Abstract Interpretation Repair

Roberto Bruni
University of Pisa, Pisa, Italy
roberto.bruni@unipi.it

Roberto Giacobazzi
University of Verona, Verona, Italy
roberto.giacobazzi@univr.it

Roberta Gori
University of Pisa, Pisa, Italy
roberta.gori@unipi.it

Francesco Ranzato
University of Padova, Padova, Italy
francesco.ranzato@unipd.it

Abstract

Abstract interpretation is a sound-by-construction method for program verification: any erroneous program will raise some alarm. However, the verification of correct programs may yield false-alarms, namely it may be *incomplete*. Ideally, one would like to perform the analysis on the most abstract domain that is precise enough to avoid false-alarms. We show how to exploit a weaker notion of completeness, called *local completeness*, to optimally refine abstract domains and thus enhance the precision of program verification. Our main result establishes necessary and sufficient conditions for the existence of an optimal, locally complete refinement, called *pointed shell*. On top of this, we define two repair strategies to remove all false-alarms along a given abstract computation: the first proceeds forward, along with the concrete computation, while the second moves backward within the abstract computation. Our results pave the way for a novel *modus operandi* for automating program verification that we call Abstract Interpretation Repair (AIR): instead of choosing beforehand the right abstract domain, we can start in any abstract domain and progressively repair its local incompleteness as needed. In this regard, AIR is for abstract interpretation what CEGAR is for abstract model checking.

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

Keywords: abstract interpretation, program analysis, program verification, local completeness, CEGAR.

ACM Reference Format:
Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2022. Abstract Interpretation Repair. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3519939.3523453

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523453

1 Introduction

It is widely acknowledged that the chance of formally verifying programs is fundamental to effectively rise the confidence level that the code we use is correct [23]. However, as emerged in the last decades, this approach to program correctness becomes socially acceptable when these proofs are not only rigorous but also explainable, meaning that they have to rely upon a largely recognized proof method which has to be simple and inspectable [22]. As advocated by Vardi [61], checking program correctness "*is a cost that must be justified by the benefits*". The last 50 years have shown an impressive flourishing of formal methods and tools for achieving this ambitious goal [32]. These include, among the others: Certified compilers [42], certified analyzers [39], advanced type checkers [49, 50], sophisticated static analyzers [6, 19, 25] and software model checkers [3, 37].

A high degree of confidence in the correctness of a software system, and of its most critical components, can be obtained when the code is certified by a *sound* and *complete* (viz. precise) static analyzer [14, 25]. Abstract interpretation [17] was introduced with this purpose in mind: simplify the proof of correctness by interpreting the program in a simplified, *abstract*, domain. This provides a general methodology for the design of *sound-by-construction* analysis tools.

The Problem. The soundness of an abstract interpreter, or program analyzer, means that all true-alarms are caught. However, it is often the case that some false-alarms are reported. Actually, when false-alarms overwhelm true ones, then the program analyzer may become poorly trustworth. This is a consequence of the approximation inherent in the making of an otherwise undecidable analysis decidable. As all alarm systems, program analysis is credible when few false-alarms are reported, ideally none. The problem we address in this paper is *how to derive the most abstract domain to decide program correctness without raising false-alarms.*

The absence of false-alarms in program analysis is closely related to the property of *completeness in abstract interpretation* [33]. As an illustrative example, consider the program

for computing the absolute value of integer variables:

$$\textbf{AbsVal}(x) \triangleq \textbf{if } (x \geq 0) \textbf{ then skip else } x := -x$$

The well-known interval abstraction Int approximates any property $S \in \wp(\mathbb{Z})$ of the integer values that x may assume by the least interval $\mathsf{Int}(S) = [a, b]$ such that $S \subseteq [a, b]$, where $a \in \mathbb{Z} \cup \{-\infty\}$, $b \in \mathbb{Z} \cup \{+\infty\}$, and $a \leq b$. Assume that we know that the possible inputs range in the set $I = \{x \mid x \text{ is odd}\}$. While the approximation in Int of $\textbf{AbsVal}(I)$ is $\mathsf{Int}(\textbf{AbsVal}(I)) = [1, +\infty]$ showing that 0 is not a possible result, it turns out that the *best correct approximation* the abstract interpreter can compute in Int is not precise, because it also includes 0: $\mathsf{Int}(\textbf{AbsVal}(\mathsf{Int}(I))) = [0, +\infty]$. Technically, this means that Int is incomplete for **AbsVal** on input I. This can spawn a problem in verification: for instance, if the result of $\textbf{AbsVal}(I)$ is used as divisor in an integer division, the abstract interval analysis would raise a "division-by-0" false-alarm.

Completeness intuitively encodes the greatest achievable precision for a program analysis on a given abstract domain. The problem of making abstract domains complete by either domain refinement (i.e., increasing precision) or by domain simplification (i.e., reducing precision) has been settled in [33]. The most abstract refinement, called *complete shell*, of an abstract domain always exists for all computable functions and it can be constructively defined as the solution of a recursive domain equation. Although extremely powerful, this notion has an intrinsic global flavor: The complete shell of an abstract domain with respect to a transfer function f makes the abstract domain complete for f *on all possible inputs*. As a result, the complete shell, as currently known, yields an abstract domain that is often way too fine grained, possibly blowing up to the whole concrete domain. Furthermore, completeness holds for all programs in a Turing complete language only for trivial abstract domains [7, 28], i.e., for the "none" and "don't-know" abstractions.

From Global to Local. Our idea is to introduce a local approach to completeness, focussing on a single execution trace produced by the applications of abstract transfer functions on some input of interest. A notion of *local completeness* in abstract interpretation has been introduced by [8] to design a logical proof system that can be used to simultaneously check both program correctness and incorrectness. While (global) completeness is a requirement that must hold for every possible input, local completeness is tied to a particular input. The proof system in [8] is parametric with respect to an underlying abstract domain A: any triple $\vdash_A [I]\ \mathsf{c}\ [Q]$ derivable in the proof system is such that Q is an under-approximation of the strongest postcondition $\mathsf{post}[\mathsf{c}](I)$ for the program c on input I, while the abstraction $A(Q)$ of Q in the domain A turns out to be a locally complete over-approximation of $\mathsf{post}[\mathsf{c}](I)$. A novel inference rule drives logical derivations for $\vdash_A$ by introducing all and only those local completeness proof obligations needed to infer a triple $\vdash_A [I]\ \mathsf{c}\ [Q]$. These are *local* obligations because they are relative to a specific computation trace. Thanks to local completeness, any alarm in Q is a true-alarm and a correctness specification *Spec* expressible in A is valid for c if and only if $Q \leq \mathit{Spec}$. Since logical derivations cannot work with locally incomplete abstractions, we advocate that *these abstract domains need to be repaired to achieve local completeness.*

Main Contributions. In this work, we design algorithmic methods to optimally repair any locally incomplete abstract analysis. The idea is to repair the abstract domain whenever a local completeness proof obligation is violated: the abstract domain is replaced by its so-called *pointed shell*, that is, its most abstract refinement that is locally complete. Notably, each refinement is the Moore closure (i.e., closure under arbitrary meets) of the original abstract domain with one new abstract element only (this justifies the term "pointed"), thus limiting the blow up in size of the domain refinement, unlike complete shells. While other approaches consider an optimal refinement in a given class of domains, e.g., using interpolants and SMT solvers [1, 35, 36], pointed shells are optimal in a *general and absolute sense*, i.e., they do not limit a priori the way in which new abstract elements are represented, e.g. by formulae of some theory.

In our simple example for **AbsVal**, we would refine the interval abstraction Int by adding a new abstraction for the input $I = \{x \mid x \text{ is odd}\}$, which is more precise than $[-\infty, +\infty]$ and guarantees that the guard $x \geq 0$ is complete on input I. Different choices could be available: in this case, the most abstract refinement would consist in adding to Int the new abstract element $\mathbb{Z}_{\neq 0} \triangleq [-\infty, -1] \cup [1, +\infty]$. The verification of **AbsVal** in this repaired domain will not raise alarms.

Our Abstract Interpretation Repair (AIR) shares similarities with the principles of the well-known *Counter-Example Guided Abstraction Refinement* (CEGAR) methodology [9, 11], which is indeed shown to be an instance of AIR. More precisely, we prove that: any CEGAR abstract counterexample is spurious iff it contains a locally incomplete step for the post transformer, and each repair strategy defines an heuristics for CEGAR's refinement. In a slogan we may say that:

AIR is for abstract interpretation what
CEGAR is for abstract model checking.

By leveraging this idea of local completeness repair, we define two general purpose verification strategies, called *forward* and *backward repair*.

The forward repair of an abstract domain A driven by an input property for a given program c consists in refining A so that local completeness is preserved by assignments and Boolean guards occurring in the analysis of c. For the above program **AbsVal** on input I this forward repair coincides with our previous example. Actually, it collects the proof obligations raised by completeness of the guard $x \geq 0$ local to input I and refine Int accordingly. No proof obligation

for the assignment $x := -x$ is collected because it is already globally complete for Int. The repaired analysis of **AbsVal** on input I will output $[1, +\infty]$, thus proving that $x \neq 0$ holds.

The backward repair for **AbsVal** on input I consists, instead, in tracing back the bad value $x = 0$ that is present in the result of the abstract analysis of **AbsVal** on Int, and then refining the abstraction Int accordingly. Intuitively, the bad value $x = 0$ is reported back to the guard $x \geq 0$ hinting that the abstraction has to be refined so that the input I after the filtering $x \geq 0$ does not have to contain the value $x = 0$, i.e., the abstraction of I itself cannot contain 0. This provides a refinement of Int that still adds the above point $\mathbb{Z}_{\neq 0}$, as in the forward repair case, but this is purely incidental. In backward repair, local completeness refers to the largest correct input, so that the pointed shell corresponds to adding this point itself. Contrary to the forward repair, the backward strategy proceeds with the next repair (if necessary) along the existing abstract computation.

Structure of the Paper. After an illustrative example in Section 2 and some background in Section 3: Section 4 introduces pointed shells; Section 5 defines AIR strategies; Section 6 draws connections with CEGAR; Section 7 applies AIR to program verification; Section 8 discusses some implementation challenges. Concluding remarks are in Section 9.

Related Work. In general, repairing computations means modifying program executions in order to remove flaws, possibly without human intervention. The problem of (automatic) program/model repair has been extensively investigated in software engineering as a solution for efficiently maintaining software (see, e.g., [27, 47]). These works are intended to repair the program in order to achieve that transformed code does not expose bugs. This means either modifying the program code or modifying the execution state at runtime. AIR can be viewed as an instance of the process of automatically removing flaws in the abstract interpreter by acting at the level of its abstract domain(s). Because abstract interpretation is sound-by-construction, the flaws amount to the possible false-alarms raised by the analyzer. A standard way to fix these flaws is through abstract domain refinements. The completeness refinement designed in [33] refines the whole abstract domain in order to remove *all* possible false-alarms that may be produced for *any* program and for *all* possible inputs. Even when applied to the specific case of a CTL formula on state partitions in abstract model checking, the result is a global partition refinement which is independent from the initial and intermediate states of traces satisfying or violating that formula, hence often boiling down to the concrete model [29–31]. By contrast, AIR is a kind of *surgical refinement* of the abstract domain, making a specific abstract transfer function complete (i.e., producing no false-alarms) w.r.t. a given input property of interest. It is therefore possible to tailor the abstract domain along a given execution trace, e.g., by repairing the analysis at runtime. AIR is inherently dynamic as compared to the global completeness refinements in [33] and provides local fixes of the abstract domain, resulting in a more abstract and therefore efficient program analysis algorithm. Under this perspective, the most closely related approach is the well-known CEGAR method pioneered in [9, 11]. The use of pointed shells goes beyond the relation settled by [29] between completeness shell and CEGAR. Moreover, AIR can be applied to arbitrary Galois connection-based abstract domains and transfer functions defined on generic complete lattices, hence going beyond the state partitions used in early abstract model checking. Modern abstraction refinement heuristics improving on the original partition-based CEGAR method have been later proposed [1, 35, 36, 38, 57, 59, 60]. However, contrary to AIR, they focus on some specific class of abstract domains and refinements, typically domains of logical formulae and interpolant-based refinements.

The logical proof system LCL_A for local completeness of [8] extends the early proof system for global completeness defined in [28]. LCL_A is applicable to prove both program correctness, as in Hoare logic for partial correctness, and, when this fails, to detect true-alarms, as in O'Hearn's incorrectness logic [48]. However, each proof in LCL_A corresponds to a locally complete abstract computation in the abstract domain A. How to make a generic abstract domain locally complete was set as an open problem in [8, Section VII]. AIR solves this problem by exhibiting the most abstract domain that allows LCL_A to complete a derivation.

2 Illustrative Example

Let us consider the following program computing the sum of the first 5 positive integers, taken verbatim from [46, 58] and that will be generalized after a preliminary discussion:

$$\mathrm{c} \triangleq i := 1; j := 0; \textbf{while } (i \leq 5) \textbf{ do } \{j := j + i; i := i + 1\}$$

Recalling that the numbers $T_k \triangleq \sum_{i=1}^{k} i = k(k+1)/2$ are called *triangular*, this program c therefore outputs $j = T_5 = 15$ and $i = 6$. Assume we want to prove that c meets the postcondition $Spec \triangleq j \leq 15$. The analysis of c on Int, using the standard widening for intervals, infers the abstract loop invariant $\widehat{P} \triangleq i \in [1, 6] \wedge j \in [0, \infty]$, and outputs the abstract store $\widehat{Q} \triangleq i \in [6, 6] \wedge j \in [0, \infty]$. Since Int is a nonrelational domain, its elements cannot represent relationships between the variables i and j. The weakly relational octagon domain Oct [44] refines the interval domain by representing relational constraints of type $\pm x \pm y \leq k$ between any two variables x and y. The analysis with Oct infers a more precise abstract loop invariant: $1 \leq i \leq 6 \wedge i - j \leq 1 \wedge i + j \geq 1 \wedge j \geq 0$. As observed in [58, Sect. 1], even if the convex polyhedra abstraction Poly [21] is more precise than both Int and Oct, an analysis of c with Poly infers a weaker loop invariant: $i \leq 6$. This may appear counterintuitive, but there exist many incomparable polyhedra representing a finite set of states

and the combined use of joins and widenings makes it difficult for the generalization step to pick the best polyhedron. Anyway, none of the above analyses is conclusive for *Spec*.

