Local Completeness in Abstract Interpretation

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato

Abstract Completeness of an abstract interpretation is an ideal situation where the abstract interpreter is guaranteed to be compositional and producing no false alarm when used for verifying program correctness. Completeness for all possible programs and inputs is a very rare condition, met only by straightforward abstractions. In this paper we make a journey in the different forms of completeness in abstract interpretation that emerged in recent years. In particular, we consider the case of local completeness, requiring precision only on some specific, rather than all, program inputs. By leveraging this notion of local completeness, a logical proof system parameterized by an abstraction A, called LCL_A, for Local Completeness Logic on A, has been put forward to prove or disprove program correctness. In this program logic a provable triple $[p]\ \mathtt{c}\ [q]$ not only ensures that all alarms raised for the postcondition q are true ones, but also that if q does not raise alarms then the program $\mathtt{c}$ cannot go wrong with the precondition p.

Key words: Abstract interpretation, program logic, completeness, incorrectness.

Roberto Bruni
University of Pisa, Pisa, Italy e-mail: `roberto.bruni@unipi.it`

Roberto Giacobazzi
University of Verona, Verona, Italy e-mail: `roberto.giacobazzi@univr.it`

Roberta Gori
University of Pisa, Pisa, Italy e-mail: `roberta.gori@unipi.it`

Francesco Ranzato
University of Padova, Padova, Italy e-mail: `francesco.ranzato@unipd.it`

1 Completeness, Fallacy, and Approximation

Formal methods are fundamental to have rigorous methods and correct algorithms to reason about programs, e.g., to prove their correctness or even incorrectness. These include program logics, model checking, and abstract interpretation as the most prominent examples. Abstract interpretation [8, 23] and most of well-known program logic, notably Hoare logic [28], have a lot in common. On the one hand, abstract interpretation can be seen as a fixpoint strategy for implementing a Hoare logic-based verifier. On the other hand, Hoare logic can be seen as a logical, i.e. rule-based, presentation of an abstract interpreter, deprived of a fixpoint extrapolation strategy. In this sense, abstract interpretation solves a harder problem than Hoare logic, and, more in general, program logics: Abstract interpretation provides an effective algorithmic construction of a witness invariant which can be used both in program analysis and in program verification [10]. It is in this perspective that abstract interpretation provides the most general framework to reason about program semantics at different levels of abstraction, including program analysis and program verification as a special case.

It is precisely in the ambition of inferring a program invariant that the need of approximation lies: approximation is inevitable to make tractable (e.g., decidable) problems that are typically intractable. The well known and inherent undecidability of all non-straightforward extensional properties of programs provides an intrinsic limitation in the use of approximated and decidable formal methods for proving program properties [39]. This is particularly clear in program analysis, where the required decidability of the analysis may introduce false positives/negatives. The soundness of a program analyser, which is guaranteed by construction in abstract interpretation, means that all true alarms (also called true positives) are caught, but it is often the case that false alarms (also called false positives) are reported. Of course, as in all alarming system, program analysis is credible when few false alarms are reported, ideally none. Completeness holds when the abstract semantics precisely describes, in the abstract domain of approximate properties, the property of the concrete semantics, namely when no false alarm can be raised.

To substantiate the role of completeness, let us introduce some formal notation. We consider Galois insertion-based abstract interpretations [7,8], where the concrete domain C and the abstract domain A are complete lattices related by a pair of monotonic functions $\alpha : C \longrightarrow A$ and $\gamma : A \longrightarrow C$, called, resp., abstraction and concretization maps, forming a Galois insertion. 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 abstraction and concretization maps. To simplify notation, in the following, we often use A as a function in place of $\gamma\alpha : C \longrightarrow C$. When the concrete domain is a lattice of program properties, a property $P \in C$ is *expressible* in the abstract domain A if $A(P) = P$. Given $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ and a predicate transformer $f : C \longrightarrow C$, an abstract function $f^\sharp : A \longrightarrow A$ is a *correct* (or *sound*) approximation of f if $\alpha f \leq f^\sharp\alpha$ holds. The *best correct approximation* (bca) of f in A is defined to be $f^A \triangleq \alpha f \gamma$, and any correct approximation $f^\sharp$ of f in A is such that $f^A \leq f^\sharp$ holds. The function $f^\sharp$ is a *complete* approximation of f (or just complete) if $Af = \gamma f^\sharp \alpha$ holds. The abstract domain A is called 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 for a sound abstract semantics defined in A. In this case, the loss of precision is only due to the abstract domain and not to the abstract function $f^\sharp$ itself, which must necessarily be the bca of f, namely if $f^\sharp$ is complete in A then $f^\sharp = f^A$ and $Af = AfA$ hold [26].

