]$ between different variables cannot be complete in a relational abstraction such as octagons Oct which is able to represent precisely a variable relation like $x = y$. It is enough to note that:

$$\begin{aligned}\alpha_{\mathsf{Oct}}([\![x = y]\!]\{(x/0, y/2), (x/2, y/0)\}) &= \alpha_{\mathsf{Oct}}(\varnothing) = \bot_{\mathsf{Oct}}\\ &\sqsubset [\![x = y]\!]^{\alpha_{\mathsf{Oct}}}\alpha_{\mathsf{Oct}}(\{(x/0, y/2), (x/2, y/0)\})\\ &= [\![x = y]\!]^{\alpha_{\mathsf{Oct}}}\langle 0 \leq x \leq 2,\ 0 \leq y \leq 2, x + y = 2\rangle\\ &= \langle x = 1, y = 1\rangle.\end{aligned}$$

A similar example may show that the test $[\![x = y]\!]$ is not complete for intervals as well, although this could be somewhat expected since intervals are not relational. □

5.2 Proving Completeness of Assignments

Assignment commands are not handled by the core proof system $\vdash_A$. The deductive system $\vdash_A$ is compositional on program's syntax and the problem for assignments stems from the fact a compositional rule for deriving the completeness of $x := a$ from the completeness of the expression a (and/or $x - a$) cannot be sound for a generic abstract domain A. This is shown by the following example dealing with the relational octagon abstraction.

Example 5.3. Let $Var = \{x, y, z\}$ and let us represent a generic store $(x/v_x, y/v_y, z/v_z)$ simply by $(v_x, v_y, v_z) \in \mathbb{Z}^3$. We consider the arithmetic expression $x + y \in \mathrm{AExp}$ and the abstraction Oct. It turns out that $x + y \in \mathbb{A}(\mathsf{Oct})$. In fact, for any nonempty set of stores $S \in \wp(\mathbb{S})$, consider the constraint $m \leq x + y \leq n$ which is expressed by $\mathsf{Oct}(S)$: this means that there exist $\rho_1, \rho_2 \in S$ such that $m = \rho_1(x) + \rho_1(y)$, $n = \rho_2(x) + \rho_2(y)$, and for any $\rho \in S$, $m \leq \rho(x) + \rho(y) \leq n$. Thus, $\mathsf{Oct}([\![x + y]\!]S) = [m, n]$. On the other hand, since, for any $\rho \in \mathsf{Oct}(S)$, $m \leq \rho(x) + \rho(y) \leq n$, we also have that $\mathsf{Oct}([\![x + y]\!]\mathsf{Oct}(S)) = [m, n]$.

Let us consider now $S = \{(2,1,0), (1,4,2)\} \in \wp(\mathbb{S})$ and the assignment $z := x + y \in \mathsf{Imp}$, whose concrete semantics on S gives $[\![z := x + y]\!]S = \{(2,1,3), (1,4,5)\}$. The abstraction of $[\![z := x + y]\!]S$ in Oct is therefore as follows:

$$\begin{aligned}\mathsf{Oct}([\![z := x + y]\!]S) = \langle &1 \leq x \leq 2, 1 \leq y \leq 4, 3 \leq z \leq 5,\\ &3 \leq x + y \leq 5, -3 \leq x - y \leq 1, 5 \leq x + z \leq 6,\\ &-4 \leq x - z \leq -1, 4 \leq y + z \leq 9, -2 \leq y - z \leq -1\rangle.\end{aligned}$$

On the abstract side, the abstraction of S in Oct is as follows:

$$\begin{aligned}\mathsf{Oct}(S) = \langle &1 \leq x \leq 2, 1 \leq y \leq 4, 0 \leq z \leq 2,\\ &3 \leq x + y \leq 5, -3 \leq x - y \leq 1, 2 \leq x + z \leq 3,\\ &-1 \leq x - z \leq 2, 1 \leq y + z \leq 6, 1 \leq y - z \leq 2\rangle.\end{aligned}$$

Here $(2,3,1) \in \mathsf{Oct}(S)$ and $(2,3,5) \in [\![z := x + y]\!]\mathsf{Oct}(S) \subseteq \mathsf{Oct}([\![z := x + y]\!]\mathsf{Oct}(S))$. But $(2,3,5) \notin \mathsf{Oct}([\![z := x + y]\!]S)$ because the relation $5 \leq x + z \leq 6$ is not satisfied. Hence, $\mathsf{Oct}([\![z := x + y]\!]S) \subsetneq \mathsf{Oct}([\![z := x + y]\!]\mathsf{Oct}(S))$, namely $z := x + y \notin \mathbb{C}(\mathsf{Oct})$. □

Nonrelational Abstractions. An abstraction $A_\alpha \in \mathrm{Abs}(\wp(\mathbb{S}))$ is nonrelational when it does not take into account any relationship between different variables. Let us formalize this notion. For any

$x \in Var$, let $\bar{x} \triangleq Var \setminus \{x\}$ and let $\alpha^x : \wp(\mathbb{S}) \to A$ be defined as $\alpha^x(S) \triangleq \alpha(\exists_{\bar{x}}(S))$. Then, A is defined to be nonrelational when:

$$\forall S \in \wp(\mathbb{S}).\ \alpha(S) = \bigwedge_{x \in Var} \alpha^x(S).$$

Null, Sign and Int are examples of nonrelational abstractions, while Oct is not nonrelational. For a nonrelational abstraction A, we introduce the following compositional rule for assignments:

$$\frac{a \in \mathbb{A}(A)}{\vdash_A x := a} \ [\textbf{assignNR}]$$

We denote by $\vdash_A^{\mathbf{NR}}$ the core proof system $\vdash_A$ enhanced with the rule [**assignNR**] and we show its soundness for deriving completeness.

