A Correctness and Incorrectness Program Logic

ROBERTO BRUNI, University of Pisa, Italy
ROBERTO GIACOBAZZI, University of Verona, Italy
ROBERTA GORI, University of Pisa, Italy
FRANCESCO RANZATO, University of Padova, Italy

Abstract interpretation is a well known and extensively used method to extract over-approximate program invariants by a sound program analysis algorithm. Soundness means that no program errors are lost and it is, in principle, guaranteed by construction. Completeness means that the abstract interpreter reports no false alarms for all possible inputs, but this is extremely rare because it requires a very precise analysis. We introduce a weaker notion of completeness, called *local completeness*, which requires that no false alarms are produced only relatively to some fixed program inputs. Based on this idea, we introduce a program logic, called LCL_A (Local Completeness Logic for an abstract domain A), for proving both the correctness and incorrectness of program specifications. Our proof system, which is parameterized by an abstract domain A, combines over- and under-approximating reasoning. In a provable triple $\vdash_A [p]\ \mathsf{c}\ [q]$, c is a program, q is an under-approximation of the strongest post-condition of c on input p such that their abstractions in A coincide. This means that q is never too coarse, namely, under some mild assumptions, *the abstract interpretation of* c *does not yield false alarms for the input p iff q has no alarm.* Therefore, proving $\vdash_A [p]\ \mathsf{c}\ [q]$ not only ensures that all the alarms raised in q are true ones, but also that if q does not raise alarms then c is correct. We also prove that if A is the straightforward abstraction making all program properties equivalent then our program logic coincides with O'Hearn's incorrectness logic, while for any other abstraction, contrary to the case of incorrectness logic, our logic can also establish program correctness.

CCS Concepts: • **Theory of computation → Logic and verification**; **Abstraction**; **Programming logic**; **Semantics and reasoning**; **Program analysis**; *Hoare logic*; Axiomatic semantics; **Abstraction**; **Program reasoning**.

Additional Key Words and Phrases: Abstract interpretation, abstract domain, program analysis, program verification, program logic, local completeness, best correct approximation, incorrectness logic.

1 INTRODUCTION

"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness" [Dijkstra 1972b]. This statement, given by E.W. Dijkstra in 1972 in his Turing Award lecture [Dijkstra 1972c], is universally valid and nowadays felt as an indispensable necessity and still a major challenge in modern information societies [Hoare 2003; Jones et al. 2006]. The idea of having programs that certify other programs is due to A.M. Turing [Turing 1989], and put forward by McCarthy [1962], Floyd [1967], and Hoare [1969] in the 1960s. The last 50 years have witnessed an incredible flourishing of formal methods and tools for achieving this ambitious goal. These include, among others: Certified compilers [Leroy 2006], certified analyzers [Jourdan et al. 2015], advanced type checkers [Pierce 2002], software model checkers [Ball et al. 2005], and abstract interpretation-based methods for static program analysis [Cousot 2021; Cousot and Cousot 1977; Giacobazzi and Ranzato 2022]. Proving program correctness means inferring an adequate invariant which has to be strong enough to imply the desired correctness property for our code. In program correctness proofs, the ambition to automate the inference of a "good" invariant immediately leads us to the need of

Authors' addresses: Roberto Bruni, University of Pisa, Pisa, Italy, bruni@di.unipi.it; Roberto Giacobazzi, University of Verona, Verona, Italy, roberto.giacobazzi@univr.it; Roberta Gori, University of Pisa, Pisa, Italy, gori@di.unipi.it; Francesco Ranzato, University of Padova, Padova, Italy, ranzato@math.unipd.it.

2023. Manuscript submitted to ACM

some sort of over-approximation, making tractable (e.g., decidable) problems that are otherwise intractable. The well known and inherent undecidability of all non-straightforward extensional properties of programs provides an intrinsic limitation in the use of approximated and decidable formal methods for proving program properties [Rice 1953] (see also [Cousot et al. 2018]). This is particularly clear in program analysis, where the necessary termination of the analysis algorithm may introduce false alarms. The soundness of a program analyser, which is guaranteed by construction in abstract interpretation, means that all true alarms (also called true positives) are caught, but it is often the case that false alarms (also called false positives) are reported. There exists a range of successful applications of formal methods for proving the absence of bugs in programs [Calcagno et al. 2015; Cousot 2021; Distefano et al. 2019; Hoare 1969; O'Hearn 2018; Rival and Yi 2020; Sadowski et al. 2018]. Of course, as in all verification systems, program analysis is credible when few false alarms are reported, ideally none. Completeness holds when no false alarm is ever raised, and this represents in many ways the *holy grail* of program analysis and verification [Jones et al. 2006].