Although being desirable, completeness is extremely rare and hard to achieve in practice. It is indeed worth noting that the existance of false positives is not just a consequence of the required decidability of program analysis. In [16], the authors proved that for *any* non-straightforward abstraction A there always exists a program $\mathtt{c}$ for which any sound abstract interpretation of $\mathtt{c}$ on A yields one of more false positives. Later, in [1] the authors showed that any program equivalence induced by an abstract interpreter built over a non-straightforward abstraction violates program extensionality. Here, non-straightforward abstractions correspond to those abstract domains A that are able to distinguish at least two programs, i.e., if $[\![\cdot]\!]^\sharp_A$ is a sound abstract semantics defined on A of a concrete semantics $[\![\cdot]\!]$, then there exist two programs $\mathtt{c}_1$ and $\mathtt{c}_2$ such that $[\![\mathtt{c}_1]\!]^\sharp_A \neq [\![\mathtt{c}_2]\!]^\sharp_A$, and A does not coincide with the identical abstraction, i.e., $[\![\cdot]\!]^\sharp_A \neq [\![\cdot]\!]$.

Making abstract interpretation complete is the holy grail in program analysis and verification by approximate methods [23]. Since the very first non-constructive solution to the problem of making abstract interpretations complete given in [21], a constructive optimal abstraction refinement has been introduced in [25] and finalized in [26], which had several applications in program analysis [11, 24, 27], model checking [20,22,34,36,37,38], language-based security [17,32], code protection [14, 15, 18], and program semantics [19]. The interested reader can refer to [35] to realize how *completeness arises everywhere* in programming languages. The key result in [26] is the notion of most abstract completeness refinement, called *complete shell*, of an abstract domain A. This operation, which belongs to the family of abstract domain refinements [12], always exists for any Scott-continuous predicate transformer — therefore for all computable transfer functions — and it can be constructively defined as the solution of a recursive abstract domain equation. Although extremely powerful, this notion has an intrinsic global flavour: 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, this complete shell yields an abstract domain that is often way too fine grain to work for a program or a set of programs, possibly blowing up to the whole concrete domain in many cases.

On the side of program logics, several over-approximation techniques are known since the pioneering works by Floyd and Hoare [13, 29]. In classical Hoare correctness logic, a triple $\{p\}\ \mathtt{c}\ \{q\}$ asserts that the postcondition q over-approximates the states reachable starting from states satisfying the precondition p when executing the command $\mathtt{c}$. Letting $[\![\mathtt{c}]\!]$ denote the collecting semantics of $\mathtt{c}$, this means that $[\![\mathtt{c}]\!]p \leq q$ holds. In an inductive setting, like forward program analysis, over-approximations are useful for proving correctness w.r.t. a specification *spec*: in fact, $q \leq \mathit{spec}$ implies $[\![\mathtt{c}]\!]p \leq \mathit{spec}$. Likewise abstract interpretation, Hoare triples may exhibit false positives, meaning that from $q \not\leq spec$ we cannot conclude that elements of $q \smallsetminus spec$ are real violations to the correctness specification *spec*.

In a dual setting, a theory of *program incorrectness* has been recently investigated by O'Hearn [33]. A program specification $[p]\ \mathsf{c}\ [q]$ in O'Hearn incorrectness logic states that the postcondition q is an under-approximation of the states reachable from some states satisfying p, i.e. $q \leq [\![\mathsf{c}]\!]p$ must hold, therefore exposing only real program bugs: if $q \not\leq spec$, then $[\![\mathsf{c}]\!]p \not\leq spec$. Dually to Hoare correctness logic, O'Hearn incorrectness logic cannot be used to prove program correctness because it may exhibit false negatives, in the sense that from $q \leq spec$ we cannot conclude that $[\![\mathsf{c}]\!]p \leq spec$ holds.