Theorem 5.4. *For any nonrelational abstraction $A \in \mathrm{Abs}(\wp(\mathbb{S}))$, $\vdash_A^{\mathbf{NR}}$ is sound,* i.e.*, if $\vdash_A P$ then $P \in \mathbb{C}(A)$.*

Proof. The soundness of the rules in $\vdash_A$ follows by Theorem 5.1. The soundness of [**assignNR**] follows from these equalities:

$$\begin{aligned}
&\alpha([\![x := a]\!]S) = \\
&\quad = \alpha^x([\![x := a]\!]S) \wedge \bigwedge_{y \neq x} \alpha^y([\![x := a]\!]S) \\
&\quad = \alpha(\exists_{\bar{x}}([\![x := a]\!]S)) \wedge \bigwedge_{y \neq x} \alpha(\exists_{\bar{y}}([\![x := a]\!]S)) \\
&\qquad\qquad \big[\text{by def. of variable projection and } \alpha^{\mathbb{Z}}\big] \\
&\quad = \alpha^{\mathbb{Z}}([\![a]\!]S) \wedge \bigwedge_{y \neq x} \alpha(\exists_{\bar{y}}(S)) \\
&\qquad\qquad \big[\text{by } a \in \mathbb{A}(A) \text{ and completeness of } \exists_{\bar{y}}\big] \\
&\quad = \alpha^{\mathbb{Z}}([\![a]\!]\gamma(\alpha(S))) \wedge \bigwedge_{y \neq x} \alpha(\exists_{\bar{y}}(\gamma(\alpha(S)))) \\
&\qquad\qquad \big[\text{by def. of variable projection and } \alpha^{\mathbb{Z}}\big] \\
&\quad = \alpha(\exists_{\bar{x}}([\![x := a]\!]\gamma(\alpha(S)))) \wedge \bigwedge_{y \neq x} \alpha(\exists_{\bar{y}}([\![x := a]\!]\gamma(\alpha(S)))) \\
&\quad = \alpha^x([\![x := a]\!]\gamma(\alpha(S))) \wedge \bigwedge_{y \neq x} \alpha^y([\![x := a]\!]\gamma(\alpha(S))) \\
&\quad = \alpha([\![x := a]\!]\gamma(\alpha(S))).
\end{aligned}$$

□

Thus, for nonrelational abstract domains A, $\vdash_A^{\mathbf{NR}}$ is a fully compositional proof system for checking completeness of programs. Let us give a simple example of derivation in $\vdash_A^{\mathbf{NR}}$.

Example 5.5. Let us consider the nonrelational abstract domain Null for nullness analysis described in Section 1 and the simple program

$$P \equiv x := 0; \textbf{if } (x = 0) \textbf{ then } x := 1$$

where, with a slight abuse of notation, we assume that 0 stands for a null reference while 1 for a non-null reference. The proof tree in Figure 6 can be derived in $\vdash_{\mathsf{Null}}^{\mathbf{NR}}$, and this entails that the abstract semantics of P on the abstraction Null is complete. Let us stress that the key point in proving the completeness of P is that the abstraction Null is complete for the Boolean guard $x = 0$ *and* its negation $\neg(x = 0)$. □

Nevertheless, for the program *Dec* in Figure 2, we are still not able to derive that $\vdash_{\mathsf{Int}}$ *Dec*: the rule [**assignNR**] allows us to derive that the assignment $x := x - 1$ is complete for Int, but this is not enough since the Boolean guard $x > 0$ is not complete (cf. Example 5.2). The fact that $\mathbb{C}(A)$ is a productive set even for (nontrivial) nonrelational abstractions implies that an effective complete proof system for deriving completeness of programs cannot be defined. The proof system $\vdash_A^{\mathbf{NR}}$ is indeed sound but not complete, and the program *Dec* shows this incompleteness.

Relational Abstractions. As shown in Example 5.3, the completeness of an expression a does not imply that an assignment $x := a$ is complete for a relational abstract domain. It can be shown that more restrictive hypotheses such as the assumption that $x \notin vars(a)$ and the completeness of the Boolean expression $x = a$ on A, would imply that $x := a$ is complete on A (the proof is omitted). However, this would yield a sound but far too restrictive derivation rule. Indeed, although the syntactic condition $x \notin vars(a)$ can always be met for any program P simply by replacing an assignment $x := a$ where $x \in vars(a)$ with the composition $x' := a; x := x'$, where x' is a fresh variable, this rule would require the hypothesis $(x = y) \in \mathbb{B}(A)$, and we observed in Example 5.2 that the equality test is not complete even for most known abstract domains.

Thus, for the generic case of a nonrelational abstraction A, it can be argued that a reasonable rule for deriving the completeness of assignments $x := a$ from the completeness of some other arithmetic expression and/or Boolean guard and/or command induced by $x := a$ simply cannot be found. Hence, a nonrelational abstraction therefore needs a specific analysis of the completeness of its assignments. We present a significant sample of this analysis for the case of the octagon abstraction.

Octagon Abstraction. In Example 5.3 we have shown that the assignment $z := x + y$ is not complete for Oct. A similar example can be found for $x := x + y$. These observations show that any generic linear assignment $x := a_1 * x_1 + \ldots + a_n * x_n$ is not complete for Oct when $n \geq 2$. The intuitive reason is that for a relation $a \leq x + y \leq b$, the substitution of x with $a_1 * x_1 + \ldots + a_n * x_n$ provides a relation for more than two variables which cannot be represented by Oct. A fortiori, this is also true for nonlinear assignments like $x := y * y$.

Thus, we are left to scrutinize linear assignments of the following shape: $x := a * y + k$ and $x := a * x + k$. If $a \notin \{-1, 0, 1\}$, *e.g.*, $x := 2 * y$, the following example shows that, as expected, we do not have completeness.

