A Logic for Locally Complete Abstract Interpretations

Roberto Bruni*, Roberto Giacobazzi‡, Roberta Gori*, Francesco Ranzato§
*University of Pisa, Italy
‡University of Verona, Italy
§University of Padova, Italy

In loving memory of Anna Maria De Paolis and Dina Gorini

***Abstract*—We introduce the notion of *local completeness* in abstract interpretation and define a logic for proving both the correctness and incorrectness of some program specification. Abstract interpretation is extensively used to design sound-by-construction program analyses that over-approximate program behaviours. Completeness of an abstract interpretation A for all possible programs and inputs would be an ideal situation for verifying correctness specifications, because the analysis can be done compositionally and no false alert will arise. Our first result shows that the class of programs whose abstract analysis on A is complete for all inputs has a severely limited expressiveness. A novel notion of *local completeness* weakens the above requirements by considering only some specific, rather than all, program inputs and thus finds wider applicability. In fact, our main contribution is the design of a proof system, parameterized by an abstraction A, that, for the first time, combines over- and under-approximations of program behaviours. Thanks to local completeness, in a provable triple $\vdash_A [P]\ \mathsf{c}\ [Q]$, the assertion Q is an under-approximation of the strongest post-condition $\mathsf{post}[\mathsf{c}](P)$ such that the abstractions in A of Q and $\mathsf{post}[\mathsf{c}](P)$ coincide. This means that Q is never too coarse, namely, under mild assumptions, *the abstract interpretation of* c *does not yield false alerts for the input* P *iff* Q *has no alert*. Thus, $\vdash_A [P]\ \mathsf{c}\ [Q]$ not only ensures that all the alerts raised in Q are true ones, but also that if Q does not raise alerts then c is correct.**

I. Introduction

Technology, you can't live without. But any coin has two sides and software failures are increasingly more frequent and their consequences are more disruptive in the Digital Age than ever before. Quoting Dijkstra's speech at the Turing Award lecture [11], *the only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness*. Since correctness proof attempts may fail even when the program is correct, also incorrectness proofs would be needed to catch actual bugs, because you can't fix what you can't see. Code-review processes and test-driven development are widely adopted best practices in software companies. Nevertheless, the problem is far from being solved and static reasoning should be extended to bug catching, as advocated by O'Hearn's incorrectness logic (IL) [24].

Static program analysis has been investigated and used for over half century and is a major methodology to help programmers and software engineers in producing reliable code [4], [12], [15], [18], [23], [27], [28]. Static analysis is based on symbolic reasoning techniques to prove program properties without running them. Given a program c and a correctness specification *Spec*, the aim of a static verification is either to prove that the behaviour of c satisfies *Spec* or to raise some alerts that point out which circumstances *may* cause a violation of *Spec*. The conditional is needed because, starting with the fundamental works by Hoare [18], program verifiers tend to over-approximate the program behaviour: this is an unavoidable consequence of the will to solve an otherwise undecidable analysis problem. As any alerting system, program analysis turns out to be *credible*, when few, ideally zero, false alerts are reported to the user [9]. The dual perspective has been recently tackled by incorrectness logic [24]: exploiting under-approximations, any violation exposed by the analysis is a true alert. This makes IL a credible support for code-review, but *Spec* may be violated even when no alert is reported.

Abstract interpretation [6]–[8] is a well-established framework for designing sound-by-construction over-approximations of the program behaviour. Given an abstraction A, instead of verifying whether the strongest post-condition $\mathsf{post}[\mathsf{c}](P)$ for a program c and a pre-condition P (also written $[\![\mathsf{c}]\!]P$) satisfies a correctness specification *Spec*, a (sound) abstract over-approximation $A(\mathsf{post}[\mathsf{c}](P))$ is considered. While it is obvious that if $A(\mathsf{post}[\mathsf{c}](P))$ satisfies *Spec* then the program is correct, it may happen that $A(\mathsf{post}[\mathsf{c}](P))$ does not satisfy *Spec* even if the program is correct, because A introduced false alerts. Once the specification *Spec* and its abstract approximation in A coincide, the ideal program analysis is achieved by assuring that a sound analysis is also *complete*, so that no false alert is ever raised.

Technically, in a domain A of abstract program stores, with abstraction and concretization maps α and γ resp., any store property P is, in general, over-approximated by $A(P) = \gamma\alpha(P) \supseteq P$. Assuming that *Spec* is expressible in A means that $Spec = A(Spec)$ holds. For instance, in the abstract domain of intervals Int (see Example III.5) the property $x \geq 0$ is expressible by the infinite interval $[0, +\infty]$. By contrast, $x \neq 0$ is not expressible in Int, since the least over-approximating interval is $\mathsf{Int}(x \neq 0) = \mathbb{Z} \supsetneq \mathbb{Z} \smallsetminus \{0\}$. An abstract semantics associates with each program c a computable function $\mathsf{post}_A[\mathsf{c}] : A \to A$ on the abstraction A (also written $[\![\mathsf{c}]\!]^\sharp_A$). By soundness of abstract interpretation, if $\gamma(\mathsf{post}_A[\mathsf{c}]\alpha(P)) \subseteq Spec$ then $\{P\}\ \mathsf{c}\ \{Spec\}$ is a valid Hoare triple. However, when $\gamma(\mathsf{post}_A[\mathsf{c}]\alpha(P)) \not\subseteq Spec$ we cannot conclude that $\{P\}\ \mathsf{c}\ \{Spec\}$ is not valid, because any witness in $\gamma(\mathsf{post}_A[\mathsf{c}]\alpha(P)) \smallsetminus Spec$ is just a *potentially false alert*.

978-1-6654-4895-6/21/$31.00

Complete abstract interpretations, instead, do not raise false alerts for expressible specifications *Spec*, namely completeness guarantees that $\{P\}\ \mathsf{c}\ \{Spec\}$ is valid iff $\gamma(\mathsf{post}_A[\mathsf{c}]\alpha(P)) \subseteq Spec$ holds. Thus, any (complete) abstract analysis of *Spec* is the same as verifying *Spec* concretely.

The Problem: According to a well-established definition [6], [7], completeness in abstract interpretation is a *global notion*, meaning that it involves *all* possible pre-conditions. More precisely, A is complete for a program c when:

$$\text{for all } P,\ \mathsf{post}_A[\mathsf{c}]\alpha(P) = \alpha(\mathsf{post}[\mathsf{c}]P) \text{ holds.}$$

As pre-conditions P are universally quantified, completeness is very hard to achieve in practice, even for very simple programs. Moreover, while completeness is expressed extensionally, namely referring to the semantics $\mathsf{post}[\mathsf{c}]$ of the whole program c, any effective abstract interpretation $\mathsf{post}_A[\mathsf{c}]$ is intensional, meaning that it needs to analyze the program c inductively on its syntax. Therefore, a complete and compositional abstract interpretation is even harder to design [3].

The case of Boolean guards occurring in programs is particularly challenging, because they are rarely complete. Letting Σ be the set of (concrete) stores, completeness of a guard $\mathsf{b?}$, viewed as predicate transformer $[\![\mathsf{b?}]\!] : \wp(\Sigma) \to \wp(\Sigma)$, in an abstraction A can be reduced to check if for any input P,

$$\gamma\alpha([\![\mathsf{b?}]\!]P) = \gamma\alpha([\![\mathsf{b?}]\!]\gamma\alpha(P)) \tag{1}$$

more concisely written as $A([\![\mathsf{b?}]\!]P) = A([\![\mathsf{b?}]\!]A(P))$. Because both branches of a conditional or loop statement must be taken into account, the same condition applies to the negative test $\neg\mathsf{b?}$. For instance, the interval domain Int is complete for a simple rectifier program, known as ReLU in (deep) neural networks, although its Boolean guards are not complete:

$$\mathsf{ReLU}(x) \triangleq \textbf{if } (x < 0) \textbf{ then } x := 0 \textbf{ else skip}$$

Int is trivially complete for both $x := 0$ and **skip**, the guards $x < 0?$ and $x \geq 0?$ are both expressible in Int as, resp., $x \in [-\infty, -1]$ and $x \in [0, \infty]$, but the lack of completeness of the abstract interpretation of the two guards prevents us to *inductively prove completeness* for ReLU. For example, when $P \equiv x \in \{-1, 1\}$, we have that $\mathsf{Int}([\![x \geq 0?]\!]P) = \mathsf{Int}(\{1\}) = [1, 1]$, but $\mathsf{Int}([\![x \geq 0?]\!]\mathsf{Int}(P)) = \mathsf{Int}([\![x \geq 0?]\!][-1, 1]) = [0, 1]$, so that Int is not complete for $x \geq 0?$ It is precisely the inductive reasoning, necessary to design a generic abstract interpreter for a programming language, that hampers recognizing the completeness of ReLU in Int.

Main Results: In Section III, we provide some necessary and sufficient conditions that guarantee completeness of Boolean guards on an abstract domain A. This condition requires that $\mathsf{b?}$ and $\neg\mathsf{b?}$ are both expressible in the abstract domain and the same has to apply to the union of any two concretizations of abstract points below $\alpha([\![\mathsf{b?}]\!])$ and $\alpha([\![\neg\mathsf{b?}]\!])$. This requirement is very strong: for example, this condition allows us to prove that any guard on the interval domain Int is incomplete (cf. Example III.5).

Since completeness is sporadic, in Section IV we investigate *locally complete abstract interpretations*, a natural weakening of completeness. Instead of requiring completeness on all possible inputs, a locally complete abstract interpretation is complete just *locally* to some given set of possible inputs. In the case of a guard $\mathsf{b?}$ with input P, local completeness amounts to check that condition (1) holds *for that particular* P. For example, any guard is trivially locally complete w.r.t. any input P that is expressible in the abstract domain. In the latter case, for the above program ReLU it is easy to check that when the input is any interval or any set of integers all having the same sign, e.g., either all nonnegative or all negative, then both guards $x < 0?$ and $x \geq 0?$ turn out to be locally complete and therefore we can inductively conclude that ReLU is locally complete with respect to any such input.

It turns out that local completeness allows us to prove the absence of false alerts for programs which are globally incomplete on a given abstract domain. As a simple example, consider the program Abs for computing the absolute value:

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

Abs is globally incomplete on the abstract domain Int. For instance, with input $P \equiv x \in \{-7, 7\}$, $\mathsf{Int}(\mathsf{post}[\mathsf{Abs}]P) = [7, 7]$ while $\mathsf{Int}(\mathsf{post}[\mathsf{Abs}]\mathsf{Int}(P)) = \mathsf{Int}(\mathsf{post}[\mathsf{Abs}][-7, 7]) = [0, 7]$. Therefore, even if the pre-condition P does not include zero, an interval analysis of Abs produce a false alert when checking the specification $Spec = x > 0$. As above for ReLU, this is not the case when the inputs have all the same sign. Instead, a (true) alert for *Spec* can be raised only if the set of input values truly contains zero.

Our main contribution is presented in Section V: a logical proof system $\vdash_A$ for locally complete abstract computations parameterized by an abstraction A, called LCL_A (Local Completeness Logic on A). The statements are Hoare-like triples $\vdash_A [P]\ \mathsf{c}\ [Q]$ asserting that:

(i) Q is an *under-approximation* of $\mathsf{post}[\mathsf{c}]P$;
(ii) $\mathsf{post}[\mathsf{c}]$ is *locally complete* for input P on A;
(iii) Q and $\mathsf{post}[\mathsf{c}]P$ have the same *over-approximation* in A.

These properties of any provable triple $\vdash_A [P]\ \mathsf{c}\ [Q]$ allow us to distinguish between true and false alerts raised by an abstract interpretation $\mathsf{post}_A[\mathsf{c}]\alpha(P)$ for verifying any correctness specification *Spec* that is expressible in A.

The two distinctive rules of LCL_A are:

$$\frac{\mathsf{post}[\mathsf{e}] \text{ locally complete for } P}{\vdash_A [P]\ \mathsf{e}\ [\mathsf{post}[\mathsf{e}](P)]}\ (\mathsf{transfer})$$

$$\frac{P' \Rightarrow P \Rightarrow A(P') \quad \vdash_A [P']\ \mathsf{c}\ [Q'] \quad Q \Rightarrow Q' \Rightarrow A(Q)}{\vdash_A [P]\ \mathsf{c}\ [Q]}\ (\mathsf{relax})$$

The rule $(\mathsf{transfer})$ checks that a basic transfer expression e, such as a Boolean test $\mathsf{b?}$ or an assignment $x := \mathsf{a}$, is locally complete for P before providing the output of $\mathsf{post}[\mathsf{e}]$ on P as post-condition. The consequence rule (relax) is the key principle of LCL_A that combines an over- and under-approximating reasoning: (relax) allows us to infer a post-condition that defines an under-approximation Q of the exact behavior as well as a sound over-approximation $A(Q)$ of it, i.e., such that $Q \Rightarrow \mathsf{post}[\mathsf{c}](P) \Rightarrow A(\mathsf{post}[\mathsf{c}](P)) = A(Q)$ holds. Likewise

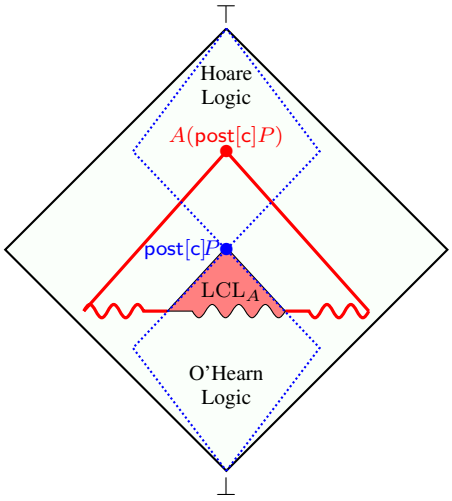


Fig. 1: Local Completeness Logic LCL_A.

the consequence rules of de Vries and Koutavas' reverse Hoare logic [10], O'Hearn's incorrectness logic [24] and Raad et al.'s incorrectness separation logic [25], the logical ordering between pre-conditions $P' \Rightarrow P$ and post-conditions $Q \Rightarrow Q'$ in the premises of (relax) is reversed w.r.t. the canonical consequence rule of Hoare logic and this is needed because our post-conditions Q are always under-approximations.

