Abstract Interpretation Repair

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

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

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

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

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:

\[ \text{AbsVal}(x) \equiv \text{if } (x \geq 0) \text{ then skip else } x := -x \]

The well-known interval abstraction \( \text{Int} \) approximates any property \( S \in \wp(\mathbb{Z}) \) of the integer values that \( x \) may assume by the least interval \( \text{Int}(S) = [a, b] \) 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 \( \text{AbsVal}(I) \) is \( \text{Int}(\text{AbsVal}(I)) = [1, +\infty] \) showing that 0 is not a possible result, it turns out that the \emph{best correct approximation} the abstract interpreter can compute in Int is not precise, because it also includes 0: \( \text{Int}(\text{AbsVal}\{\text{Int}(I)\}) = [0, +\infty] \). Technically, this means that Int is incomplete for \( \text{AbsVal} \) on input \( I \). This can spawn a problem in verification: for instance, if the result of \( \text{AbsVal}(I) \) 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 \emph{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.

\textbf{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 \( \tau_A [I] c [Q] \) derivable in the proof system is such that \( Q \) is an under-approximation of the strongest postcondition \( \text{post}_c[I] \) 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 \( \text{post}_c[I] \). A novel inference rule drives logical derivations for \( \tau_A \) by introducing all and only those

local completeness proof obligations needed to infer a triple \( \tau_A [I] c [Q] \). These are \emph{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 \( \text{Spec} \) expressible in \( A \) is valid for \( c \) if and only if \( Q \leq \text{Spec} \). Since logical derivations cannot work with locally incomplete abstractions, we advocate that these abstract domains need to be repaired to achieve local completeness.

\textbf{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 \emph{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 \emph{general and absolute sense}, i.e., they do not limit a priori the way in which new abstract elements are represented, e.g., by formulæ of some theory.

In our simple example for \( \text{AbsVal} \), 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 \( \mathbb{Z}_{\neq 0} = [−\infty, −1] \cup [1, +\infty] \). The verification of \( \text{AbsVal} \) in this repaired domain will not raise alarms.

Our Abstract Interpretation Repair (AIR) shares similarities with the principles of the well-known \textit{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 \textit{forward} and \textit{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 \( \text{AbsVal} \) 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 \( Z_{\neq 0} \), 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:

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

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 \( \text{Spec} \triangleq j \leq 15 \). The analysis of \( c \) on Int, using the standard widening for intervals, infers the abstract loop invariant \( \overline{P} \triangleq i \in [1, 6] \land j \in [0, \infty) \), and outputs the abstract store \( \overline{Q} \triangleq i \in [6, 6] \land j \in [0, \infty] \). 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: \( 1 \leq i \leq 6 \land i-j \leq 1 \land i+j \geq 1 \land j \geq 0 \). 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 \( T_{i-1} = \frac{i(i - 1)}{2} \), we observe that the concrete (i.e., strongest) loop invariant of \( c \) is:

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

AIR allows us to refine the aforementioned abstract domains \( \text{Int} \) and \( \text{Oct} \) 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):

\[
r \triangleq i := 1; \quad j := 0; \quad (\{ i \leq 5? \} \land \{ j = i \land i := i + 1 \} \ast \{ i > 5 \}).
\]

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

\[
\begin{align*}
P & \triangleq i \in [1, 6] \land j \in [0, T_{i-1}] \\
R_1 & \triangleq i \in [1, 5] \land j \in [0, T_{i-1}] \\
R_2 & \triangleq i \in [1, 5] \land j \in [1, T_i] \\
R_3 & \triangleq i \in [2, 6] \land j \in [1, T_{i-1}] \\
V & \triangleq (i \in [1, 5] \land j \in [0, \infty)] \lor (i = 6 \land j \in [0, T_5]).
\end{align*}
\]

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 \( \text{Int} \) returns \( Q = i \in [6, 6] \land j \in [0, \infty] \). Since we aim to prove \( j \leq 15 \), AIR first computes the point \( Q \triangleq \hat{Q} \land \text{Spec} = i \in [6, 6] \land j \in [0, 15] \), which is already in \( \text{Int} \). Since \( Q \) was obtained by the abstract evaluation of the exit condition (\( i > 5 \)) on the abstract loop invariant \( \hat{V} \triangleq i \in [1, 6] \land j \in [0, \infty] \) for \( r_1 \), this repair step introduces the largest precondition \( V \) below (i.e., entailing \( \hat{V} \)) such that the guard (\( i > 5 \))? filters \( V \) into a result below \( \hat{Q} \). 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 \( R_0 \) are introduced because the repair procedure is inductively defined and they are needed to guarantee local completeness. For example, \( R_3 \) is the point necessary for making the guard (\( i \leq 5 \))? locally complete on \( P \). Similarly, \( R_3 \) makes the assignment \( j := j + 1 \) locally complete on \( R_1 \), and, in turn, \( R_3 \) is necessary for making \( i := i + 1 \) locally complete on \( R_3 \) (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 \( \text{Inv} \) in place of \( P \) and restricted the range of \( j \) in \( R_0 \), \( V \) in a similar way.

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

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

\[
\begin{align*}
P[K] & \triangleq n = K \land i \in [1, K + 1] \land j \in [0, D_K + T_{i-1}] \\
R_1[K] & \triangleq n = K \land i \in [1, K] \land j \in [0, D_K + T_{i-1}] \\
R_2[K] & \triangleq n = K \land i \in [1, K] \land j \in [1, D_K + T] \\
R_3[K] & \triangleq n = K \land i \in [2, K + 1] \land j \in [1, D_K + T_{i-1}] \\
V[K] & \triangleq n = K \land ((i \in [1, K] \land j \in [0, \infty]) \lor S)
\end{align*}
\]

where \( S \triangleq i = K + 1 \land j \in [0, T] \). 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 \( P, R_0, V \).

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

\[
\begin{align*}
P[K_1, K_2] & \triangleq n \in [K_1, K_2] \land (P[n] \lor S[n]) \\
V[K_1, K_2] & \triangleq n \in [K_1, K_2] \land (V[n] \lor S[n])
\end{align*}
\]

where \( S[n] \triangleq i \in [n+1, K_2+1] \land j \in [0, T] \). 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 \( R_0[K_1, K_2] \) is analogous. When \( K_1 = K_2 \) they all boil down to \( P[K_2], R_0[K_2], V[K_2], V[K_2], \) respectively.