Example 5.6. We follow the notation of Example 5.3 where $Var = \{x, y, z\}$. Let us consider the assignment $x := 2 * y \in \mathsf{Imp}$ and the set of input stores $S = \{(0, 2, 0), (3, 0, 4)\} \in \wp(\mathbb{S})$. On the concrete side, we have that $[\![x := 2 * y]\!]S = \{(4, 2, 0), (0, 0, 4)\}$. The abstraction of $[\![x := 2 * y]\!]S$ in Oct is therefore as follows:

$$\begin{aligned}
\mathsf{Oct}([\![x := 2 * y]\!]S) = \\
&\langle 0 \leq x \leq 4, 0 \leq y \leq 2, 0 \leq z \leq 4, \\
&\ 0 \leq x + y \leq 6, 0 \leq x - y \leq 2, x + z = 4, \\
&\ -4 \leq x - z \leq 4, 2 \leq y + z \leq 4, -4 \leq y - z \leq 2\rangle.
\end{aligned}$$

On the abstract side, the abstraction of S in Oct is as follows:

$$\begin{aligned}
\mathsf{Oct}(S) = \langle &0 \leq x \leq 3, 0 \leq y \leq 2, 0 \leq z \leq 4, \\
&2 \leq x + y \leq 3, -2 \leq x - y \leq 3, 0 \leq x + z \leq 7, \\
&-1 \leq x - z \leq 0, 2 \leq y + z \leq 4, -4 \leq y - z \leq 2\rangle.
\end{aligned}$$

We consider the store $(1, 2, 2) \in \mathsf{Oct}(S)$ so that $(4, 2, 2) \in [\![x := 2 * y]\!]\mathsf{Oct}(S) \subseteq \mathsf{Oct}([\![x := 2 * y]\!]\mathsf{Oct}(S))$. However, we have that $(4, 2, 2) \notin \mathsf{Oct}([\![x := 2 * y]\!]S)$ because the relation $x + z = 4$ is not satisfied. This shows that $\mathsf{Oct}([\![x := 2 * y]\!]S) \subsetneq \mathsf{Oct}([\![x := 2 * y]\!]\mathsf{Oct}(S))$, *i.e.*, $x := 2 * y \notin \mathbb{C}(\mathsf{Oct})$. □

It turns out that the remaining assignments are the only which are complete for octagons.

Lemma 5.7. *The only complete assignments for* Oct *are: $x := a * y + k$ and $x := a * x + k$, where $a \in \{-1, 0, 1\}$ and $k \in \mathbb{Z}$.*

$$\dfrac{\dfrac{0 \in \mathbb{A}(\mathsf{Null})}{\vdash^{\mathbf{NR}}_{\mathsf{Null}} x := 0} \quad \dfrac{(x=0) \in \mathbb{B}(\mathsf{Null}) \quad \neg(x=0) \in \mathbb{B}(\mathsf{Null}) \quad \dfrac{1 \in \mathbb{A}(\mathsf{Null})}{\vdash^{\mathbf{NR}}_{\mathsf{Null}} x := 1}}{\vdash^{\mathbf{NR}}_{\mathsf{Null}} \textbf{if } (x=0) \textbf{ then } x := 1}}{\vdash^{\mathbf{NR}}_{\mathsf{Null}} P \equiv x := 0;\ \textbf{if}(x=0) \textbf{ then } x := 1}$$

Figure 6. The derivation tree proving the completeness of the program P on the abstraction Null.

Proof. Consider $x := a * y + k$ with $a \in \{-1, 0, 1\}$. By the characterization of completeness given in [13] and recalled in Section 2, we have that $[\![x := a * y + k]\!] \in \mathbb{C}(\mathsf{Oct})$ iff for any octagon $\mathsf{oct} \in \mathsf{Oct}$:

$$\max(\{T \in \wp(\mathbb{S}) \mid [\![x := a * y + k]\!]T \subseteq \mathsf{oct}\}) \in \mathsf{Oct}.$$

We observe that this maximal set of store can be obtained from oct by replacing in any relation involving the variable x the expression $ay + k$. If $a = 0$ then it is clear that this replacement still provides relations of the shape $\pm x \pm y \leq k$. It turns out that this also happens for $a \in \{-1, 1\}$. Let us consider the case $a = 1$, namely $x := y + k$. The following cases are possible. The replacement for $a \leq x \leq b$ provides the relation $a - k \leq y \leq b - k$; for $a \leq x - y \leq b$ the relation $a \leq k \leq b$, which can be true or false (and in this case we obtain the empty octagon); for $a \leq x + y \leq b$ the relation $a - k \leq 2y \leq b - k$, which is equivalent (integer variables) to $\lceil (a-k)/2 \rceil \leq y \leq \lfloor (b-k)/2 \rfloor$; for $a \leq x \pm z \leq b$, where $z \neq y$, we obtain $a - k \leq y \pm z \leq b - k$. Thus, in each case we obtain a variable relation of the shape $\pm x \pm y \leq k$, *i.e.*, $\max(\{T \in \wp(\mathbb{S}) \mid [\![x := y + k]\!]T \subseteq \mathsf{oct}\})$ is an octagon. A similar analysis applies to the remaining cases $x := -y + k$ and $x := \pm x + k$. □

Lemma 5.7 therefore provides an exhaustive (static) analysis of the completeness of assignments for the octagon abstraction. It is worthwhile to observe that [20, Figure 15] describes algorithms that compute the best correct approximations (called exact approximations in [20]) on octagons for precisely the set of complete assignments characterized by Lemma 5.7. This may suggest an interesting relationship between the property of being complete for an assignment and that of admitting a computable best correct approximation.