The ingenuity of (relax) is to constrain the under-approximating post-condition Q to have *the same abstraction* as the exact behaviour, which is needed for preserving local completeness. This twist is fundamental to guarantee the effectiveness of any triple derivable in LCL_A. Fig. 1 illustrates the interplay between LCL_A approximations (the red-filled region), Hoare logic for correctness (upper diamond, any over-approximation of $\mathsf{post}[\mathsf{c}](P)$), O'Hearn logic for incorrectness (lower diamond, any under-approximation of $\mathsf{post}[\mathsf{c}](P)$) and locally complete abstract interpretation A (red bordered region, any Q such that $A(Q) = A(\mathsf{post}[\mathsf{c}]P)$). In particular, it shows the effect of the (relax) rule that allows us to shrink the post-condition of our triples, up to some boundary that fringes the assertions of O'Hearn's IL. This is made without much loss of precision: under mild conditions on *Spec*, any approximation Q in the red-filled region guarantees that if $A(\mathsf{post}_A[\mathsf{c}]P)$ reports some alert, then Q contains a *true* alert and, viceversa, any alert in Q is a true one. The key point is that any triple $\vdash_A [P]\ \mathsf{c}\ [Q]$ of LCL_A provides an under-approximation Q which is not too coarse. More precisely, given a correctness specification *Spec* expressible in A, two scenarios can occur:

(a) *Spec* is satisfied: Abstract interpretation in A, as well as any triple $\vdash_A [P]\ \mathsf{c}\ [Q]$ derivable in LCL_A, allow to conclude that *Spec* holds. This is not true for Hoare logic: although $\{P\}\ \mathsf{c}\ \{Spec\}$ is a valid triple, it is also possible to derive some triple $\{P\}\ \mathsf{c}\ \{Q\}$ whose post-condition Q includes some false alert. Using incorrectness logic, no true alert can be found and no conclusion can be drawn about the validity of *Spec*.

(b) *Spec* is violated: Since *Spec* is expressible in A, then *any triple* $\vdash_A [P]\ \mathsf{c}\ [Q]$ of LCL_A will expose a true alert witnessing that *Spec* is violated. On the contrary, abstract interpretation in A, as well as Hoare logic, can also expose false alerts together with true ones. In IL, although it is possible to derive triples $\vdash_{\mathrm{IL}} [P]\ \mathsf{c}\ [Q]$ where Q exhibits some (true) alerts, other triples for P and c may have no alert at all, e.g., $\vdash_{\mathrm{IL}} [P]\ \mathsf{c}\ [\mathbf{ff}]$.

Theorem V.4 states that our proof system is sound for the previously emphasized properties (i–iii). To prove a result of logical completeness[1] of LCL_A for a program c, we add two more ingredients: (1) an infinitary rule (limit) for loops and (2) the assumption that all the basic transfer functions occurring in c are globally complete on A.

Finally, in Section VI, we show that IL coincides with LCL_A when the singleton trivial abstraction $A_{tr} \triangleq \lambda X.\Sigma$ is taken as abstract domain. The key observation is that A_{tr} is globally complete for every transfer function, and thus the premises of the rule (transfer) are always trivially satisfied. A_{tr} is also the only abstraction for which the proof system can be logical complete for a Turing complete language. Due to space constraints, all the proofs are omitted.

II. Background

The identity function on a set X is denoted id_X and the subscript is omitted when given by the context. If $f : X \to Y$ then f is overloaded to denote its (often called collecting) lifting $f : \wp(X) \to \wp(Y)$ to sets of values: $f(S) \triangleq \{f(x) \mid x \in S\}$. Given two functions $f : X \to Y$ and $g : Y \to Z$ we denote with $g \circ f$, or simply gf, their composition. For $f : X \to X$ and $n \in \mathbb{N}$ we let $f^n : X \to X$ be defined inductively as: $f^0 \triangleq id_X$ and $f^{n+1} \triangleq f \circ f^n$.

In ordered structures such as posets, CPOs, and complete lattices, we typically use $\leq$ to denote a partial order relation, $\vee$ for lub, $\wedge$ for glb, $\top$ and $\bot$ for, resp., greatest and least elements. We will use $\wp(X)$ to denote the powerset complete lattice over a set X ordered by inclusion, in which case the standard symbols $\subseteq$, $\cup$, etc., will be used to denote its order-theoretic structure. If C is a poset and $l, u \in C$ then $[l, u] \triangleq \{x \in C \mid l \leq x \leq u\}$ denotes the interval between l and u. Order-preserving functions between posets are called monotone. For $f, g : C \to C$ the notation $f \leq g$ means that for all $c \in C$, $f(c) \leq g(c)$. A function f between complete lattices is additive (resp. co-additive) when f preserves arbitrary lubs (resp. glbs). The least fixpoint of a function $f : C \to C$ on a poset C is denoted, when it exists, by $\mathrm{lfp}(f)$. If f is (Scott) continuous on a complete lattice, then $\mathrm{lfp}(f) = \vee_{n \in \mathbb{N}} f^n(\bot)$.

A. Abstract Interpretation

Abstract interpretation [6], [21], [27] is used to approximate a program semantics on a domain C on some abstraction A of C. Since different abstractions are possible, we use subscripts such as $\leq_A$ and $\vee_A$ to disambiguate the underlying carrier set A and omit the subscripts in the case of C. Given complete lattices C and A, a pair of functions $\alpha : C \to A$ and $\gamma : A \to C$ forms a Galois connection (GC, also called adjunction) when for all $c \in C$, $a \in A$, $\alpha(c) \leq_A a \Leftrightarrow c \leq \gamma(a)$ holds. In a GC C and A are called, resp., concrete and abstract domain

[1] The adjective *logical* serves to distinguish the usual notion of completeness for a proof system from the notion of completeness in abstract interpretation.

and α and γ are called, resp., abstraction and concretization maps. We only consider GCs such that $\alpha\gamma = id_A$, called Galois insertions (GIs), where α is onto and γ is 1-1. Let us recall that α is additive and γ is co-additive. We use $\mathrm{Abs}(C)$ to denote the class of abstract domains of C and write $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ to make explicit the maps $\langle \alpha, \gamma \rangle$. An abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is *strict* when $\gamma(\bot_A) = \bot$ and a concrete value $c \in C$ is *expressible* in A when $\gamma\alpha(c) = c$, while if $c < \gamma\alpha(c)$ holds then c is (strictly) *approximated* in A. Notice that $\gamma(A)$ and $C \smallsetminus \gamma(A)$ are the sets of concrete values which are, resp., expressible and approximated in A. An abstract domain A is *trivial* if it is isomorphic to the concrete domain C (i.e., $\gamma\alpha = id$) or it is a singleton (i.e., $\gamma\alpha = \lambda x.\top$).

1) Correctness: Given an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ and a concrete operation $f : C \to C$ (a generalization to n-ary functions of type $C^n \to C$ can be easily done pointwise), an abstract function $f^\sharp : A \to A$ is a *correct* (or *sound*) approximation of f when $\alpha f \leq_A f^\sharp \alpha$ holds. It is known that if $f^\sharp$ is a correct approximation of f then we also have fixpoint correctness when least fixpoints exist, i.e., $\alpha(\mathrm{lfp}(f)) \leq_A \mathrm{lfp}(f^\sharp)$ holds. The *best correct approximation* (bca) of f in A is the abstract function $f^A \triangleq \alpha \circ f \circ \gamma$. Any other abstract function $f^\sharp : A \to A$ is a correct approximation of f iff $f^A \leq_A f^\sharp$, i.e. $f^\sharp$ is less precise than f^A.

2) Completeness: The abstract function $f^\sharp$ is a *complete* approximation of f (or just complete) if $\alpha \circ f = f^\sharp \circ \alpha$ 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 when abstracting the concrete behaviour of f on the abstract domain A. In a complete approximation $f^\sharp$ the only loss of precision is due to the abstract domain and not to the abstract function itself. Analogously to soundness, completeness transfers to fixpoints, meaning that if $f^\sharp$ is complete for f then fixpoint completeness $\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^\sharp)$ holds. It turns out that there exists an abstract function $f^\sharp : A \to A$ such that completeness $\alpha \circ f = f^\sharp \circ \alpha$ holds iff $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$ iff $(\gamma\alpha) \circ f = (\gamma\alpha) \circ f \circ (\gamma\alpha) = \gamma \circ f^A \circ \alpha$. Thus, the possibility of defining a complete approximation $f^\sharp$ of f on some abstract domain $A \in \mathrm{Abs}(C)$ only depends upon the bca f^A of f in A, i.e., completeness is a property of the abstract domain only and any trivial abstract domain is complete for any f. In the following, we write both "A is complete for f" and "f is complete on A", and, when convenient, we use A in place of the function $\gamma\alpha : C \to C$ (which is an upper closure operator on C), as we did in the Introduction, e.g., for $\mathsf{Int}(\{-7, 7\})$. We write $\mathbb{C}^A(f)$ to denote that A is complete for f:

$$\mathbb{C}^A(f) \quad \Leftrightarrow \quad A \circ f = A \circ f \circ A\,. \tag{2}$$

B. Regular Commands

Following O'Hearn [24] (see also Winskel [29, Chapter 14, Exercise 14.4]) we consider a language of *regular commands*:

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

which is general enough to cover deterministic imperative languages as well as other programming paradigms that include, e.g., nondeterministic and probabilistic computations, and equational systems such as Kleene algebras with tests [19], [20]. The language is parametric on the syntax of basic transfer expressions (or functions) $\mathsf{e} \in \mathsf{Exp}$, which provide the basic commands and can be instantiated with different kinds of instructions such as (nondeterministic or parallel) assignments, (Boolean) guards or assumptions, error generation primitives, etc. More generally, regular commands can be used in the context of strategy languages for rewrite systems, where basic instructions can serve to represent enabling conditions for the applicability of rewrite rules or their application [2], [14], [13, Section 4]. Then, the term $\mathsf{r}_1; \mathsf{r}_2$ represents sequential composition, the term $\mathsf{r}_1 \oplus \mathsf{r}_2$ represents a choice command that can behave as either r_1 or r_2, and the term r^* is the Kleene iteration of r where r can be executed 0 or any bounded number of times in a sequence. As an abbreviation, we write r^n for the sequence $\mathsf{r}; ...; \mathsf{r}$ of n instances of r and let $\mathsf{Exp}(\mathsf{r})$ be the set of basic transfer expressions occurring in $\mathsf{r} \in \mathsf{Reg}$.