Since at the beginning of each loop iteration, j stores the $(i-1)$-th triangular number $T_{i-1} = i(i-1)/2$, we observe that the concrete (i.e., strongest) loop invariant of c is:

$$Inv \quad \triangleq \quad i \in [1,6] \wedge j = T_{i-1}.$$

AIR allows us to refine the aforementioned abstract domains Int and Oct by adding to them the least amount of information necessary to prove the validity of *Spec* in a *compositional way* w.r.t. the program structure, that is, our solution is inductively computed on the program syntax to guarantee that each abstract computation step (i.e., abstract transfer functions of the Boolean guard and assignments) is locally complete. First, the program c is encoded as the following regular command (cf. Section 3.2):

$$\mathsf{r} \triangleq i := 1;\ j := 0;\ \overbrace{\big((i \le 5)?;\ j := j + i;\ i := i + 1\big)^*}^{\mathsf{r}_1};\ (i > 5)?$$

For the case of Int, our backward repair (cf. Algorithm 2) proves that *Spec* holds by refining Int with the addition of five new abstract elements:

$$\begin{array}{rcl} P & \triangleq & i \in [1,6] \wedge j \in [0, T_{i-1}] \\ R_1 & \triangleq & i \in [1,5] \wedge j \in [0, T_{i-1}] \\ R_2 & \triangleq & i \in [1,5] \wedge j \in [1, T_i] \\ R_3 & \triangleq & i \in [2,6] \wedge j \in [1, T_{i-1}] \\ V & \triangleq & (i \in [1,5] \wedge j \in [0,\infty]) \vee (i = 6 \wedge j \in [0, T_5]). \end{array}$$

Therefore, P and R_1, R_2, R_3 encode relational polynomial constraints between i and j, while V is a disjunction of intervals. Let us describe how these points are obtained. Recall that the abstract analysis of r in Int returns $\widehat{Q} = i \in [6,6] \wedge j \in [0,\infty]$. Since we aim to prove $j \le 15$, AIR first computes the point $Q \triangleq \widehat{Q} \wedge Spec = i \in [6,6] \wedge j \in [0,15]$, which is already in Int. Since $\widehat{Q}$ was obtained by the abstract evaluation of the exit condition $(i > 5)?$ on the abstract loop invariant $\widehat{V} \triangleq i \in [1,6] \wedge j \in [0,\infty]$ for r_1^*, this repair step introduces the largest precondition V below (i.e., entailing) $\widehat{V}$ such that the guard $(i > 5)?$ filters V into a result below Q. This explains why and how the new point V is introduced. Since V is not an invariant for r_1^*, the next repair step infers the largest loop invariant for r_1^* below V, and this is the property P. The new points R_h are introduced because the repair procedure is inductively defined and they are needed to guarantee local completeness. For example, R_1 is the point necessary for making the guard $(i \le 5)?$ locally complete on P. Similarly, R_2 makes the assignment $j := j + i$ locally complete on R_1, and, in turn, R_3 is necessary for making $i := i + 1$ locally complete on R_2 (more details are given in Example 7.13). If we started the repair in Oct, then we would have obtained a more concrete result. If the specification $j = 15$ was considered, our repair would have instead introduced the concrete invariant Inv in place of P and restricted the range of j in R_h, V in a similar way.

Let us discuss what happens when we generalize the example. To this aim, we replace the guard $i \le 5$ (constant boundary) by $i \le n$, where n is a variable, and consider the specification $Spec \triangleq j \le T$ for a generic constant T.

As a first investigation, we consider the precondition $n \in [K,K]$ for some constant $K \ge 0$ such that $T_K \le T$. Letting $D_K \triangleq T - T_K \ge 0$, our repair strategy is still able to prove that *Spec* holds by refining Int with the addition of five new abstract elements that can be represented as follows:

$$\begin{array}{rcl} P[K] & \triangleq & n = K \wedge i \in [1, K+1] \wedge j \in [0, D_K + T_{i-1}] \\ R_1[K] & \triangleq & n = K \wedge i \in [1, K] \wedge j \in [0, D_K + T_{i-1}] \\ R_2[K] & \triangleq & n = K \wedge i \in [1, K] \wedge j \in [1, D_K + T_i] \\ R_3[K] & \triangleq & n = K \wedge i \in [2, K+1] \wedge j \in [1, D_K + T_{i-1}] \\ V[K] & \triangleq & n = K \wedge ((i \in [1,K] \wedge j \in [0,\infty]) \vee S) \end{array}$$

where $S \triangleq i = K+1 \wedge j \in [0,T]$. An easy generalization of the arguments illustrated above for r explains why and how these new elements are computed. As expected, when $K = 5$ and $T = T_5$ (so that $D_K = 0$), they coincide with P, R_h, V.

Next, consider the precondition $n \in [K_1, K_2]$, for some constants $0 \le K_1 \le K_2$ such that $T_{K_2} \le T$. Even in this case our repair strategy proves *Spec* by refining Int with the addition of five abstract elements, such as:

$$\begin{array}{rcl} P[K_1,K_2] & \triangleq & n \in [K_1,K_2] \wedge (P[n] \vee S[n]) \\ V[K_1,K_2] & \triangleq & n \in [K_1,K_2] \wedge (V[n] \vee S[n]) \end{array}$$

where $S[n] \triangleq i \in [n{+}1, K_2{+}1] \wedge j \in [0,T]$. Here, the notation $P[n]$ denotes the syntactic replacement of the parameter of $P[\cdot]$ with n, so as to establish a relational constraint between the variables i, j and n. Clearly, for each value of n in the range $[K_1, K_2]$ the values of i and j are constrained by the new abstract points $P[K_1,K_2]$ and $V[K_1,K_2]$ in the same way as they are, respectively, in $P[n]$ and $V[n]$, except for the cases $i \in [n{+}1, K_2{+}1]$ when the guard of the loop is false and the postcondition is enforced. The reasoning for the other three points $R_h[K_1,K_2]$ is analogous. When $K_1 = K_2$ they all boil down to $P[K_2], R_h[K_2], V[K_2]$, respectively.

To conclude, we remark that: (i) AIR provides the *most abstract refinement* (cf. Section 4) of Int for proving that *Spec* holds; (ii) even if the analysis starts in the nonrelational domain Int, AIR yields a relational refinement and infers polynomial constraints; (iii) our findings have been validated by a proof-of-concept implementation of AIR based on an explicit enumerative representation of abstract elements; (iv) an advanced tool implementing AIR would necessarily require the ability of dealing in practice with specific classes of symbolic abstract domains, e.g., domains of SMT logical formulae, constraints, bit-vectors, etc.; this challenge is briefly discussed in Section 8 and not addressed by this work, whose goal is to characterize *general optimal refinements* with no restriction about their symbolic representations.

3 Background

Notation. Any function $f : X \to Y$ is overloaded to denote its lifting $f : \wp(X) \to \wp(Y)$ to powersets: $f(S) \triangleq \{f(x) \mid x \in S\}$. A complete lattice C is denoted by $\langle C, \leq, \vee, \wedge, \top, \bot\rangle$, with partial order $\leq$, lub $\vee$, glb $\wedge$, greatest element $\top$ and least element $\bot$. Given $X \subseteq C$, its set of maximal elements is $\max(X) \triangleq \{x \in X \mid \forall y \in X.\, x \leq y \Rightarrow x = y\}$. A lub-preserving (resp. glb-preserving) function between complete lattices is called additive (resp. co-additive). If C is a lattice and $f, g : X \to C$ then $f \sqsubseteq g$ denotes pointwise ordering, i.e., $\forall x \in X.\, f(x) \leq_C g(x)$. The least fixpoint of a monotone function $f : C \to C$ is denoted by $\mathrm{lfp}(f)$.

3.1 Abstract Interpretation

We just recall some basics of abstract interpretation [17] (see the book [16] for a comprehensive introduction). In abstract interpretation the concrete domain C and the abstract domain A are complete lattices related by a pair of monotone maps $\langle \alpha : C \to A, \gamma : A \to C\rangle$ forming a Galois insertion (GI), i.e., $\forall c \in C, a \in A.\, \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$ and $\alpha\gamma = id_A$. We let $\mathrm{Abs}(C)$ denote the class of abstract domains of C, where the notation $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ makes explicit the maps of a GI.

A concrete value $c \in C$ is *expressible* in A if $\gamma\alpha(c) = c$.

Soundness and Completeness. Given $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ and a function $f : C \to C$, an abstract function $f^\sharp : A \to A$ is a *correct* (or *sound*) approximation of f if $\alpha f \sqsubseteq f^\sharp\alpha$ holds. If $f^\sharp$ is a correct approximation of f then fixpoint correctness $\alpha(\mathrm{lfp}(f)) \leq_A \mathrm{lfp}(f^\sharp)$ holds. The *best correct approximation* (bca) of f in A is defined by $f^A \triangleq \alpha f\gamma : A \to A$.

The function $f^\sharp$ is a *complete* approximation of f (or just complete) if $\alpha f = f^\sharp\alpha$ holds. The abstract domain A is a complete abstraction for f if there exists a complete approximation $f^\sharp : A \to A$ of f. Completeness of $f^\sharp$ intuitively encodes the greatest achievable precision when abstracting the concrete behaviour of f on the abstract domain A. In complete abstractions the loss of precision is only due to the abstract domain and not to the abstract function itself, which is necessarily the bca of f. Indeed, it turns out that completeness $\alpha f = f^\sharp\alpha$ holds for some (sound) $f^\sharp$ iff $\alpha f = \alpha f\gamma\alpha$ iff $(\gamma\alpha)f = (\gamma\alpha)f(\gamma\alpha)$, which, by a slight abuse of notation, will be denoted by $Af = AfA$. It is known that completeness transfers to fixpoints, meaning that if $f^\sharp$ is complete for f then fixpoint completeness $\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^\sharp)$ holds.

Domains as Closures. An abstract domain refinement is a transform that yields a more precise abstract domain [26]. Because increasing precision means adding new elements to abstract domains, refinements are simpler to define when abstract domains are viewed as *closure operators*. An upper closure operator (uco) on a complete lattice $\langle C, \leq\rangle$ is a function $\mu : C \to C$ that is monotone, idempotent and extensive (i.e., $\forall c \in C.\, c \leq \mu(c)$). Let $\mathrm{uco}(C)$ denote the set of ucos on C. The isomorphism between GIs and uco's is known:

(1) if $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is a GI then $\gamma\alpha \in \mathrm{uco}(C)$, and
(2) if $\mu \in \mathrm{uco}(C)$ then $\mu(C)_{\mu,id} \in \mathrm{Abs}(C)$ is a GI.

Thus, GIs and uco's are completely equivalent approaches for defining abstract domains. We will often abuse notation, and write A for its underlying closure $\gamma\alpha$. For instance, we write $\mathrm{Int}(\{-2, 5\}) = [-2, 5]$. Each closure $\mu \in \mathrm{uco}(C)$ is uniquely determined by the set of its fixpoints, which also coincides with its image $\mu(C)$, because $\mu = \lambda x. \wedge\{y \in \mu(C) \mid x \leq y\}$ holds. Let us also recall that a subset $X \subseteq C$ is the set of fixpoints of a uco on C iff X is meet-closed, i.e., $X = \mathcal{M}(X) \triangleq \{\wedge Y \mid Y \subseteq X\}$ (note that $\top = \wedge\varnothing \in \mathcal{M}(X)$). When C is a complete lattice, $\langle \mathrm{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, id\rangle$ is a complete lattice as well. By the above isomorphism, $\langle \mathrm{Abs}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top_C, id_C\rangle$ denotes the so-called *lattice of abstract interpretations* [18, Section 8], where $A' \sqsubseteq A$ means that A' is more precise than (i.e., is a refinement of) A, and one can consider the most concrete simplification (i.e., lub $\sqcup$) and the most abstract refinement (i.e., glb $\sqcap$) of any family of abstract domains. In particular, recall that $A' \sqsubseteq A$ holds when $\gamma_A(A) \subseteq \gamma_{A'}(A')$ and, consequently, that a lub of a set $\mathcal{X} \subseteq \mathrm{Abs}(C)$ of abstract domains coincides with their intersection, i.e., $\sqcup\{A \mid A \in \mathcal{X}\} = \cap\{\gamma_A(A) \mid A \in \mathcal{X}\}$.

Domain Refinements. Given $A \in \mathrm{Abs}(C)$ and $N \subseteq C$ we define $A \boxplus N \triangleq \mathcal{M}(\gamma_A(A) \cup N)$ as the least refinement of A including the (possibly) new elements in N. Using the shorthand A_N for $A \boxplus N$, it holds that for all $c \in C$, $A_N(c) = \wedge\{x \in (N \cup \{A(c)\}) \mid c \leq x\}$. As a special case, we write A_z for $A \boxplus \{z\}$, with $A_z(c) = z \wedge A(c)$ if $c \leq z$ and $A_z(c) = A(c)$ otherwise. The domain A_z is called a *pointed refinement* of A.

3.2 Regular Commands

Following [8, 48, 51] (see also [62, Chapter 14]), we consider a language of *regular commands* that covers imperative languages as well as other programming paradigms [40, 41].

$$\mathsf{Reg} \ni \mathsf{r} ::= \mathsf{e} \mid \mathsf{r}; \mathsf{r} \mid \mathsf{r} \oplus \mathsf{r} \mid \mathsf{r}^*$$

This language is parametric on the syntax of basic transfer expressions $\mathsf{e} \in \mathsf{Exp}$, which define the basic commands. These can be instantiated, e.g., with (deterministic or nondeterministic or parallel) assignments, Boolean guards, error generation primitives, etc. Moreover, regular commands represent in a compact way the structure of control-flow graphs (CFGs) of imperative programs. The term $\mathsf{r}_1; \mathsf{r}_2$ represents sequential composition, $\mathsf{r}_1 \oplus \mathsf{r}_2$ a choice command that can behave as either r_1 or r_2, and r^* is Kleene iteration, where r can be executed 0 or any bounded number of times.