6. Proving Completeness of Guards

We argued that the main problem in proving $\vdash_A P$ lies in the fact that in general it is hard to prove that $b \in \mathbb{B}(A)$ and $\neg b \in \mathbb{B}(A)$ for an arbitrary store abstraction A. Actually, most of the times, only trivial guards—true, false, nondeterministic choices and, e.g., $x = \mathit{Null}$ in the nullness domain—turn out to be complete. For instance, one may expect that a Boolean condition which is exactly representable in some abstraction, as $x > 0$ in Int, turns out to be complete. However, in general, this is not the case, as shown by Example 5.2. Our goal is to take into account explicitly the Boolean guards of programs by adding suitable assumptions to completeness proofs in $\vdash_A$. Thus, we derive a conditional proof of completeness in A for a program P, whose assumptions on the completeness of the Boolean guards of P then need to be validated in a further distinct step.

Let us consider a store abstraction $A_{\alpha,\gamma} \in \mathrm{Abs}(\wp(\mathbb{S}))$ and a Boolean predicate $b \in \mathrm{BExp}$. We use the following notation: $[\![b]\!]^{\mathbf{t}} \triangleq \{\rho \in \mathbb{S} \mid (\!|b|\!)\rho = \mathbf{t}\}$. If $\alpha([\![b]\!]S) = [\![b]\!]^{\alpha}\alpha(S)$ holds for a given $S \in \wp(\mathbb{S})$ then b is called complete for A in S. Since this condition is equivalent to $\alpha([\![b]\!]S) = \alpha([\![b]\!]\gamma(\alpha(S)))$ and $[\![b]\!]S = [\![b]\!]^{\mathbf{t}} \cap S$, we have that b is complete for A in S when $\alpha([\![b]\!]^{\mathbf{t}} \cap S) = \alpha([\![b]\!]^{\mathbf{t}} \cap \gamma(\alpha(S)))$.

Now, let us consider some $b \in \mathrm{BExp}$ which is representable in the abstract domain A, that is, $[\![b]\!]^{\mathbf{t}} = \gamma(\alpha([\![b]\!]^{\mathbf{t}}))$ holds — we call them A-Boolean expressions. For example, $x \leq k_1 \wedge y > k_2$ is a Int- and Oct-Boolean expression, while $k \leq x \wedge x \leq y$ is a Oct-Boolean expression but not a Int-Boolean expression. In this case, it turns out that

$$\begin{aligned}\alpha([\![b]\!]^{\mathbf{t}} \cap \gamma(\alpha(S))) &= \alpha(\gamma(\alpha([\![b]\!]^{\mathbf{t}})) \cap \gamma(\alpha(S))) \\ &= \alpha(\gamma(\alpha([\![b]\!]^{\mathbf{t}}) \wedge_A \alpha(S))) = \alpha([\![b]\!]^{\mathbf{t}}) \wedge_A \alpha(S)\end{aligned}$$

so that completeness of b for A in S corresponds to require:

$$\alpha([\![b]\!]^{\mathbf{t}} \cap S) = \alpha([\![b]\!]^{\mathbf{t}}) \wedge_A \alpha(S).$$

We instrument our proof system $\vdash_A$ with suitable assumptions that guarantee that the sets of possible input states for a Boolean guard make it complete. This allows us to derive a conditional proof of completeness for a program P under these assumptions on the completeness of the Boolean guards of P. These proof assumptions can then be proved in a later step, *e.g.* by a static analysis focused on Boolean guards. Let $\mathrm{BExp}(P)$ denote the set of Boolean guards occurring in some program point of P and assume that the set Γ_P of Boolean guards of P which are not complete on A, *i.e.*, $\Gamma_P \triangleq \{b \in \mathrm{BExp}(P) \mid b \notin \mathbb{B}(A)\}$, consists of A-Boolean expressions only. If $\not\vdash_A P$ then the completeness proof for P may fail along some guard $b \in \Gamma_P$. If we are able to guarantee that this guard b is complete for any set S of possible input stores at the program point where the guard b occurs in P then we can safely conclude that b is complete on A for the purpose of proving that P is complete on A. We therefore add the following conditional meta-rule [**gc**] for the completeness of guards $b \in \Gamma_P$:

$$\dfrac{\mathit{assume}[S : \alpha([\![b]\!]^{\mathbf{t}} \cap S) = \alpha([\![b]\!]^{\mathbf{t}}) \wedge_A \alpha(S)]}{b \in \mathbb{B}(A)}\ [\mathbf{gc}]$$

Hence, a conditional completeness proof of P in $\vdash_A \cup \{[\mathbf{gc}]\}$ depends on the collection $\mathcal{G}_P$ of all the assumptions made for the guards $b \in \Gamma_P$. The next step consists in designing some domain-specific—possibly statically checkable—conditions that allow to validate all the assumptions in $\mathcal{G}_P$ made in a conditional proof of P, so as to establish an unconditional proof for P on A.

6.1 Completeness of Int- and Oct-Boolean Guards

Given a set of store, we want to provide a characterization of the completeness of Boolean expressions which are representable as intervals or octagons. Namely, we characterize the sets of stores S that make a Int- and Oct-Boolean expression complete, hence providing specific conditions for the assumptions of the conditional rule [**gc**] for Int and Oct.