1) Concrete semantics: We assume that basic transfer expressions have a semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ on a complete lattice C such that $(\!|\mathsf{e}|\!)$ is an additive function. This assumption can be done w.l.o.g. in Hoare-like (or collecting) program semantics, since the basic transfer functions are always defined by an additive lifting, namely, they are defined as successor or predecessor transformer of a transition relation The concrete semantics $[\![\cdot]\!] : \mathsf{Reg} \to C \to C$ of regular commands is inductively defined as follows:

$$\begin{array}{ll} [\![\mathsf{e}]\!]c \triangleq (\!|\mathsf{e}|\!)c & [\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]c \triangleq [\![\mathsf{r}_1]\!]c \vee [\![\mathsf{r}_2]\!]c \\ [\![\mathsf{r}_1; \mathsf{r}_2]\!]c \triangleq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]c) & [\![\mathsf{r}^*]\!]c \triangleq \bigvee\{[\![\mathsf{r}]\!]^n c \mid n \in \mathbb{N}\} \end{array} \tag{3}$$

2) Abstract Semantics: The abstract semantics of regular commands $[\![\cdot]\!]^\sharp_A : \mathsf{Reg} \to A \to A$ on an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is defined by structural induction as follows:

$$\begin{array}{rl} [\![\mathsf{e}]\!]^\sharp_A a \triangleq [\![\mathsf{e}]\!]^A a = (\alpha \circ [\![\mathsf{e}]\!] \circ \gamma)a \\ [\![\mathsf{r}_1; \mathsf{r}_2]\!]^\sharp_A a \triangleq [\![\mathsf{r}_2]\!]^\sharp_A([\![\mathsf{r}_1]\!]^\sharp_A a) \\ [\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^\sharp_A a \triangleq [\![\mathsf{r}_1]\!]^\sharp_A a \vee_A [\![\mathsf{r}_2]\!]^\sharp_A a \\ [\![\mathsf{r}^*]\!]^\sharp_A a \triangleq \bigvee_A\{([\![\mathsf{r}]\!]^\sharp_A)^n a \mid n \in \mathbb{N}\} \end{array} \tag{4}$$

It is also easy to check by structural induction that the abstract semantics in (4) is monotonic and correct, i.e., $\alpha \circ [\![\mathsf{r}]\!] \leq_A [\![\mathsf{r}]\!]^\sharp_A \circ \alpha$ holds. Let us point out that as abstract semantics of basic expressions e we consider their bcas on A, i.e., we assume that no additional loss of precision is due to their interpretation. We remark that this is the standard definition by *structural induction* of abstract semantics used in abstract interpretation, adapted to the language of regular commands. Therefore, it turns out that the abstract semantics of the choice command preserves bcas, namely $[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^A a = [\![\mathsf{r}_1]\!]^A a \vee_A [\![\mathsf{r}_2]\!]^A a$. This property of preserving bcas, in general, does not hold for sequential composition and Kleene iteration: for example, $[\![\mathsf{r}_2]\!]^A \circ [\![\mathsf{r}_1]\!]^A$ is not guaranteed to be the bca $[\![\mathsf{r}_1; \mathsf{r}_2]\!]^A$. On the other hand, it can be easily seen, by structural induction, that all the definitions in (4) preserve completeness, meaning that if $[\![\mathsf{r}_1]\!]^\sharp_A, [\![\mathsf{r}_2]\!]^\sharp_A, [\![\mathsf{r}]\!]^\sharp_A$ are complete, then $[\![\mathsf{r}_1; \mathsf{r}_2]\!]^\sharp_A$, $[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^\sharp_A$ and $[\![\mathsf{r}^*]\!]^\sharp_A$ are complete as well.

3) Programs: We consider standard basic transfer expressions used in deterministic while programs: no-op instruction, assignments and Boolean guards, as defined below:

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

where, for simplicity, we consider just integer values and variables and Var is a denumerable set of program variables. Hence, a standard deterministic imperative language Imp (cf. [29]) can be defined using guarded branching and loop commands as syntactic sugar (cf. [19, Section 2.2]):

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

To improve readability, in our running examples we will use this syntactic sugar whenever possible.

A program store $\sigma : V \to \mathbb{Z}$ is a total function from a finite set of variables of interest $V \subseteq \mathit{Var}$ to values and $\Sigma \triangleq V \to \mathbb{Z}$ denotes the set of stores on the variables ranging in a set V that is left implicit. The concrete domain is $\mathbb{S} \triangleq \wp(\Sigma)$, ordered by inclusion. Store update $[x \mapsto v]$ is defined as usual: $S[x \mapsto v] \triangleq \{\sigma[x \mapsto v] \mid \sigma \in S\}$ where

$$\sigma[x \mapsto v](y) \triangleq \begin{cases} v & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

The semantics $(\!|\mathsf{e}|\!) : \mathbb{S} \to \mathbb{S}$ of basic transfer expressions is:

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

where $\{\!|\mathsf{a}|\!\} : \Sigma \to \mathbb{Z}$ and $\{\!|\mathsf{b}|\!\} : \Sigma \to \{\mathbf{tt}, \mathbf{ff}\}$ are inductively defined as expected. For brevity, we overload b to denote the set $(\!|\mathsf{b}?|\!)\,\Sigma$ of all and only stores that satisfy b, so that $(\!|\mathsf{b}?|\!)S = S \cap \mathsf{b}$ filters the concrete stores in S making b true. The usual strongest post-condition for r on pre-condition $P \in \mathbb{S}$ is thus $\mathsf{post}[\mathsf{r}]P \triangleq [\![\mathsf{r}]\!]P$. Analogously, $\mathsf{post}_A[\mathsf{r}]\alpha(P) \triangleq [\![\mathsf{r}]\!]^\sharp_A\alpha(P)$.

In the following, we will present some simple running examples involving programs with just one variable, so that $V = \{x\}$. In these cases, to simplify the notation, $\wp(\mathbb{Z})$ will be used to represent sets of stores in $\mathbb{S}$, i.e., $S \in \wp(\mathbb{Z})$ represents the set $\{\sigma \in \Sigma \mid \sigma(x) \in S\} \in \mathbb{S}$. Accordingly, $\mathrm{Abs}(\wp(\mathbb{Z}))$ will represent $\mathrm{Abs}(\mathbb{S})$. For example, $\{-2, 2\}$ will be used to represent a more verbose expression such as $x = -2 \vee x = 2$.

III. On the Limits of (Global) Completeness

It has been proven in [3], [16] that completeness holds for all programs in a Turing complete programming language only for trivial abstract domains. This means that the only abstract domains that are complete for all programs are the straightforward ones: the identical abstraction, making abstract and concrete semantics the same, and the top abstraction, making all programs equivalent by abstract semantics. In [16] the authors observed that since **skip** is always trivially complete and composition, conditional and loop statements all preserve the completeness of their subprograms, the only sources of incompleteness may arise from assignments and Boolean guards. Nevertheless, one can logically prove the completeness of specific programs by structural induction on their syntax, as done by the basic proof system in [16]. In this case, the completeness of (the semantic functions associated with) assignments and Boolean guards occurring in a program is a sufficient condition to guarantee the completeness of the whole program. While the completeness of assignments has been extensively studied (e.g., the completeness conditions for assignments in major numerical domains such as intervals, congruences, octagons and affine relations have been fully settled [16], [21]), the case of Boolean guards is troublesome and largely unexplored. In particular, in the case of conditional and loop statements, the completeness on a store abstraction A calls for the validity of the conditions $\mathbb{C}^A([\![\mathsf{b}?]\!])$ and $\mathbb{C}^A([\![\neg\mathsf{b}?]\!])$, or, equivalently,

$$\begin{aligned}\forall S \in \mathbb{S}.\ & A(S \cap \mathsf{b}) = A(A(S) \cap \mathsf{b}) \quad \&\\ & A(S \cap \neg\mathsf{b}) = A(A(S) \cap \neg\mathsf{b})\end{aligned} \tag{5}$$

The adjective *global* in the section title refers to the universal quantification over any possible set S of stores in (5), which we prove to be a major limitation. The following results provide a sufficient and necessary condition on the abstract domain A for guaranteeing both $\mathbb{C}^A([\![\mathsf{b}?]\!])$ and $\mathbb{C}^A([\![\neg\mathsf{b}?]\!])$. It is worth remarking that the same result extends to any arbitrary distributive concrete domain C whenever b admits a complement $\neg\mathsf{b}$, even if the whole lattice C is not complemented.

We first observe that when the functions $[\![\mathsf{b}?]\!]$ and $[\![\neg\mathsf{b}?]\!]$ are complete in a strict abstract domain A, then b and $\neg\mathsf{b}$ are necessarily expressible in A.

Lemma III.1 *Let $A \in \mathrm{Abs}(\mathbb{S})$ be strict. If $\mathbb{C}^A([\![\mathsf{b}?]\!])$ and $\mathbb{C}^A([\![\neg\mathsf{b}?]\!])$ hold then b and $\neg\mathsf{b}$ are expressible in A.*

Furthermore, when b and $\neg\mathsf{b}$ are both expressible in A, it turns out that the completeness of $[\![\mathsf{b}?]\!]$ and $[\![\neg\mathsf{b}?]\!]$ boils down to a co-additivity requirement of the abstraction map α or, equivalently, an additivity requirement for the concretization map γ. This is clearly a way too strong requirement in abstract interpretation as co-additive abstractions imply that the abstract domain is a complete join and meet sublattice of the concrete domain.

Lemma III.2 *Let b and $\neg\mathsf{b}$ be expressible in $A_{\alpha,\gamma} \in \mathrm{Abs}(\mathbb{S})$. Then, $\mathbb{C}^A([\![\mathsf{b}?]\!])$ and $\mathbb{C}^A([\![\neg\mathsf{b}?]\!])$ hold iff*

$$\begin{aligned}\forall S \in \mathbb{S}.\ & \alpha(S \cap \mathsf{b}) = \alpha(S) \wedge_A \alpha(\mathsf{b}) \quad \&\\ & \alpha(S \cap \neg\mathsf{b}) = \alpha(S) \wedge_A \alpha(\neg\mathsf{b})\end{aligned} \tag{6}$$

The next characterization result gives an effective way to verify whether an abstract domain A is complete w.r.t. a Boolean guard b. It amounts to check that b and $\neg\mathsf{b}$ are both expressible in A and that the union of the concretizations of any two abstract points in A below, resp., $\alpha(\mathsf{b})$ and $\alpha(\neg\mathsf{b})$, is also expressible in A (see Example III.4).

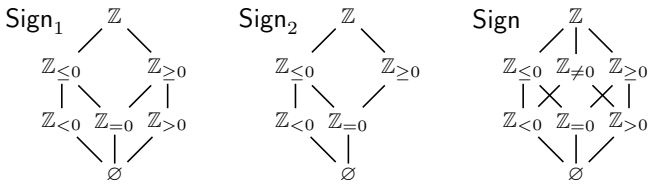


Fig. 2: Abstract Domains for Sign Analysis.

Theorem III.3 (Complete Guards) *Let* b, $\neg\mathsf{b}$ *be expressible in* $A_{\alpha,\gamma} \in \mathrm{Abs}(\mathbb{S})$. *Then,* $\mathbb{C}^A(\llbracket \mathsf{b}? \rrbracket)$ *and* $\mathbb{C}^A(\llbracket \neg\mathsf{b}? \rrbracket)$ *hold iff*

$$\forall S_1^\sharp, S_2^\sharp \in A.\ (\ S_1^\sharp \leq_A \alpha(\mathsf{b})\ \&\ S_2^\sharp \leq_A \alpha(\neg\mathsf{b}) \Rightarrow \gamma(S_1^\sharp \vee_A S_2^\sharp) = \gamma(S_1^\sharp) \cup \gamma(S_2^\sharp)\ ) \tag{7}$$

Example III.4 Let $\mathsf{Sign}_1, \mathsf{Sign}_2, \mathsf{Sign} \in \mathrm{Abs}(\wp(\mathbb{Z}))$ be the abstract domains depicted in Fig. 2. In Sign_1 no expressible Boolean guard is complete (except the trivial ones **tt** and **ff**). Indeed, only the elements $\varnothing$ and $\mathbb{Z}$ satisfy condition (7). The negation of $\mathbb{Z}_{=0}$ is $\mathbb{Z}_{\neq 0}$, which does not belong to Sign_1. For the guard $\mathbb{Z}_{<0}$, its negation $\mathbb{Z}_{\geq 0}$ is in Sign_1, but the join of $\mathbb{Z}_{<0}$ with $\mathbb{Z}_{>0} \leq_{\mathsf{Sign}_1} \mathbb{Z}_{\geq 0}$ is again $\mathbb{Z}_{\neq 0}$. Dually for $\mathbb{Z}_{>0}$.
In Sign_2 the guards $\mathbb{Z}_{\geq 0}$ and $\mathbb{Z}_{<0}$ are complete, while $\mathbb{Z}_{\leq 0}$ and $\mathbb{Z}_{=0}$ are not. For example, for $\mathbb{Z}_{=0} \leq_{\mathsf{Sign}_2} \mathbb{Z}_{\geq 0}$ and $\mathbb{Z}_{<0}$ we have $\gamma(\mathbb{Z}_{=0} \vee_{\mathsf{Sign}_2} \mathbb{Z}_{<0}) = \gamma(\mathbb{Z}_{\leq 0}) = \gamma(\mathbb{Z}_{=0}) \cup \gamma(\mathbb{Z}_{<0})$. It follows that, even if both ReLU and Abs in Section I are complete in Sign_2, only ReLU can be inductively proved to be complete, because all its basic transfer functions are complete in Sign_2. In the case of Abs instead, the assignment $x := -x$ is not complete for $x > 0$, even if such input will never be provided to that branch of code.
In Sign all expressible Boolean guards are complete and the completeness of Abs and ReLU can be proved inductively. □

Theorem III.3 displays a major drawback of refining an abstract domain in order to achieve completeness for Boolean guards. Because all interesting programs contain Boolean guards, complete abstract domains refining a given domain may indeed become very close to the concrete domain, therefore limiting the effectiveness of this notion of completeness in program analysis. The following example shows this phenomenon for the case of the abstract domain of intervals.