2 Proving Completeness

The key fact that complete abstract functions compose [9], makes it possible to use structural induction to check whether an abstract interpretation is complete for a program. This idea was originally applied in [16] to define a very first sound proof system for checking whether an abstract domain is complete for a given program. Given an arbitrary abstract domain $A \in \mathrm{Abs}(C)$, the logical proof system $\vdash_A$ in Figure 1 is such that if $\vdash_A \mathsf{c}$ can be proved then the abstract interpreter on the domain A is complete for the program c. Herem c is a program in a simple imperative language and a proof obligation $\mathbb{C}^A(f)$ is set as the base inductive case, meaning that the abstract domain A is complete for the predicate transformer f.

$$\frac{}{\vdash_A \mathbf{skip}}\ [\mathbf{skip}] \qquad \frac{\vdash_A \mathsf{c}_1 \quad \vdash_A \mathsf{c}_2}{\vdash_A \mathsf{c}_1 ; \mathsf{c}_2}\ [\mathbf{seq}] \qquad \frac{\vdash_A \mathsf{c} \quad \mathbb{C}^A(b) \quad \mathbb{C}^A(\neg b)}{\vdash_A \mathbf{if}\ b\ \mathbf{then}\ \mathsf{c}}\ [\mathbf{if}]$$

$$\frac{\mathbb{C}^A(x := a)}{\vdash_A x := a}\ [:=] \qquad \frac{\vdash_A \mathsf{c} \quad \mathbb{C}^A(b) \quad \mathbb{C}^A(\neg b)}{\vdash_A \mathbf{while}\ b\ \mathbf{do}\ \mathsf{c}\ \mathbf{ew}}\ [\mathbf{while}]$$

Fig. 1 The core proof system $\vdash_A$ defined in [16].

This proof system $\vdash_A$ is very simple and basically postpones the checking of completeness for a given program c, no matter how complex c can be, to the checking of completeness of the basic predicate transformers occurring in c, namely those associated with boolean guards, i.e. $\mathbb{C}^A(b)$ and $\mathbb{C}^A(\neg b)$, and with variable assignments, i.e. $\mathbb{C}^A(x := a)$.

All the aforementioned methods to make an abstract interpreter complete or to check its completeness, as in the case of the proof system in Figure 1, consider completeness as a property that has to hold *for all* possible inputs. This, as we remarked above, is extremely hard to achieve. Some recent works put forward the idea of weakening the notion of completeness with the goal of making it more easily attainable. The basic idea is not to require completeness of an abstract interpreter for all possible input properties, hence on the whole abstract domain, but rather to demand completeness along only the sequence of store properties computed by the program on a specific computation path. This approach makes completeness a "local" notion, hence the name of *local completeness* introduced in [2]. An abstract domain $A \in \mathrm{Abs}(C)$ is *locally complete* for a predicate transformer f on a precondition $p \in C$ if the following condition holds:

$$\mathbb{C}^A_p(f) \overset{\triangle}{\Leftrightarrow} Af(p) = AfA(p).$$

An abstract interpreter is therefore locally complete relatively to a given precondition p when no false positives for the verification of a postcondition are produced by running the abstract interpreter with p as input. The soundness of the abstraction guarantees that $[\![\mathsf{c}]\!]p \le A([\![\mathsf{c}]\!]p) \le A([\![\mathsf{c}]\!]A(p)) \le [\![\mathsf{c}]\!]^\sharp_A A(p)$ hold, where $A([\![\mathsf{c}]\!]A(p))$ is the best correct approximating semantics of the program c on the input precondition p. The novel objective here is to define a program logic where an available under-approximation q of the postcondition $[\![\mathsf{c}]\!]p$ is tied to the over-approximation of q in the abstraction A. Hence, the aim is to design a program logic where a provable triple $\vdash_A [p]\ \mathsf{c}\ [q]$ guarantees that

$$q \le [\![\mathsf{c}]\!]p \le A([\![\mathsf{c}]\!]p) = A(q) \qquad (1)$$

holds. To provide these guarantees, this proof system requires that any computational step of the abstract interpreter is locally complete on the approximated input. As an illustrative example, consider the program for computing the absolute value of integer variables:

$$\mathsf{Abs}(x) \triangleq \textbf{if } (x \ge 0) \textbf{ then } ; \textbf{ if } x < 0 \textbf{ then } x := -x$$

The ubiquitous interval abstraction Int [8,23] approximates any property $s \in \wp(\mathbb{Z})$ of the integer values that the variable x may assume by the least interval $\mathsf{Int}(s) = [a, b]$ over-approximating S, i.e. such that $S \subseteq [a, b]$, where $a \le b$, $a \in \mathbb{Z} \cup \{-\infty\}$ and $b \in \mathbb{Z} \cup \{+\infty\}$. Let us assume that the possible inputs for $\mathsf{Abs}(x)$ range just in the set $i = \{x \mid x \text{ is odd}\}$. While the interval approximation of the outputs $\mathsf{Abs}(i)$ is $\mathsf{Int}(\mathsf{Abs}(i)) = [1, +\infty]$, showing that 0 is not a possible result, it turns out that the best correct approximation in Int of the concrete semantics is less precise, because it also includes 0: in fact, $\mathsf{Int}(\mathsf{Abs}(\mathsf{Int}(i))) = [0, +\infty]$. Technically, this means that Int is incomplete for Abs on input i. This can spawn a problem in program verification: for instance, if the result is used as divisor in an integer division, the abstract interval analysis would raise a "division-by-0" false alarm. However, for different sets of input, we can derive more precise results. For example, if we consider $j = \{x < 0 \mid x \text{ is odd}\}$, we have that $\mathsf{Int}(\mathsf{Abs}(\mathsf{Int}(j))) = [1, +\infty] = \mathsf{Int}(\mathsf{Abs}(j))$ holds. This entails that the abstract domain Int is locally complete for $\mathsf{Abs}(x)$ on input j but not on input i. Proving local completeness means, e.g., to prove the following two triples:

$$\vdash_{\mathsf{Int}} [\{-7, -5, -3, -1\}]\ \mathsf{Abs}(x)\ [\{1, 7\}] \quad \text{and} \quad \vdash_{\mathsf{Int}} [\{-1, 0, 7\}]\ \mathsf{Abs}(x)\ [\{0, 7\}]$$

but not the triple

$$\vdash_{\mathsf{Int}} [\{-3,-1,5,7\}]\ \mathsf{Abs}(x)\ [\{1,7\}]$$

because some local completeness requirements are not met in this latter case, even if the property $\{1,7\} \subseteq [\![\mathsf{Abs}(x)]\!]\{-3,-1,5,7\} = \mathsf{Int}(\{1,7\})$ is indeed valid. Notably, the triple $\vdash_{\mathsf{Int}} [\{-7,-5,-3,-1\}]\ \mathsf{Abs}(x)\ [\{1,7\}]$ can be used to prove correctness w.r.t. the specification $x > 0$, while the triple $\vdash_{\mathsf{Int}} [\{-1,0,7]\ \mathsf{Abs}(x)\ [\{0,7\}]$ exhibits a true counterexample for the same property.

Proving local completeness raises two main challenges. The first is obvious: the proof obligations of the basic program components (boolean guards and assignments in a simple imperative language) depend upon the input preconditions. The second is more interesting and sheds a deeper insight in the way approximation and computation interplay: the locality assumption closely tights completeness to the proof system, which is inductively defined on program's syntax, thus going well beyond the basic program logic in Figure 1 for "global" completeness [16]. In particular, a logic for locally complete abstract interpretations has to combine the standard over-approximation of abstract interpretation with under-approximations used in incorrectness logic, to encompass over- and under-approximating program reasoning in a unified program logic.

3 LCL: Local Completeness Logic

We consider a simple language Reg of *regular commands* that covers imperative languages as well as other programming paradigms [30, 31, 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 and can be instantiated, e.g., with (deterministic or nondeterministic or parallel) assignments, boolean guards, error generation primitives, etc. For simplicity, we consider integer variables $x \in \mathrm{Var}$ and let a and b range over, resp., arithmetic and Boolean expressions, so that:

$$\mathsf{Exp} \ni \mathsf{e} ::= \textbf{skip} \mid x := \mathsf{a} \mid \mathsf{b}?$$

The term $\mathsf{r}_1;\mathsf{r}_2$ represents sequential composition, $\mathsf{r}_1 \oplus \mathsf{r}_2$ a choice that can behave as either r_1 or r_2, and r^* is the Kleene iteration, where r can be executed 0 or any bounded number of times. Moreover, regular commands represent in a compact way the structure of control-flow graphs (CFGs) of imperative programs, and standard while-based languages, such as Imp in [41], can be retrieved by the following standard encodings (cf. [30, Section 2.2]):

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

The concrete semantics $[\![\cdot]\!] : \mathsf{Reg} \to C \to C$ is inductively defined for any c ranging in a concrete domain C by assuming that the basic commands have a semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ defined on C such that $(\!|\mathsf{e}|\!)$ is an additive function for any $\mathsf{e} \in \mathsf{Exp}$:

$$\begin{array}{ll} [\![\mathsf{e}]\!]c \triangleq (\!|\mathsf{e}|\!)c & [\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]c \triangleq [\![\mathsf{r}_1]\!]c \vee_C [\![\mathsf{r}_2]\!]c \\ [\![\mathsf{r}_1;\mathsf{r}_2]\!]c \triangleq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]c) & [\![\mathsf{r}^*]\!]c \triangleq \bigvee_C \{[\![\mathsf{r}]\!]^n c \mid n \in \mathbb{N}\} \end{array}$$

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

$$\begin{aligned} (\!|\mathbf{skip}|\!)s &\triangleq s \\ (\!|x := \mathsf{a}|\!)s &\triangleq \{\sigma[x \mapsto \{\!|\mathsf{a}|\!\}\sigma] \mid \sigma \in s\} \\ (\!|\mathsf{b}?|\!)s &\triangleq \{\sigma \in s \mid \{\!|\mathsf{b}|\!\}\sigma = \mathbf{tt}\} \end{aligned}$$

where $\{\!|\mathsf{a}|\!\} : \Sigma \to \mathbb{Z}$ and $\{\!|\mathsf{b}|\!\} : \Sigma \to \{\mathbf{tt}, \mathbf{ff}\}$ are defined as expected.