Concrete Semantics. We assume that basic expressions have a semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ on a complete lattice C such that $(\!|\mathsf{e}|\!)$ is an additive function for any $\mathsf{e} \in \mathsf{Exp}$. This assumption can be done w.l.o.g. in Hoare-like collecting semantics, since the basic transfer functions are defined by additive lifting. In turn, the concrete semantics $[\![\cdot]\!] : \mathsf{Reg} \to$

$C \longrightarrow C$ of regular commands is defined as follows:

$$\llbracket \mathrm{e} \rrbracket c \triangleq (\!|\mathrm{e}|\!)c \qquad \llbracket \mathrm{r}_1 \oplus \mathrm{r}_2 \rrbracket c \triangleq \llbracket \mathrm{r}_1 \rrbracket c \vee_C \llbracket \mathrm{r}_2 \rrbracket c$$
$$\llbracket \mathrm{r}_1; \mathrm{r}_2 \rrbracket c \triangleq \llbracket \mathrm{r}_2 \rrbracket (\llbracket \mathrm{r}_1 \rrbracket c) \qquad \llbracket \mathrm{r}^* \rrbracket c \triangleq \bigvee_C \{ \llbracket \mathrm{r} \rrbracket^n c \mid n \in \mathbb{N} \}$$

Abstract Semantics. The abstract semantics of regular commands $\llbracket \cdot \rrbracket_A^\sharp : \mathrm{Reg} \longrightarrow A \longrightarrow A$ on an abstract domain $A \in \mathrm{Abs}(C)$ is defined by structural induction as follows:

$$\llbracket \mathrm{e} \rrbracket_A^\sharp a \triangleq \llbracket \mathrm{e} \rrbracket^A a = A((\!|\mathrm{e}|\!)a) \qquad \llbracket \mathrm{r}_1 \oplus \mathrm{r}_2 \rrbracket_A^\sharp a \triangleq \llbracket \mathrm{r}_1 \rrbracket_A^\sharp a \vee_A \llbracket \mathrm{r}_2 \rrbracket_A^\sharp a$$
$$\llbracket \mathrm{r}_1; \mathrm{r}_2 \rrbracket_A^\sharp a \triangleq \llbracket \mathrm{r}_2 \rrbracket_A^\sharp (\llbracket \mathrm{r}_1 \rrbracket_A^\sharp a) \qquad \llbracket \mathrm{r}^* \rrbracket_A^\sharp a \triangleq \bigvee_A \{ (\llbracket \mathrm{r} \rrbracket_A^\sharp)^n a \mid n \in \mathbb{N} \}$$

The above abstract semantics is monotonic and sound, i.e., $A \circ \llbracket \mathrm{r} \rrbracket \sqsubseteq \llbracket \mathrm{r} \rrbracket_A^\sharp \circ A$ holds. Note that the abstract semantics of a basic expression e is its bca $\llbracket \mathrm{e} \rrbracket^A$ on A: this agrees with the standard definition by *structural induction* of abstract semantics used in abstract interpretation [16, 54]. Best correct approximations are preserved by choice commands, i.e., $\llbracket \mathrm{r}_1 \oplus \mathrm{r}_2 \rrbracket^A a = \llbracket \mathrm{r}_1 \rrbracket^A a \vee_A \llbracket \mathrm{r}_2 \rrbracket^A a$, but, in general, not by sequential composition and Kleene iteration: for example, $\llbracket \mathrm{r}_2 \rrbracket^A \circ \llbracket \mathrm{r}_1 \rrbracket^A$ is not guaranteed to be the bca $\llbracket \mathrm{r}_1; \mathrm{r}_2 \rrbracket^A$.

Imp. We consider standard basic transfer expressions used in deterministic while programs, as defined below, where, for simplicity, we consider just integer variables $x \in \mathit{Var}$:

$$\begin{aligned} \mathrm{AExp} \ni \mathrm{a} &::= v \in \mathbb{Z} \mid x \in \mathit{Var} \mid \mathrm{a} + \mathrm{a} \mid \mathrm{a} - \mathrm{a} \mid \mathrm{a} * \mathrm{a} \\ \mathrm{BExp} \ni \mathrm{b} &::= \mathbf{tt} \mid \mathbf{ff} \mid \mathrm{a} = \mathrm{a} \mid \mathrm{a} < \mathrm{a} \mid \mathrm{a} \leq \mathrm{a} \mid \mathrm{b} \wedge \mathrm{b} \mid \neg \mathrm{b} \\ \mathrm{Exp} \ni \mathrm{e} &::= \mathbf{skip} \mid x := \mathrm{a} \mid \mathrm{b}? \end{aligned}$$

A standard deterministic imperative language like Winskel's Imp [62] can be easily retrieved using guarded branching and loop commands as syntactic sugar (cf. [40]):

$$\begin{aligned} &\mathbf{if}\ (\mathrm{b})\ \mathbf{then}\ \mathrm{c}_1\ \mathbf{else}\ \mathrm{c}_2 \triangleq\ (\mathrm{b}?; \mathrm{c}_1) \oplus (\neg \mathrm{b}?; \mathrm{c}_2) \\ &\mathbf{while}\ (\mathrm{b})\ \mathbf{do}\ \mathrm{c} \triangleq\ (\mathrm{b}?; \mathrm{c})^*; \neg \mathrm{b}? \\ &\mathbf{do}\ \mathrm{c}\ \mathbf{while}\ (\mathrm{b}) \triangleq\ \mathrm{c};\ (\mathrm{b}?; \mathrm{c})^*; \neg \mathrm{b}? \end{aligned}$$

A program store $\sigma : V \longrightarrow \mathbb{Z}$ is a total function from a finite set of variables of interest $V \subseteq \mathit{Var}$ to values and $\Sigma \triangleq V \longrightarrow \mathbb{Z}$ denotes the set of stores. The concrete domain is $\mathbb{S} \triangleq \wp(\Sigma)$, ordered by inclusion. Store update is defined as usual and lifted to sets $S \in \mathbb{S}$: $S[x \mapsto v] \triangleq \{\sigma[x \mapsto v] \mid \sigma \in S\}$. The semantics $(\!|\mathrm{e}|\!) : \mathbb{S} \longrightarrow \mathbb{S}$ of basic commands is standard:

$$(\!|\mathbf{skip}|\!)S \triangleq S, \qquad (\!|x := \mathrm{a}|\!)S \triangleq \{\sigma[x \mapsto \{\!|\mathrm{a}|\!\}\,\sigma] \mid \sigma \in S\},$$
$$(\!|\mathrm{b}?|\!)S \triangleq \{\sigma \in S \mid \{\!|\mathrm{b}|\!\}\,\sigma = \mathbf{tt}\},$$

where $\{\!|\mathrm{a}|\!\} : \Sigma \longrightarrow \mathbb{Z}$ and $\{\!|\mathrm{b}|\!\} : \Sigma \longrightarrow \{\mathbf{tt}, \mathbf{ff}\}$ are inductively defined as expected. For brevity, we sometimes overload b to denote the set $(\!|\mathrm{b}?|\!)\,\Sigma$ of all and only the stores that satisfy b, so that $(\!|\mathrm{b}?|\!)S = S \cap \mathrm{b}$. When $V = \{x\}$, we let $S \in \wp(\mathbb{Z})$ denote the set $\{\sigma \in \Sigma \mid \sigma(x) \in S\} \in \mathbb{S}$.

4 Making AI Locally Complete

The completeness property $Af = AfA$ for an abstract domain A w.r.t. a transfer function f is, to some extent, a *global* property, meaning that it is universally quantified on all possible concrete inputs, i.e., $\forall c \in C.\ Af(c) = AfA(c)$. In program analysis, the semantic transfer functions f are given by the collecting semantics $\llbracket \mathrm{e} \rrbracket$ of basic commands e, and the corresponding concrete domain $\mathbb{S}$ is the set of all possible input store properties. While completeness can be hard/impossible to achieve globally, as argued by [8], it may well happen that completeness holds *locally*, i.e. just for some input properties, thus giving grounds for investigating a more permissive notion of *local completeness* in program analysis. We investigate how to "repair" an abstract domain through a refinement when local completeness fails. In particular, we provide necessary and sufficient conditions for the existence of a novel type of most abstract domain refinement, called *pointed* (locally complete) *shell*, that compares domain refinements on the basis of their new points rather than their relative precision order $\sqsubseteq$ in the standard lattice of abstract interpretation.

4.1 Local Completeness

Definition 4.1 (Local Completeness [8]). An abstract domain $A \in \mathrm{Abs}(C)$ is *locally complete* for a function $f : C \longrightarrow C$ on a value $c \in C$, denoted by $\mathbb{C}_c^A(f)$, if $Af(c) = AfA(c)$, i.e.

$$\mathbb{C}_c^A(f) \triangleq Af(c) = AfA(c).$$

When the abstraction A is clear from the context, we simply write that f is locally complete on c. □

Local completeness enjoys a sort of convexity property, meaning that whenever $\mathbb{C}_c^A(f)$ holds then $\mathbb{C}_x^A(f)$ holds for any other concrete point $x \in C$ such that $c \leq x \leq A(c)$. We remark that, w.r.t. a compositional program analysis, global and local completeness disclose a significant disparity: while the composition (via generic regular commands, and, consequently, via conditionals and loops of CFGs) of globally complete transfer functions is still globally complete, the same does not necessarily happen for local completeness.

Example 4.2 (Local Completeness is not Compositional). Consider the Imp program

$$\mathrm{c} \triangleq \mathbf{if}\ (0 < x)\ \mathbf{then}\ x := x - 2\ \mathbf{else}\ x := x + 1$$

and the interval abstraction Int. While the Boolean guard $\llbracket 0 < x? \rrbracket$ is not globally complete, it turns out that $\llbracket 0 < x? \rrbracket$ is locally complete for any $P \in \wp(\mathbb{Z})$ such that:

$$(1)\ P \subseteq \mathbb{Z}_{>0},\ \text{or}\ (2)\ P \subseteq \mathbb{Z}_{\leq 0},\ \text{or}\ (3)\ \{0, 1\} \subseteq P.$$

Since the transfer functions for $x \pm k$ are globally complete for Int, c is locally complete for any P satisfying one of the above conditions (1)–(3). For example, if $P_1 = \{2, 5\} \subseteq \mathbb{Z}_{>0}$, then $\mathsf{Int}(\llbracket \mathrm{c} \rrbracket P_1) = \mathsf{Int}(\{0, 3\}) = [0, 3]$ and $\mathsf{Int}(\llbracket \mathrm{c} \rrbracket \mathsf{Int}(P_1)) = \mathsf{Int}(\llbracket \mathrm{c} \rrbracket [2, 5]) = \mathsf{Int}(\{0, 1, 2, 3\}) = [0, 3]$. Contrariwise, if $P_2 = \{0, 3\}$ we have that $\mathsf{Int}(\llbracket \mathrm{c} \rrbracket P_2) = \mathsf{Int}(\{1\}) = [1, 1]$, but $\mathsf{Int}(\llbracket \mathrm{c} \rrbracket \mathsf{Int}(P_2)) = \mathsf{Int}(\llbracket \mathrm{c} \rrbracket [0, 3]) = \mathsf{Int}(\{1, -1, 0\}) = [-1, 1]$. Consider now the composite command c; c. While Int is locally complete for c on P_1, it is not locally complete for c; c on P_1, because $\mathsf{Int}(\llbracket \mathrm{c}; \mathrm{c} \rrbracket \{2, 5\}) = \mathsf{Int}(\llbracket \mathrm{c} \rrbracket \{0, 3\}) = [1, 1]$, but $\mathsf{Int}(\llbracket \mathrm{c}; \mathrm{c} \rrbracket \mathsf{Int}(\{2, 5\})) = \mathsf{Int}(\llbracket \mathrm{c}; \mathrm{c} \rrbracket [2, 5]) = [-1, 1]$. □

Similarly to what is known for global completeness [33], local completeness for an abstraction A can be *constructively characterized*, meaning that there exists a set of concrete points that are expressible in A if and only if A is locally complete. In order to achieve this, transfer functions need to be merely continuous, so that our result is widely applicable.

Definition 4.3 (Local Completeness Set). Let $f : C \longrightarrow C$, $A \in \mathrm{Abs}(C)$ and $c \in C$. The *local completeness set* $\mathbb{L}^A_{c,f} \subseteq C$ is defined by $\mathbb{L}^A_{c,f} \triangleq \{x \in C \mid x \leq A(c),\ f(x) \leq Af(c)\}$. □

The local completeness set $\mathbb{L}^A_{c,f}$ includes c as well as any concrete element x below $A(c)$ (notice that x is not necessarily comparable with c) such that its image $f(x)$ is bounded from above in A by $Af(c)$. By resorting to an equivalent formulation of the axiom of choice called Hausdorff maximal principle, it turns out that any element in $\mathbb{L}^A_{c,f}$ is dominated by a maximal one in $\max(\mathbb{L}^A_{c,f})$, and this technically enables the following characterization for continuous transfer functions, which is further specialized for additive functions.

Theorem 4.4 (Characterization of Local Completeness). *Let $f : C \longrightarrow C$, $A \in \mathrm{Abs}(C)$ and $c \in C$.*

(i) *If f is continuous then, $\mathbb{C}^A_c(f) \Leftrightarrow \max(\mathbb{L}^A_{c,f}) \subseteq A$.*
(ii) *In particular, if f is additive then, $\mathbb{C}^A_c(f) \Leftrightarrow \vee\mathbb{L}^A_{c,f} \in A$. Moreover, $\vee\mathbb{L}^A_{c,f} = A(c) \wedge \big(\vee\{x \in C \mid f(x) \leq Af(c)\}\big)$.*

Example 4.5. Let us consider the interval abstraction Int and the function $[\![\mathrm{c}]\!] : \wp(\mathbb{Z}) \longrightarrow \wp(\mathbb{Z})$ of Example 4.2 which has been shown locally complete on $P_1 = \{2, 5\}$, but not on $P_2 = \{0, 3\}$. In fact, coherently with Theorem 4.4 (ii), it turns out that (recall that $[\![\mathrm{c}]\!]$ is an additive function):
(1) $\mathbb{L}^{\mathsf{Int}}_{P_1,[\![\mathrm{c}]\!]} = \{X \in \wp(\mathbb{Z}) \mid X \subseteq \mathsf{Int}(P_1),\ [\![\mathrm{c}]\!]X \subseteq \mathsf{Int}([\![\mathrm{c}]\!]P_1)\} = \{X \in \wp(\mathbb{Z}) \mid X \subseteq [2, 5],\ [\![\mathrm{c}]\!]X \subseteq [0, 3]\}$ and $\cup\mathbb{L}^{\mathsf{Int}}_{P_1,[\![\mathrm{c}]\!]} = [2, 5]$
(2) $\mathbb{L}^{\mathsf{Int}}_{P_2,[\![\mathrm{c}]\!]} = \{X \in \wp(\mathbb{Z}) \mid X \subseteq \mathsf{Int}(P_2),\ [\![\mathrm{c}]\!]X \subseteq \mathsf{Int}([\![\mathrm{c}]\!]P_2)\} = \{X \in \wp(\mathbb{Z}) \mid X \subseteq [0, 3],\ [\![\mathrm{c}]\!]X \subseteq [1, 1]\}$ and $\cup\mathbb{L}^{\mathsf{Int}}_{P_2,[\![\mathrm{c}]\!]} = \{0, 3\}$
so that $\cup\mathbb{L}^{\mathsf{Int}}_{P_1,[\![\mathrm{c}]\!]} \in \mathsf{Int}$, while $\cup\mathbb{L}^{\mathsf{Int}}_{P_2,[\![\mathrm{c}]\!]} \notin \mathsf{Int}$. □

4.2 Exact Locally Complete Shells

If local completeness $\mathbb{C}^A_c(f)$ fails during some step of a program abstract interpretation on a domain A, we are interested in *repairing* the abstract domain A in order to retrieve local completeness in that failure step. When this repair procedure is iterated along an abstract trace, the objective is to make our abstract domain locally complete along that abstract computation. Domain refinement is an obvious choice to repair the local completeness of some abstraction. The ideal goal would be to enhance A by adding the minimum number of new abstract points that allows us to attain local completeness. This kind of minimal refinement of abstract domains to achieve some domain property is known as *exact shell* [26] once abstract domains are compared w.r.t. their relative precision ordering $\sqsubseteq$ in the lattice of abstract interpretation $\mathrm{Abs}(C)$ (cf. Section 3.1). For the property of local completeness of a transfer function $f : C \longrightarrow C$ on some point $c \in C$, the exact locally complete shell of A exists when its minimal refinement $A_{\mathrm{ref}} \triangleq \sqcup\{B \in \mathrm{Abs}(C) \mid B \sqsubseteq A,\ \mathbb{C}^B_c(f)\}$ turns out to be locally complete on c. Unfortunately, the following example shows that, in general, this minimal refinement does not exist, even under strong hypotheses.