To conclude, we remark that: (i) AIR provides the most abstract refinement (cf. Section 4) of \( \text{Int} \) for proving that Spec holds; (ii) even if the analysis starts in the nonrelational domain \( \text{Int} \), 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 \( \langle C, \leq, \lor, \land, \top, \bot \rangle \), with partial order \( \leq \), lub \( \lor \), glb \( \land \), greatest element \( \top \) and least element \( \bot \). Given \( X \subseteq C \), its set of maximal elements is \( \text{max}(X) \triangleq \{ x \in X \mid \forall y \in X, x \leq y \Rightarrow x = y \} \). 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 \( f \lesssim g \) 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 \( \text{lfp}(f) \).

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 \( \langle \alpha, \gamma \rangle : C \to A \) 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 \( \gamma \alpha = \text{id}_A \). Let \( \text{Abs}(C) \) denote the class of abstract domains of \( C \), where the notation \( \text{Abs}_\alpha(C) \in \text{Abs}(C) \) 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 \( \alpha, \gamma \in \text{Abs}(C) \) and a function \( f : C \to C \), an abstract function \( f^A : A \to A \) is a correct (or sound) approximation of \( f \) if \( \alpha f^A \subseteq f^A \alpha \) holds. If \( f^A \) is a correct approximation of \( f \) then any split correctness \( \alpha(\text{lfp}(f)) \leq_A \text{lfp}(f^A) \) holds. The best correct approximation \( (\text{bca}) \) of \( f \) in \( A \) is defined by \( f^A = \alpha f \gamma : A \to A \).

The function \( f^A \) is a complete approximation of \( f \) if \( \alpha f^A = f^A \alpha \) holds. The abstract domain \( A \) is a complete abstraction for \( f \) if there exists a complete approximation \( f^A : A \to A \) of \( f \). Completeness of \( f^A \) 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 \( \text{bca} \) of \( f \). Indeed, it turns out that completeness \( \alpha f^A = f^A \alpha \) holds for some (sound) \( f^A \) iff \( \alpha f = \alpha f \gamma \alpha \Leftrightarrow (\gamma \alpha f = (\gamma \alpha) f \gamma(\alpha) \gamma) \), which, by a slight abuse of notation, will be denoted by \( Af = AfA \). It is known that completeness transfers to fixpoints, meaning that if \( f^A \) is complete for \( f \) then any split completeness \( \alpha(\text{lfp}(f)) = \text{lfp}(f^A) \) 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 \( \text{uco}(C) \) denote the set of ucos on \( C \). The isomorphism between GIs and uco’s is known:

1. If \( \text{uco}_\alpha(C) \in \text{ucc}(C) \) is a GL then \( \gamma \alpha \in \text{uco}(C) \), and
2. If \( \mu \in \text{uco}(C) \) then \( \mu(C) = \text{uco}(\mu) \).

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 \( \text{lnt}[-2, 5] = \text{lnt}(-2, 5] \). Each closure \( \mu \in \text{uco}(C) \) is uniquely determined by the set of its fixpoints, which also coincides with its image \( \mu(C) \), because \( \mu = \text{lfp} \mu = \text{lfp} \mu(C) \). 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., \( X = \mathcal{M}(X) = \{ \gamma y \mid Y \subseteq X \} \) (note that \( \top = \text{lfp} \eta \in \mathcal{M}(X) \)). When \( C \) is a complete lattice, \( \text{uco}(C), \subseteq, \lor, \land, \text{ax}, \tau, \text{id} \) is a complete lattice as well. By the above isomorphism, \( \text{ucc}(C), \subseteq, \lor, \land, \text{ax}, \tau, \text{id} \) denotes the so-called lattice of abstract interpretations [18, Section 8], where \( A' \subseteq A \) 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 \( \cup \)) and the most abstract refinement (i.e., glb \( \cap \)) of any family of abstract domains. In particular, recall that \( A' \subseteq A \) holds when \( \gamma A(\alpha) \subseteq \gamma A'(\alpha) \) and, consequently, that a lub of a set \( X \subseteq \text{ucc}(C) \) of abstract domains coincides with their intersection, i.e., \( \bigcup \{ A \mid A \in \mathcal{X} \} = \cap \{ \gamma A(\alpha) \mid A \in \mathcal{X} \} \).

Domain Refinements. Given \( A \in \text{ucc}(C) \) and \( N \subseteq C \) we define \( A \bot N = \mathcal{M}(\gamma A(\alpha) \cup N) \) as the least refinement of \( A \) including the (possibly) new elements in \( N \). Using the shorthand \( A_N \) for \( A \bot N \), it holds that for all \( c \in C, A_N(c) = \{ x \in (N \cup \{ A(c) \}) \mid c \leq x \} \). As a special case, we write \( A_r \) for \( A \bot \{ z \} \), with \( A_r(c) = z \land A(c) \) if \( c \leq z \) and \( A_r(c) = A(c) \) otherwise. The domain \( A_r \) 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].

\[
\text{Reg} \ni r := e \mid r; r \mid r @ r \mid r^*
\]

This language is parametric on the syntax of basic transfer expressions \( e \in \text{Exp} \), which define the basic commands. These can be instantiated, e.g., with (deterministic or nonde
terministic 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 \( t_1; t_2 \) represents sequential composition, \( t_1 @ t_2 \) a choice command that can behave as either \( t_1 \) or \( t_2 \), 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 \( \llbracket \cdot \rrbracket : \text{Exp} \to C \to C \) on a complete lattice \( C \) such that \( \llbracket e \rrbracket \) is an additive function for any \( e \in \text{Exp} \). 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 \( \llbracket \cdot \rrbracket : \text{Reg} \to \text{ucc}(C) \)
C → C of regular commands is defined as follows:
\[
\begin{align*}
[l]c & \triangleq \langle l \rangle c \\
[l_1 \circ \circ]r_1 & \triangleright l_2 \circ r_2 \circ c \triangleq [l_1]c \triangleright [l_2]c
\end{align*}
\]

Abstract Semantics. The abstract semantics of regular commands \( [L]A \): \( \text{Reg} \rightarrow A \rightarrow A \) on an abstract domain \( A \in \text{Abs}(C) \) is defined by structural induction as follows:
\[
\begin{align*}
[l]c & \triangleq \langle l \rangle c \\
[l_1 \circ \circ]r_1 & \triangleright l_2 \circ r_2 \circ c \triangleq [l_1]c \triangleright [l_2]c \\
[l_1; r_2] & \circ c \triangleq [l_1]c \triangleright [r_2]c \\
[r_1 \circ \circ]c & \circ c \triangleq \bigvee_{n \in \mathbb{N}} ([r_1]^n c) \circ n
\end{align*}
\]

The above abstract semantics is monotonic and sound, i.e., \( A \circ [l]A \subseteq [l]A \circ A \) holds. Note that the abstract semantics of a basic expression \( e \) is its bca \( [e]A \) on \( A \): this agrees with the standard definition by structural induction of abstract semantics used in abstract interpretations [16, 54]. Best correct approximations are preserved by choice commands, i.e., \( [l_1 \circ \circ]r_1 \circ l_2 \circ r_2 \circ c = [l_1]A \circ A \circ [l_2]A \circ A \circ [r_2]A \circ A \), but, in general, not by sequential composition and Kleene iteration: for example, \( [l_1]A \circ [r_2]A \circ [l_1]A \) is not guaranteed to be the bca \( [l_1; r_2]A \).