The abstract semantics $[\![\cdot]\!]^\sharp_A : \mathsf{Reg} \to A \to A$ on an abstraction $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is defined similarly, by structural induction, as follows: for any $a \in A$,

$$\begin{array}{ll} [\![\mathsf{e}]\!]^\sharp_A a \triangleq [\![\mathsf{e}]\!]^A a = A((\!|\mathsf{e}|\!)a) & [\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^\sharp_A a \triangleq [\![\mathsf{r}_1]\!]^\sharp_A a \vee_A [\![\mathsf{r}_2]\!]^\sharp_A a \\ [\![\mathsf{r}_1;\mathsf{r}_2]\!]^\sharp_A a \triangleq [\![\mathsf{r}_2]\!]^\sharp_A([\![\mathsf{r}_1]\!]^\sharp_A a) & [\![\mathsf{r}^*]\!]^\sharp_A a \triangleq \bigvee_A \{([\![\mathsf{r}]\!]^\sharp_A)^n a \mid n \in \mathbb{N}\} \end{array} \tag{2}$$

It turns out that the above abstract semantics is monotonic and sound, i.e., $A[\![\mathsf{r}]\!] \leq [\![\mathsf{r}]\!]^\sharp_A A$ holds. Note that the abstract semantics of a basic expression e is its bca $[\![\mathsf{e}]\!]^A$ on A. This definition (2) agrees with the standard compositional definition by structural induction of abstract semantics used in abstract interpretation [7, 40]. Best correct abstractions are preserved by choice commands, i.e., $[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^A a = [\![\mathsf{r}_1]\!]^A a \vee_A [\![\mathsf{r}_2]\!]^A a$, but generally not by sequential composition and Kleene iteration: for example, $[\![\mathsf{r}_2]\!]^A \circ [\![\mathsf{r}_1]\!]^A$ is not guaranteed to be the bca of $[\![\mathsf{r}_1;\mathsf{r}_2]\!]$.

Local completeness enjoys an "abstract convexity" property, that is, local completeness on a precondition p implies local completeness on any assertion s in between p and its abstraction $A(p)$. It is this key observation that led us to the design of LCL_A, the Local Completeness Logic on A. LCL_A combines over- and under-approximation of program properties: in LCL_A we can prove triples $\vdash_A [p]\ \mathsf{r}\ [q]$ ensuring that:

(i) q is an under-approximation of the concrete semantics $[\![\mathsf{r}]\!]p$, i.e., $q \leq [\![\mathsf{r}]\!]p$;
(ii) q and $[\![\mathsf{r}]\!]P$ have the same over-approximation in A, i.e., $A([\![\mathsf{r}]\!]p) = A(q)$;
(iii) A is locally complete for $[\![\mathsf{r}]\!]$ on input p, i.e., $[\![\mathsf{r}]\!]^\sharp_A \alpha(P) = \alpha(Q)$.

Points (i–iii) guarantee that, given a specification *spec* expressible in A, any provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ either proves correctness of r with respect to *spec* or exposes an alert in $q \smallsetminus spec$. This alert, in turn, must correspond to a true alert because q

is an under-approximation of the concrete semantics $[\![\mathsf{r}]\!]p$. The full proof system is here omitted and can be found in [2]. Here, we focus on the two most relevant rules: (relax), that allows us to generalize a proof, and (transfer), that checks for local completeness of basic transfer functions.

The main idea of provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ in LCL_A is to constrain the under-approximation q as postcondition in such a way that it has the same abstraction of the concrete semantics $[\![\mathsf{r}]\!]p$. The rule (relax) allows us to weaken the premises and strengthen the conclusions of the deductions in this proof system:

$$\frac{d \leq p \leq A(d) \quad \vdash_A [d]\ \mathsf{r}\ [s] \quad q \leq s \leq A(q)}{\vdash_A [p]\ \mathsf{r}\ [q]}\ (\mathsf{relax})$$

Since the proof system infers a s that has the same abstraction of the concrete semantics $[\![\mathsf{r}]\!]p$, by the abstract convexity property mentioned above, we have that local completeness of $[\![\mathsf{r}]\!]$ on the *under-approximation* d is enough to prove local completeness on p. The conclusion s can then be strengthened to any under-approximation q preserving the abstraction (we have $A(s) = A(q)$).