Example III.5 Consider the well-known abstract domain $\mathsf{Int} \in \mathrm{Abs}(\wp(\mathbb{Z}))$ of integer intervals [6], [21]. The only Boolean guards b and $\neg\mathsf{b}$ that are expressible in Int are the infinite intervals $[-\infty, k]$ and $[k, \infty]$ together with the trivial intervals $\mathbb{Z}$ and $\varnothing$. In fact, in Int, the complement of any finite interval $[a, b] \in \mathsf{Int}$, with $a \leq b$, must be necessarily approximated. However, if we consider $\mathsf{b} = [-\infty, k]$ and, correspondingly, $\neg\mathsf{b} = [k+1, \infty]$ then condition (7) of Theorem III.3 is not satisfied. As an example, let us fix $k = 0$, i.e. $\mathsf{b} = [-\infty, 0]$ and, correspondingly, $\neg\mathsf{b} = [1, \infty]$: condition (7) of Theorem III.3 would require the presence of all the concrete joins $[n_1, n_2] \cup [m_1, m_2]$ with $n_1 \leq n_2 \leq 0 < m_1 \leq m_2$, because $[n_1, n_2] \leq_{\mathsf{Int}} [-\infty, 0]$ and $[m_1, m_2] \leq_{\mathsf{Int}} [1, \infty]$, but such joins are not intervals, unless $n_2 = 0$ and $m_1 = 1$.
Even a basic guard such as $\mathsf{b} = [0, 0]$ would need its complement $\neg\mathsf{b} = [-\infty, -1] \cup [1, \infty]$ as well as the concrete joins $[n_1, n_2] \cup [0, 0]$ for any $n_1 \leq n_2 < -1$ or $1 < n_1 \leq n_2$, because any such interval $[n_1, n_2]$ is below $[-\infty, -1] \cup [1, \infty]$. Moreover, since abstract domains are closed by meet, the intersection $[n, -1] \cup [1, m]$ of $\neg\mathsf{b}$ and any interval $[n, m]$ with $n < 0 < m$ must be present. □

IV. Local Completeness

Section III shows that the standard notion of (global) completeness (2) for Boolean guards is a too strong requirement for abstract domains, often met in practice just by trivial guards or domains. While completeness can be hard/impossible to achieve *globally*, i.e. for *all* possible sets of stores, it could well happen that completeness holds *locally*, i.e. just for *some* store properties. We therefore put forward a notion of *local completeness* which in program analysis corresponds to consider completeness only along certain program traces.

Definition IV.1 (Local Completeness) An abstract domain $A \in \mathrm{Abs}(C)$ is *locally complete* for $f : C \to C$ on a concrete value $c \in C$, written $\mathbb{C}_c^A(f)$, if the following condition holds:

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

Let us observe that A is trivially locally complete for any f on any abstract value $A(c)$ (i.e., $\mathbb{C}^A_{A(c)}(f)$ always holds). As discussed in Section I, in program analysis it may well happen that $\llbracket \mathsf{c} \rrbracket$ is not globally complete w.r.t. the abstract domain A but it is locally complete for a particular class of inputs P.

Example IV.2 Consider the Imp program

$$\mathsf{c} \triangleq \textbf{if } (0 < x) \textbf{ then } x := x - 2 \textbf{ else } x := -x$$

and the interval abstraction Int. While the transfer function $\llbracket 0 < x? \rrbracket$ is not globally complete (see Example III.5), it is locally complete for any set $P \in \wp(\mathbb{Z})$ such that: (1) $P \subseteq \mathbb{Z}_{>0}$, or (2) $P \subseteq \mathbb{Z}_{\leq 0}$, or (3) $\{0, 1\} \subseteq P$. Since the transfer functions for constant addition and multiplication are globally complete, the program c is locally complete for any P satisfying one of the above conditions (1–3). For example, if $P = \{0, 1, 4\}$, condition (3) holds and we have that $\mathsf{Int}(\llbracket \mathsf{c} \rrbracket P) = \mathsf{Int}(\{-1, 0, 2\}) = [-1, 2]$ and $\mathsf{Int}(\llbracket \mathsf{c} \rrbracket \mathsf{Int}(P)) = \mathsf{Int}(\llbracket \mathsf{c} \rrbracket [0, 4]) = \mathsf{Int}(\{-1, 0, 1, 2\}) = [-1, 2]$. As an example of local incompleteness, if $P = \{0, 4\}$ we have that $\mathsf{Int}(\llbracket \mathsf{c} \rrbracket P) = \mathsf{Int}(\{0, 2\}) = [0, 2]$, but $\mathsf{Int}(\llbracket \mathsf{c} \rrbracket \mathsf{Int}(P)) = [-1, 2]$. □

Let us remark that, with respect to a compositional reasoning, there is a significant key difference between global and local completeness: while the composition (via generic regular commands operators, and consequently via conditionals and loops) of globally complete transfer functions is always globally complete, the same does not necessarily hold for local completeness that depends on a given input property. Equivalently, local completeness of a composite program may well depend on the partial store properties met during the computation, as shown by the following example.

Example IV.3 Consider a composition $\mathsf{c};\mathsf{c}$, where c is defined in Example IV.2. Int is locally complete for c on the input property $P = \{2, 6\}$, because condition (1) of Example IV.2 holds. However, Int is not locally complete for $\mathsf{c};\mathsf{c}$ on P, because $\mathsf{Int}([\![\mathsf{c};\mathsf{c}]\!]\{2,6\}) = \mathsf{Int}([\![\mathsf{c}]\!]\{0,4\}) = [0,2]$ while $\mathsf{Int}([\![\mathsf{c};\mathsf{c}]\!]\mathsf{Int}(\{2,6\})) = \mathsf{Int}([\![\mathsf{c};\mathsf{c}]\!][2,6]) = \mathsf{Int}([\![\mathsf{c}]\!][0,4]) = \mathsf{Int}(\{-1,0,1,2\}) = [-1,2]$. □

V. Local Completeness Logic

We define a logical proof system for program analysis of regular commands, parameterized by an abstraction A, whose provable triples $\vdash_A [P]\ \mathsf{r}\ [Q]$ guarantee that:

(i) Q is an *under-approximation* of $[\![\mathsf{r}]\!]P$;
(ii) $[\![\mathsf{r}]\!]$ is *locally complete* for input P and abstraction A;
(iii) Q and $[\![\mathsf{r}]\!]P$ have the same *over-approximation* in A.

It turns out that these properties of $\vdash_A [P]\ \mathsf{r}\ [Q]$ allow us to distinguish between true and false alerts raised by the abstract interpretation $[\![\mathsf{r}]\!]^\sharp_A\alpha(P)$ for checking a correctness specification *Spec*, as explained below:

Case 1: If the over-approximation $[\![\mathsf{r}]\!]^\sharp_A\alpha(P)$ does not raise alerts, i.e. $\gamma([\![\mathsf{r}]\!]^\sharp_A\alpha(P)) \subseteq \mathit{Spec}$ holds, then the program r does not exhibit unwanted behaviours. It should be remarked that this already holds for any sound and possibly incomplete over-approximating abstract interpretation.

Case 2: If *Spec* is expressible in A and the abstract interpretation $[\![\mathsf{r}]\!]^\sharp_A\alpha(P)$ raises some alerts because $\gamma([\![\mathsf{r}]\!]^\sharp_A\alpha(P)) \not\subseteq \mathit{Spec}$, then, by local completeness, any provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ is such that $Q \cap \neg\, \mathit{Spec} \neq \varnothing$ and all the stores in $Q \cap \neg\, \mathit{Spec}$ are true alerts. Let us stress that by using ordinary abstract interpretation we could not distinguish which alerts in $\gamma([\![\mathsf{r}]\!]^\sharp_A\alpha(P)) \smallsetminus \mathit{Spec}$ are true ones and which are false.

Case 3: If *Spec* is expressible in A, some alert is raised because $\gamma([\![\mathsf{r}]\!]^\sharp_A A(P)) \not\subseteq \mathit{Spec}$ but any attempt to derive a triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ for some under-approximation Q fails because some proof obligations of local completeness $\mathbb{C}^A_c(f)$ are not met, then the abstraction A is not precise enough to distinguish between true and false alerts in a compositional way (for r on P). In this case, one could refine the abstraction A to enhance its precision and repeat the analysis, possibly guided by the failed proof obligations (see Example V.8).

The logical proof system $\vdash_A$ is defined in Fig. 3 and called LCL_A (Local Completeness Logic on A). Our objective in designing this deductive system has been to track the assumptions of local completeness needed for having a compositional proof. The distinctive rules are (transfer) and (relax) whose premises directly depend on the underlying abstraction A.

The combined consequence rule (relax) is the key principle which allows us to adapt and generalize partial proofs to broader contexts. The novelty of (relax) lies in combining an over- and under-approximating reasoning: (relax) allows to infer a post-condition Q that is an under-approximation of the exact behaviour but whose abstraction $A(Q)$ is a sound over-approximation of it, i.e., such that $Q \subseteq [\![\mathsf{r}]\!]P \subseteq A([\![\mathsf{r}]\!]P) = A(Q)$ holds. Likewise the consequence rules of [10], [24], [25] the logical ordering between pre-conditions $P' \leq P$ and post-conditions $Q \leq Q'$ in the premises of (relax) is reversed w.r.t. the canonical consequence rule of Hoare logic and this is needed because our post-conditions Q are always under-approximations. Let us also remark that the premises of (relax) imply that $A(P) = A(P')$ and $A(Q) = A(Q')$. Example V.9 will show that a dual version of (relax) with P strengthening P' and Q weakening Q' as in the classical consequence rule of Hoare logic would not be sound w.r.t. local completeness.

The crux of (relax) is to constrain this under-approximating post-condition Q to have *the same abstraction* as the exact behaviour, in order to preserve the precision of the deduction. This opens up an interesting perspective about the generality of our proof system that will be tackled in Section VI for showing how to recover O'Hearn's IL [24] as an instance of our proof system. Moreover, in Section V-C we also show how an easy dualization of LCL_A allows to accomodate backward abstract reasoning as used in backward program analysis.

Technically, the validity of the rule (relax) relies on observing that local completeness is a kind of "abstract convex property": if A is locally complete for some $c \in C$ then A is locally complete for all $d \in C$ such that $c \leq d \leq A(c)$ holds.

Lemma V.1 *If* $\mathbb{C}^A_c(f)$ *and* $d \in [c, A(c)]$ *then* $\mathbb{C}^A_d(f)$.

The rule (transfer) checks that the basic transfer expressions e are locally complete on P and, in that case, provides the output of the corresponding transfer function $[\![\mathsf{e}]\!]$ on P as post-condition. Of course, for no-ops, assignments and Boolean guards, the rule (transfer) can be equivalently stated in symbolic form as follows:

$$\frac{}{\vdash_A [P]\ \mathbf{skip}\ [P]}\ (\mathsf{skip}) \qquad \frac{\mathbb{C}^A_P([\![\mathsf{b}?]\!])}{\vdash_A [P]\ \mathsf{b}?\ [P \wedge \mathsf{b}]}\ (\mathsf{assume})$$

$$\frac{\mathbb{C}^A_P([\![x := \mathsf{a}]\!])}{\vdash_A [P]\ x := \mathsf{a}\ [\exists v.(P[^v/_x] \wedge x = \mathsf{a}[^v/_x])]}\ (\mathsf{assign})$$

where $[^v/_x]$ denotes the substitution for replacing x by v.

The rule (seq) for sequential composition and the rule (join) for choice are standard. The rule (rec) allows us to unfold one step of Kleene iteration, until the rule (iterate) can be applied.

(iterate) is a distinguishing rule of LCL_A and is as much fundamental as rule (relax) for several reasons: both rules have premises depending on the abstraction A; under-approximated post-conditions are only introduced by these two rules (all the other rules are otherwise "exact"); while the concrete semantics of r^* can be infinitary (e.g., consider $(x := x+1)^*$), (iterate) can exploit the abstraction A to stop the proof when the abstraction of a finitary input P is already an infinitary abstract invariant (a maybe non-obvious consequence of the condition $Q \leq A(P)$), returning a finite under-approximation of the concrete invariant; the combination of under- and over-approximations in the rule (iterate) is therefore more expressive than the sum of its parts, as it allows us to speed up both program analysis and alert detection.