Imp. We consider standard basic transfer expressions used in deterministic while programs, as defined below, where, for simplicity, we consider just integer variables \( x \in \mathbb{V} \):
\[
\begin{align*}
A\text{Exp} & \equiv a := o \in \mathbb{Z} \mid x \in \mathbb{V} \mid a + a + a \mid a + a - a - a \\
B\text{Exp} & \equiv b := \text{tt} \mid \text{ff} \\
\text{Exp} & \equiv e := \text{skip} \mid x := a \mid b
\end{align*}
\]
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]):
\[
\begin{align*}
\text{if} (b) & \text{then } c_1 \text{ else } c_2 \triangleq (b_1) \circ c_1 \oplus (\neg b_2 \circ c_2) \\
\text{while} (b) & \text{ do } c \triangleq (b_1) \circ c \oplus (\neg b_2) \\
\text{do } c & \text{ while } (b) \triangleq c \circ (b_1) \circ c \oplus (\neg b_2)
\end{align*}
\]
A program store \( \sigma : \mathbb{V} \rightarrow \mathbb{Z} \) is a total function from a finite set of variables of interest \( \mathbb{V} \subseteq \mathbb{V} \) to values and \( \Sigma \equiv \mathbb{V} \rightarrow \mathbb{Z} \) denotes the set of stores. The concrete domain is \( \mathbb{C} \equiv \phi(\Sigma) \), ordered by inclusion. Store update is defined as usual and lifted to sets \( S \subseteq \Sigma, S[x \mapsto o] \equiv \{ \sigma[x \mapsto o] \mid \sigma \in S \} \). The semantics \( (l) : \mathbb{C} \rightarrow \mathbb{C} \) of basic commands is standard:
\[
\begin{align*}
(\text{skip}) S & \equiv S \\
(x := a) S & \equiv \{ \sigma[x \mapsto \{a\} \sigma] \mid \sigma \in S \}, \\
(b_1) S & \equiv \{ \sigma \in S \mid \{b_1\} \sigma = \text{tt} \}
\end{align*}
\]
where \( \{a\} : \Sigma \rightarrow \mathbb{Z} \) and \( \{b_1\} : \Sigma \rightarrow \{\text{tt, ff}\} \) are inductively defined as expected. For brevity, we sometimes overload \( b \) to denote the set \( \{b\} \Sigma \) of all and only the stores that satisfy \( b \), so that \( \{b\} S = S \cap b \).

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 \( [e]A \) 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 non 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 \( \subseteq \) in the standard lattice of abstract interpretation.

4.1 Local Completeness
Definition 4.1 (Local Completeness [8]). An abstract domain \( A \in \text{Abs}(C) \) is locally complete for a function \( f : C \rightarrow C \) on a value \( c \in C \), denoted by \( C_A^l(f) \), if \( Af(c) = AfA(c) \), i.e.
\[
C_A^l(f) \Rightarrow Af(c) = AfA(c).
\]
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 \( C_A^l(f) \) holds then \( C_A^l(f) \) 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
\[
\begin{align*}
&c \triangleq (0 < x) \text{ then } x := x - 2 \text{ else } x := x + 1 \\
&\text{and the interval abstraction Int. While the Boolean guard }0 < x?\text{ is not globally complete, it turns out that }[0 < x?]\text{ is locally complete for any } P \in \phi(\Sigma) \text{ such that:}
\end{align*}
\]
\[
\begin{align*}
&(1) P \subseteq \mathbb{Z}_{>0}, \text{ or } (2) P \subseteq \mathbb{Z}_{\leq 0}, \text{ or } (3) \{0, 1\} \subseteq P.
\end{align*}
\]
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 \( \text{Int}(\{c\}P_1) = \text{Int}(\{0, 3\}) = \{0, 3\} \) and \( \text{Int}([c]\text{Int}(P_1)) = \text{Int}(\{0, 2, 5\}) = \{0, 1, 2, 3\} = \{0, 3\} \). Contrariwise, if \( P_2 = \{0, 3\} \) we have that \( \text{Int}(\{c\}P_2) = \text{Int}(\{1\}) = \{1, 1\} \), but \( \text{Int}([c]\text{Int}(P_2)) = \text{Int}(\{c\}0, 3\}) = \text{Int}(\{1, -1, 0\}) = \{-1, 1\} \).

Consider now the composite command \( c : c \). While \( c \) is locally complete for \( c \) on \( P_1 \), it is not locally complete for \( c \) on \( P_1 \), because \( \text{Int}(\{c,c\}2, 5\}) = \text{Int}(\{c\}0, 3\}) = \{1, 1\} \), but \( \text{Int}(\{c,c\}\text{Int}(\{2, 5\}) = \text{Int}(\{c,c\}2, 5\}) = \{-1, 1\} \).
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 $f : C \to C$, $A \in \text{Abs}(C)$ and $c \in C$. The local completeness set $L^A_{c,f}$ is defined by $L^A_{c,f} = \{ x \in C \mid x \leq A(c), f(x) \leq Af(c) \}$. □

The local completeness set $L^A_{c,f}$ 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 $L^A_{c,f}$ is dominated by a maximal one in $\text{max}(L^A_{c,f})$, 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 $f : C \to C$, $A \in \text{Abs}(C)$ and $c \in C$.

(i) If $f$ is continuous then $C^A_c(f) \Leftrightarrow \text{max}(L^A_{c,f}) \subseteq A$.

(ii) In particular, if $f$ is additive then $C^A_c(f) \Leftrightarrow L^A_{c,f} \subseteq A$.

Furthermore, $\forall x \in C \mid f(x) \leq Af(c)$.