All local completeness proof obligations are introduced by the rule (transfer), in correspondence of each basic transfer function e, which is nothing else than the local completeness version of the proof obligations $\mathbb{C}^A(b)$, $\mathbb{C}^A(\neg b)$ and $\mathbb{C}^A(x := a)$ in the proof system for global completeness in Figure 1:

$$\frac{\mathbb{C}^A_p([\![\mathsf{e}]\!])}{\vdash_A [p]\ \mathsf{e}\ [[\![\mathsf{e}]\!]p]}\ (\mathsf{transfer})$$

The main consequence of this construction is that, given a specification *spec* expressible in the abstract domain A, a provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ can determine both correctness and incorrectness of the program r, that is,

$$[\![\mathsf{r}]\!]p \leq spec \iff q \leq spec \qquad (3)$$

holds. In equivalent terms:

- If $q \leq spec$, then we have also $[\![\mathsf{r}]\!]p \leq spec$, so that the program is correct with respect to *spec*.
- If $q \not\leq spec$, then $[\![\mathsf{r}]\!]p \not\leq spec$ also holds, thus meaning that $[\![\mathsf{r}]\!]p \smallsetminus spec$ is not empty, and therefore includes a true alarm for the program. Moreover, because $q \leq [\![\mathsf{r}]\!]p$, we have that $q \smallsetminus spec \leq [\![\mathsf{r}]\!]p \smallsetminus spec$. This means that already q is able to pinpoint some issues.

To illustrate the approach, we consider the following example (discussed in [2] where the reader can find all the details of the derivation). Let us consider the command $\mathsf{r} \triangleq (\mathsf{r}_1 \oplus \mathsf{r}_2)^*$ where

$$\mathsf{r}_1 \triangleq (0 < x?; x := x - 1)$$
$$\mathsf{r}_2 \triangleq (x < 1000?; x := x + 1)$$

The concrete domain is $C = \wp(\mathbb{Z})$ and the abstract domain is $A = \mathsf{Int}$. Using the pre-condition $p \triangleq \{1, 999\}$, we can derive the triple $\vdash_{\mathsf{Int}} [p]\ \mathsf{r}\ [\{0, 2, 1000\}]$. This allows us to prove, for instance, that $spec_1 = (x \leq 1000)$ is met, and exhibits two true violations of $spec_2 = (100 \leq x)$, namely 0 and 2.

4 Concluding Remarks

In this extended abstract, we presented the genesis and the main ideas behind the local completeness logic LCL_A that we introduced in [2]. Local completeness represents a notable weakening of the notion of completeness originally introduced in [9] and later studied in [26]. The key point in using LCL_A is that the proof obligations ensuring local completeness for the basic expressions occurring in a program have to be guaranteed. Of course, this is an issue when the abstract domain A is not expressive enough to entail these proof obligations. This problem has been settled in [3], where a strategy has been proposed to *repair* the abstract domain when a local completeness proof obligation fails. The goal here is to refine the abstract domain by adding a new abstract element, which must be as abstract as possible, such that the proof obligation is satisfied in the new refined domain. This strategy is called *forward repair* since it repairs the domain A along a derivation attempt as soon as a proof obligation of local completeness is found. After one such repair step, a new derivation must be started in the refined domain A', so that in general the process is iterative and is not guaranteed to terminate, similarly to what happens program verification based on counterexample-guided abstract refinement (CEGAR) [5, 6]. In fact, the repair A' of a given proof obligation may compromise the satisfaction of previously encountered proof obligations that were valid in A but maybe not in A'. A *backward repair* strategy has been therefore designed to overcome this limitation. Because of the analogy with partition refinement, in a sentence we may argue that *abstract interpretation repair is for abstract interpretation what CEGAR is for abstract model checking*.