Example 4.6 (Exact Shells May Not Exist). Let us consider a simple collecting transfer function $f : \wp(\mathbb{Z}) \longrightarrow \wp(\mathbb{Z})$ defined as follows: $f(X) \triangleq \{x + 1 \mid x \in X\}$. Thus, f is the additive transformer of the imperative assignment $x := x+1$, namely, $f = [\![x := x+1]\!]$. We consider the toy abstract domain $A \triangleq \{\mathbb{Z}, [0, 4], [1, 3]\} \in \mathrm{Abs}(\wp(\mathbb{Z}))$, which is a finite chain of three integer intervals and is deliberately artificial for the ease of exposition. For the store property $P = \{0, 2\}$, observe that $\mathbb{C}^A_P(f)$ does not hold, because: $Af(P) = A(\{1, 3\}) = [1, 3]$ while $AfA(P) = Af([0, 4]) = A([1, 5]) = \mathbb{Z}$.
Consider the following two pointed refinements of A:

$$A_{[0,2]} = A \cup \{[0, 2], [1, 2]\}, \quad A_{\{0,2\}} = A \cup \{\{0, 2\}, \{2\}\}.$$

Since $P \in A_{\{0,2\}}$, we have that $A_{\{0,2\}}$ is trivially locally complete on P. Moreover, $A_{[0,2]}$ also achieves local completeness on P: $A_{[0,2]}fA_{[0,2]}(P) = A_{[0,2]}f([0, 2]) = A_{[0,2]}([1, 3]) = [1, 3] = A_{[0,2]}(\{1, 3\}) = A_{[0,2]}f(P)$. Therefore, $A_{[0,2]}, A_{\{0,2\}} \in \{B \in \mathrm{Abs}(\wp(\mathbb{Z})) \mid B \sqsubseteq A,\ \mathbb{C}^B_P(f)\}$. Since $A_{[0,2]} \sqcup A_{\{0,2\}} = A_{[0,2]} \cap A_{\{0,2\}} = A$, it turns out that $A_{\mathrm{ref}} = \sqcup\{B \in \mathrm{Abs}(\wp(\mathbb{Z})) \mid B \sqsubseteq A,\ \mathbb{C}^B_P(f)\} = A$. This entails that the exact shell refinement of A which is locally complete on P does not exist. □

The above example shows that even when the concrete domain is a complete Boolean algebra, the transfer function is both additive and co-additive and the abstract domain is a finite chain, exact shells of local completeness do not necessarily exist. This observation makes it impossible to find some adequate sufficient conditions on the transfer function and/or the abstract domain guaranteeing the existence of locally complete shells. Nevertheless, it is worth remarking that the locally complete shell for some specific abstract domain, transfer function and concrete point may well exist.

4.3 Pointed Locally Complete Shells

Let us focus on pointed refinements A_x of a given abstraction A. The following result shows which pointed refinements A_x may achieve local completeness.

Lemma 4.7. *Let $f : C \longrightarrow C$ be monotone, $A \in \mathrm{Abs}(C)$ and $c \in C$. Assume that local completeness does not hold, i.e., $Af(c) \neq AfA(c)$. If $x \in C$ is such that $\mathbb{C}^{A_x}_c(f)$, i.e. local completeness holds, and $x \not\leq A(c)$ then there exists some $y < A(c)$ such that $A_x \sqsubseteq A_y$ and $\mathbb{C}^{A_y}_c(f)$.*

Hence, this result tells us that if A is not locally complete on c and we are looking for a pointed refinement A_x that achieves local completeness on c then we can restrict our search to those new points x which are below $A(c)$, because for $x \not\leq A(c)$ a more abstract pointed refinement A_y which is locally complete and with $y < A(c)$ can always be found. However, as shown in Section 4.2, the most abstract pointed

refinement of A which is locally complete on c, in general, does not exist. For instance, in Example 4.6 we have shown that the two pointed refinements $A_{[0,2]}$ and $A_{\{0,2\}}$ are both locally complete but their common abstraction $A_{[0,2]} \sqcup A_{\{0,2\}}$ is A itself, which is not locally complete. Notice that for the standard precision ordering $\sqsubseteq$ in the lattice of abstract interpretations $\mathrm{Abs}(C)$, the locally complete pointed refinements $A_{[0,2]}$ and $A_{\{0,2\}}$ turn out to be incomparable because $A_{[0,2]} \not\subseteq A_{\{0,2\}}$ and $A_{\{0,2\}} \not\subseteq A_{[0,2]}$. Nevertheless, we argue that *the pointed refinement $A_{[0,2]}$ should be preferred to $A_{\{0,2\}}$* because the new point $[0,2]$ is more abstract than $\{0,2\}$, namely, $\{0,2\} \subsetneq [0,2]$ holds. One further reason supporting our claim is that, due to the convexity property of local completeness mentioned after Definition 4.1, the choice of a more abstract approximation $c^\sharp$ for c guarantees that local completeness holds not only for c but also for any concrete element in between c and $c^\sharp$. We therefore put forward a novel notion of shell refinement of abstract domains, which is restricted to pointed refinements A_x and compares them by the relative precision ordering of their new point x.

Definition 4.8 (Pointed Shells). Let $f : C \longrightarrow C$ be monotone, $A \in \mathrm{Abs}(C)$ and $c \in C$. The *pointed locally complete shell*, *pointed shell* for short, of A in c exists when

$$\max(\{x \in C \mid x \leq A(c),\ \mathbb{C}_c^{A_x}(f)\}) = \{u\} \tag{1}$$

and, in such a case, this pointed shell is $A_u \in \mathrm{Abs}(C)$. □

Let us remark that the condition $x \leq A(c)$ in (1) is justified by Lemma 4.7, as discussed above. We therefore give the following main result that: (i) by leveraging Theorem 4.4, characterizes the point u of a pointed shell A_u, and (ii) when f is additive, provides a necessary and sufficient condition for the existence of a pointed shell.

Theorem 4.9 (Existence of Pointed Shells). *Let us consider $f : C \longrightarrow C$, $A \in \mathrm{Abs}(C)$ and $c \in C$, and let $u \triangleq \vee \mathbb{L}^A_{c,f}$.*

(i) *If f is monotone then: $\mathbb{C}_c^{A_u}(f) \Leftrightarrow A_u$ is the pointed shell of A for f on c.*
(ii) *If f is additive then: $\mathbb{C}_c^{A_u}(f) \Leftrightarrow (f(c) \leq u \Rightarrow f(u) \leq u)$.*

Hence, for an additive function f, such as a collecting semantics, and the new concrete element $u = \vee \mathbb{L}^A_{c,f}$, the pointed shell of A exists exactly when $f(c) \not\leq u$ or $f(u) \leq u$ holds, and in such a case A_u is the pointed shell.

Example 4.10. Consider again Example 4.6 dealing with local completeness of the function $f = [\![x := x + 1]\!]$ on input $P = \{0, 2\}$ in the toy abstract domain $A = \{\mathbb{Z}, [0,4], [1,3]\}$. We have: $\mathbb{L}^A_{P,f} = \{X \in \wp(\mathbb{Z}) \mid X \subseteq A(P),\ f(X) \subseteq Af(P)\} = \{X \in \wp(\mathbb{Z}) \mid X \subseteq [0,4],\ f(X) \subseteq [1,3]\}$, so that $u = \cup \mathbb{L}^A_{P,f} = [0,2]$. Since $f(P) = \{1,3\} \not\subseteq [0,2] = u$, the condition of Theorem 4.9 (ii) is satisfied, because its premise $f(P) \subseteq u$ is false, thus $A_{[0,2]}$ is the pointed shell of A for f on P.
Consider now Example 4.5, showing that Int is not locally complete on $P_2 = \{0, 3\}$ for the program c in Example 4.2. We have already seen that $u = \cup \mathbb{L}^{\mathsf{Int}}_{P_2,[\![\mathrm{c}]\!]} = \{0, 3\}$. Since $[\![\mathrm{c}]\!]P_2 = \{1\} \not\subseteq \{0, 3\} = u$ holds, the condition $[\![\mathrm{c}]\!]P_2 \subseteq u \Rightarrow [\![\mathrm{c}]\!]u \subseteq u$ of Theorem 4.9 (ii) is satisfied, and $\mathsf{Int}_{\{0,3\}} = \mathsf{Int} \cup \{0, 3\}$ is the pointed shell of Int for $[\![\mathrm{c}]\!]$ on P_2. □

4.4 Boolean Guards

Boolean guards are particularly important in this scenario because, as argued in [28], they represent the major source of incompleteness. In this case we are interested in achieving local completeness for two transfer functions $[\![\mathrm{b?}]\!]$ and $[\![\neg \mathrm{b?}]\!]$. Definition 4.8 can be slightly generalized as follows: the pointed locally complete shell of A in c exists for a set of functions $F \subseteq C \longrightarrow C$ when:

$$\max(\{x \in C \mid x \leq A(c), \forall f \in F.\, A_x f(c) = A_x f A_x(c)\}) = \{u\}.$$

If this is the case, then A_u is defined to be the pointed (locally complete) shell of A for F. Interestingly, we constructively prove that pointed shells for Boolean guards always exist.

Theorem 4.11 (Pointed Shell for Boolean Guards). *Given $\mathrm{b} \in \mathrm{BExp}$, $A \in \mathrm{Abs}(\mathbb{S})$ and $P \in \mathbb{S}$, let*

$$u \triangleq (A(P \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(P \cap \neg \mathrm{b}) \cap \neg \mathrm{b}) \in \mathbb{S}.$$

Then, A_u is the pointed shell for $\{[\![\mathrm{b?}]\!], [\![\neg \mathrm{b?}]\!]\}$ on P.

Example 4.12. Let us apply Theorem 4.11 to the case of the interval domain and the Boolean guard $\mathrm{b} \triangleq (x > 0)$? on the point of incompleteness $P = \{-3, -1, 2\}$. Letting

$$\begin{aligned} u &\triangleq (\mathsf{Int}(P \cap x \leq 0) \cap x \leq 0) \cup (\mathsf{Int}(P \cap x > 0) \cap x > 0) \\ &= (\mathsf{Int}(\{-3,-1\}) \cap \neg \mathrm{b}) \cup (\mathsf{Int}(\{2\}) \cap \mathrm{b}) = [-3,-1] \cup \{2\}, \end{aligned}$$

by closing under meets, we obtain that the pointed shell is $\mathsf{Int}_u = \mathsf{Int} \cup \{[-3,-1] \cup \{2\}, [-2,-1] \cup \{2\}, \{-1,2\}\}$. □

5 Repair Strategies

Whenever the current abstract domain is not precise enough to prevent false-alarms, we advocate AIR as a way to optimally refine the abstract domain so to remove false-alarms.

The general scenario consists of a composite transfer function $f \triangleq f_n \circ \ldots \circ f_1$, e.g., modeling the sequential composition of $f_1, \ldots, f_n : C \longrightarrow C$, a concrete input c and a correctness specification a for which we want to decide whether $f(c) \leq a$ is satisfied, without actually computing $f(c)$. In CEGAR, the functions f_k are the post transformers of a transition system, while in program verification each f_k is the transfer function of some basic command. In abstract interpretation, we select an abstract domain $A \in \mathrm{Abs}(C)$ such that the property a is expressible in A and then we check whether $f^\sharp_A A(c) \leq_A a$ holds, where $f^\sharp_A \triangleq f^A_n \circ \ldots \circ f^A_1$ and each f^A_i is the bca in A of f_i. In the positive case, by soundness of abstract interpretation, we are done. Otherwise, we cannot tell if the specification a is met or not, because the bca f^A of f in A does not coincide, in general, with $f^\sharp_A$. However, if $f^\sharp_A A(c) = A(f(c))$ holds, then from $f^\sharp_A A(c) \not\leq_A a$ we can conclude that a is not met, because $A(f(c)) \not\leq_A a$ implies $f(c) \not\leq a$ when $a \in A$.

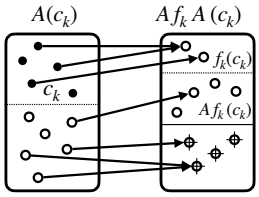


Figure 1. A bca repair.

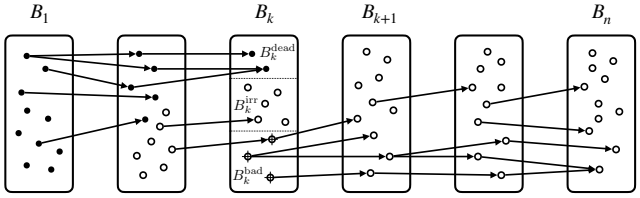


Figure 2. A spurious counterexample.

Forward Repair Strategy. By letting $c_k \triangleq f_{k-1} \ldots f_1(c)$, with $c_1 \triangleq c$, the equality $f_A^\sharp A(c) = A(f(c))$ is a consequence of n local completeness proof obligations:

$$\forall k \in [1, n].\ Af_k(c_k) = Af_k A(c_k).$$

The forward repair strategy processes these proof obligations in increasing index order: as soon as the smallest index k is found such that $Af_k(c_k) \neq Af_k A(c_k)$, then A is repaired by some pointed shell A_u and the analysis is restarted in the refined domain A_u, now locally complete for f_k on c_k.