The next two examples illustrate the key features of LCL_A: the first one exploits all the rules and the second one is applied to a classical while-loop. They will be revisited in Section V-A to show how LCL_A can help in program analysis.

$$\frac{\mathbb{C}^A_P(\mathsf{e})}{\vdash_A [P]\ \mathsf{e}\ [[\![\mathsf{e}]\!]P]}\ (\mathsf{transfer}) \qquad \frac{P' \le P \le A(P') \quad \vdash_A [P']\ \mathsf{r}\ [Q'] \quad Q \le Q' \le A(Q)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\mathsf{relax})$$

$$\frac{\vdash_A [P]\ \mathsf{r}_1\ [R] \quad \vdash_A [R]\ \mathsf{r}_2\ [Q]}{\vdash_A [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q]}\ (\mathsf{seq}) \qquad \frac{\vdash_A [P]\ \mathsf{r}_1\ [Q_1] \quad \vdash_A [P]\ \mathsf{r}_2\ [Q_2]}{\vdash_A [P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [Q_1 \vee Q_2]}\ (\mathsf{join})$$

$$\frac{\vdash_A [P]\ \mathsf{r}\ [R] \quad \vdash_A [P \vee R]\ \mathsf{r}^*\ [Q]}{\vdash_A [P]\ \mathsf{r}^*\ [Q]}\ (\mathsf{rec}) \qquad \frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad Q \le A(P)}{\vdash_A [P]\ \mathsf{r}^*\ [P \vee Q]}\ (\mathsf{iterate})$$

Fig. 3: The Proof System LCL_A.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}}_{P_1}([\![\mathsf{b}_1?]\!])}{\vdash_{\mathsf{Int}} [P_1]\ \mathsf{b}_1?\ [\{1,999,1000\}]}\ (\mathsf{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Int}}_{\{1,999,1000\}}([\![\mathsf{e}_1]\!])}{\vdash_{\mathsf{Int}} [\{1,999,1000\}]\ \mathsf{e}_1?\ [\{0,998,999\}]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}} [P_1]\ \mathsf{r}_1\ [\{0,998,999\}]}\ (\mathsf{seq}) \quad \dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}}_{P_1}([\![\mathsf{b}_2?]\!])}{\vdash_{\mathsf{Int}} [P_1]\ \mathsf{b}_2?\ [\{0,1,999\}]}\ (\mathsf{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Int}}_{\{0,1,999\}}([\![\mathsf{e}_2]\!])}{\vdash_{\mathsf{Int}} [\{0,1,999\}]\ \mathsf{e}_2\ [\{1,2,1000\}]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}} [P_1]\ \mathsf{r}_2\ [\{1,2,1000\}]}\ (\mathsf{seq})}{\vdash_{\mathsf{Int}} [P_1]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [\{0,1,2,998,999,1000\}]}\ (\mathsf{join})}{(\star)}\ (\mathsf{iterate})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}}_{P}([\![\mathsf{b}_1?]\!])}{\vdash_{\mathsf{Int}} [P]\ \mathsf{b}_1?\ [P]}\ (\mathsf{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Int}}_{P}([\![\mathsf{e}_1]\!])}{\vdash_{\mathsf{Int}} [P]\ \mathsf{e}_1\ [\{0,998\}]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1\ [\{0,998\}]}\ (\mathsf{seq}) \quad \dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}}_{P}([\![\mathsf{b}_2?]\!])}{\vdash_{\mathsf{Int}} [P]\ \mathsf{b}_2?\ [P]}\ (\mathsf{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Int}}_{P}([\![\mathsf{e}_2]\!])}{\vdash_{\mathsf{Int}} [P]\ \mathsf{e}_2\ [\{2,1000\}]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_2\ [\{2,1000\}]}\ (\mathsf{seq})}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [\{0,2,998,1000\}]}\ (\mathsf{join}) \quad \dfrac{(\star)}{\vdash_{\mathsf{Int}} [\{0,1,999,1000\}]\ \mathsf{r}\ [\{0,1,2,998,999,1000\}]}\ (\mathsf{iterate})}{\vdash_{\mathsf{Int}} [\{0,1,2,998,999,1000\}]\ \mathsf{r}\ [\{0,2,1000\}]}\ (\mathsf{relax})}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [\{0,2,1000\}]}\ (\mathsf{rec})$$

Fig. 4: Derivation of $\vdash_{\mathsf{Int}} [P = \{1,999\}]\ \ \mathsf{r}\ [\{0,2,1000\}]$ for Example V.2.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Sign}}_{P}([\![x \le 0?]\!])}{\vdash_{\mathsf{Sign}} [P]\ x \le 0?\ [\{-10,-1\}]}\ (\mathsf{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Sign}}_{\{-10,-1\}}([\![x := x*10]\!])}{\vdash_{\mathsf{Sign}} [\{-10,-1\}]\ x := x*10\ [\{-100,-10\}]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Sign}} [P]\ x \le 0?; x := x*10\ [\{-100,-10\}] \quad \{-100,-10\} \subseteq \mathsf{Sign}(P) = \mathbb{Z}_{\neq 0}}\ (\mathsf{seq})}{\vdash_{\mathsf{Sign}} [P]\ (x \le 0?; x := x*10)^*\ [\{-100,-10,-1,100\}] \quad \{-100,100\} \subseteq \{-100,-10,-1,100\} \subseteq \mathsf{Sign}(\{-100,100\}) = \mathbb{Z}_{\neq 0}}\ (\mathsf{iterate})}{\vdash_{\mathsf{Sign}} [P]\ (x \le 0?; x := x*10)^*\ [\{-100,100\}]}\ (\mathsf{relax}) \quad \dfrac{\mathbb{C}^{\mathsf{Sign}}_{\{-100,100\}}([\![0 < x?]\!])}{\vdash_{\mathsf{Sign}} [\{-100,100\}]\ 0 < x?\ [\{100\}]}\ (\mathsf{transfer})$$

$$\vdash_{\mathsf{Sign}} [P]\ \mathsf{c}\ [\{100\}]\ (\mathsf{seq})$$

Fig. 5: Derivation of $\vdash_{\mathsf{Sign}} [P = \{-10,-1,100\}]\ \ \mathsf{c}\ [\{100\}]$ for Example V.3.

Example V.2 Let us consider the interval domain Int, the pre-condition $P \triangleq \{1,999\}$ and the command $\mathsf{r} \triangleq (\mathsf{r}_1 \oplus \mathsf{r}_2)^*$ where

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

The triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [\{0,2,1000\}]$ can be derived as shown in Fig. 4, where for brevity we let:

$$\mathsf{b}_1 \triangleq 0 < x \qquad \mathsf{e}_1 \triangleq x := x-1 \quad P_1 \triangleq \{0,1,999,1000\}$$
$$\mathsf{b}_2 \triangleq x < 1000 \quad \mathsf{e}_2 \triangleq x := x+1$$

Notably, each instance of rule (transfer) used in the derivation exposes a proof obligation (such as $\mathbb{C}^{\mathsf{Int}}_P([\![\mathsf{e}_2]\!])$, $\mathbb{C}^{\mathsf{Int}}_{P_1}([\![\mathsf{b}_1?]\!])$, etc.) concerning the local completeness of a basic transfer function. This proof needs just one application of (rec) to compute an under-approximation of $\mathsf{post}[\mathsf{r}]P = [\![\mathsf{r}]\!]P$, because the rule (iterate) can stop the unfolding of the Kleene iterate operator as soon as an abstract invariant is detected, before the actual concrete invariant is fully computed (in this case the abstract invariant is detected by $\{0,1,2,998,999,1000\} \subseteq [0,1000]$). Moreover, (relax) is exploited to reduce the number of values to be taken into account (along the pre-conditions by navigating the derivation tree bottom-up and along the post-conditions when the tree is explored top-down).
Finally, a similar result is soon obtained on any input $P_k \triangleq \{k,999\}$ for some $k \in \mathbb{N}$, by applying the rule (rec) for k times: then the rule (iterate) can be used. □

Example V.3 Let us consider the domain Sign, the pre-condition $P \triangleq \{-10,-1,100\}$ and the Imp program

$$\mathsf{c} \triangleq \textbf{while}\ (x \le 0)\ \textbf{do}\ x := x*10$$
$$= (x \le 0?\,;\, x := x*10)^*\,;\, 0 < x?$$

Let us verify that c does not satisfy the correctness specification $\mathit{Spec} \triangleq x < 10$, even if the loop c diverges on inputs $\{-10,-1\}$. The derivation in Fig. 5 proves the triple $\vdash_{\mathsf{Sign}} [P]\ \mathsf{c}\ [\{100\}]$. As the post-condition $\{100\}$ is an under-approximation of $[\![\mathsf{r}]\!]P$ (cf. Theorem V.4 (1)), we conclude that $100 \notin \mathit{Spec}$ is a true alert. Observe that all proof obligations about local completeness due to rule (transfer) are satisfied, as e.g., letting $\mathsf{b} \triangleq x \le 0$, for $\mathbb{C}^{\mathsf{Sign}}_P([\![\mathsf{b}?]\!])$, we have

$$\mathsf{Sign}([\![\mathsf{b}?]\!]\mathsf{Sign}(P)) = \mathsf{Sign}([\![\mathsf{b}?]\!]\mathbb{Z}_{\neq 0}) = \mathsf{Sign}(\mathbb{Z}_{<0}) = \mathbb{Z}_{<0}$$
$$\mathsf{Sign}([\![\mathsf{b}?]\!]P) = \mathsf{Sign}(\{-10,-1\}) = \mathbb{Z}_{<0}.\ \square$$

Of course, let us point out that some additional valid rules could be added to our proof system, for example the following two rules can be easily proved to be valid:

$$\frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad Q \le P}{\vdash_A [P]\ \mathsf{r}^*\ [P]}\ (\mathsf{invariant})$$

$$\frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad A(P) = A(Q)}{\vdash_A [P]\ \mathsf{r}^*\ [Q]}\ (\mathsf{abs\text{-}fix})$$

Rule (invariant) is the analogous of the loop invariant rule in Hoare logic, while (abs-fix) allows us to accelerate the convergence of Kleene iteration to an abstract fixpoint.

A. Soundness

Our logic LCL_A turns out to be sound for the target properties (i–iii) stated at the beginning of Section V, as formalized below, where (1) and (2) embody (i) and (ii)+(iii), respectively.

Theorem V.4 (Soundness) *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. *For all* $\mathsf{r} \in \mathsf{Reg}$, $P, Q \in C$, *if* $\vdash_A [P]\ \mathsf{r}\ [Q]$ *then:* (1) $Q \leq [\![\mathsf{r}]\!](P)$ *and* (2) $[\![\mathsf{r}]\!]^\sharp_A \alpha(P) = \alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$.

As a consequence, if a correctness specification *Spec* is expressible in A, i.e., if $\mathit{Spec} = \gamma(a)$ for some abstract element a, then any provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ allows us to use Q to decide *both the correctness or the incorrectness* of r for the pre-condition P, as stated by the following result.

Corollary V.5 (Precision) *For all* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$, $\mathsf{r} \in \mathsf{Reg}$, $P, Q \in C$, *if* $\vdash_A [P]\ \mathsf{r}\ [Q]$, *then:*

$$\forall a \in A.\ [\![\mathsf{r}]\!]P \leq \gamma(a) \text{ iff } Q \leq \gamma(a).$$

Example V.6 In Example V.3 we already noticed that, by Theorem V.4 (1), $[\![\mathsf{c}]\!]P$ does not satisfy *Spec*. Note that for $P' \triangleq \{-10, -1, 5, 100\}$ we could also prove, e.g., $\vdash_{\mathsf{Sign}} [P']\ \mathsf{c}\ [\{5\}]$ where the post-condition $\{5\}$ has no alert, even if *Spec* is not satisfied: since *Spec* is not expressible in Sign, then Corollary V.5 is not applicable. □

Example V.7 Let us consider again Example V.2 and the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [\{0, 2, 1000\}]$ (see Fig. V.2) to discuss the cases of three different correctness specifications: $\mathit{Spec}_1 \triangleq x \neq 2$, $\mathit{Spec}_2 \triangleq x \leq 1000$ and $\mathit{Spec}_3 \triangleq 100 \leq x$.
Despite that Spec_1 is not expressible in Int, the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [\{0, 2, 1000\}]$ exposes the true alert $2 \notin \mathit{Spec}_1$, since $2 \in \{0, 2, 1000\} \subseteq [\![\mathsf{r}]\!]P$ (cf. Theorem V.4 (1)).
By Theorem V.4 (2), we know that $[\![\mathsf{r}]\!]P$ satisfies Spec_2, because $\mathsf{Int}([\![\mathsf{r}]\!]P) = \mathsf{Int}(\{0, 2, 1000\}) = [0, 1000] \subseteq \mathit{Spec}_2$. Since Spec_2 is expressible in Int, by Corollary V.5, any other provable triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$ will guarantee that Spec_2 holds. Finally, the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [\{0, 2, 1000\}]$ exposes two true alerts $0, 2 \notin \mathit{Spec}_3$. Likewise Spec_2, by Corollary V.5, the post-condition Q of any provable triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$ will contain a true alert for Spec_3. In particular, any such triple is such that $0 \in Q$, contradicting the assertion Spec_3, because it must be $\mathsf{Int}(Q) = [0, 1000] = \mathsf{Int}([\![\mathsf{r}]\!]P)$. □

Example V.8 Let us consider the program $\mathsf{r} \triangleq \mathsf{r}_1^*; \mathsf{r}_2$ where:

$$\mathsf{r}_1 \triangleq (x < 1000?; x := x + 10)$$
$$\mathsf{r}_2 \triangleq (x = 30?; x := -1) \oplus (x \neq 30?)$$