In order to keep the notation as simple as possible, in what follows we consider the abstractions Int and Oct on stores over two variables (x, y), *i.e.*, $\mathbb{S} = \mathbb{Z}^2$ and $\mathsf{Int}, \mathsf{Oct} \in \mathrm{Abs}(\langle \wp(\mathbb{Z}^2), \subseteq \rangle)$. Moreover, for any set of concrete points $S \in \wp(\mathbb{Z}^2)$, we use $\mathsf{Int}(S), \mathsf{Oct}(S) \in \wp(\mathbb{Z}^2)$ to denote the corresponding interval and octagon abstractions. The generalization to $N > 2$ variables is conceptually simple but notationally tedious. In two variables, a (possibly infinite) interval is a (possibly infinite) rectangle R in the

(x, y) plane. A rectangle R can be represented by its set of edges, denoted by $\mathrm{edges}(R)$, which are at most four (this is the case of finite rectangles). Similarly, an octagon O in two variables can be represented by its edges in $\mathrm{edges}(O)$, which are at most eight. Any $E \in \mathrm{edges}(R)$ determines a line in the (x, y) integer plane which is denoted by l_E. If l is a line in the (x, y) integer plane and $S \in \wp(\mathbb{Z}^2)$ is any set of points then $\pi^l(S)$ denotes the orthogonal projection of S onto l. Completeness of Int-Boolean guards is then geometrically characterized as follows.

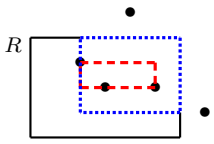
Theorem 6.1. *Let R be a Int-Boolean expression. Then, R is complete for Int in a set of stores $S \in \wp(\mathbb{Z}^2)$ if and only if the following condition holds:*

$$\forall E \in \mathrm{edges}(R).\ \pi^{l_E}(\mathsf{Int}(S) \cap R) \subseteq \mathsf{Int}(\pi^{l_E}(S \cap R)). \quad (*)$$

Proof. Since $\mathsf{Int}(S \cap R) \subseteq \mathsf{Int}(\mathsf{Int}(S) \cap R) = \mathsf{Int}(S) \cap R$ always holds, let us first observe that R is not complete for Int in S when

$$\mathsf{Int}(S \cap R) \neq \mathsf{Int}(\mathsf{Int}(S) \cap R) \Leftrightarrow$$
$$\mathsf{Int}(S \cap R) \neq \mathsf{Int}(S) \cap R \Leftrightarrow$$
$$\mathsf{Int}(S \cap R) \subsetneq \mathsf{Int}(S) \cap R.$$

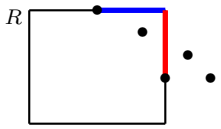
Moreover, it turns out that the rectangle $\mathsf{Int}(S \cap R)$ is strictly contained into the rectangle $\mathsf{Int}(S) \cap R$ iff there exists an edge $E' \in \mathrm{edges}(\mathsf{Int}(S) \cap R)$ such that $\pi^{l_{E'}}(\mathsf{Int}(S \cap R)) \subsetneq E' = \pi^{l_{E'}}(\mathsf{Int}(S) \cap R)$. The following figure helps in explaining this, where S is given by the set of bullets, $\mathsf{Int}(S \cap R)$ is the (red) dashed inner rectangle, and $\mathsf{Int}(S) \cap R$ is the (blue) dotted rectangle.



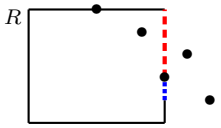
Since the rectangle $\mathsf{Int}(S) \cap R$ is contained in R, the last condition $\mathsf{Int}(S \cap R) \subsetneq \mathsf{Int}(S) \cap R$ holds iff there exists an edge $E \in \mathrm{edges}(R)$ such that $\pi^{l_E}(\mathsf{Int}(S \cap R)) \subsetneq \pi^{l_E}(\mathsf{Int}(S) \cap R)$. Also, the orthogonal projection π^{l_E} onto the line l_E satisfies the following property: $\pi^{l_E}(\mathsf{Int}(S \cap R)) = \mathsf{Int}(\pi^{l_E}(S \cap R))$, so that $\mathsf{Int}(S \cap R) \subsetneq \mathsf{Int}(S) \cap R$ holds iff $\mathsf{Int}(\pi^{l_E}(S \cap R)) \subsetneq \pi^{l_E}(\mathsf{Int}(S) \cap R)$. It is now easy to check that this latter condition holds iff the negation of $(*)$ holds. □

Observe that in the condition $(*)$ we have that $\pi^{l_E}(\mathsf{Int}(S) \cap R)$ is always an interval because $\mathsf{Int}(S) \cap R$ is always a rectangle. Furthermore, if S is already a rectangle, namely $\mathsf{Int}(S) = S$, then condition $(*)$ holds so that we have completeness. Let us also observe that $(*)$ is trivially satisfied when one of the following conditions hold: $\mathsf{Int}(S) \cap R = \varnothing$ or $S \subseteq R$ or $R \subseteq S$. Finally, if $S \cap R = \varnothing$ but $\mathsf{Int}(S) \cap R \neq \varnothing$, then $(*)$ boils down to $\forall E \in \mathrm{edges}(R).\ \pi^{l_E}(\mathsf{Int}(S) \cap R) = \varnothing$, which is always false, so that completeness does not hold.

Example 6.2. In the following pictures, the set of points S in the (x, y) plane is given by the set of bullets. Let us first consider the following example.



In this case, the rectangle R turns out to be complete for Int in S. In fact, for any edge $E \in \mathrm{edges}(R)$, the two projections $\pi^{l_E}(\mathsf{Int}(S) \cap R)$ and $\pi^{l_E}(S \cap R)$ give the same intervals, which are depicted with (blue and red) thick lines. Let us now consider the following modified picture, where the rightmost point of S is moved one step down.