**Example 4.5.** Let us consider the interval abstraction Int and the function $\llbracket c \rrbracket : \varphi(\mathbb{Z}) \to \varphi(\mathbb{Z})$ 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 $\llbracket c \rrbracket$ is an additive function):

1. $\llbracket 1 \rrbracket_{P_1,c} = \{ x \in \varphi(\mathbb{Z}) \mid x \subseteq \text{Int}(P_1), \llbracket c \rrbracket x \subseteq \text{Int}(\llbracket c \rrbracket P_1) \} = \{ x \in \varphi(\mathbb{Z}) \mid x \subseteq [2, 5], \llbracket c \rrbracket x \subseteq [0, 3] \}$ and $\llbracket 2 \rrbracket_{P_1,c} = [2, 5]$

2. $\llbracket 3 \rrbracket_{P_1,c} = \{ x \in \varphi(\mathbb{Z}) \mid x \subseteq \text{Int}(P_2), \llbracket c \rrbracket x \subseteq \text{Int}(\llbracket c \rrbracket P_2) \} = \{ x \in \varphi(\mathbb{Z}) \mid x \subseteq [0, 3], \llbracket c \rrbracket x \subseteq [1, 1] \}$ and $\llbracket 4 \rrbracket_{P_1,c} = [0, 3]$ so that $\cup \llbracket 1 \rrbracket_{P_1,c} \subseteq \text{Int}$, while $\cup \llbracket 3 \rrbracket_{P_1,c} \not\subseteq \text{Int}$. □

### 4.2 Exact Locally Complete Shells

If local completeness $C^A_c(f)$ 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 $\text{Abs}(C)$ (cf. Section 3.1). For the property of local completeness of a transfer function $f : C \to C$ on some point $c \in C$, the exact locally complete shell of $A$ exists when its minimal refinement $A_{\text{ref}} \sqcup \{ B \in \text{Abs}(C) \mid B \subseteq A, C^A_c(f) \}$ 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 $f : \varphi(\mathbb{Z}) \to \varphi(\mathbb{Z})$ defined as follows: $f(X) = \{ x + 1 \mid x \in X \}$. Thus, $f$ is the additive transformer of the imperative assignment $x := x + 1$, namely, $f = \{ x := x + 1 \}$. We consider the toy abstract domain $A = \{ \mathbb{Z}, [0, 4], [1, 3] \} \in \text{Abs}(\varphi(\mathbb{Z}))$, 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 $C^A_c(f)$ 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 $A$:

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

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 $P$: $A_{[0,2]}fA_{[0,2]}(P) = A_{[0,2]}f(\{ 0, 2 \}) = A_{[0,2]}(\{ 1, 3 \}) = \{ 1, 3 \} = A_{[0,2]}(\{ 1, 3 \}) = A_{[0,2]}(P)$. Therefore, $A_{[0,2]}, A_{[0,2]} \in \{ B \in \text{Abs}(\varphi(\mathbb{Z})) \mid B \subseteq A, C^B_c(f) \}$. Since $A_{[0,2]} \cup A_{[0,2]} = A_{[0,2]} \cap A_{[0,2]} = A$, it turns out that $A_{\text{ref}} = \{ B \in \text{Abs}(\varphi(\mathbb{Z})) \mid B \subseteq A, C^B_c(f) \} = A$. 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 $A_p$ of a given abstraction $A$. The following result shows which pointed refinements $A_p$ may achieve local completeness.

**Lemma 4.7.** Let $f : C \to C$ be monotone, $A \in \text{Abs}(C)$ 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 $C^A_c(f)$, i.e. local completeness holds, and $x \notin A(c)$ then there exists some $y < A(c)$ such that $A_y \subseteq A_p$ and $C^A_c(f)$.

Hence, this result tells us that if $A$ is not locally complete on $c$ and we are looking for a pointed refinement $A_p$ that achieves local completeness on $c$ then we can restrict our search to those new points $x$ which are below $A(c)$, because for $x \notin A(c)$ 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 $A_{[0.2]} \cup A_{[0.2]}$ is $A$ itself, which is not locally complete. Notice that for the standard precision ordering $\subseteq$ in the lattice of abstract interpretations $\text{Abs}(C)$, 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, $\{0, 2\} \not\subseteq [0, 2]$ 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 $c^\#$ for $c$ guarantees that local completeness holds not only for $c$ but also for any concrete element in between $c$ and $c^\#$. We therefore put forward a novel notion of shell refinement of abstract domains, which is restricted to pointed refinements $A_u$ and compares them by the relative precision ordering of their new point $x$.

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

$$\max\{\{x \in C \mid x \leq A(c), C^A_c(f)) = \{u\} \quad (1)$$

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

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 $f : C \rightarrow C, A \in \text{Abs}(C)$ and $c \in C$, and let $u \triangleq \vee A^A_c(f)$. Then

(i) If $f$ is monotone then: $C^A_u(f) \Leftrightarrow A_u$ is the pointed shell of $A$ for $f$ on $c$.

(ii) If $f$ is additive then: $C^A_u(f) \Leftrightarrow (f(c) \leq u \Rightarrow f(u) \leq u)$. 

Hence, for an additive function $f$, such as a collecting semantics, and the new concrete element $u = \vee L^A_c(f)$, the pointed shell of $A$ exists exactly when $f(c) \leq u$ 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: $L^A_{[0, 2]} = \{X \in \mathbb{Z} \mid X \subseteq \text{Abs}(P), f(X) \subseteq A_f(P) = \{X \in \mathbb{Z} \mid X \subseteq [0, 4], f(X) \subseteq [1, 3]\}$, so that $u = \cup L^A_{[0, 2]} = \{0, 2\}$. 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 $\text{Int}$ is not locally complete on $P_2 = \{0, 3\}$ for the program $c$ in Example 4.2. We have already seen that $u = \cup \text{Int}_{P_2} = \{0, 3\}$. Since $\text{Int}_{P_2} = \{1\} \not\subseteq \{0, 3\} = u$, the condition $\text{Int}_{P_2} \subseteq u \Rightarrow \{c\} u \subseteq u$ of Theorem 4.9 (ii) is satisfied, and $\text{Int}_{\{0, 3\}} = \text{Int} \cup \{0, 3\}$ is the pointed shell of $\text{Int}$ for $\{c\}$ 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 $[b?]$ and $[\neg b?]$. Definition 4.8 can be slightly generalized as follows: the pointed locally complete shell of $A$ in $c$ exists for a set of functions $F \subseteq C \rightarrow C$ when

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

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 $b \in \text{BExp}, A \in \text{Abs}(\mathbb{S})$ and $P \in \mathbb{S}$, let

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

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

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

$$u \triangleq (\text{Int}(P \cap x \leq 0) \cap x \leq 0) \cup (\text{Int}(P \cap x > 0) \cap x > 0)$$

$$= (\text{Int}([3, -1]) \cap \neg b) \cup (\text{Int}([2]) \cap b) = \{[-3, 1] \cap \{2\}, [-2, 1] \cup \{2\}, \{1, 2\}\}. □$$