The general goal is to make LCL_A an effective method for program analysis by securing a good trade-off between precision and efficiency. In particular, the overall objective is to reduce/minimize the presence of false positives/negatives. In our proof system in order to guarantee that the property (1) holds, the stronger requirement

$$q \leq [\![\mathsf{c}]\!]p \leq A([\![\mathsf{c}]\!]p) = A(q) = [\![\mathsf{c}]\!]^{\sharp}_{A} A(p).$$

is enforced, that corresponds to ask for the actual abstract interpreter $[\![\mathsf{c}]\!]^{\sharp}_{A}$ to be locally complete. This might be not strictly necessary, as the condition can be weakened to require that the best correct abstraction is locally complete. Unfortunately, computing the best correct abstraction $[\![\mathsf{c}]\!]^{A}$ is not always possible, but we are investigating sufficient conditions to assure local completeness of best correct abstractions. The idea here is to exploit different refinements of the abstract domain for different parts of the derivations, so that precision can be improved whenever needed as far as the result can be transferred to the original domain. Furthermore, in the vein of the so-called core domain simplification for global completeness introduced in [26], we plan to investigate the chance to simplify the domain rather than refining it for ensuring that local completeness holds on a given input.

Finally, let us mention that different types of weakening of the notion of completeness are possible. A notion of partial completeness has been introduced in [4], meaning that (local) completeness holds up to some measurable error $\varepsilon \geq 0$. [4] studied a quantitative proof system which allows us to measure the imprecision of an abstract interpretation and can be used to estimate an upper bound on the error accumulated by the abstract interpreter during a program analysis. This quantitative framework is general enough to be instantiated to most known metrics for abstract domains.

Acknowledgements 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* unrestricted gift.

References

1. Bruni, R., Giacobazzi, R., Gori, R., Garcia-Contreras, I., Pavlovic, D.: Abstract extensionality: on the properties of incomplete abstract interpretations. Proc. ACM Program. Lang. **4**(POPL), 28:1–28:28 (2020). URL https://doi.org/10.1145/3371096
2. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: Proceedings of LICS 2021, 36th Annual ACM/IEEE Symposium on Logic in Computer Science, pp. 1–13. IEEE (2021). Distinguished paper
3. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: Abstract interpretation repair. In: R. Jhala, I. Dillig (eds.) PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022, pp. 426–441. ACM (2022). URL https://doi.org/10.1145/3519939.3523453
4. Campion, M., Preda, M.D., Giacobazzi, R.: Partial (in)completeness in abstract interpretation: limiting the imprecision in program analysis. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). URL https://doi.org/10.1145/3498721
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of CAV 2000, 12th International Conference on Computer Aided Verification, *Lecture Notes in Computer Science*, vol. 1855, pp. 154–169. Springer-Verlag (2000). URL https://doi.org/10.1007/10722167_15
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). URL https://doi.org/10.1145/876638.876643
7. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of ACM POPL'77, pp. 238–252. ACM (1977). URL https://doi.org/10.1145/512950.512973
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of ACM POPL'79, pp. 269–282. ACM (1979). URL https://doi.org/10.1145/567752.567778
10. Cousot, P., Giacobazzi, R., Ranzato, F.: Program analysis is harder than verification: A computability perspective. In: H. Chockler, G. Weissenbacher (eds.) Computer Aided

Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II, *Lecture Notes in Computer Science*, vol. 10982, pp. 75–95. Springer (2018). URL `https://doi.org/10.1007/978-3-319-96142-2\_8`