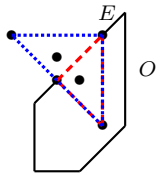
Here, completeness of R for Int in S is lost. In fact, for the rightmost vertical edge $E \in \mathrm{edges}(R)$, $\mathsf{Int}(\pi^{l_E}(S \cap R))$ is the (red) dashed interval which is strictly contained in the interval $\pi^{l_E}(\mathsf{Int}(S) \cap R)$ obtained by projecting S onto E: the gap between these two intervals is depicted by the (blue) dotted segment. □

Of course, Theorem 6.1 can be stated in general for a N-dimensional space, with $N \geq 1$: in the general geometric formulation, R is a N-dimensional hyperrectangle (also called orthotope), $S \in \wp(\mathbb{Z}^N)$ is any set of points in the N-dimensional space, edges of R are replaced by the $2N$ facets of the hyperrectangle R and lines determined by edges are the $(N-1)$-dimensional hyperplanes determined by facets.

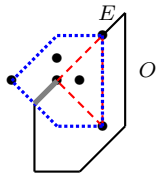
Furthermore, Theorem 6.1 also holds for octagons with the same statement where the abstraction Oct replaces Int. Here, in the 2-dimensional case, the edges of an octagon are at most eight while the abstraction in Oct of a projection onto a line boils down to an interval abstraction. Thus, if O is a Oct-Boolean guard then O is complete for Oct in S iff the following condition holds:

$$\forall E \in \mathrm{edges}(O).\ \pi^{l_E}(\mathsf{Oct}(S) \cap O) \subseteq \mathsf{Int}(\pi^{l_E}(S \cap O)). \quad (*)$$

Example 6.3. In the following picture, the octagon O is complete for Oct in the set S of points depicted as bullets. In the picture, $\mathsf{Oct}(S)$ is depicted with (blue) dotted edges, while $\mathsf{Oct}(S \cap O)$ is depicted with (red) dashed edges. Condition $(*)$ holds: for example, for the edge $E \in \mathrm{edges}(O)$ in the picture $\pi^{l_E}(\mathsf{Oct}(S) \cap O) = \mathsf{Int}(\pi^{l_E}(S \cap O))$ is the (red) dashed interval.



On the other hand, let us consider the following modified picture, where the leftmost point of S is moved two steps down.



Here, completeness of O for Oct in S is lost: the gap between the two orthogonal projections on the edge $E \in \mathrm{edges}(O)$ is depicted with a (gray) thick segment, *i.e.*, condition $(*)$ does not hold. □

Theorem 6.1 explains precisely why the Int/Oct-Boolean expression $R \equiv x \leq 0$, which is the negation of the loop guard of the program *DecInc* in Figure 4 discussed in Section 1, is not complete for Int in the set of input stores $S = \{(9, 0), (8, 1), \ldots, (0, 9)\}$. Here, $\mathsf{Int}(S) \cap R = \langle x = 0,\ 0 \leq x \leq 9 \rangle$ and $S \cap R = \{(0, 9)\}$, and for the edge $E \equiv \langle x = 0 \rangle \in \mathrm{edges}(R)$, we have that $\pi^{l_E}(\mathsf{Int}(S) \cap R) = \langle x = 0,\ 0 \leq y \leq 9 \rangle$ is not contained in $\mathsf{Int}(\pi^{l_E}(S \cap R)) = \{(0, 9)\}$. Instead, this incompleteness does

not arise with Oct because $\pi^{l_E}(\mathsf{Oct}(S) \cap R) = \{(0,9)\} = \mathsf{Int}(\pi^{l_E}(S \cap R))$.

Validating Assumptions on Guards. Consider the following program template

$$P \triangleq x := k_1; y := k_2; \textbf{while } (x, y \in R) \textbf{ do } \{x := a_1; y := a_2\}$$

which subsumes the examples of Figures 2, 3 and 4. We assume that: (i) $k_i \in \mathbb{Z}$ are constant values; (ii) R is any (finite or infinite) 2-dimensional rectangle in Int such that its complement $\neg R$ is a rectangle as well; (iii) the arithmetic expressions $a_i \in \mathrm{AExp}$ are complete for Int, *i.e.*, $a_i \in \mathbb{A}(\mathsf{Int})$. By assumption (i):

$$\dfrac{\dfrac{k_1 \in \mathbb{A}(\mathsf{Int})}{\vdash^{\mathbf{NR}}_{\mathsf{Int}} x := k_1} \quad \dfrac{k_2 \in \mathbb{A}(\mathsf{Int})}{\vdash^{\mathbf{NR}}_{\mathsf{Int}} y := k_2}}{\vdash^{\mathbf{NR}}_{\mathsf{Int}} x := k_1; y := k_2}$$

Also, by assumption (iii), the body of the while-loop can be proved complete for Int:

$$\dfrac{\dfrac{a_1 \in \mathbb{A}(\mathsf{Int})}{\vdash^{\mathbf{NR}}_{\mathsf{Int}} x := a_1} \quad \dfrac{a_2 \in \mathbb{A}(\mathsf{Int})}{\vdash^{\mathbf{NR}}_{\mathsf{Int}} y := a_2}}{\vdash^{\mathbf{NR}}_{\mathsf{Int}} x := a_1; y := a_2}$$

Let us then consider the set S of stores, *i.e.*, concrete (x, y)-points, computed by the body of the while-loop of P, *i.e.*, if $I = \{(k_1, k_2)\} \in \wp(\mathbb{Z}^2)$ is the set of input stores for the while-loop then S is given by

$$S = \mathsf{lfp}(\lambda X.I \cup [\![x := a_1; y := a_2]\!][\![R]\!]X).$$

We thus have a conditional proof $\vdash^{\mathbf{NR}}_{\mathsf{Int}} P$ which depends on the assumptions on the Boolean guard R and its negation $\neg R$:

$$\textit{assume}[S : \mathsf{Int}(R \cap S) = R \cap \mathsf{Int}(S)],$$
$$\textit{assume}[S : \mathsf{Int}(\neg R \cap S) = \neg R \cap \mathsf{Int}(S)].$$

We apply Theorem 6.1 to derive the validity of these assumptions. Let us consider the following cases.