### 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 $f = f_n \circ \ldots \circ f_1$, e.g., modeling the sequential composition of $f_1, \ldots, f_n : C \rightarrow C$, 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 CEGRAR, the functions $\hat{f}_k$ are the post transformers of a transition system, while in program verification each $\hat{f}_k$ is the transfer function of some basic command. In abstract interpretation, we select an abstract domain $A \in \text{Abs}(C)$ such that the property $a$ is expressible in $A$ and then we check whether $\hat{f}_A A(c) \leq A a$ holds, where $f_A \triangleq f_A \circ \ldots \circ f_1 A$ and each $f_A$ 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 $f_A$. However, if $\hat{f}_A A(c) = A(f(c))$ holds, then from $\hat{f}_A A(c) \nleq A a$ we can conclude that $a$ is not met, because $\hat{f}_A A(c) \nleq A a$ implies $f(c) \nleq a$ when $a \in A$. 433
Forward Repair Strategy. By letting $c_k \equiv f_{k-1} \ldots f_1(c)$, with $c_1 \equiv c$, the equality $f^A_A(c) = A(f(c))$ is a consequence of $n$ local completeness proof obligations:

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

The forward repair strategy processes these proof obligations in increasing index order: as soon as the smallest index $k$ is found such that $A_f(c_k) \neq A_f(A(c_k))$, 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 $A_f(c_k) \neq A_f(A(c_k))$, 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 $A_f(c_k)$ and $A_f(A(c_k))$. 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 $u \equiv \forall \ell \in \mathcal{L}_{A_u}$ (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 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 $f^A_A(c_k) \leq a$, where, by local completeness of $A_f$ for $f_k$ on $c_k$, we have that $f^A_A(c_k) = f^A_f(c_k)$ holds. Backward repair relies instead on weakest liberal preconditions [24] and looks for a refinement $A_h$ such that $A_h(f^A_f(c_k) \ldots f^A_1 A_1(c_k)) \leq a$ holds, and $A_h(f^A_f(c_k) \ldots f^A_1 A_1(c_k)) = f^A_f(c_k)$ follows by local completeness. Recall that, given $f : C \rightarrow C$ and $z \in C$, the weakest liberal precondition of $f$ for $z$ is

$$\wlp(f, z) = \exists \{ x \in C \mid f(x) \leq z \}.$$

When $f$ is additive, the key property of $\wlp$ is that $f(c) \leq a$ is equivalent to $c \leq \wlp(f, a)$, and, analogously, $f^A_A(c) \leq a$ iff $A(c) \leq \wlp(f^A_A, a)$. Letting $a_k \equiv f^A_A(c_k)$, we inductively define $a_k \equiv a_1 \wedge \wlp(f_{k+1}, a_{k+1})$, with $a_n \equiv a_n \wedge a$. Thus, backward repair amounts to check and possibly repair $n$ local completeness proof obligations:

$$\forall k \in [1, n]. \ A_f(\delta_k) = A_f(A(\delta_k)),$$

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 $c_k \neq \delta_k$. 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 $A_h$ 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 $\delta_k$ is just $A_h \delta_k$; $\delta_k$ 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 $S = \langle E, \rightarrow \rangle$ whose successor/predecessor transformer post/pre are defined as $\text{post}(X) = \{ t \mid \exists x \in X, s \rightarrow t \}$ and $\text{pre}(X) = \{ s \mid \exists t \in X, s \rightarrow t \}$, for all $X \in \wp(E)$. Given a partitioning abstraction $A \in \text{Abs}(\wp(E))$ [43, 52, 53], whose abstract elements are unions of blocks of a partition of the state space $E$, abstract model checking [2, 12, 13] exploits the existential abstract transition relation $\rightarrow^A \subseteq A \times A$ defined as follows: for any $B, B' \in A$, $B \rightarrow^A B'$ iff $\exists x \in B.\exists y \in B'.x \rightarrow y$. The resulting abstract transition system $\mathcal{A} = \langle A, \rightarrow^A \rangle$ is then
we have that $S$ An abstract path (crossed circles in Fig. 2, those with some arcs to nodes in $B$). For any $i \in [2, n]$ (for simplicity, loop path counterexamples are not treated here). Its underlying concrete paths are defined as follows: $\text{paths}(\pi) = \{(s_1, ..., s_n) \mid \forall i \in [1, n], s_i \in B_i, \forall i \in [2, n], s_{i-1} → s_i\}$. An abstract path $\pi$ is spurious if $\text{paths}(\pi) = \emptyset$ and we define a corresponding sequence $\text{sp}(\pi) = (S_1, ..., S_n) \in \varphi(\Sigma)^n$ as:

$$S_i \triangleq B_i \neq \emptyset; \quad \forall i \in [1, n-1]. S_{i+1} = \text{post}(S_i) \cap B_{i+1}. \quad (2)$$

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 $B_i$. As observed in [11, Lemma 4.10], it turns out that $\pi$ is spurious if there exists a least $k \in [1, n-1]$ such that $S_{k+1} = \emptyset$. Fig. 2 depicts a spurious counterexample $\pi = (B_1, ..., B_n)$, 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 $B_{k+1}$, and, in turn, the model checker is run on this refined abstract transition system. The refinement of the block $B_{k+1}$ relies on the three following subsets of states: $\text{dead-end states}$ $B^\text{dead}_k \triangleq S_k \neq \emptyset$ (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 $\text{bad states}$ $B^\text{bad}_k \triangleq B_k \cap \text{pre}(B_{k+1}) \neq \emptyset$ (crossed circles in Fig. 2, those with some arcs to nodes in $B_{k+1}$); all the other states $B^\text{irr}_k \triangleq B_k \setminus (B^\text{dead}_k \cup B^\text{bad}_k)$ 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], $B_i$ is split into $B^\text{dead}_i$ and $B^\text{bad}_i \cup B^\text{irr}_i$. 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 $B_k \cup B^\text{irr}_k$).