Let us focus on a step where $Af_k(c_k) \neq Af_k A(c_k)$, as depicted in Fig. 1: the states in c_k are black-filled while the source of incompleteness is due to the crossed states in $Af_k A(c_k) \smallsetminus Af_k(c_k)$. To repair this incompleteness we could thus introduce any approximation u of c_k such that $c_k \leq u \leq A(c_k)$ and $A_u f_k(u) = A_u f_k(c_k)$. Our methodology is simple: when a local completeness violation is detected, we refine the domain A to the pointed shell A_u by adding $u \triangleq \vee \mathbb{L}^A_{c_k, f_k}$ (cf. Theorem 4.9), when it exists, otherwise we can just take $u = c_k$, which is the most concrete pointed refinement. Note that by taking the pointed shell, the refined domain turns out to be locally complete for f_k on any concrete x in between c_k and u (by abstract convexity), so this refinement is, e.g., more likely reusable within abstract fixpoint computations. Moreover, as explained in Section 4.3, the *pointed shell is the best possible refinement strategy* we can locally apply for removing the incompleteness of f_k on c_k. However, this forward method requires to compute the concrete element c_k on which local completeness is violated.

Forward repair stops as soon as we obtain a domain that is precise enough to prove the correctness specification a or such that all the local completeness requirements are met. The worst case happens when $f(c) \not\leq a$, because this means that all the local completeness equations must be inspected and, consequently, all the concrete elements c_k must be computed.

Backward Repair Strategy. Forward repair focuses on the strongest postconditions c_k and aims at a refinement A_f such that $f_n^{A_\mathrm{f}} \ldots f_{k+1}^{A_\mathrm{f}} A_\mathrm{f}(c_k) \leq a$, where, by local completeness of A_f for f_k on c_k, we have that $f_n^{A_\mathrm{f}} \ldots f_{k+1}^{A_\mathrm{f}} A_\mathrm{f}(c_k) = f_n^{A_\mathrm{f}} \ldots f_1^{A_\mathrm{f}} A_\mathrm{f}(c) = f_{A_\mathrm{f}}^\sharp A_\mathrm{f}(c)$ holds. Backward repair relies instead on weakest liberal preconditions [24] and looks for a refinement A_b such that $A_\mathrm{b}(f_n \ldots f_{k+1} f_k^{A_\mathrm{b}} \ldots f_1^{A_\mathrm{b}} A_\mathrm{b}(c)) \leq a$ holds, and $A_\mathrm{b}(f_n \ldots f_{k+1} f_k^{A_\mathrm{b}} \ldots f_1^{A_\mathrm{b}} A_\mathrm{b}(c)) = f_{A_\mathrm{b}}^\sharp A_\mathrm{b}(c)$ follows by local completeness. Recall that, given $f : C \longrightarrow C$ and $z \in C$, the *weakest liberal precondition* of f for z is

$$\textsc{wlp}(f, z) \triangleq \vee\{x \in C \mid f(x) \leq z\}.$$

When f is additive, the key property of WLP is that $f(c) \leq a$ is equivalent to $c \leq \textsc{wlp}(f, a)$, and, analogously, $f_A^\sharp A(c) \leq a$ iff $A(c) \leq \textsc{wlp}(f_A^\sharp, a)$. Letting $a_k \triangleq f_k^A \ldots f_1^A A(c)$, we inductively define $v_k \triangleq a_k \wedge \textsc{wlp}(f_{k+1}, v_{k+1})$, with $v_n \triangleq a_n \wedge a$. Thus, backward repair amounts to check and possibly repair n local completeness proof obligations:

$$\forall k \in [1, n].\ Af_k(v_k) = Af_k A(v_k),$$

which are best processed in decreasing order, from n to 1.

Forward and backward strategies compute the pointed shells for different concrete elements, as in general $c_k \neq v_k$. Backward repair looks more appealing for several reasons: (i) backward repair mainly operates on abstractions $A(c)$ and not on concrete elements c; (ii) the abstract domain A_b can be used to decide the (in)correctness of any $d \leq A(c)$, whereas the domain A_f of a forward repair can only be used to decide the (in)correctness of any element d such that $c \leq d \leq A(c)$; (iii) the pointed shell for f_k on v_k is just A_{v_k}: v_k is the largest element that satisfies the correctness specification, so any larger approximation would compromise the validity of that property; (iv) finally, a major distinction is that, after any repair, the forward strategy *must redo the abstract interpretation*, while the backward strategy can apply the next repair (if necessary) along the *existing abstract computation*.

6 CEGAR as AIR

CounterExample-Guided Abstraction Refinement (CEGAR) [9, 11] is a popular abstraction refinement strategy applicable to the verification of temporal logic formulas. CEGAR tackles the critical state explosion problem [10] by model checking an abstract model instead of the concrete one.

CEGAR in a Nutshell. Consider a finite state transition system $\mathcal{S} = \langle \Sigma, \rightarrow \rangle$ whose successor/predecessor transformers post/pre are defined as $\mathrm{post}(X) \triangleq \{t \mid \exists s \in X, s \rightarrow t\}$ and $\mathrm{pre}(X) \triangleq \{s \mid \exists t \in X, s \rightarrow t\}$, for all $X \in \wp(\Sigma)$. Given a partitioning abstraction $A \in \mathrm{Abs}(\wp(\Sigma))$ [43, 52, 53], whose abstract elements are unions of blocks of a partition of the state space Σ, abstract model checking [2, 12, 13] exploits the existential abstract transition relation $\rightarrow^\sharp \subseteq A \times A$ defined as follows: for any $B, B' \in A$, $B \rightarrow^\sharp B'$ iff $\exists x \in B. \exists y \in B'. x \rightarrow y$. The resulting abstract transition system $\mathcal{A} = \langle A, \rightarrow^\sharp \rangle$ is then

model checked to prove the validity of a temporal specification. Typically, a counterexample is a finite abstract path $\pi = \langle B_1, ..., B_n \rangle$ such that $B_{i-1} \to^\sharp B_i$ for any $i \in [2, n]$ (for simplicity, loop path counterexamples are not treated here). Its underlying concrete paths are defined as follows: $\mathrm{paths}(\pi) \triangleq \{\langle s_1, ..., s_n \rangle \in \Sigma^n \mid \forall i \in [1, n].\, s_i \in B_i,\ \forall i \in [2, n].\, s_{i-1} \to s_i\}$. An abstract path π is *spurious* if $\mathrm{paths}(\pi) = \varnothing$ and we define a corresponding sequence $\mathrm{sp}(\pi) = \langle S_1, ..., S_n \rangle \in \wp(\Sigma)^n$ as:

$$S_1 \triangleq B_1 \neq \varnothing; \quad \forall i \in [1, n-1].\, S_{i+1} \triangleq \mathrm{post}(S_i) \cap B_{i+1}. \quad (2)$$

Hence, S_i is the set of states in a block B_i that are reachable (in $i-1$ concrete steps) from a state in B_1. As observed in [11, Lemma 4.10], it turns out that π is spurious iff there exists a least $k \in [1, n-1]$ such that $S_{k+1} = \varnothing$. Fig. 2 depicts a spurious counterexample $\pi = \langle B_1, ..., B_n \rangle$, where for all $i \in [1, k]$, the nodes in S_i are black-filled. The partitioning abstract domain A is refined by splitting the block B_k, and, in turn, the model checker is run on this refined abstract transition system. The refinement of the block B_k relies on the three following subsets of states: *dead-end* states $B_k^{\mathrm{dead}} \triangleq S_k \neq \varnothing$ (black-filled circles in Fig. 2, those reachable by some underlying concrete trace, but with no arcs to nodes in B_{k+1}), must necessarily be separated from *bad* states $B_k^{\mathrm{bad}} \triangleq B_k \cap \mathrm{pre}(B_{k+1}) \neq \varnothing$ (crossed circles in Fig. 2, those with some arcs to nodes in B_{k+1}); all the other states $B_k^{\mathrm{irr}} \triangleq B_k \setminus (B_k^{\mathrm{dead}} \cup B_k^{\mathrm{bad}})$ are called *irrelevant*. However, when states are memory stores for program variables, the problem of finding the coarsest (and therefore cheapest) refinement of A that separates dead-end and bad states turns out to be NP-hard [11, Theorem 4.17]. This entails that, in practice, some refinement heuristics must necessarily be used. According to the basic heuristics of CEGAR [11, Section 4], B_k is split into B_k^{dead} and $B_k^{\mathrm{bad}} \cup B_k^{\mathrm{irr}}$. By looking at Fig. 2, notice that other spurious counterexamples may still arise after the refinement (due to the arcs from nodes in B_{k-1} to those in $B_k^{\mathrm{bad}} \cup B_k^{\mathrm{irr}}$).

CEGAR by Forward Repair. Given a finite abstract path $\pi = \langle B_1, ..., B_n \rangle$ on a partitioning abstraction A, let

$$\mathrm{post}_{\pi_k}(X) \triangleq \mathrm{post}(X) \cap B_{k+1} \quad (3)$$

for any $k \in [1, n-1]$ and $X \in \wp(\Sigma)$. Obviously, for any k, we have that $S_{k+1} = \mathrm{post}_{\pi_k}(S_k)$.

Lemma 6.1. *π is not spurious iff for all $k \in [1, n-1]$,*

$$A(\mathrm{post}_{\pi_k}(S_k)) = A(\mathrm{post}_{\pi_k}(A(S_k))). \quad (4)$$

Notably, (4) expresses the local completeness of the abstraction A for post_{π_k} on the concrete set of states $S_k \in \wp(\Sigma)$. For applying the forward repair of Section 5 to abstract model checking we consider $c = B_1$, $f_i = \mathrm{post}_{\pi_i}$ and $a = \varnothing$. To discharge the counterexample π, we have to guarantee that there is no underlying path from states in $c = B_1$ to states in B_n, i.e., π is spurious iff $f_n \cdots f_1(B_1) \subseteq \varnothing$. For a spurious counterexample we then have a least $k \in [1, n-1]$ such that:

$$\varnothing = A(\varnothing) = A(\mathrm{post}_{\pi_k}(S_k)) \neq A(\mathrm{post}_{\pi_k}(A(S_k)) = A(\mathrm{post}_{\pi_k}(B_k)) = B_{k+1}.$$

To repair this local incompleteness of A, we need to compute for S_k a more precise abstraction than the current block B_k. Recall that $S_k = B_k^{\mathrm{dead}}$ and that any state in B_k^{bad} cannot be part of the abstraction of S_k (otherwise local incompleteness persists). Thus, our forward repair strategy takes the pointed shell refinement $A \boxplus \{B_k^{\mathrm{dead}} \cup B_k^{\mathrm{irr}}\} = A \boxplus \{B_k^{\mathrm{dead}} \cup B_k^{\mathrm{irr}}, B_k^{\mathrm{bad}}\}$. Again, other spurious counterexamples may still arise (see the arcs in Fig. 2 from B_{k-1} to crossed nodes in B_k^{bad}).

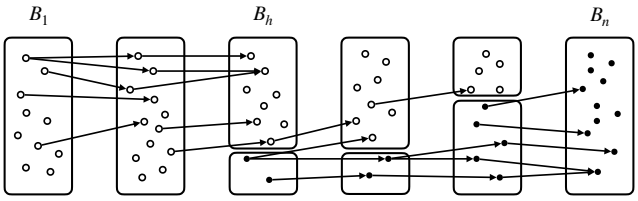


Figure 3. Backward repair from block B_n to block B_h.

Theorem 6.2. *Let $k \in [1, n-1]$ such that $S_k \neq \varnothing$ and $S_{k+1} = \varnothing$. Then, $A \boxplus \{B_k^{\mathrm{dead}} \cup B_k^{\mathrm{irr}}\}$ is the pointed shell of A on S_k.*

CEGAR by Backward Repair. In backward repair, the elements $V_k = B_k \cap \mathrm{WLP}(\mathrm{post}_{\pi_k}, V_{k+1})$ (with $V_n = \varnothing$) can just be defined as $V_k \triangleq B_k \setminus T_k$, where $T_k \subseteq B_k$ is the set of states with a path of length $n-k$ to some state in B_n. Formally:

$$T_n \triangleq B_n \neq \varnothing; \quad \forall i \in [1, n-1].\, T_i \triangleq \mathrm{pre}(T_{i+1}) \cap B_i.$$

Lemma 6.3. *π is spurious iff there exists $A' \sqsubseteq A$ such that $A'(V_1) = B_1$ and for all $k \in [1, n-1]$,*

$$A'(\mathrm{post}_{\pi_k}(V_k)) = A'(\mathrm{post}_{\pi_k}(A'(V_k))). \quad (5)$$

Since V_k is the largest subset of B_k such that $\mathrm{post}_{\pi_k}(V_k) \subseteq V_{k+1}$, when V_{k+1} is expressible (as $V_n = \varnothing$), condition (5) boils down to $A'(V_k) = V_k$, so that $A \boxplus \{V_k\}$ is the pointed shell.

Theorem 6.4. *Let $k \in [1, n-1]$ be the largest index s.t. (5) does not hold. Then, $A' \boxplus \{V_k\}$ is the pointed shell of A' on V_k.*

Fig. 3, where nodes in T_i are black-filled, shows how backward repair iterates on the example of Fig. 2: notably, *no residual spurious abstract path abides in the abstract model.*

7 Program Verification Repair

Notation. In the following, we overload the abstract semantic function $[\![\mathsf{r}]\!]_A^\sharp : A \to A$ to also denote the function $[\![\mathsf{r}]\!]_A^\sharp \circ A : C \to A$ applicable to concrete input properties. Moreover, we write a simpler $\mathbb{C}_R^A(\mathsf{r})$ instead of $\mathbb{C}_R^A([\![\mathsf{r}]\!])$ for local completeness proof obligations. Note that when A is repaired to A_u then the new transfer functions for A_u can be computed in terms of those for A by letting $[\![\mathsf{e}]\!]_{A_u}^\sharp a = u \wedge [\![\mathsf{e}]\!]_A^\sharp a$ if $[\![\mathsf{e}]\!]_A^\sharp a \leq u$ and $[\![\mathsf{e}]\!]_{A_u}^\sharp a = [\![\mathsf{e}]\!]_A^\sharp a$ otherwise.