Let $\mathit{Spec} \triangleq x \neq -1$, $P_1 \triangleq \{10\}$ and $P_2 \triangleq \{11\}$. In the case of P_1, the value 30 for x is actually computed by the Kleene iteration r_1^*, causing a violation of *Spec*. In the case P_2, any alert raised by a static analysis would be a false positive since r_1^* will never store the value 30 in x. Let us use the abstract domain Sign for the analysis of r. Firstly, consider P_1 and observe that by applying (rec) and (iterate) we can derive $\vdash_{\mathsf{Sign}} [\{10\}]\ \mathsf{r}_1^*\ [\{10, 20, 30\}]$. Then, by (relax), we prove $\vdash_{\mathsf{Sign}} [\{10\}]\ \mathsf{r}_1^*\ [\{10, 30\}]$. Note that $\mathbb{C}^{\mathsf{Sign}}_{\{10,30\}}([\![x = 30?]\!])$ and $\mathbb{C}^{\mathsf{Sign}}_{\{10,30\}}([\![x \neq 30?]\!])$, so that $\vdash_{\mathsf{Sign}} [\{10, 30\}]\ \mathsf{r}_2\ [\{-1, 10\}]$ can be proved. Finally, by applying (seq), we derive the triple $\vdash_{\mathsf{Sign}} [\{10\}]\ \mathsf{r}_1^*; \mathsf{r}_2\ [\{-1, 10\}]$, whose post-condition includes the true alert -1 violating *Spec*.
Consider now the pre-condition P_2. Here, we apply (rec), (iterate) and (relax) so as to derive $\vdash_{\mathsf{Sign}} [\{11\}]\ \mathsf{r}_1^*\ [\{11, 31\}]$. In this case, since local completeness $\mathbb{C}^{\mathsf{Sign}}_{\{11,31\}}([\![x = 30?]\!])$ does not hold (indeed note that any nonempty subset of $\{11, 31\}$ is not locally complete for $x = 30?$ on Sign), the proof cannot be completed in Sign. It is worth observing that in order to prove that r satisfies *Spec* for input P_2 we need to resort to a more precise abstract domain where the guard $x = 30?$ is locally complete for $\{11, 31\}$. In this sense, we remark that the failed proof obligation $\mathbb{C}^{\mathsf{Sign}}_{\{11,31\}}([\![x = 30?]\!])$ could be exploited to find a refinement of the current abstraction Sign where the derivation can be completed. For example, a possible choice is to extend the abstract domain Sign by taking Sign_{30} as the Moore closure of $\mathsf{Sign} \cup \{x \neq 30\}$: then, the triple $\vdash_{\mathsf{Sign}_{30}} [P_2]\ \mathsf{r}\ [\{11\}]$ could be derived to witness that r satisfies *Spec* (by Corollary V.5). □

The next example shows that the classical consequence rule of Hoare logic for strengthening pre-conditions and weakening post-conditions is in general not compatible with the local completeness property (2) of Theorem V.4.

Example V.9 Consider the abstract domain Int, the program

$$\mathsf{c} \triangleq \textbf{if } (x < 0) \textbf{ then } x := -x \textbf{ else skip}$$

and the pre-conditions $P \triangleq \{-5, 5\}$ and $P' \triangleq \{-5, -1, 0, 5\}$. By applying (transfer) (to $x < 0?$, $x \geq 0?$, $x := -x$ and **skip**), (seq) twice and (join), one can easily derive $\vdash_{\mathsf{Int}} [P']\ \mathsf{c}\ [Q']$ with $Q' = \{0, 1, 5\}$. In fact, Int is locally complete for $[\![\mathsf{c}]\!]$ on P', because $\mathsf{Int}([\![\mathsf{c}]\!]\mathsf{Int}\ (P')) = [0, 5] = \mathsf{Int}(Q')$.
Imagine now to replace the rule (relax) by its dual (convex) version, inspired by the consequence rule of Hoare logic

$$\frac{P \leq P' \leq A(P) \quad \vdash_A [P']\ \mathsf{r}\ [Q'] \quad Q' \leq Q \leq A(Q')}{\vdash_A [P]\ \mathsf{r}\ [Q]}$$

If we let $Q = \mathsf{Int}(Q')$, since $P \subseteq P' \subseteq \mathsf{Int}(P) = [-5, 5]$, and $Q' \subseteq Q = \mathsf{Int}(Q')$, then we could derive, e.g., $\vdash_{\mathsf{Int}} [P]\ \mathsf{c}\ [Q]$. However, Int is not locally complete for $[\![\mathsf{c}]\!]$ on P, because $\mathsf{Int}([\![\mathsf{c}]\!]P) = [5, 5] \neq [0, 5] = \mathsf{Int}([\![\mathsf{c}]\!]\mathsf{Int}(P))$. □

B. On the Completeness of LCL_A

We now study the completeness, in the logical sense, of LCL_A as a proof system. Our logic LCL_A is not logically complete in general, i.e., the converse of Theorem V.4 does not hold, as shown by the following example.

Example V.10 Let $\mathsf{e}_1 \triangleq x := x - 1$, $\mathsf{e}_2 \triangleq x := x + 1$, and $P \triangleq \mathbb{Z}_{\geq 2}$. The abstract domain $A = \mathsf{Sign}$, is locally complete for $[\![\mathsf{e}_1; \mathsf{e}_2]\!]$ on P:

$$\alpha([\![\mathsf{e}_1; \mathsf{e}_2]\!]P) = \alpha([\![\mathsf{e}_2]\!]([\![\mathsf{e}_1]\!]P)) = \alpha(P) = \mathbb{Z}_{>0}$$
$$[\![\mathsf{e}_1; \mathsf{e}_2]\!]^\sharp_A \alpha(P) = [\![\mathsf{e}_2]\!]^\sharp_A([\![\mathsf{e}_1]\!]^\sharp_A \mathbb{Z}_{>0}) = [\![\mathsf{e}_2]\!]^\sharp_A \mathbb{Z}_{\geq 0} = \mathbb{Z}_{>0}$$

but the triple $\vdash_{\mathsf{Sign}} [P]\ \mathsf{e}_1; \mathsf{e}_2\ [P]$ cannot be derived. This is because, in the attempt to derive the triple $\vdash_{\mathsf{Sign}} [P]\ \mathsf{e}_1\ [[\![\mathsf{e}_1]\!]P]$ by rule (transfer), the proof obligation $\mathbb{C}^{\mathsf{Sign}}_P([\![\mathsf{e}_1]\!])$ fails: $\alpha([\![\mathsf{e}_1]\!]P) = \mathbb{Z}_{>0} \neq \mathbb{Z}_{\geq 0} = \alpha(\mathbb{Z}_{\geq 0}) = \alpha([\![\mathsf{e}_1]\!]\mathsf{Sign}(P))$. Note that the rule (relax) cannot help. In fact, let us assume that there exists some P' such that $P' \subseteq P \subseteq \mathsf{Sign}(P')$ and the triple $\vdash_{\mathsf{Sign}} [P']\ \mathsf{e}_1\ [[\![\mathsf{e}_1]\!]P']$ is provable by (transfer). Then, $\mathsf{Sign}(P') = \mathsf{Sign}(P) = \mathbb{Z}_{>0}$ and $P' \subseteq P = \mathbb{Z}_{\geq 2}$ imply that $[\![\mathsf{e}_1]\!]P' \subseteq \mathbb{Z}_{\geq 1}$. Hence, we would have that $\alpha([\![\mathsf{e}_1]\!]P') \leq \alpha(\mathbb{Z}_{\geq 1}) = \mathbb{Z}_{\geq 0}$ while $\alpha([\![\mathsf{e}_1]\!]\mathsf{Sign}(P')) = \alpha([\![\mathsf{e}_1]\!]\mathbb{Z}_{>0}) = \mathbb{Z}_{\geq 0}$. Thus, $\mathbb{C}^{\mathsf{Sign}}_{P'}([\![\mathsf{e}_1]\!])$ does not hold, contradicting the hypothesis that $\vdash_{\mathsf{Sign}} [P']\ \mathsf{e}_1\ [[\![\mathsf{e}_1]\!]P']$ is provable. □

For a result of logical completeness for LCL_A, we need:

(1) to add the sound, infinitary rule (limit) for Kleene star:

$$\frac{\forall n \in \mathbb{N}.\ \vdash_A [P_n]\ \mathsf{r}\ [P_{n+1}]}{\vdash_A [P_0]\ \mathsf{r}^*\ [\bigvee_{i \in \mathbb{N}} P_i]}\ (\mathsf{limit})$$

(2) to assume that all the basic transfer expressions occurring in a command $\mathsf{r} \in \mathsf{Reg}$ are globally complete on A.

We are therefore able to obtain a result of logical completeness for LCL_A when this includes the powerful infinitary rule (limit) and under an assumption of global completeness for the basic transfer expressions occurring in the program. Let us remark that incorrectness logic [24] also includes this infinitary rule (limit), there called *backwards variant* rule. Likewise (limit), the backwards variant rule of IL allows to derive other finitary rules (e.g., *Iterate zero*) and plays a crucial role for proving the logical completeness of IL [24, Theorem 6].

Theorem V.11 (Logical Completeness) *Let $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ and $\mathsf{r} \in \mathsf{Reg}$ such that any $\mathsf{e} \in \mathsf{Exp}(\mathsf{r})$ is globally complete on A. For any $P, Q \in C$, if $Q \leq [\![\mathsf{r}]\!]P$ and $[\![\mathsf{r}]\!]^\sharp_A \alpha(P) = \alpha(Q)$ then $\vdash_A [P]\ \mathsf{r}\ [Q]$.*

It is worth noting that if any $\mathsf{e} \in \mathsf{Exp}(\mathsf{r})$ is globally complete on A then, as proved in [16, Theorem 5.1], $[\![\mathsf{r}]\!]^\sharp_A \circ \alpha = \alpha \circ [\![\mathsf{r}]\!]$ also holds. Thus, the hypotheses of Theorem V.11 imply that that properties (1–2) of the soundness Theorem V.4 all hold, i.e. Theorem V.11 provides a result of (limited) logical completeness for LCL_A. Viceversa, whenever the language is Turing complete, $C = \mathbb{S}$ and the abstraction A is not trivial, LCL_A turns out to be *intrinsically incomplete*, meaning that it is always possible to find a valid triple (w.r.t. properties (1–2) of Theorem V.4) but LCL_A is unable to prove it.

Theorem V.12 (Intrinsic Incompleteness) *Let* Reg *be a Turing complete language. Let $A_{\alpha,\gamma} \in \mathrm{Abs}(\mathbb{S})$. If $\gamma\alpha \neq id$ and $\gamma\alpha \neq \lambda x.\Sigma$ then there exist $P, Q \in \mathbb{S}$ and $\mathsf{r} \in \mathsf{Reg}$ such that $Q \leq [\![\mathsf{r}]\!]P$, $[\![\mathsf{r}]\!]^\sharp_A \alpha(P) = \alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$ but $\not\vdash_A [P]\ \mathsf{r}\ [Q]$.*

Turing completeness is crucial here because the proof relies upon the possibility of: (1) specifying in Reg an arbitrary store in Σ, (2) effectively checking store inequality, and (3) specifying in Reg an undefined transfer function. The proof of Theorem V.12 generalizes to LCL_A and to Turing complete regular commands the proofs in [3] and [16] showing that the class of all programs for which an abstract interpretation on A is globally complete is the set of all programs (or, equivalently, an index set for partial recursive functions) if and only if A is trivial. As a consequence of Theorems V.11 and V.12, LCL_A cannot be a logically complete proof system for a Turing complete language unless the abstraction A is trivial. Of course, the identical abstraction is unfeasible here as both rules (transfer) and (relax) would become vacuous. Hence, the only meaningful straightforward abstraction for LCL_A is $A = \lambda x.\Sigma$. Under this light we may therefore observe that logical completeness of IL [24, Theorem 6] follows from the choice made in its consequence rule of letting pre-conditions and post-conditions be, resp., arbitrarily weakened and strengthened, i.e., the choice of the abstraction $A = \lambda x.\Sigma$ in rule (relax). Section VI provides additional details on the relationship with incorrectness logic.

C. A Backward Proof System

As pioneered by Cousot [5, Section 3.4], it is known that abstract interpretation-based program analysis can be defined backward rather than forward [1], [21]. Instead of propagating forward an abstract store in a program control flow graph (CFG) from its entry point, a backward analysis can start from any program point q of the CFG (possibly, but not necessarily, an exit point) and from any input abstract value a it abstractly computes backward in the CFG to derive *necessary* abstract conditions for the executions reaching q and satisfying the store property a. Backward analysis is typically used after a preliminary forward analysis to refine its results, as acknowledged by Bourdoncle [1] for abstract program debugging. An under-approximating backward program analysis by abstract interpretation has been put forward by Miné [22]. In the following, we show how LCL_A can be easily dualized in order to accomodate backward analyses.