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

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

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

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

$$A(\text{post}_{n_k}(S_k)) = A(\text{post}_{n_k}(A(S_k))). \quad (4)$$

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

$$\emptyset = A(\emptyset) = A(\text{post}_{n_k}(S_k)) \neq \emptyset = A(\text{post}_{n_k}(B_k)) \cap B_k \neq \emptyset.$$ 

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 $S_k = B^\text{dead}_k$ and that any state in $B^\text{dead}_k$ cannot be part of the abstraction of $S_k$ (otherwise local incompleteness persists). Thus, our forward repair strategy takes the pointed shell refinement $A \sqsupseteq \{B^\text{dead}_k \cup B^\text{irr}_k\} = A \sqsupseteq \{B^\text{dead}_k \cup B^\text{irr}_k \cup B^\text{bad}_k\}$. Again, other spurious counterexamples may still arise (see the arcs in Fig. 2 from $B_{k-1}$ to crossed nodes in $B^\text{bad}_k$).

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

**CEGAR by Backward Repair.** In backward repair, the elements $V_k = B_k \cap \text{wp}(\text{post}_{n_k}(V_{k+1}))$ (with $V_1 = \emptyset$) can just be defined as $V_k \subseteq B_k \setminus T_k$, where $T_k \subseteq B_k$ is the set of states with a path of length $n-k$ to some state in $B_k$. Formally:

$$T_k \triangleq B_k \setminus \emptyset; \quad \forall i \in [1, n-1]. T_i \triangleq \text{pre}(T_{i+1}) \cap B_i.$$ 

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

$$A'(\text{post}_{n_k}(V_k)) = A'(\text{post}_{n_k}(A'(V_k))). \quad (5)$$

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

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

Fig. 3, where nodes in $T_k$ 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 $\llbracket r \rrbracket^A_A : A → A$ to also denote the function $\llbracket r \rrbracket^C_A : C → A$ applicable to concrete input properties. Moreover, we write a simpler $C^A_A(r)$ instead of $\llbracket r \rrbracket^A_A$ 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 $\llbracket e \rrbracket^A_A = a \Leftrightarrow \llbracket e \rrbracket^A_{A_u} = a \Leftrightarrow \llbracket e \rrbracket^C_A = a$ otherwise.

The verification problem for a program $r \in \text{Reg}$ on input $P \subseteq C$ and correctness specification $\text{Spec} \subseteq C$, consists in
checking whether $[r]P \leq \text{Spec}$ holds. Abstract interpretation exploits over-approximations to interpret $r$ on a (tractable) abstract domain $A$. By soundness of $[r]A$, if $[r]A \leq \text{Spec}$ holds then $[r]P \leq \text{Spec}$ follows, but the reverse implication is not necessarily true. Nevertheless, if the abstraction $A$ is locally complete for $[r]$ on $P$ and Spec is expressible in $A$ then $[r]P \leq \text{Spec}$ implies $[r]A \leq \text{Spec}$. 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

$$
[r]P \leq \text{Spec} \iff [r]A_N \leq \text{Spec} \quad (6)
$$

To see the analogy with abstract model checking, given a CEGAR abstract counterexample $\pi = (B_1, ..., B_n)$, assume that we define a regular command $r_\pi = e_1; e_2; ...; e_n$ such that the semantics $[e_k]_A$ of each basic expression is exactly the function post$_{s_k}$ defined in (3). Then, if we take the input property $P \triangleq B_1$ and a trivial specification $\text{Spec} \triangleq \bot$, it turns out that $[r_\pi]P \leq \text{Spec}$ holds iff $\pi$ is spurious. The correspondence with the general scenario presented in Section 5 is recovered by taking $c = P$, $a = \text{Spec}$ and $f_k = \text{post}_{s_k} = [e_k]_A$.

### 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 $A_N = A \equiv N$, simply passed as $N$, a current set of stores $P$ and a command $r$ and, either returns an under-approximation $Q$ satisfying $Q \subseteq [r]P$ and $A_N(Q) = A_N([r]P)$ — thus meaning that $A_N$ is locally complete for $[r]$ on $P$ — or returns a pair $(R, e)$ for some set of stores $R$ and basic command $e$ occurring in $r$ such that a local completeness proof obligation $\Box^{A_N}(e)$ is not met.

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

**Algorithm 1: Forward repair procedure 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 (not found);
9. return $(N, out)$;

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

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

and consequently, by meet closure, all the intervals with a hole in 0. In the next iteration, find$_{\text{Int}}(\emptyset, I, r_{\text{abs}})$ is called and this returns $Q = \{x \in \mathbb{Z} \mid x > 0, x \text{ is odd}\}$. By Theorem 7.1, we know that $[r]P \subseteq \text{Int}(\mathbb{Z}_{\geq 0}) = [1, +\infty]$ 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 $P' \leq A(P)$:

$$\forall P' \leq A(P) \Rightarrow ([r]^{A_N}P' \leq \text{Spec} \iff [r]P' \leq \text{Spec}) \quad (7)$$

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

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

$$\text{V}(P, r, Spec) \triangleq \forall C' \in C \mid P' \leq P, [r]P' \leq \text{Spec},$$

denote the greatest valid input set.

It turns out that $\text{V}(P, r, Spec) = P \land \text{wp}(\llbracket r \rrbracket, \text{Spec})$. As an example, for the basic expressions in Exp we have that:

$$\text{V}(P, \text{skip}, S) \triangleq P \cap S, \quad \text{V}(P, b, S) \triangleq P \cap (S \cup \neg b),$$

$$\text{V}(P, x := a, S) \triangleq \{s | s \in P \land s[x := a] \in S\}.$$
Algorithm 2: Backward repair procedure \( \text{bRepair}_A \).

1. Function \( \text{bRepair}_A(N, \bar{P}, r, s) \):
   2. if \( \langle \vec{t} \rangle_{\text{ABN}}^{\vec{r}} \subseteq S \) then return \( \langle \bar{P}, N \rangle \);
   3. switch \( r \) do
      4. case \( e \) do // basic expression
         5. \( V = V(\bar{P}, e, S) \); \( Q = S \land \langle \vec{t} \rangle_{\text{ABN}}^{\vec{r}} \bar{P} \);
         6. return \( (V, N \cup \{V, Q\}) \);
      7. case \( r_{t_1} r_{t_2} \) do // sequential
         8. \( (V_1, N_1) := \text{bRepair}_A(N, \langle v \rangle_{\text{ABN}}^{\vec{r}} \bar{P}, r_{t_1}, S) \);
         9. \( (V_2, N_2) := \text{bRepair}_A(N, \bar{P}, r_{t_2}, V_1) \);
        10. return \( (V_2, N_2 \cup N_1 \cup \{Q\}) \);
      11. case \( t_0 \oplus r_1 \) do // choice
         12. \( (V_0, N_0) := \text{bRepair}_A(N, \bar{P}, r_{t_0}, S) \);
         13. \( (V_1, N_1) := \text{bRepair}_A(N, \bar{P}, r_{t_1}, S) \);
        14. \( Q = S \land \langle \vec{t} \rangle_{\text{ABN}}^{\vec{r}} \bar{P} \);
        15. return \( (V_0 \lor V_1, N_0 \cup N_1 \cup \{Q\}) \);
      16. case \( r \) do // Kleene star
         17. if \( (\bar{P}) \) then return \( \text{inv}_A(N, \bar{P}, r_{t_0}, S) \);
         18. else \(/\!\!/\text{unroll} \)
         19. \( (V_1, N_1) := \text{bRepair}_A(N, \bar{P}, \text{ABN}, r_{t_1} r_{t_0}, S) \);
        20. return \( (V_1 \land N_1) \);
   21. end