The verification problem for a program $\mathsf{r} \in \mathrm{Reg}$ on input $P \in C$ and correctness specification $Spec \in C$, consists in

checking whether $[\![\mathsf{r}]\!]P \leq \mathit{Spec}$ holds. Abstract interpretation exploits over-approximations to interpret r on a (tractable) abstract domain A. By soundness of $[\![\mathsf{r}]\!]^\sharp_A$, if $[\![\mathsf{r}]\!]^\sharp_A P \leq \mathit{Spec}$ holds then $[\![\mathsf{r}]\!]P \leq \mathit{Spec}$ follows, but the reverse implication is not necessarily true. Nevertheless, if the abstraction A is locally complete for $[\![\mathsf{r}]\!]$ on P and Spec is expressible in A then $[\![\mathsf{r}]\!]P \leq \mathit{Spec}$ implies $[\![\mathsf{r}]\!]^\sharp_A P \leq \mathit{Spec}$. In this section we apply AIR to iteratively repair an initial, possibly incomplete, abstract domain A to get a refined domain A_N such that

$$[\![r]\!]P \leq \mathit{Spec} \quad \Leftrightarrow \quad [\![\mathsf{r}]\!]^\sharp_{A_N} P \leq \mathit{Spec} \tag{6}$$

To see the analogy with abstract model checking, given a CEGAR abstract counterexample $\pi = \langle B_1, ..., B_n\rangle$, assume that we define a regular command $\mathsf{r}_\pi \triangleq \mathsf{e}_1; \mathsf{e}_2; ...; \mathsf{e}_n$ such that the semantics $[\![\mathsf{e}_k]\!]$ of each basic expression is exactly the function post_{π_k} defined in (3). Then, if we take the input property $P \triangleq B_1$ and a trivial specification $\mathit{Spec} \triangleq \bot_C$, it turns out that $[\![\mathsf{r}_\pi]\!]P \leq \mathit{Spec}$ holds iff π is spurious. The correspondence with the general scenario presented in Section 5 is recovered by taking $c = P$, $a = \mathit{Spec}$ and $f_k = \mathrm{post}_{\pi_k} = [\![\mathsf{e}_k]\!]$.

7.1 Program Verification by Forward Repair

The forward repair strategy is described by the pseudocode in Algorithm 1. Here, we do not commit to a specific methodology to detect local incompleteness, because the approach is independent of the way in which such completeness counterexamples are found. Thus, we merely assume that an *oracle function* find_A is available for the abstraction A, which takes as input a refinement $A_N = A \boxplus N$, simply passed as N, a current set of stores P and a command r and, either returns an under-approximation Q satisfying $Q \leq [\![\mathsf{r}]\!]P$ and $A_N(Q) = A_N([\![\mathsf{r}]\!]P)$ — thus meaning that A_N is locally complete for $[\![\mathsf{r}]\!]$ on P — or returns a pair $\langle R, \mathsf{e}\rangle$ for some set of stores R and basic command e occurring in r such that a local completeness proof obligation $\mathbb{C}^{A_N}_R(\mathsf{e})$ is not met.

The procedure fRepair calls the oracle find_A, and if this returns a pair $\langle R, \mathsf{e}\rangle$ for a failed proof obligation $\mathbb{C}^{A}_R(\mathsf{e})$, then the abstraction A is repaired by taking the pointed shell returned by $\mathsf{refine}_A(N, R, \mathsf{e})$. More precisely, $\mathsf{refine}_A(N, R, \mathsf{e})$ takes as input the current domain refinement $A_N = A \boxplus N$, the current set of stores R and a basic expression e, and outputs a finite set $N' \supseteq N$ such that $\mathbb{C}^{A_{N'}}_R(\mathsf{e})$ holds. This oracle function find_A is iteratively called until an under-approximation Q is eventually output by find_A. Summing up, $\mathsf{fRepair}_A(N, P, \mathsf{r})$ takes as input a refinement $A_N = A \boxplus N$, a command r and a concrete input P, and returns a pair $\langle N', Q\rangle$ such that $N' \supseteq N$, $Q \leq [\![\mathsf{r}]\!]P$, and $A_{N'}(Q) = A_{N'}([\![\mathsf{r}]\!]P)$.

Theorem 7.1 (fRepair **is Sound).** *For all $A \in \mathrm{Abs}(C)$, finite $N \subseteq C$, $P \in C$, and $\mathsf{r} \in \mathrm{Reg}$, if $\mathsf{fRepair}_A(N, P, \mathsf{r}) = \langle Q, N'\rangle$, then $N' \supseteq N$, $\mathbb{C}^{A_{N'}}_P(\mathsf{r})$ and $Q \leq [\![\mathsf{r}]\!]P \leq A_{N'}(Q) = A_{N'}([\![\mathsf{r}]\!]P)$.*

Example 7.2. Consider the regular command for **AbsVal**:

$$\mathsf{r}_{\mathrm{Abs}}(x) \triangleq ((x \geq 0)?; \mathbf{skip}) \oplus ((x < 0)?; x := -x)$$

Algorithm 1: Forward repair procedure $\mathsf{fRepair}_A$

```
1 Function fRepair_A(N, P, r)
2     found := false;
3     do
4         out := find_A(N, P, r);
5         switch out do
6             case Q do found := true;            // underapprox.
7             case ⟨R, e⟩ do N := refine_A(N, R, e); // incompl.
8     while (¬found);
9     return ⟨N, out⟩;
```

As discussed in Section 1, the analysis on Int of $\mathsf{r}_{\mathrm{Abs}}$ on input $I \triangleq \{x \mid x \text{ is odd}\}$ and $\mathit{Spec} \triangleq [1, +\infty]$ raises an alarm for the allowed output $x = 0 \notin \mathit{Spec}$. To recognize whether this is a true- or false-alarm, we apply the forward repair Algorithm 1, where the oracle $\mathsf{find}_{\mathsf{Int}}(\varnothing, I, \mathsf{r}_{\mathrm{Abs}})$ returns, as expected, the proof obligation $\mathbb{C}^{\mathsf{Int}}_I((x \geq 0)?)$. Therefore, at line 7 the function $\mathsf{refine}_{\mathsf{Int}}(\varnothing, I, (x \geq 0)?)$ is called. By Theorem 4.11, this refinement adds the new concrete element:

$$(\mathsf{Int}(I \cap (x \geq 0)) \cap (x \geq 0)) \cup (\mathsf{Int}(I \cap (x < 0)) \cap (x < 0)) = [1, +\infty] \cup [-\infty, -1] = \mathbb{Z}_{\neq 0},$$

and consequently, by meet closure, all the intervals with a hole in 0. In the next iteration, $\mathsf{find}_{\mathsf{Int}}(\{\mathbb{Z}_{\neq 0}\}, I, \mathsf{r}_{\mathrm{Abs}})$ is called and this returns $Q = \{x \in \mathbb{Z} \mid x > 0, x \text{ is odd}\}$. By Theorem 7.1, we know that $[\![\mathsf{r}]\!]P \leq \mathsf{Int}_{\{\mathbb{Z}_{\neq 0}\}}(Q) = [1, +\infty]$ holds, so that we infer that $x = 0$ *was a false-alarm.* □

Whenever the abstract domain is refined, at the next iteration the procedure find_A performs a new full analysis in the refined domain. Backward repair will overcome this issue.

7.2 Program Verification by Backward Repair

The key idea of backward repair is to exploit as much as possible the abstract reasoning, disregarding the concrete input P and the actual elements traversed by a concrete computation. To achieve this, the target equivalence (6) is replaced by the following stronger condition (7), guaranteeing that the refinement A_N is precise enough to decide the (in)correctness of r not only for input P but also for any $P' \leq A(P)$:

$$\forall P' \leq A(P) \;\Rightarrow\; \left([\![\mathsf{r}]\!]^\sharp_{A_N} P' \leq \mathit{Spec} \;\Leftrightarrow\; [\![\mathsf{r}]\!]P' \leq \mathit{Spec}\right) \tag{7}$$

This condition (7) admits an equivalent formulation in terms of weakest liberal precondition (see Theorem 7.4).

Definition 7.3 (Valid Input). Given $\mathsf{r} \in \mathrm{Reg}$, an input set $P \in C$, and $\mathit{Spec} \in C$, we let:

$$\mathbf{V}\langle P, \mathsf{r}, \mathit{Spec}\rangle \triangleq \bigvee_C \{P' \in C \mid P' \leq P, [\![\mathsf{r}]\!]P' \leq \mathit{Spec}\}$$

denote the *greatest valid input* set. □

It turns out that $\mathbf{V}\langle P, \mathsf{r}, \mathit{Spec}\rangle = P \wedge \mathrm{wlp}([\![\mathsf{r}]\!], \mathit{Spec})$. As an example, for the basic expressions in Exp we have that:

$$\mathbf{V}\langle P, \mathbf{skip}, S\rangle \triangleq P \cap S, \qquad \mathbf{V}\langle P, \mathsf{b}?, S\rangle \triangleq P \cap (S \cup \neg\mathsf{b}),$$
$$\mathbf{V}\langle P, x := \mathsf{a}, S\rangle \triangleq \{\sigma \in P \mid \sigma[x \mapsto \{\!|\mathsf{a}|\!\}\,\sigma] \in S\}.$$

Algorithm 2: Backward repair procedure $\mathsf{bRepair}_A$.

```
1  Function bRepair_A(N, P̂, r, S)
2      if ([[r]]^♯_{A⊞N} P̂ ≤ S) then return ⟨P̂, N⟩;
3      switch r do
4          case e do                                   // basic expression
5              V := V⟨P̂, e, S⟩;  Q := S ∧ [[e]]^♯_{A⊞N} P̂;
6              return ⟨V, N ∪ {V, Q}⟩;
7          case r0; r1 do                              // sequential
8              ⟨V1, N1⟩ := bRepair_A(N, [[r0]]^♯_{A⊞N} P̂, r1, S);
9              ⟨V0, N0⟩ := bRepair_A(N, P̂, r0, V1);
10             return ⟨V0, N0 ∪ N1⟩;
11         case r0 ⊕ r1 do                             // choice
12             ⟨V0, N0⟩ := bRepair_A(N, P̂, r0, S);
13             ⟨V1, N1⟩ := bRepair_A(N, P̂, r1, S);
14             Q := S ∧ [[r]]^♯_{A⊞N} P̂;
15             return ⟨V0 ∧ V1, N0 ∪ N1 ∪ {Q}⟩;
16         case r0* do                                 // Kleene star
17             R̂ := [[r0]]^♯_{A⊞N} P̂;
18             if (R̂ ≤ P̂) then return inv_A(N, P̂, r0, S);
19             else                                    // unroll
20                 ⟨V1, N1⟩ := bRepair_A(N, P̂ ∨_{A⊞N} R̂, r0*, S);
21                 return ⟨P̂ ∧ V1, N1⟩
22 Function inv_A(N, P̂, r, V1)                        // loop invariants
23     do
24         V0 := P̂ ∧ V1;  N0 := N ∪ {V0};
25         ⟨V1, N1⟩ := bRepair_A(N0, V0, r, V0);
26     while (V1 ≠ V0);
27     return ⟨V1, N1⟩;
```

Theorem 7.4. *Let* $\mathsf{r} \in \mathrm{Reg}$, $A \in \mathrm{Abs}(C)$, $P, \mathit{Spec} \in C$, *and* $A_N \triangleq A \boxplus N$. *Then, condition* (7) *holds if and only if*

$$[\![\mathsf{r}]\!]^\sharp_{A_N} \mathbf{V}\langle A(P), \mathsf{r}, \mathit{Spec}\rangle \;\leq\; \mathit{Spec}. \tag{8}$$

Checking this latter condition (8) requires computing the set $\mathbf{V}\langle A(P), \mathsf{r}, \mathit{Spec}\rangle$, which can be as expensive as computing $[\![\mathsf{r}]\!]P$. Thus, we provide a necessary condition for (8) that can help to prove the validity of *Spec* without necessarily computing $\mathbf{V}\langle A(P), \mathsf{r}, \mathit{Spec}\rangle$. In the following, the hat-notation $\widehat{P}$ is used for abstract elements ranging in the abstract domain A.

Lemma 7.5. *Let* $\mathsf{r} \in \mathrm{Reg}$, $A \in \mathrm{Abs}(C)$, $\widehat{P} \in A$, $\mathit{Spec} \in C$, *and let* $A_N \triangleq A \boxplus N$ *be an abstraction refinement of* A. *If* (8) *holds (for the case* $A(P) = \widehat{P}$*) then* $\mathbf{V}\langle \widehat{P}, \mathsf{r}, \mathit{Spec}\rangle$ *is expressible in* A_N.

If we presume that $[\![\mathsf{r}]\!]P \leq \mathit{Spec}$ holds, then Lemma 7.5 suggests to consider an initial abstract domain A where P is already expressible. In fact, when $\widehat{P} = P = A(P)$ and $[\![\mathsf{r}]\!]P \leq \mathit{Spec}$ holds, it turns out that $\mathbf{V}\langle \widehat{P}, \mathsf{r}, \mathit{Spec}\rangle = \mathbf{V}\langle P, \mathsf{r}, \mathit{Spec}\rangle = P$, so that the necessary condition of Lemma 7.5 is already met by A and, therefore, by any of its refinements A_N.

The backward repair strategy $\mathsf{bRepair}_A$ is defined by the pseudocode in Algorithm 2. It exploits the auxiliary function inv_A to deal with loop invariants of r^*. This function inv_A has in input a refinement $A \boxplus N \sqsubseteq A$, an abstract invariant $\widehat{P} \in A$ for r^*, a command r and a concrete specification *Spec* and finds the greatest concrete element $V \leq \widehat{P}$ such that r^* will not yield alarms when executed on V. In fact, similarly to $\mathsf{bRepair}_A$, inv_A returns a pair $\langle V, N'\rangle$ that comprises the greatest valid input $V = \mathbf{V}\langle \widehat{P}, \mathsf{r}^*, \mathit{Spec}\rangle$ and some necessary points N' such that $N' \supseteq N$ and $[\![\mathsf{r}^*]\!]^\sharp_{A\boxplus N'} V \leq V \leq \mathit{Spec}$.