11. Dalla Preda, M., Giacobazzi, R., Mastroeni, I.: Completeness in approximated transductions. In: Static Analysis, 23rd International Symposium, SAS 2016., *LNCS*, vol. 9837, pp. 126–146 (2016)
12. Filé, G., Giacobazzi, R., Ranzato, F.: A unifying view of abstract domain design. ACM Comput. Surv. **28**(2), 333–336 (1996). URL `https://doi.org/10.1145/234528.234742`
13. Floyd, R.W.: Assigning meanings to programs. Proceedings of Symposium on Applied Mathematics **19**, 19–32 (1967)
14. Giacobazzi, R.: Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In: Proc. of the 6th IEEE Int. Conferences on Software Engineering and Formal Methods (SEFM '08), pp. 7–20. IEEE Press (2008)
15. Giacobazzi, R., Jones, N.D., Mastroeni, I.: Obfuscation by partial evaluation of distorted interpreters. In: Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12), pp. 63–72. ACM Press (2012)
16. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. In: Proceedings of POPL 2015, 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 261–273. ACM (2015). URL `http://doi.acm.org/10.1145/2676726.2676987`
17. Giacobazzi, R., Mastroeni, I.: Adjoining classified and unclassified information by abstract interpretation. Journal of Computer Security **18**(5), 751–797 (2010)
18. Giacobazzi, R., Mastroeni, I.: Making abstract interpretation incomplete: Modeling the potency of obfuscation. In: A. Miné, D. Schmidt (eds.) Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7460, pp. 129–145. Springer (2012). URL `https://doi.org/10.1007/978-3-642-33125-1\_11`
19. Giacobazzi, R., Mastroeni, I.: Making abstract models complete. Mathematical Structures in Computer Science **26**(4), 658–701 (2016). URL `https://doi.org/10.1017/S0960129514000358`
20. Giacobazzi, R., Quintarelli, E.: 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, pp. 356–373. Springer (2001). URL `https://doi.org/10.1007/3-540-47764-0\_20`
21. Giacobazzi, R., Ranzato, F.: Completeness in abstract interpretation: A domain perspective. In: M. Johnson (ed.) Proc. of the 6th Internat. Conf. on Algebraic Methodology and Software Technology (*AMAST '97*), *Lecture Notes in Computer Science*, vol. 1349, pp. 231–245. Springer-Verlag (1997)
22. Giacobazzi, R., Ranzato, F.: Incompleteness of states w.r.t. traces in model checking. Inf. Comput. **204**(3), 376–407 (2006). URL `https://doi.org/10.1016/j.ic.2006.01.001`
23. Giacobazzi, R., Ranzato, F.: History of abstract interpretation. IEEE Ann. Hist. Comput. **44**(2), 33–43 (2022)
24. Giacobazzi, R., Ranzato, F., Scozzari, F.: Building complete abstract interpretations in a linear logic-based setting. In: G. Levi (ed.) Static Analysis, Proceedings of the Fifth International Static Analysis Symposium SAS 98, *Lecture Notes in Computer Science*, vol. 1503, pp. 215–229. Springer-Verlag (1998)
25. Giacobazzi, R., Ranzato, F., Scozzari, F.: Complete abstract interpretations made constructive. In: L. Brim, J. Gruska, J. Zlatuška (eds.) Proc. of the 23rd Internat. Symp. on Mathematical Foundations of Computer Science (*MFCS '98*), *Lecture Notes in Computer Science*, vol. 1450, pp. 366–377. Springer-Verlag (1998)
26. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretation complete. Journal of the ACM **47**(2), 361–416 (2000). URL `https://doi.org/10.1145/333979.333989`

27. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract domains condensing. ACM Transactions on Computational Logic **6**(1), 33–60 (2005). URL `https://doi.org/10.1145/1042038.1042040`
28. Hoare, C.: An axiomatic basis for computer programming. Comm. of The ACM **12**(10), 576–580 (1969)
29. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
30. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997). DOI 10.1145/256167.256195. URL `https://doi.org/10.1145/256167.256195`
31. Kozen, D.: On Hoare logic and Kleene algebra with tests. ACM Trans. Comput. Logic **1**(1), 60–76 (2000). DOI 10.1145/343369.343378. URL `https://doi.org/10.1145/343369.343378`
32. Mastroeni, I., Banerjee, A.: Modelling declassification policies using abstract domain completeness. Mathematical Structures in Computer Science **21**(6), 1253–1299 (2011). URL `https://doi.org/10.1017/S096012951100020X`
33. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL), 10:1–10:32 (2020). URL `https://doi.org/10.1145/3371078`
34. Ranzato, F.: On the completeness of model checking. In: D. Sands (ed.) Proc. of the European Symp. on Programming (ESOP'01), *Lecture Notes in Computer Science*, vol. 2028, pp. 137–154. Springer-Verlag (2001)
35. Ranzato, F.: Complete abstractions everywhere. In: Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013, *Lecture Notes in Computer Science*, vol. 7737, pp. 15–26. Springer (2013)
36. Ranzato, F., Tapparo, F.: Strong preservation as completeness in abstract interpretation. In: Proceedings of ESOP 2004, 13th European Symposium on Programming, *Lecture Notes in Computer Science*, vol. 2986, pp. 18–32. Springer (2004). URL `https://doi.org/10.1007/978-3-540-24725-8\_3`
37. Ranzato, F., Tapparo, F.: An abstract interpretation-based refinement algorithm for strong preservation. In: N. Halbwachs, L. Zuck (eds.) Proceedings of TACAS 2005, Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 3440, pp. 140–156. Springer-Verlag (2005)
38. Ranzato, F., Tapparo, F.: Generalized strong preservation by abstract interpretation. J. Log. Comput. **17**(1), 157–197 (2007). URL `https://doi.org/10.1093/logcom/exl035`
39. Rice, H.: Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. **74**, 358–366 (1953)
40. Rival, X., Yi, K.: Introduction to Static Analysis – An Abstract Interpretation Perspective. MIT Press (2020)
41. Winskel, G.: The Formal Semantics of Programming Languages: an Introduction. MIT press (1993)