22. Function \( \text{inv}_A(N, \bar{P}, r, V_1) \) // loop invariants
   23. do
      24. \( V_0 := \bar{P} \land V_1 ; N_0 := N \cup \{V_0\} \);
      25. \( (V_i, N_i) := \text{bRepair}_A(N_0, V_0, V_i, r, V_1) \);
   26. while \( V_i \neq V_0 \);
   27. return \( (V_i \land N_i) \);

Theorem 7.4. Let \( r \in \text{Reg}, A \in \text{Abs}(C), P, \text{Spec} \in C, \) and \( A_N \triangleq A \oplus N. \) Then, condition (7) holds if and only if
\[
\langle \vec{r} \rangle_{\text{ABN}}^{\vec{r}} V(A(P), r, \text{Spec}) \leq \text{Spec}. \tag{8}
\]

7.6 (bRepair_{A} and inv_{A} are Sound). For any \( A \in \text{Abs}(C), r \in \text{Reg}, \bar{P} \in A, S, \text{Spec} \in C, \) and \( N, N' \subseteq C. \)

1. If \( \text{bRepair}_{A}(N, \bar{P}, r, s) = \langle V, N' \rangle \) then:
   a. \( \langle V \rangle A \subseteq N' \); \( b. \langle \vec{r} \rangle_{\text{ABN}}^{\vec{r}} V \leq S \); \( c. \langle V \rangle \bar{P} \land r, S \).
2. If \( \langle \vec{r} \rangle_{\text{ABN}}^{\vec{r}} \bar{P} \leq \bar{P} \) and \( \text{inv}_{A}(N, \bar{P}, r, S) = \langle V, N' \rangle \) then:
   a. \( \langle V \rangle A \subseteq N' \); \( b. \langle \vec{r} \rangle_{\text{ABN}}^{\vec{r}} V \leq S \); \( c. \langle V \rangle \bar{P} \land r, S \).

7.7 (Program (In)Correctness). Let \( A \in \text{Abs}(C), r \in \text{Reg}, \bar{P} \in A, \) and \( \text{Spec} \in C. \) For any \( V \in C, N' \subseteq C \) such that \( \text{bRepair}_{A}(\bar{P}, \bar{P}, r, \text{Spec}) = \langle V, N' \rangle, \) we have that:
\[
\forall P' \leq \bar{P}. \langle \vec{r} \rangle P' \leq \text{Spec} \iff \langle \vec{r} \rangle_{\text{ABN}}^{\vec{r}} P' \leq \text{Spec} \iff P' \leq V.
\]

As a special case of Corollary 7.7, taking \( \bar{P} = A(P) \) it turns out that \( \langle \vec{r} \rangle P \leq \text{Spec} \) 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]:
\[
c \doteq \text{while} \ (x > 0) \ \{ x := x - 1; \ y := y - 1 \}
\]
with input \( \bar{P} \triangleq x < 100 \) and \( \text{Spec} \triangleq y = 0. \) 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 \( \text{Spec}. \) We therefore apply backward repair to determine the valid inputs for \( \text{Spec}. \) The Imp program \( c \) is encoded by the following regular command:
\[
r \doteq (x > 0); \ x := x - 1; \ y := y - 1 \}; \ (x \leq 0)?
\]
The call to \( \text{bRepair}_{\text{Int}}(\tilde{\bar{P}}, \bar{R}, r, \text{Spec}) \) selects the case for sequential composition and, letting \( \tilde{R} \doteq \langle \vec{r} \rangle_{\text{Int}} \bar{P} = x \in [0, 100], \) invokes \( \text{bRepair}_{\text{Int}}(\tilde{\bar{P}}, \tilde{R}, (x \leq 0)?, \text{Spec}) \), which returns the pair \((Q, \{Q\})\) with \( Q \doteq x \in [0, 100] \land (x = 0 \Rightarrow y = 0). \) In turn, \( \text{bRepair}_{\text{Int}}(\tilde{\bar{P}}, \tilde{R}, r_{t_1}, Q) \) is then called. After one recursive call to unroll the loop, we call \( \text{inv}_{\text{Int}}(\tilde{\bar{P}}, \tilde{R}, r_1, Q). \) Iteratively, the largest invariant under \( \tilde{R} \land Q = Q \) is found.
that is \( P \vdash x \in [0, 100] \land y = x \). 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 \( R_1 \vdash x \in [1, 100] \land y = x, R_2 \vdash x \in [0, 99] \land y = x + 1, \) and \( R_3 \vdash x \in [0, 99] \land y = x \) are also added. In the end, the call to \( \text{invInt} \left( P, \{P, R_1, R_2, R_3\} \right) \), so that the original call to \( \text{bRepairInt} \) returns \( \{R_1, \{P, R_1, R_2, R_3, Q\}\} \) because \( R_1 = P \land \overline{P} \). By Corollary 7.7, for any \( P' \leq P \), we have that \( \lceil r \rceil \rightarrow P' \leq \text{Spec} \) if \( P' \leq R_1 \). Remarkably, backward repair is able to add the minimal relational information in a nonrelational domain as needed to prove \( \text{Spec} \). Since all the new elements are octagons, incidentally, the analysis on Oct with input \( R_1 \) (instead of \( P \)) is also able to prove that \( \text{Spec} \) 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 \( p \vdash (z = 0) \) and \( q \vdash (x = y) \), as depicted below:

\[
\begin{align*}
c & \vdash \text{do} \{ \begin{array}{l} z := 0; \ \ x := y; \\
\text{if} \ (w \neq 0) \ \ \text{then} \{ x := x + 1; \ \ z := 1 \} \end{array} \} \\
& \text{while} \ (x \neq y).
\end{align*}
\]

We want to prove that \( \lceil c \rceil \rightarrow A = p \), whereas it turns out that \( \lceil c \rceil_A \rightarrow \otimes = q \). The domain refinement used in [4, 5] is the reduced disjunctive completion of \( A \), which is isomorphic to the Boolean abstraction \( B = \langle \phi (p \land q, p \land \overline{q}, \overline{p} \land q, \overline{p} \land \overline{q}) \rangle \). The analysis with \( B \) leads exactly to the same result as with \( A \), namely \( \lceil c \rceil_{\overline{B}} \rightarrow \otimes = p \land q \).