(A) If $[\![R]\!]I \neq I$, *i.e.* $[\![R]\!]I = \varnothing$, then the while-loop is not entered, so that $S = I$, and $R \cap S = \varnothing$ and $\neg R \cap S = S$. For example, this happens with:

$$x := 2; y := 4; \textbf{while } (x < 0) \textbf{ do } \{x := x + 1; y := y + 1\}$$

In this case, completeness of R and $\neg R$ for S is a trivial case of condition $(*)$ in Theorem 6.1.

(B) If $[\![R]\!]I = I$ then the while-loop is entered. If S is an infinite set then we have nontermination. This means that $S \subseteq R$ and $\neg R \cap S = \varnothing$. This happens, for example, with the program:

$$x := 2; y := 4; \textbf{while } (x > 0) \textbf{ do } \{x := x + 1; y := y + 1\}$$

Here, condition $(*)$ for R and $\neg R$ follows trivially because we have that, respectively, $S \subseteq R$ and $\neg R \cap S = \varnothing$.

(C) If $[\![R]\!]I = I$ and S is finite then the while-loop is entered and we have termination. This means that $R \cap S \neq \varnothing$ and $\neg R \cap S \neq \varnothing$. This is the case of the following two programs:

$$P_1 \triangleq x := 2; y := 4; \textbf{while } (x > 0) \textbf{ do} \{x := x-1; y := y-1\}$$
$$P_2 \triangleq x := 2; y := 4; \textbf{while } (x > 0) \textbf{ do } \{x := x-1; y := y\}$$

Here, we have that:

$$S_{P_1} = \{(2,4), (1,3), (0,2)\}; \quad S_{P_2} = \{(2,4), (1,4), (0,4)\}.$$

By condition $(*)$, we obtain that R is not complete for S_{P_1} while both R and $\neg R$ are complete for S_{P_2}. While for P_1 the assumptions of its conditional proof cannot be validated, the assumptions for P_2 are validated by Theorem 6.1, hence we have $\vdash^{\mathbf{NR}}_{\mathsf{Int}} P_2$.

(D) Finally, let us consider the analyses of the programs P_1 and P_2 of point (C) for the abstraction Oct. In this case, by Lemma 5.7, we have that all the assignments in P_1 and P_2 are complete for Oct. Moreover, while $\vdash_{\mathsf{Oct}} P_2$ can obviously be inferred as in point (C), here we are also able to infer that $\vdash_{\mathsf{Oct}} P_1$. In fact, since the set S_{P_1} is an octagon, *i.e.*, $\mathsf{Oct}(S_{P_1}) = S_{P_1}$, the condition $(*)$ is trivially satisfied both for R and $\neg R$, so that the assumptions for R and $\neg R$ of the conditional proof of P_1 can be validated.

7. Related Work

To the best of our knowledge, this is the first attempt to systematically extract a property concerning the precision of a static program analysis through a proof system. The subject of our analysis is a given abstract interpretation of a program, while the property to analyze is the completeness of this abstract interpretation. Completeness for an abstract interpretation is particularly important because it formalizes the absence of false alarms, and therefore it is deeply related with the quality of the program analysis. The most related approach is in [13], where the authors define a constructive optimal abstraction refinement for deriving the most abstract complete domain which refines a given (incomplete) abstraction for a generic program statement. Our method follows an orthogonal pattern. We are not interested in refining an abstract domain for obtaining completeness with respect to a given class of programs, but rather in studying the class of programs for which a given abstraction is complete. This different starting point leads us to design a deductive system for deriving a completeness proof for programs on a given abstraction. This new perspective allows us to decompose the problem of attaining a complete abstraction for a given program, which becomes modular and inductive on the program's syntax. The ubiquity of completeness properties in static analysis is also studied in [22], where it is argued how completeness can play a beneficial role for designing static analyses by reasoning on the completeness properties of their underlying abstract domains.

Provably precise static analyses usually make some assumption on the syntactic form of the analyzed program. For instance the precise interprocedural analysis of [21] computes, for each program point, all the affine relations among program variables. What the authors call "precision" is effectively our notion of completeness. They achieve precision by focusing on a particular class of programs, namely affine programs. Affine programs are such that: (i) all guards are non-deterministic; and (ii) the right-hand side of assignments are either affine expressions or unknown values. Similarly, in type systems, it is customary to ignore the guards in order to prove the completeness of the type inference algorithm, *e.g.*, in [9]. We argued in this article that the main problem for proving completeness is the handling of assignments and of Boolean guards.

Our research follows the lines of a recent approach by Cousot and Cousot [5] who put forward a type system for typing the structure of an abstract interpretation. A type represents inductively the way an abstraction has been built by composing simpler abstractions through systematic domain operations like reduced product. It could be interesting to investigate the possibility of combining our proof system with a structural type system for abstract interpretations as in [5], with the aim of providing along a derivation in $\vdash_A$ some additional information about the used sub-domains and their composition through domain operations.

8. Conclusion

Static analysis is, by design, incomplete. Nevertheless, experience has shown that it can be made precise enough to be used for verification [6, 11]. We envision static analyses which in addition to

the inferred invariants also provide completeness certificates. The completeness certificate is used to provide confidence to the analysis of alarms. As a foundational step towards this goal, we introduced a theoretical framework to prove the completeness of a static analysis. We have shown that the source of incompleteness lies in the handling of Boolean guards and, for relational abstractions, in assignments. For nonrelational abstractions we introduced an abstraction-independent core proof system which pushes the completeness of the analysis to the numerical expressions and the Boolean guards of conditionals and loops. For relational abstractions, instead, the structure of complete assignments has to be derived in advance in order to obtain a sound proof system. We argued that the designer of a static analyzer should also provide completeness conditions for the Boolean guards and that these conditions could be automatically checked by further, yet less sophisticated, static analyses—we leave the design of such automated analyses for future work. We studied the completeness of Boolean guards in widely used numerical abstractions such as Intervals and Octagons. Most known abstract domains have been indeed designed to precisely capture properties of some given programs. This is justified by the fact that the class of completeness $\mathbb{C}(A)$ of any abstraction A is always an infinite set. Therefore, deriving an abstract domain which is complete for a specific program P provides at the same time a domain which is complete for an infinite class of programs.