In backward analysis, the basic transfer expressions e have a co-additive backward concrete semantics $(\!|\mathsf{e}|\!)_\leftarrow : C \to C$. Notably, $(\!|\mathsf{e}|\!)_\leftarrow Y \triangleq \widetilde{\mathsf{pre}}_{R_\mathsf{e}}(Y) = \{\sigma \mid (\sigma, \sigma') \in R_\mathsf{e} \Rightarrow \sigma' \in Y\}$, where R_e is the transition relation for e. For example, for the basic transfer functions of Imp (Section II-B3) we have that:

$$(\!|\mathbf{skip}|\!)_\leftarrow Y \triangleq Y$$
$$(\!|x := \mathsf{a}|\!)_\leftarrow Y \triangleq \{\sigma \in \Sigma \mid \sigma[x \mapsto \{\!|\mathsf{a}|\!\}\,\sigma] \in Y\},$$
$$(\!|\mathsf{b}?|\!)_\leftarrow Y \triangleq \{\sigma \in \Sigma \mid \{\!|\mathsf{b}|\!\}\,\sigma = \mathbf{tt} \Rightarrow \sigma \in Y\} = Y \cup \neg\mathsf{b}.$$

Furthermore, in backward concrete and abstract semantics: (1) the control flows backwards and (2) meets replace joins. Hence, the backward semantics of regular commands is given as the following dual definition of (3):

$$[\![\mathsf{e}]\!]_\leftarrow c \triangleq (\!|\mathsf{e}|\!)_\leftarrow c \qquad [\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]_\leftarrow c \triangleq [\![\mathsf{r}_1]\!]_\leftarrow c \wedge [\![\mathsf{r}_2]\!]_\leftarrow c$$
$$[\![\mathsf{r}_1; \mathsf{r}_2]\!]_\leftarrow c \triangleq [\![\mathsf{r}_1]\!]_\leftarrow([\![\mathsf{r}_2]\!]_\leftarrow c) \qquad [\![\mathsf{r}^*]\!]_\leftarrow c \triangleq \bigwedge\{[\![\mathsf{r}]\!]^n_\leftarrow c \mid n \in \mathbb{N}\}$$

The backward abstract semantics uses an *under-approximating* abstraction $U_{\alpha,\gamma} \in \mathrm{Abs}^{\leftarrow}(C)$, meaning that U is defined by a Galois insertion w.r.t. the concrete domain $\langle C, \geq \rangle$ where $\geq \triangleq \leq^{-1}$. This means that for all $x \in C$, an under-approximation relation $\gamma(\alpha(x)) \leq x$ replaces an over-approximating approximation $x \leq \gamma(\alpha(x))$.

Example V.13 The interval domain Int can be viewed as an under-approximating abstraction by dualizing its maps $\alpha^{\leftarrow} : \wp(\mathbb{Z}) \to \mathsf{Int}$ and $\gamma^{\leftarrow} : \mathsf{Int} \to \wp(\mathbb{Z})$ as follows:

$$\gamma^{\leftarrow}([l,u]) \triangleq \neg\gamma([l,u]) = \{z \in \mathbb{Z} \mid z < l \vee u > z\}$$
$$\alpha^{\leftarrow}(X) \triangleq \alpha(\neg X) = [\min(\neg X), \max(\neg X)]$$

For example, $\alpha^{\leftarrow}(\{x \in \mathbb{Z} \mid x < -3 \vee x > 5\} \cup \{0,1,2\}) = [-3,5]$ and $\alpha^{\leftarrow}(\{x \in \mathbb{Z} \mid x < -3\} \cup \{0,1,2\} = [-3,+\infty]$. □

Accordingly, the abstract semantics $[\![\mathsf{r}]\!]^{\sharp}_{U}$ for an under-approximating abstraction $U \in \mathrm{Abs}^{\leftarrow}(C)$ is defined as dual of (4). Correspondingly, when our proof system $\vdash_U$ is instantiated to $U \in \mathrm{Abs}^{\leftarrow}(C)$, we need to replace $\leq$, which models logical implication, with $\geq$ and logical disjunction $\vee$ with conjunction $\wedge$. Hence, the fundamental (relax) rule becomes:

$$\frac{U(P') \leq P \leq P' \quad \vdash_U [P']\ \mathsf{r}\ [Q'] \quad U(Q) \leq Q' \leq Q}{\vdash_U [P]\ \mathsf{r}\ [Q]}$$

Thus, here the condition P is logically stronger than P' and weaker than the under-approximation $U(P')$, and dually for Q. By duality, as a direct consequence of Theorems V.4 and V.11, a result of soundness and limited logical completeness for the dual logic $\vdash_U$ can be derived. Hence, a provable triple $\vdash_U [P]\ \mathsf{r}\ [Q]$ for an under-approximation U implies that: $[\![\mathsf{r}]\!]_{\leftarrow} Q \leq P$, i.e. P is an over-approximation of the backward semantics, and $[\![\mathsf{r}]\!]^{\sharp}_{U}\alpha(Q) = \alpha(P) = \alpha([\![\mathsf{r}]\!]_{\leftarrow} Q)$, i.e. local completeness holds.

VI. Relationship with Incorrectness Logic

The idea to reverse the direction of implication in Hoare's consequence rule was investigated by de Vries and Koutavas' reverse Hoare logic [10] to study reachability specifications for randomized algorithms. They first put forward the rule:

$$\frac{P' \leq P \quad [P']\ \mathsf{r}\ [Q'] \quad Q \leq Q'}{[P]\ \mathsf{r}\ [Q]}\ (\mathsf{cons})$$

O'Hearn's incorrectness logic extends the approach of reverse Hoare logic with an explicit handling of error detection and propagation. As pointed out by O'Hearn in [24], "*Program correctness and incorrectness are two sides of the same coin [...] Incorrectness Logic is so basic that it could have been defined and studied immediately after or alongside the fundamental works of Floyd and Hoare on correctness in the 1960s*". Because IL is tailored to under-approximations, it can be used to prove the presence of bugs but not their absence.

In [24], programs are regular commands that include primitives such as: the error() statement, to halt execution and raise an error signal **er**; assume(b) statements, analogous to our Boolean guards b?; and nondeterministic assignments

$$(\!|\mathbf{skip}|\!)(\mathbf{ok}:Q, \mathbf{er}:R) \triangleq \mathbf{ok}:Q, \mathbf{er}:R$$
$$(\!|x := \mathsf{a}|\!)(\mathbf{ok}:Q, \mathbf{er}:R) \triangleq \mathbf{ok}:\textstyle\bigcup_{\sigma\in Q}\{\sigma[x \mapsto (\!|\mathsf{a}|\!)\sigma]\}, \mathbf{er}:R$$
$$(\!|\mathsf{error}()|\!)(\mathbf{ok}:Q, \mathbf{er}:R) \triangleq \mathbf{er}:(Q \cup R)$$
$$(\!|\mathsf{assume(b)}|\!)(\mathbf{ok}:Q, \mathbf{er}:R) \triangleq \mathbf{ok}:(Q \cap \mathsf{b}), \mathbf{er}:R$$
$$(\!|x := \mathsf{nond}()|\!)(\mathbf{ok}:Q, \mathbf{er}:R) \triangleq \mathbf{ok}:\textstyle\bigcup_{v\in\mathbb{Z}} Q[x \mapsto v], \mathbf{er}:R$$

Fig. 6: Basic transfer functions for additional statements.

$x := \mathsf{nond}()$ also present in the setting of reverse Hoare logic. Thus, we set:

$$\mathsf{Exp} \ni \mathsf{e} ::= \mathbf{skip} \mid x := \mathsf{a} \mid \mathsf{error}() \mid \mathsf{assume(b)} \mid x := \mathsf{nond}()$$

Incorrectness logic triples take the form $\vdash_{\mathrm{IL}} [P]\ \mathsf{c}\ [\epsilon : Q]$, as their post-conditions are extended with labels $\epsilon \in \{\mathbf{ok}, \mathbf{er}\}$ (following [24], we use colors for a visual differentiation) to distinguish the case of error-free termination $\mathbf{ok}:Q$ leading to an under-approximating post-condition Q, from interrupted computations $\mathbf{er}:Q$, meaning that some error occurred under the circumstances reported by Q. We write $\vdash_{\mathrm{IL}} [P]\ \mathsf{c}\ [\mathbf{ok}:Q][\mathbf{er}:R]$ when $\vdash_{\mathrm{IL}} [P]\ \mathsf{c}\ [\mathbf{ok}:Q]$ and $\vdash_{\mathrm{IL}} [P]\ \mathsf{c}\ [\mathbf{er}:R]$ are derivable for the same pre-condition P. Notably, the proof system of IL is proved to be sound and complete (in the logical sense): an error can arise iff it can be exposed by some provable triple.

Next, we sketch how IL can be seen as a particular instance of LCL_A. Due to error handling, the domain $\mathbb{S}$ is not informative enough, but this is not a problem given the generality of LCL_A. Therefore, we lay the following four ingredients:

(i) A suitable concrete domain $\mathcal{C} \triangleq \wp(\{\mathbf{ok}, \mathbf{er}\} \times \Sigma)$ that can distinguish between normal and erroneous termination. For $Q \in \wp(\Sigma)$ and $\epsilon \in \{\mathbf{ok}, \mathbf{er}\}$ we write $\epsilon : Q$ as a shorthand for $\{\epsilon : \sigma \mid \sigma \in Q\} = \{\epsilon\} \times Q$. Clearly, a generic $S \in \mathcal{C}$ takes the form $\mathbf{ok}:Q \cup \mathbf{er}:R$ for suitable $Q, R \in \wp(\Sigma)$, so we denote elements in $\mathcal{C}$ more concisely as $\mathbf{ok}:Q, \mathbf{er}:R$.

(ii) Transfer functions $f : \mathcal{C} \to \mathcal{C}$ are additive functions s.t.

$$f(\mathbf{ok}:Q, \mathbf{er}:R) = \mathbf{er}:R \cup \textstyle\bigcup_{\sigma\in Q} f(\mathbf{ok}:\sigma)$$

meaning that all the errors R are preserved. Possibly further errors are generated by the application of f to some $\sigma \in Q$.

(iii) The semantics of basic transfer functions is defined in Fig. 6. For any $\mathsf{r} \in \mathsf{Reg}$ and any $Q, R \in \wp(\Sigma)$, by structural induction on r, it follows that $[\![\mathsf{r}]\!](\mathbf{er}:R) = \mathbf{er}:R$ and $[\![\mathsf{r}]\!](\mathbf{ok}:Q, \mathbf{er}:R) = [\![\mathsf{r}]\!](\mathbf{ok}:Q) \cup \mathbf{er}:R$.

(iv) The abstract domain A_{tr} is the trivial abstraction such that $\gamma\alpha = \lambda X.\top = \lambda X.(\{\mathbf{ok}, \mathbf{er}\} \times \Sigma)$. Note that A_{tr} is (globally) complete for every transfer function, therefore all proof obligations $\mathbb{C}^{A}_{c}(\mathsf{e})$ are trivially satisfied, and (transfer) becomes an axiom. Moreover, by $A_{tr}(P') = \top = A_{tr}(Q)$, the rule (relax) boils down to the rule (cons).

At the level of concrete semantics, we can establish a tight connection between the transfer function $[\![\mathsf{r}]\!] : \mathcal{C} \to \mathcal{C}$ associated with $\mathsf{r} \in \mathsf{Reg}$ and its relational denotational semantics that was taken as a reference model for IL. Let us denote by $[\![\mathsf{r}]\!]\epsilon \subseteq \Sigma \times \Sigma$ the relational semantics defined in [24, Figure 4]. In order to formalize the correspondence, we find

it convenient to let $[\![\mathsf{r}]\!]_\epsilon : \mathbb{S} \to \mathbb{S}$ denote the functional version of $[\![\mathsf{r}]\!]\epsilon$ defined by: $[\![\mathsf{r}]\!]_\epsilon P \triangleq \{\sigma' \in \Sigma \mid \sigma \in P,\ (\sigma, \sigma') \in [\![\mathsf{r}]\!]\epsilon\}$.

Lemma VI.1 *For any* $\mathsf{r} \in \mathsf{Reg}$ *and* $P \in \mathbb{S}$ *we have:*

$$[\![\mathsf{r}]\!](\mathbf{ok} : P) = \mathbf{ok} : [\![\mathsf{r}]\!]_{\mathbf{ok}} P, \mathbf{er} : [\![\mathsf{r}]\!]_{\mathbf{er}} P$$

Lemma VI.1 is instrumental for proving the equivalence between IL and the particular instance $\mathrm{LCL}_{A_{tr}}$ of our logic as determined by (i–iv) above. This correspondence necessarily follows as a consequence of the (logical) completeness of IL [24, Theorem 6] and our Theorem V.11 of limited completeness, which guarantee that any under-approximation of the strongest post-condition $\mathsf{post}[c]P$ is provable in both logics.