Theorem 7.6 ($\mathsf{bRepair}_A$ **and** inv_A **are Sound).** *For any* $A \in \mathrm{Abs}(C)$, $\mathsf{r} \in \mathrm{Reg}$, $\widehat{P} \in A$, $S, V \in C$, *and* $N, N' \subseteq C$:

(1) *If* $\mathsf{bRepair}_A(N, \widehat{P}, \mathsf{r}, S) = \langle V, N'\rangle$ *then:*
   (a) $V \in A \boxplus N'$; (b) $[\![\mathsf{r}]\!]^\sharp_{A\boxplus N'} V \leq S$; (c) $V = \mathbf{V}\langle \widehat{P}, \mathsf{r}, S\rangle$.
(2) *If* $[\![\mathsf{r}]\!]^\sharp_{A\boxplus N} \widehat{P} \leq \widehat{P}$ *and* $\mathsf{inv}_A(N, \widehat{P}, \mathsf{r}, S) = \langle V, N'\rangle$ *then:*
   (a) $V \in A \boxplus N'$; (b) $[\![\mathsf{r}]\!]^\sharp_{A\boxplus N'} V \leq V$; (c) $V = \mathbf{V}\langle \widehat{P}, \mathsf{r}^*, S\rangle$.

Corollary 7.7 (Program (In)Correctness). *Let* $A \in \mathrm{Abs}(C)$, $\mathsf{r} \in \mathrm{Reg}$, $\widehat{P} \in A$, *and* $\mathit{Spec} \in C$. *For any* $V \in C$, $N' \subseteq C$ *such that* $\mathsf{bRepair}_A(\varnothing, \widehat{P}, \mathsf{r}, \mathit{Spec}) = \langle V, N'\rangle$, *we have that:*

$$\forall P' \leq \widehat{P}.\;\; [\![r]\!]P' \leq \mathit{Spec} \;\Leftrightarrow\; [\![\mathsf{r}]\!]^\sharp_{A\boxplus N'} P' \leq \mathit{Spec} \;\Leftrightarrow\; P' \leq V.$$

As a special case of Corollary 7.7, taking $\widehat{P} = A(P)$ it turns out that $[\![\mathsf{r}]\!]P \leq \mathit{Spec}$ if and only if $P \leq V$.

The following examples, whose programs are taken from well-known literature, show how backward repair actually works. All of them have been *automatically* verified with a proof-of-concept Haskell implementation that exploits *finite* concrete domains and *explicit* enumerative representations of new abstract elements. The symbolic representations of the new elements added by pointed shells have been obtained by inspecting the output of this tool. Let us remark that the main goal of this work is to set the general foundations of AIR, independently of any specific class of abstract domains and of the symbolic representation of their elements. The application of AIR to symbolic frameworks for representing abstract elements — notably logical formulas whose SMT problem is decidable — is left as a stimulating future work.

Example 7.8 (Intervals). Let us consider the analysis of the following Imp program, adapted from [38, 57]:

$$\mathsf{c} \triangleq \textbf{while } (x > 0) \textbf{ do } \{x := x - 1;\; y := y - 1\}$$

with input $\widehat{P} \triangleq 0 < x \leq 100$ and $\mathit{Spec} \triangleq y = 0$. The analysis of c in the domain Int of intervals returns $x = 0$, and the same happens for octagons Oct, so, in both cases, obviously, we cannot infer the validity (and certainly not the invalidity) of *Spec*. We therefore apply backward repair to determine the valid inputs for *Spec*. The Imp program c is encoded by the following regular command:

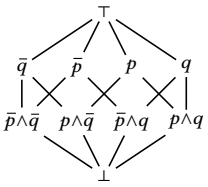
$$\mathsf{r} \triangleq \underbrace{\big((x > 0)?;\; x := x - 1;\; y := y - 1\big)}_{\mathsf{r}_1}{}^*;\; (x \leq 0)?$$

The call to $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \widehat{P}, \mathsf{r}, \mathit{Spec})$ selects the case for sequential composition and, letting $\widehat{R} \triangleq [\![\mathsf{r}_1^*]\!]^\sharp_{\mathsf{Int}} \widehat{P} = x \in [0, 100]$, invokes $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \widehat{R}, (x \leq 0)?, \mathit{Spec})$, which returns the pair $\langle Q, \{Q\}\rangle$ with $Q \triangleq x \in [0, 100] \wedge (x = 0 \Rightarrow y = 0)$. In turn, $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \widehat{P}, \mathsf{r}_1^*, Q)$ is then called. After one recursive call to unroll the loop, we call $\mathsf{inv}_{\mathsf{Int}}(\varnothing, \widehat{R}, \mathsf{r}_1, Q)$. Iteratively, the largest invariant under $\widehat{R} \wedge Q = Q$ is found,

that is $P \triangleq x \in [0,100] \wedge y = x$. Note that the addition of P to Int is not enough, because intermediate approximations for the evaluation of basic commands in r_1 are also needed by compositionality. Hence, the new elements $R_1 \triangleq x \in [1,100] \wedge y = x$, $R_2 \triangleq x \in [0,99] \wedge y = x+1$, and $R_3 \triangleq x \in [0,99] \wedge y = x$ are also added. In the end, the call to $\mathsf{inv}_{\mathsf{Int}}$ returns $\langle P, \{P, R_1, R_2, R_3\}\rangle$, so that the original call to $\mathsf{bRepair}_{\mathsf{Int}}$ returns $\langle R_1, \{P, R_1, R_2, R_3, Q\}\rangle$ because $R_1 = P \wedge \widehat{P}$. By Corollary 7.7, for any $P' \leq \widehat{P}$, we have that $[\![\mathsf{r}]\!]P' \leq \mathit{Spec}$ iff $P' \leq R_1$. Remarkably, backward repair is able to add the minimal relational information in a nonrelational domain as needed to prove *Spec*. Since all the new elements are octagons, incidentally, the analysis on Oct with input R_1 (instead of $\widehat{P}$) is also able to prove that *Spec* holds. □

Example 7.9 (Predicate Abstraction). Consider the following program c from Fig. 1 of [4, 5] and the Cartesian predicate abstraction domain A induced by $p \triangleq (z = 0)$ and $q \triangleq (x = y)$, as depicted below:

```
c ≜ do { z := 0; x := y;
         if (w ≠ 0) then {
             x := x + 1; z := 1
         }
       } while (x ≠ y)
```



We want to prove that $[\![\mathrm{c}]\!]\top \leq p$, whereas it turns out that $[\![\mathrm{c}]\!]^\sharp_A \top = q$. The domain refinement used in [4, 5] is the reduced disjunctive completion of A, which is isomorphic to the Boolean abstraction $B \triangleq \langle \wp(\{p \wedge q, p \wedge \bar{q}, \bar{p} \wedge q, \bar{p} \wedge \bar{q}\}), \subseteq\rangle$. The analysis with B leads exactly to the same analysis with $A_1 \triangleq A \boxplus \{p \leftrightarrow q\}$, namely $[\![\mathrm{c}]\!]^\sharp_{A_1} \top = p \wedge q$.
By invoking $\mathsf{bRepair}_A(\varnothing, \top, \mathrm{c}, p)$ we get as a result the pair $\langle \top, \{q \longrightarrow p\}\rangle$. In fact, letting $A_2 \triangleq A \boxplus \{q \longrightarrow p\}$ we get $[\![\mathrm{c}]\!]^\sharp_{A_2} \top = p \wedge q = A_2(p \wedge q) = A_2([\![\mathrm{c}]\!]\top)$. Let us remark that the point $q \longrightarrow p$ is indeed more abstract than $p \leftrightarrow q$. □

Widening. It is well known that the convergence of fixpoint computations in non-ACC domains can be forced or accelerated by means of *widening operators* [16, 17]. A widening over-approximates an abstract join $x \vee_A y$ by a more abstract element $x \nabla_A y$. We show that backward repair is compatible with widening operators, and that, in this case, the repaired abstract interpreter is guaranteed to terminate.

Definition 7.10 (Widening Operator). Given $A \in \mathrm{Abs}(C)$, a *widening operator* $\nabla_A : A \times A \longrightarrow A$ is a function such that: (i) for all $x, y \in A$, $x, y \leq_A x \nabla_A y$; and (ii) for every sequence $\{x_i\}_{i\in\mathbb{N}} \subseteq A$, the chain $\{y_i\}_{i\in\mathbb{N}}$ inductively defined by $y_0 \triangleq x_0$ and $y_{i+1} \triangleq y_i \nabla_A x_{i+1}$ finitely converges (i.e., $\exists k \in \mathbb{N}.\ \forall j \in \mathbb{N}.\ y_{j+k} = y_k$). □

The abstract semantics with widening of the Kleene star is updated to (see, e.g., [45, Section 3.5]):

$$[\![\mathsf{r}^*]\!]^\sharp_A \widehat{S} \triangleq \mathrm{lfp}\left(\lambda \widehat{X} \in A.\ \widehat{X} \nabla_A (\widehat{S} \vee_A [\![\mathsf{r}]\!]^\sharp_A \widehat{X})\right).$$

$\langle \top, \{P, R_1, R_2, R_3, V\}\rangle \leftarrow \mathsf{bRepair}(\varnothing, \top, \mathsf{r}, \mathit{Spec})$
  $\langle V, \{V\}\rangle \leftarrow \mathsf{bRepair}(\cdots (i > 5)?, \mathit{Spec})$
  $\langle \top, \{P, R_1, R_2, R_3\}\rangle \leftarrow \mathsf{bRepair}(\cdots, \mathsf{r}_2, V)$
    $\langle \widehat{P}_1, \{P, R_1, R_2, R_3\}\rangle \leftarrow \mathsf{bRepair}(\cdots, \mathsf{r}_1^*, V)$
      ...
        $\langle P, \{P, R_1, R_2, R_3\}\rangle \leftarrow \mathsf{bRepair}(\cdots, \mathsf{r}_1^*, V)$
          $\langle P, \{P, R_1, R_2, R_3\}\rangle \leftarrow \mathsf{inv}(\varnothing, \widehat{P}, \mathsf{r}_1, V)$
            $\langle P, \{P, R_1, R_2, R_3\}\rangle \leftarrow \mathsf{bRepair}(\{V\}, V, \mathsf{r}_1, V)$
            ...
            $\langle P, \{P, R_1, R_2, R_3\}\rangle \leftarrow \mathsf{bRepair}(\{P\}, P, \mathsf{r}_1, P)$
              $\langle R_3, \{P, R_3\}\rangle \leftarrow \mathsf{bRepair}(\cdots, i := i+1, P)$
              $\langle R_1, \{P, R_1, R_2\}\rangle \leftarrow \mathsf{bRepair}(\cdots, (i \leq 5)?; j := j+i, R_3)$
                $\langle R_2, \{P, R_2\}\rangle \leftarrow \mathsf{bRepair}(\cdots, j := j+i, R_3)$
                $\langle R_1, \{P, R_1\}\rangle \leftarrow \mathsf{bRepair}(\cdots, (i \leq 5)?, R_2)$
  $\langle \top, \emptyset\rangle \leftarrow \mathsf{bRepair}(\cdots, \mathsf{r}_3, \widehat{P}_1)$

Figure 4. Backward repair call scheme for Example 7.13.

By termination condition (ii) of ∇_A, the iteration sequence for the least fixpoint of $\lambda \widehat{X}.\ \widehat{X} \nabla_A (\widehat{S} \vee_A [\![\mathsf{r}]\!]^\sharp_A \widehat{X})$ always finitely converges. We settle the problem of how a suitable widening operator for $A \boxplus N$ can be derived from a widening for A.

Definition 7.11 (Pointed Widening). Given a refinement $A_N \triangleq A \boxplus N$ of $A \in \mathrm{Abs}(C)$ and a widening ∇_A on A, the *pointed widening* $\nabla^N_A : A_N \times A_N \longrightarrow A_N$ is defined as follows: $x \nabla^N_A y \triangleq \wedge_{A_N}\{z \in (N \cup \{A(x) \nabla_A A(y)\}) \mid x, y \leq_{A_N} z\}$. □

Theorem 7.12 (Soundness of Pointed Widening). *If $N \subseteq C$ is finite, then ∇^N_A is a widening operator for $A \boxplus N$.*

In Algorithm 2, line 20, we simply replace $\widehat{P} \vee_{A\boxplus N} \widehat{R}$ with $\widehat{P} \nabla^N_A (\widehat{P} \vee_{A\boxplus N} \widehat{R})$ in the clause dealing with the unroll of Kleene star. This change has no consequence in the proof of Theorem 7.6, so the main results are seamlessly extended to widening with the additional guarantee that whenever $\mathsf{bRepair}_A(N, \widehat{P}, \mathsf{r}, S)$ returns $\langle V, N'\rangle$, then termination of the abstract interpreter for $A_{N'}$ is guaranteed by Theorem 7.12.

Example 7.13 (Widening). Let us revisit the illustrative example in Section 2, where $\mathit{Spec} \triangleq j \leq 15$ and

$$\mathsf{r} \triangleq \overbrace{\underbrace{i := 1;\ j := 0;}_{\mathsf{r}_3}\ \underbrace{((i \leq 5)?;\ j := j + i;\ i := i + 1)^*;}_{\mathsf{r}_1}\ \underbrace{(i > 5)?}_{\mathsf{b}}}^{\mathsf{r}_2}.$$

The call to $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \top, \mathsf{r}, \mathit{Spec})$ recursively computes $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \widehat{P}, \mathsf{b}?, \mathit{Spec})$, where $\widehat{P} = [\![\mathsf{r}_2]\!]^\sharp_{\mathsf{Int}} \top = i \in [1,6] \wedge j \in [0,\infty]$ (using the widening on intervals for j), which returns $\langle V, \{V\}\rangle$: the element $V \triangleq (i \in [1,5] \wedge j \in [0,\infty]) \vee (i = 6 \wedge j \in [0,15])$ is introduced to repair b?. This leads to call $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \top, \mathsf{r}_2, V)$, and, then, $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \widehat{P}_1, \mathsf{r}_1^*, V)$, with $\widehat{P}_1 \triangleq [\![\mathsf{r}_3]\!]^\sharp_{\mathsf{Int}} \top = (i = 1 \wedge j = 0)$. After unrolling r_1^* to reach the abstract invariant $\widehat{P}$, the call to $\mathsf{inv}_{\mathsf{Int}}(\varnothing, \widehat{P}, \mathsf{r}_1, V)$ computes the largest invariant under $\widehat{P} \wedge V$ for r_1^*, which is $P \triangleq i \in [1,6] \wedge j \in [0, i(i-1)/2]$. Then, $\mathsf{bRepair}_{\mathsf{Int}}(\{P\}, P, \mathsf{r}_1, P)$ is invoked, which adds the new abstract elements R_1, R_2 and R_3 (cf. Section 2) to repair the local completeness of the instructions in r_1. Thus, $\mathsf{inv}_{\mathsf{Int}}(\varnothing, \widehat{P}, \mathsf{r}_1, V)$ returns $\langle P, \{P, R_1, R_2, R_3\}\rangle$. The point P is then intersected with the approximations

computed by the unrolling of r_1^* until $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \widehat{P_1}, \mathsf{r}_1^*, V)$ outputs $\langle \widehat{P_1}, \{P, R_1, R_2, R_3\}\rangle$, so that $\mathsf{bRepair}_{\mathsf{Int}}(\varnothing, \top, \mathsf{r}_3, \widehat{P_1})$ returns $\langle \top, \emptyset\rangle$, because $[\![\mathsf{r}_3]\!]^\sharp_{\mathsf{Int}} \top = \widehat{P_1}$, and the original call to bRepair outputs $\langle \top, \{P, R_1, R_2, R_3, V\}\rangle$. By Corollary 7.7, for all $P' \leq \top$, $[\![\mathsf{r}]\!]P' \leq \mathit{Spec}$ holds. The series of recursive calls is sketched in Fig. 4, omitting most details for readability. □

8 Implementation Challenges

When shaping the application examples of AIR, we developed a pilot Haskell implementation to gain confidence in our findings, although it was not conceived to scale up to practical domains and real programs. This tool works on finite integer domains, relies on an *explicit* enumeration of the abstract elements and supports backward repair.