As future work, we think that proving completeness of static analyses could be also beneficial to: (i) automatically apply abstract code repairs [18]—if the analysis of the original and the repaired programs can both be proven complete, then the repair is very likely to have fixed a concrete bug; (ii) validate refactorings [8]—among different program refactorings one may only keep the one(s) for which she can prove it preserves the completeness of the analysis; (iii) provide a better understanding of why over-approximating analyses of arrays [7] works well in practice even without performing under-approximations, argued as necessary in [15].

Acknowledgments

Roberto Giacobazzi and Francesco Ranzato have been partially supported by the Microsoft Research Software Engineering Innovation Foundation 2013 Award (SEIF 2013).

References

[1] COUSOT, P., AND COUSOT, R. Static verification of dynamic type properties of variables. Research Report no. 25, Laboratoire IMAG, University of Grenoble, France, 1975.

[2] COUSOT, P., AND COUSOT, R. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming* (1976), Dunod, Paris, pp. 106–130.

[3] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (*POPL '77*) (1977), ACM Press, pp. 238–252.

[4] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (*POPL '79*) (1979), ACM Press, pp. 269–282.

[5] COUSOT, P., AND COUSOT, R. A Galois connection calculus for abstract interpretation. In *Conference Record of the 41st ACM Symposium on Principles of Programming Languages (POPL '14)* (2014), S. Jagannathan and P. Sewell, Eds., ACM Press, pp. 3–4.

[6] COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. The ASTRÉE analyzer. In *Proceedings of the European Symposium on Programming (ESOP '05)* (2005), vol. 3444 of *Lecture Notes in Computer Science*, Springer, pp. 21–30.

[7] COUSOT, P., COUSOT, R., AND LOGOZZO, F. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Conference Record of the 38th ACM Symposium on Principles of Programming Languages (POPL '11)* (2011), ACM Press, pp. 105–118.

[8] COUSOT, P., COUSOT, R., LOGOZZO, F., AND BARNETT, M. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *Proceedings of the 27th ACM Inter. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)* (2012), ACM Press, pp. 213–232.

[9] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *Conference Record of the 9th ACM Symposium on Principles of Programming Languages (POPL '82)* (1982), ACM Press, pp. 207–212.

[10] DEKKER, J. C. E. Productive sets. *Trans. of the American Mathematical Society 78* (1955), 129–149.

[11] FAHNDRICH, M., AND LOGOZZO, F. Static contract checking with abstract interpretation. In *Proceedings of the Internat. Conf. on Formal Verification of Object-oriented Software (FoVeOOS 10)* (2010), Springer.

[12] GIACOBAZZI, R. Hiding information in completeness holes – New perspectives in code obfuscation and watermarking. In *Proc. of the 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM '08)* (2008), IEEE Press, pp. 7–20.

[13] GIACOBAZZI, R., RANZATO, F., AND SCOZZARI, F. Making abstract interpretation complete. *Journal of the ACM 47*, 2 (2000), 361–416.

[14] GRANGER, P. Static analysis of arithmetical congruences. *Intern. J. Computer Math. 30* (1989), 165–190.

[15] GULWANI, S., MCCLOSKEY, B., AND TIWARI, A. Lifting abstract interpreters to quantified logical domains. In *Conference Record of the 35th ACM Symposium on Principles of Programming Languages (POPL '08)* (2008), ACM Press, pp. 35–46.

[16] KARR, M. Affine relationships among variables of a program. *Acta Informatica 6* (1976), 133–151.

[17] LAVIRON, V., AND LOGOZZO, F. Refining abstract interpretation-based static analyses with hints. In *Proc. of the 2009 Asian Symp. on Programming Languages and Systems (APLAS '09)* (2009), vol. 5904 of *Lecture Notes in Computer Science*, Springer, pp. 343–358.

[18] LOGOZZO, F., AND BALL, T. Modular and verified automatic program repair. In *Proceedings of the 27th ACM Internat. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)* (2012), ACM Press, pp. 133–146.

[19] MINÉ, A. A new numerical abstract domain based on difference-bound matrices. In *Proc. of the 2nd Symp. on Programs as Data Objects (PADO '01)* (2001), vol. 2053 of *Lecture Notes in Computer Science*, Springer, pp. 155–172.

[20] MINÉ, A. The octagon abstract domain. *Higher Order and Symbolic Computation 19*, 1 (2006), 31–100.

[21] MÜLLER-OLM, M., AND SEIDL, H. Precise interprocedural analysis through linear algebra. In *Conference Record of the 31st ACM Symposium on Principles of Programming Languages (POPL '04)* (2004), ACM Press, pp. 330–341.

[22] RANZATO, F. Complete abstractions everywhere. In *Proc. of the 14th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'13)* (2013), vol. 7737 of *Lecture Notes in Computer Science*, Springer, pp. 15–26.

[23] ROGERS, H. *Theory of Recursive Functions and Effective Computability*. The MIT press, 1992.

[24] SOARE, R. I. *Recursively Enumerable Sets and Degrees*. Springer-Verlag, 1980.

[25] WINSKEL, G. *The Formal Semantics of Programming Languages: an Introduction*. MIT press, 1993.