Corollary VI.2 *For any* $P, Q, R \in \mathbb{S}$ *and* $\mathsf{r} \in \mathsf{Reg}$*:*

$$\vdash_{\mathrm{IL}} [P]\ \mathsf{r}\ [\mathbf{ok} : Q][\mathbf{er} : R] \quad \textit{iff} \quad \vdash_{A_{tr}} [\mathbf{ok} : P]\ \mathsf{r}\ [\mathbf{ok} : Q, \mathbf{er} : R].$$

It is interesting to observe that any instance LCL_A of our logic using an abstraction $A \neq A_{tr}$ would only be able to derive some triples of IL *but not all of them* (while, of course, any triple derived in $\vdash_A$ would also be derivable in $\vdash_{\mathrm{IL}}$). This is a consequence of the intrinsic incompleteness Theorem V.12 that extends the impossibility results of [3], [16] to regular commands. Therefore, whenever $A \neq A_{tr}$ there will always exist some program r and some triple $\vdash_{\mathrm{IL}} [P]\ \mathsf{r}\ [\mathbf{ok} : Q][\mathbf{er} : R]$ such that it will not be possible to derive $\vdash_A [\mathbf{ok} : P]\ \mathsf{r}\ [\mathbf{ok} : Q, \mathbf{er} : R]$ because some proof obligation $\mathbb{C}_S^A([\![\mathsf{e}]\!])$ of local completeness will fail for some basic transfer expression e appearing in r.

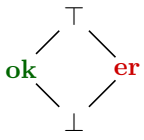
The correspondence provided by Corollary VI.2 is interesting, because although the rules of IL and ours share some similarities, they also display significant differences:

(a) The most visible difference is that the pre-conditions of the triples in $\vdash_{\mathrm{IL}}$ are elements of $\mathbb{S}$ while pre-conditions of triples in $\vdash_{A_{tr}}$ are elements of $\mathcal{C}$, meaning that the rules in $\vdash_{\mathrm{IL}}$ are concerned only with normal inputs, while the rules in $\vdash_{A_{tr}}$ must deal also with (the propagation of) erroneous inputs.

(b) Building on (a), the rules of IL are tailored to error propagation, while our rules are designed for any concrete domain. Both logical frameworks can be extended to deal with different kinds of error and error recovery mechanisms. Because our rule (transfer) is parametric on basic transfer expressions, it should not be necessary to change any rule of our proof system to implement such extensions. For example, IL exploits two rules for sequential composition while LCL_A just needs a single composition rule but it delegates error propagation to the underlying concrete domain.

(c) Some rules of IL, such as *Disjunction*, *Choice* and *Iterate zero* in [24, Fig. 2], are designed to incrementally build the under-approximation bottom-up by composing smaller under-approximations into larger ones. On the contrary, LCL_A is constrained to work in the opposite direction by the requirement of preserving the over-approximation of the strongest post-condition in the abstraction A.

(d) Finally, incorrectness logic includes specific rules for dealing with the introduction of fresh local variables, while



$$\begin{array}{rcl} \gamma_{\mathsf{Err}}(\top) & \triangleq & \{\mathbf{ok}, \mathbf{er}\} \times \Sigma \\ \gamma_{\mathsf{Err}}(\mathbf{ok}) & \triangleq & \{\mathbf{ok}\} \times \Sigma \\ \gamma_{\mathsf{Err}}(\mathbf{er}) & \triangleq & \{\mathbf{er}\} \times \Sigma \\ \gamma_{\mathsf{Err}}(\bot) & \triangleq & \varnothing \end{array}$$

Fig. 7: The abstract domain Err

in the context of abstract interpretation it is typically assumed that the program variables are statically known. Of course, it would be possible to deal with dynamic allocation in LCL_A although its formalization would be technically more involved.

The use of an abstract over-approximation in LCL_A that constrains any under-approximation has some advantages but also carries drawbacks. The main advantages are: (1) by exploiting the over-approximation, LCL_A can also prove the absence of errors, and (2) under the hypothesis that the correctness specification *Spec* is expressible in the abstraction A, *any provable triple will suffice to establish either Spec is satisfied or violated*. However, as already mentioned, whenever A is not trivial, not all the possible under-approximations can be obtained by our system. The next example shows this phenomenon by combining the interval abstraction Int with the simple error domain Err depicted in Fig. 7.

Example VI.3 Let us revisit Example V.7 (in turn using the regular command r of Example V.2). To study $Spec_2$, in the context of $\vdash_{\mathrm{IL}}$ we focus on the command $\widehat{\mathsf{r}}_2 \triangleq \mathsf{r}; \mathsf{r}_{s2}$ where $\mathsf{r}_{s2} \triangleq (x \leq 1000?; \mathbf{skip}) \oplus (1000 < x?; \mathsf{error}())$. Using $\vdash_{\mathsf{Int}}$, it was shown in Example V.7 that r satisfies $Spec_2$, so that $\widehat{\mathsf{r}}_2$ will never issue the error signal. Since the absence of errors cannot be proved by under-approximations, incorrectness logic cannot derive any useful information in this case. Let us consider instead the abstract domain $\mathsf{Int}^+ \triangleq \mathsf{Err} \sqcap \mathsf{Int}$ defined as reduced product [7, Section 10.1] of Err and Int, that is, whose concretization map is defined as $\gamma_{\mathsf{Int}^+}(\langle a_1, a_2 \rangle) \triangleq \gamma_{\mathsf{Err}}(a_1) \cap \gamma_{\mathsf{Int}}(a_2)$. By mimicking the derivation of $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [\{0, 2, 1000\}]$, it is not hard to check that $\vdash_{\mathsf{Int}^+} [\mathbf{ok} : P]\ \widehat{\mathsf{r}}_2\ [\mathbf{ok} : \{0, 2, 1000\}]$ is derivable. Since $\mathsf{Int}^+(\mathbf{ok} : \{0, 2, 1000\}) = \mathbf{ok} : [0, 1000]$ over-approximates the strongest post-condition, this is enough for proving that no error will be issued by $\widehat{\mathsf{r}}_2$ with pre-condition P.
To study $Spec_3$, we focus on $\widehat{\mathsf{r}}_3 \triangleq \mathsf{r}; \mathsf{r}_{s3}$ where

$$\mathsf{r}_{s3} \triangleq (100 \leq x?; \mathbf{skip}) \oplus (x < 100?; \mathsf{error}())$$

We know from Example V.7, that $[\![\mathsf{r}]\!]P \not\subseteq Spec_3$, so that $\widehat{\mathsf{r}}_3$ can issue some errors. Within IL we can derive triples that exhibit some erroneous situation, like $\vdash_{\mathrm{IL}} [P]\ \widehat{\mathsf{r}}_3\ [\mathbf{er} : \{0\}]$ as well as others that do not, e.g., $\vdash_{\mathrm{IL}} [P]\ \widehat{\mathsf{r}}_3\ [\mathbf{ok} : \{1000\}]$.
By mimicking the derivation of $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [\{0, 2, 1000\}]$, but applying (rec) 100 times in order to include the values 100 and 101 that are necessary to satisfy the local completeness requirements for the tests $100 \leq x?$ and $x < 100?$, we can then derive $\vdash_{\mathsf{Int}^+} [\mathbf{ok} : P]\ \widehat{\mathsf{r}}_3\ [\mathbf{ok} : \{1000\}, \mathbf{er} : \{0, 2\}]$ using the abstract domain Int^+. Note that, since $\mathsf{Int}^+(\mathbf{ok} : \{1000\}, \mathbf{er} : \{0, 2\}) = \top : [0, 1000]$, the rule (relax) cannot be used to discard all the errors because this would

change the abstract over-approximation of the post-condition. Since $\mathbf{ok}{:}[100, +\infty]$ is expressible in Int^+, by Corollary V.5, the label $\top$ in the over-approximation provides good evidence about the occurrence of one or more errors. Moreover, because the over-approximation induced by Int is always preserved, any provable triple $\vdash_{\mathsf{Int}^+} [P]\ \widehat{\mathsf{r}}_3\ [Q]$ is such that Q will contain the true alert $\mathbf{er}{:}\{0\}$ (as well as $\mathbf{ok}{:}\{1000\}$). □

VII. Future Work

We have presented a logic for locally complete abstract interpretations, called LCL_A (parametric on an abstraction A), that can *prove the presence as well as the absence of true alerts*, i.e., LCL_A is able to prove both the correctness and incorrectness of some program specification. As far as we know, LCL_A combines for the first time over- and under-approximations in a logical proof system for abstract interpretation. We think that our work opens many promising lines of research.

Firstly, when some proof obligation $\mathbb{C}_c^A(f)$ about the local completeness in A of some basic transfer function f fails, a natural question is whether it is possible to transform A into some A' in order to satisfy $\mathbb{C}_c^{A'}(f)$ and conclude the derivation in $\vdash_{A'}$. In particular, following [17], we are interested in the problem of *minimally transforming* the abstract domain A to A' through refinements (i.e. by adding concrete values) and simplifications (i.e. by removing abstract values) in order to make A' locally complete for a set of basic transfer functions. As sketched in Example V.8, in program verification the strategy would be to iteratively transform the original abstract domain A_0 through a series of domains $A_1, ..., A_n$, until a derivation $\vdash_{A_n} [P]\ \mathsf{r}\ [Q]$ can be completed in A_n.

While completeness of abstract transfer functions is preserved by function composition, one major issue of abstract interpretation is that bcas are not compositional [26], [30]. The lack of composition for bcas has practical consequences, because the precision of a program analysis strictly depends on the granularity of program decomposition into atomic operations. Finer decompositions commonly induce more imprecise analyses, while coarser decompositions may enhance the chance of designing bcas for larger code blocks. We plan to investigate a proof system for the *property of being a bca*, notably for proving when the composition of bcas is a bca.

Finally, we plan to explore whether and how the relationship of LCL_A with IL studied in Section VI could be extended to incorrectness separation logic [25]. The main challenge will be to engineer a suitable frame rule for heap assertions that is able to preserve local completeness for the abstraction A.

Acknowledgments: The authors have been funded by the *Italian MIUR*, under the PRIN2017 project no. 201784YSZ5 "AnalysiS of PRogram Analyses (ASPRA)". The work of Francesco Ranzato and Roberto Giacobazzi has been partially funded by *Facebook Research*, under a "Probability and Programming Research Award". Francesco Ranzato has also been partially funded by the *University of Padova*, under the SID2018 project "Analysis of STatic Analyses (ASTA)".

References

[1] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. ACM SIGPLAN PLDI'93*, page 46–55. ACM, 1993.
[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.
[3] R. Bruni, R. Giacobazzi, R. Gori, I. Garcia-Contreras, and D. Pavlovic. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.*, 4(POPL):28:1–28:28, 2020.
[4] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *Proc. NFM'15*, LNCS 9058, pp. 3–11, 2015.
[5] P. Cousot. *Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique des Programmes*. PhD thesis, U. Grenoble, 1978.
[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM POPL'77*, pp. 238–252. ACM, 1977.
[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM POPL'79*, pp. 269–282. ACM, 1979.
[8] P. Cousot and R. Cousot. Abstract interpretation: past, present and future. In *Proc. Joint Meeting CSL-LICS'14*, pp. 2:1–2:10. ACM, 2014.
[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *Proc. ESOP'05*, LNCS 3444, pp. 21–30, 2005.
[10] E. de Vries and V. Koutavas. Reverse Hoare logic. In *Proc. SEFM'11*, pp. 155–171. Springer, 2011.
[11] E. W. Dijkstra. The humble programmer. *CACM*, 15(10):859–866, 1972.
[12] D. Distefano, M. Fähndrich, F. Logozzo, and P. O'Hearn. Scaling static analyses at Facebook. *CACM*, 62(8):62–70, 2019.
[13] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. L. Talcott. Programming and symbolic computation in maude. *J. Log. Algebraic Methods Program.*, 110, 2020.
[14] M. Fernández, H. Kirchner, and O. Namet. A strategy language for graph rewriting. In *LOPSTR'11*, LNCS 7225, pp. 173–188, 2011.
[15] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
[16] R. Giacobazzi, F. Logozzo, and F. Ranzato. Analyzing program analyses. In *Proc. ACM POPL'15*, pp. 261–273. ACM, 2015.
[17] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretation complete. *J. ACM*, 47(2):361–416, 2000.
[18] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 1969.
[19] D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, May 1997.
[20] D. Kozen. On Hoare logic and Kleene algebra with tests. *ACM Trans. Comput. Logic*, 1(1):60–76, July 2000.
[21] A. Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Found. Trends in Prog. Lang.*, 4(3-4):120–372, 2017.
[22] A. Miné. Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Sci. Comput. Program.*, 93:154 – 182, 2014.
[23] P. W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Proc. LICS'18*, page 13–25. ACM, 2018.
[24] P. W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, 2020.
[25] A. Raad, J. Berdine, H. Dang, D. Dreyer, P. W. O'Hearn, and J. Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Proc. CAV'20, Part II*, LNCS 12225, pp. 225–252, 2020.
[26] T. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Proc. VMCAI'04*, LNCS 2937, pp. 252–266, 2004.
[27] X. Rival and K. Yi. *Introduction to Static Analysis – An Abstract Interpretation Perspective*. MIT Press, 2020.
[28] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at Google. *CACM*, 61(4):58–66, Mar. 2018.
[29] G. Winskel. *The Formal Semantics of Programming Languages: an Introduction*. MIT press, 1993.
[30] G. Yorsh, T. Reps, and S. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Proc. TACAS'04*, LNCS 2988, pp. 530–545, 2004.