A prospective implementation of AIR must necessarily be *domain-dependent*, namely, it has to rely upon the way properties of states, i.e., sets in $\mathbb{S}$, are represented. This is a common feature of any abstraction refinement strategy. Pointed shells, as well as completeness shells and other domain refinements, require the enhancement of an input abstract domain with more concrete objects, in our case the ones repairing the abstraction, that ineluctably belong to a more concrete domain of program properties, e.g., the concrete domain itself $\mathbb{S}$. For instance, Example 7.8 shows that sometimes a nonrelational abstract domain as Int must be enriched with some elements of a more concrete relational domain like Oct in order to verify certain properties.

Of course, *being concrete is a relative notion*. Therefore, a natural way for designing an implementation of AIR is to build it on top of a "tractable" domain C abstracting $\mathbb{S}$, i.e., such that $\mathbb{S} \sqsubseteq C$. This domain C can be itself a known abstract domain which is deemed too concrete for the purposes of our analysis. In this case, AIR for an abstract domain A such that $\mathbb{S} \sqsubseteq C \sqsubseteq A$ will isolate within C, and not within $\mathbb{S}$, a locally complete refinement of A which works for our program and input. In all cases, it is necessary to figure out a specification language and symbolic representations for the concrete properties of interest, i.e., for the elements of C. In CEGAR, for instance, this comes as a consequence of the fact that in abstract model checking (e.g., in predicate abstraction) the properties of the state space in C are partitions of the same space. These can be represented by predicates in a suitable restriction of classical first order logic. Both forward and backward AIR instead use properties of states that do not necessarily correspond to partitions, such as intervals, octagons, and most domains used in abstract interpretation.

We therefore envision that the main implementation challenge of AIR is defining a specification language $\mathcal{L}$ for the new concrete properties we want to express in such a way that C is built as the set of models of formulae in $\mathcal{L}$. For example, for domains of SMT logical formulae, Cousot et al. [20] show how to unify program analysis approaches based on abstract interpretation and on SMT solvers/theorem provers, so that they can be combined and adapted to a specific application domain. In particular, the logical abstract domains defined in [20, Section 7] as well as the quantified abstract domains studied by Gulwani et al. [34, Section 2] can play the role of specification language $\mathcal{L}$, where in both cases the domain elements are formulae of a logical theory that will be computed and added by the locally complete refinements of AIR. As a further example, abstract domains for polynomial inequalities, such as those needed in Section 2, have been studied in [55, 56] relying on algorithms for Gröbner bases.

9 Conclusion

AIR means finding the most abstract domain, called pointed shell, that removes all the false-alarms along an incomplete abstract computation. Because this operation depends upon a specific execution trace, AIR features an inherent dynamic flavor: *we are not interested in refining an abstract domain for all possible programs and all possible inputs*. AIR generalizes CEGAR to arbitrary abstract domains and transfer functions, as those typically used in abstract interpretation.

Likewise CEGAR for predicate abstractions of infinite state systems, termination of forward/backward-repair cannot be guaranteed in general. Finding sufficient conditions to guarantee termination of AIR is an open research avenue. As future work, we plan to incorporate AIR in the LCL_A logic defined in [8]: whenever a local completeness proof obligation emerges, we can repair the abstract interpreter to settle such an obligation. We also plan to exploit different dynamically selected abstract domains within the same proof and to combine forward and backward repairs. We know that backward repair, when it terminates, will always offer the most abstract domain for our analysis. This leaves no choice for the abstract elements that need to be added in the refinement. On the contrary, forward repair may deal with non-optimal refinements. Hence, whenever certain pointed shells are hard to represent in the specification language $\mathcal{L}$ envisioned in Section 8, it may still be the case that the forward repair can select a less abstract but admissible representation in $\mathcal{L}$, yet satisfying local completeness and concluding the proof of (in)correctness. For example, the use of generalized widening/narrowing, in the style of [15], is worth investigating. We envisage that our research on AIR may result in the design of a very first automatic self-adaptable program verification tool based on abstract interpretation.

Acknowledgments

We thank Andrea Laretto for the tool development. This research has been funded by the *Italian MIUR*, under the PRIN2017 project no. 201784YSZ5 "AnalysiS of PRogram Analyses (ASPRA)", and by a *Meta Research* gift. The work of Francesco Ranzato and Roberto Giacobazzi has been partially funded by *Facebook Research*, under a "Probability and Programming Research Award".

References

[1] Aws Albarghouthi and Kenneth L. McMillan. 2013. Beautiful Interpolants. In *Proceedings of CAV 2013, 25th International Conference on Computer Aided Verification (Lecture Notes in Computer Science, Vol. 8044)*. Springer, 313–329. https://doi.org/10.1007/978-3-642-39799-8_22

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. The MIT Press.

[3] Thomas Ball, Todd D. Millstein, and Sriram K. Rajamani. 2005. Polymorphic predicate abstraction. *ACM Trans. Program. Lang. Syst.* 27, 2 (2005), 314–343. https://doi.org/10.1145/1057387.1057391

[4] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. 2001. Boolean and Cartesian Abstraction for Model Checking C Programs. In *Proceedings of TACAS 2001, 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 2031)*. Springer, 268–283. https://doi.org/10.1007/3-540-45319-9_19

[5] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. 2003. Boolean and Cartesian abstraction for model checking C programs. *Int. J. Softw. Tools Technol. Transf.* 5, 1 (2003), 49–58. https://doi.org/10.1007/s10009-002-0095-0

[6] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of PLDI 2003, ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 196–207. https://doi.org/10.1145/781131.781153

[7] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. 2020. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.* 4, POPL (2020), 28:1–28:28. https://doi.org/10.1145/3371096

[8] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *Proceedings of LICS 2021, 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 1–13. https://doi.org/10.1109/LICS52264.2021.9470608 Distinguished paper.

[9] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Proceedings of CAV 2000, 12th International Conference on Computer Aided Verification (Lecture Notes in Computer Science, Vol. 1855)*. Springer-Verlag, 154–169. https://doi.org/10.1007/10722167_15

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2001. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead (Lecture Notes in Computer Science, Vol. 2000)*. Springer, 176–194. https://doi.org/10.1007/3-540-44577-3_12

[11] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643

[12] Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1512–1542. https://doi.org/10.1145/186025.186051

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. The MIT Press.

[14] Patrick Cousot. 2007. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of EMSOFT 2007, 7th ACM & IEEE International conference on Embedded software*. ACM, 7–9. https://doi.org/10.1145/1289927.1289932

[15] Patrick Cousot. 2015. Abstracting Induction by Extrapolation and Interpolation. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 8931)*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer, 19–42. https://doi.org/10.1007/978-3-662-46081-8_2

[16] Patrick Cousot. 2021. *Principles of Abstract Interpretation*. MIT Press.

[17] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM POPL'77*. ACM, 238–252. https://doi.org/10.1145/512950.512973

[18] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of ACM POPL'79*. ACM, 269–282. https://doi.org/10.1145/567752.567778

[19] Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTRÉE Analyzer. In *Proceedings of ESOP 2005, 14th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 3444)*. Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

[20] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2012. Theories, solvers and static analysis by abstract interpretation. *J. ACM* 59, 6 (2012), 31:1–31:56. https://doi.org/10.1145/2395116.2395120

[21] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL'78, the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[22] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. 1979. Social Processes and Proofs of Theorems and Programs. *Commun. ACM* 22, 5 (1979), 271–280. https://doi.org/10.1145/359104.359106

[23] Edsger W. Dijkstra. 1972. The Humble Programmer. *Commun. ACM* 15, 10 (1972), 859–866. https://doi.org/10.1145/355604.361591

[24] Edsger W. Dijkstra. 1976. *A discipline of programming*. Prentice-Hall.

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

[26] Gilberto Filé, Roberto Giacobazzi, and Francesco Ranzato. 1996. A unifying view of abstract domain design. *ACM Comput. Surv.* 28, 2 (1996), 333–336. https://doi.org/10.1145/234528.234742

[27] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Trans. Software Eng.* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[28] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of POPL 2015, 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 261–273. https://doi.org/10.1145/2676726.2676987

[29] Roberto Giacobazzi and Elisa Quintarelli. 2001. Incompleteness, counterexamples and refinements in abstract model-checking. In *Proceedings of SAS 2001, 8th International Static Analysis Symposium (Lecture Notes in Computer Science, Vol. 2126)*. Springer, 356–373. https://doi.org/10.1007/3-540-47764-0_20

[30] Roberto Giacobazzi and Francesco Ranzato. 2002. States vs. Traces in Model Checking by Abstract Interpretation. In *Proceedings of the 9th International Static Analysis Symposium, SAS 2002 (Lecture Notes in Computer Science, Vol. 2477)*. Springer, 461–476. https://doi.org/10.1007/3-540-45789-5_32

[31] Roberto Giacobazzi and Francesco Ranzato. 2006. Incompleteness of states w.r.t. traces in model checking. *Inf. Comput.* 204, 3 (2006), 376–407. https://doi.org/10.1016/j.ic.2006.01.001

[32] Roberto Giacobazzi and Franceso Ranzato. 2022. History of Abstract Interpretation. *IEEE Annals of the History of Computing* 44 (2022), 13 pages. https://doi.org/10.1109/MAHC.2021.3133136

[33] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. 2000. Making Abstract Interpretation Complete. *Journal of the ACM* 47, 2 (March 2000), 361–416. https://doi.org/10.1145/333979.333989

[34] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. 2008. Lifting Abstract Interpreters to Quantified Logical Domains. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing

Machinery, 235–246. https://doi.org/10.1145/1328438.1328468
[35] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. 2004. Abstractions from proofs. In *Proceedings of POPL 2004, 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 232–244. https://doi.org/10.1145/964001.964021
[36] Krystof Hoder, Laura Kovács, and Andrei Voronkov. 2012. Playing in the grey area of proofs. In *Proceedings of POPL 2012, 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 259–272. https://doi.org/10.1145/2103656.2103689
[37] Ranjit Jhala and Rupak Majumdar. 2009. Software Model Checking. *ACM Comput. Surv.* 41, 4, Article 21 (Oct. 2009), 54 pages. https://doi.org/10.1145/1592434.1592438
[38] Ranjit Jhala and Kenneth L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *Proceedings of TACAS 2006, 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 3920)*. Springer, 459–473. https://doi.org/10.1007/11691372_33
[39] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of POPL 2015, 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 247–259. https://doi.org/10.1145/2676726.2676966
[40] Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 427–443. https://doi.org/10.1145/256167.256195
[41] Dexter Kozen. 2000. On Hoare Logic and Kleene Algebra with Tests. *ACM Trans. Comput. Logic* 1, 1 (July 2000), 60–76. https://doi.org/10.1145/343369.343378
[42] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of POPL 2006, 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042
[43] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. 1995. Property preserving abstractions for the verification of concurrent systems. *Formal Methods Syst. Des.* 6 (1995), 11–44. https://doi.org/10.1007/BF01384313
[44] Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1
[45] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372. https://doi.org/10.1561/2500000034
[46] David Monniaux and Julien Le Guen. 2012. Stratified Static Analysis Based on Variable Dependencies. *Electron. Notes Theor. Comput. Sci.* 288 (2012), 61–74. https://doi.org/10.1016/j.entcs.2012.10.008
[47] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906
[48] Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. https://doi.org/10.1145/3371078
[49] Benjamin Pierce. 2002. *Types and Programming Languages*. MIT Press.
[50] Benjamin Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.
[51] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Proc. CAV 2020 (LNCS, Vol. 12225)*. Springer, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14
[52] Francesco Ranzato and Francesco Tapparo. 2004. Strong Preservation as Completeness in Abstract Interpretation. In *Proceedings of ESOP 2004, 13th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 2986)*. Springer, 18–32. https://doi.org/10.1007/978-3-540-24725-8_3
[53] Francesco Ranzato and Francesco Tapparo. 2007. Generalized Strong Preservation by Abstract Interpretation. *J. Log. Comput.* 17, 1 (2007), 157–197. https://doi.org/10.1093/logcom/exl035
[54] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis – An Abstract Interpretation Perspective*. MIT Press.
[55] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.* 64, 1 (2007), 54–75. https://doi.org/10.1016/j.scico.2006.03.003
[56] Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Generating all polynomial invariants in simple loops. *J. Symb. Comput.* 42, 4 (2007), 443–476. https://doi.org/10.1016/j.jsc.2007.01.002
[57] Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants as Classifiers. In *Proceedings of CAV 2012, 24th International Conference on Computer Aided Verification (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 71–87. https://doi.org/10.1007/978-3-642-31424-7_11
[58] Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2014. Bias-variance tradeoffs in program analysis. In *Proceedings of POPL 2014, 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 127–138. https://doi.org/10.1145/2535838.2535853
[59] Tachio Terauchi and Hiroshi Unno. 2015. Relaxed Stratification: A New Approach to Practical Complete Predicate Refinement. In *Proceedings of ESOP 2015, 24th European Symposium on Programming (Lecture Notes in Computer Science, Vol. 9032)*. Springer, 610–633. https://doi.org/10.1007/978-3-662-46669-8_25
[60] Hiroshi Unno and Tachio Terauchi. 2015. Inferring Simple Solutions to Recursion-Free Horn Clauses via Sampling. In *Proceedings of TACAS 2015, 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 9035)*. Springer, 149–163. https://doi.org/10.1007/978-3-662-46681-0_10
[61] Moshe Y. Vardi. 2021. Program Verification: Vision and Reality. *Commun. ACM* 64, 7 (2021), 5. https://doi.org/10.1145/3469113
[62] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: an Introduction*. MIT press.