By invoking \( \text{bRepair}_{\otimes} (\otimes, T, r, c, p) \) we get as a result the pair \( \langle r, \{q \rightarrow p\} \rangle \). In fact, letting \( A_2 \supseteq A \vdash \{q \rightarrow p\} \) we get \( \lceil c \rceil_{\overline{A}} \rightarrow \otimes = A_2 \rightarrow q \). We remark that the point \( q \rightarrow p \) is indeed more abstract than \( p \rightarrow q \). □

**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 \( x \nu A y \) by a more abstract element \( x \nu A y \). 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 \( A \in \text{Abs}(C) \), a widening operator \( \nu A : A \times A \rightarrow A \) is a function such that: (i) for all \( x, y \in A \), \( x, y \subseteq A \nu A x \); and (ii) for every sequence \( x_i \) in \( A \), the chain \( y_i \) inductively defined by \( y_0 = x_0 \) and \( y_{i+1} = y_i \nu A x_{i+1} \) finitely converges (i.e., \( \exists k \in \mathbb{N}, \forall i \in \mathbb{N}, \nu A x_{i+k} = y_k \)).

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

\[
\lceil r \rceil_A \nu A (\overline{\nu A \lceil r \rceil_A X}) = \text{Inv}(\nu A X \nu A (\overline{\nu A \lceil r \rceil_A X})).
\]

**Figure 4. Backward repair call scheme for Example 7.13.**

By termination condition (ii) of \( \nabla A \), the iteration sequence for the least fixpoint of \( \lambda X. \overline{\nu A (\overline{\nu A \lceil r \rceil_A X})} \) always finitely converges. We settle the problem of how a suitable widening operator for \( A \sqsubseteq B \) can be derived from a widening for \( A \).

**Definition 7.11 (Pointed Widening).** Given a refinement \( A_N \equiv A \sqsubseteq B \) of \( A \in \text{Abs}(C) \) and a widening \( \nabla A \) on \( A \), the pointed widening \( \nabla A : A_N \times A_N \rightarrow A_N \) is defined as follows:

\[
\langle x \rangle \rightarrow \nabla A \left( \langle x \rangle \rightarrow A_N \right) \rightarrow A_N \rightarrow \langle x \rangle.
\]

**Theorem 7.12 (Soundness of Pointed Widening).** If \( N \subseteq C \) is finite, then \( \nabla A_N \rightarrow \nabla A \) is a widening operator for \( A \sqsubseteq B \).

In Algorithm 2, line 20, we simply replace \( \overline{\nu A \nu A \nu A} \) with \( \overline{\nu A \nu A \nu A} \) 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 \( \text{bRepair}_A (N, \overline{\nu A \nu A \nu A}, r, s) \) returns \( \langle V, \nu A \rangle \), 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 \( Spec \equiv j \leq 15 \) and

\[
\langle r, \{r_2 \rightarrow r_{\overline{r}} \rangle \rightarrow \langle r, \{r_{\overline{r}} \rightarrow r_2 \} \rangle.
\]

The call to \( \text{bRepairInt} (\otimes, T, r, \overline{\nu A \nu A \nu A}) \) recursively computes \( \text{bRepairInt} (\otimes, P, b') \), where \( P = \langle r_2 \rangle_\text{Int} = \langle i \in [1, 6] \land j \in [0, \infty] \rangle \) (using the widening on intervals for \( j \), which returns \( \langle V, \nu A \rangle \): the element \( \nu A \rightarrow (\overline{\nu A \nu A \nu A}) \rightarrow \langle i \in [1, 6] \land j \in [0, \infty] \rangle \) is introduced to repair \( b' \). This leads to call \( \text{bRepairInt} (\otimes, T, r, \overline{\nu A \nu A \nu A}) \). Then, \( \text{bRepairInt} (\otimes, P, r_{\overline{r}}) \), and, then \( \text{bRepairInt} (\otimes, P, r_{\overline{r}}) \), with \( P = \langle r \rangle_\text{Int} = \langle i = i \rangle \). On unrolling \( r_{\overline{r}} \) to reach the abstract invariant \( P \), the call to \( \text{invInt} (\otimes, P, r, V) \) computes the largest invariant under \( P \rightarrow V \), which is \( P = \langle i \in [1, 6] \rightarrow j \in [0, i(i-1)/2 \rangle \). Then \( \text{bRepairInt} (\langle P, P, r, s \rangle) \) 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, \( \text{invInt} (\otimes, P, r, V) \) returns \( \langle P, P, R_1, R_2, R_3 \rangle \). The point \( P \) is then intersected with the approximations

\[
\langle T, \{P, R_1, R_2, R_3, V\} \rangle \rightarrow \text{bRepair} (\otimes, T, r, \overline{\nu A \nu A \nu A}) \rightarrow \text{bRepair} (\otimes, T, r, \overline{\nu A \nu A \nu A}) \rightarrow \text{bRepair} (\otimes, T, r, \overline{\nu A \nu A \nu A})
\]
computed by the unrolling of \( r^*_i \) until \( \text{bRepair}_{\text{inf}}(\emptyset, \overline{P}_i, r^*_i, V) \) outputs \( (\overline{P}_i, \{ P, R_1, R_2, R_3 \}) \), so that \( \text{bRepair}_{\text{inf}}(\emptyset, \tau, r, \emptyset) \) returns \( (\tau, \emptyset) \), because \([r_3]_0^* \leq \tau = \overline{P}_i\), and the original call to \( \text{bRepair} \) outputs \( (\tau, \{ P, R_1, R_2, R_3 \}) \). By Corollary 7.7, for all \( P' \leq \tau, [r]_0^n P' \leq \text{Spec} \) 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 \( S \), 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 \( S \). For instance, Example 7.8 shows that sometimes a nonrelational abstract domain as \( \text{Int} \) must be enriched with some elements of a more concrete relational domain like \( \text{Oct} \) 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 \( S \), i.e., such that \( S \subseteq C \). 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 \( S \subseteq C \subseteq A \) will isolate within \( C \), and not within \( S \), 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 \( L \) 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 \( L \). 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 \( L \), 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 \( LCL_1 \) 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 \( L \) envisioned in Section 8, it may still be the case that the forward repair can select a less abstract but admissible representation in \( L \), 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”.

439
References