Understanding whether an alarm corresponds to a true bug, i.e., indirectly whether our approximation method is complete, boils down to the challenge of proving some sort of program incorrectness. The same E.W. Dijkstra asserted in 1972 that *"Program testing can be used to show the presence of bugs, but never to show their absence"* [Dijkstra 1972a]. Proving that our program contains a true bug therefore means isolating those states that will eventually trigger the fault, i.e., will make our program violating a correctness assertion. This means propagating along the computation properties of states that are strong enough to isolate those states that will produce a fault. This is the key idea in O'Hearn [2020]'s incorrectness logic (IL) and its subsequent application to separation logic in [Raad et al. 2020]. By exploiting under-approximations, any violation exposed by such program analysis corresponds to a true alarm. This makes IL a credible support for code-review and test-driven software development, although a correctness condition can still be violated even when no alarm is reported by the analysis.

The tension between proving the absence of bugs and exhibiting their actual presence shaped the research in programming languages and software engineering for decades. Nevertheless, the problem is far from being solved and static reasoning should be extended to bug catching, rather than only for proving absence of bugs, as advocated by O'Hearn [2020].

The Problem

In this work we consider abstract interpretation [Cousot and Cousot 1977] as the reference theory to design and validate algorithms capable to infer over-approximate program invariants. The problem is how to guarantee that the approximation is precise enough to avoid false alarms. Next we explain why completeness is crucial to ensure precision of the analysis and why it is hard to achieve it. The ingredients are: an abstract domain A of abstract program properties, e.g., some properties of stores, which is connected with the concrete domain of program properties C by a pair of monotone functions $\alpha : C \longrightarrow A$ and $\gamma : A \longrightarrow C$, respectively the abstraction and concretization map, such that $A = \alpha(C)$ and for any concrete property $p \in C$, $p \subseteq A(p) \triangleq \gamma\alpha(p)$[1]. Program verification by abstract interpretation means that instead of verifying whether the strongest post-condition $\mathrm{post}[\mathsf{c}](p)$ for a program c and a pre-condition p, satisfies a correctness specification *spec*, we verify whether a sound abstract over-approximation $\mathrm{post}_A[\mathsf{c}] : A \longrightarrow A$ satisfies *spec*. Using denotational semantics symbols, the original verification problem can be stated more concisely as checking

[1] Abstract interpretation is more general than this [Cousot and Cousot 1977]. There are weaker abstract interpretation frameworks where only the concretization function is assumed to exist [Cousot and Cousot 1992]. We consider the stronger formulation based on Galois insertions because it is the most widely known and, as observed later, this provides enough mathematical structure to define the key notion of completeness in abstract interpretation.

if $[\![\mathsf{c}]\!]p \subseteq \mathit{spec}$ holds, while its abstract counterpart becomes the property $\gamma([\![\mathsf{c}]\!]^\sharp_A \alpha(p)) \subseteq \mathit{spec}$, where $[\![\mathsf{c}]\!]$ denotes the strongest post-condition function for c and $[\![\mathsf{c}]\!]^\sharp_A$ is called the *abstract interpretation of* c *on* A.

As observed above, the problem here is that computing $[\![\mathsf{c}]\!]^\sharp_A$ may degrade the approximation and make the computed abstract program property too weak to imply the desired specification. This may be a consequence of the fact that abstract interpretation analysis is done by composing functions. Consider the abstract interpretation of c on A defined as $[\![\mathsf{c}]\!]^A \triangleq \alpha \circ [\![\mathsf{c}]\!] \circ \gamma$, commonly called *best correct approximation* (bca) of c in A. This is the best possible approximate semantics, namely any abstract interpretation $[\![\mathsf{c}]\!]^\sharp_A$ is sound if and only if $[\![\mathsf{c}]\!]^A \subseteq [\![\mathsf{c}]\!]^\sharp_A$ [Cousot and Cousot 1979]. It is known that, in general, the composition of two bcas is not necessarily a bca itself. Consider the case of the abstract domain of integer intervals Int whose elements approximate any property $p \in \wp(\mathbb{Z})$ of the integer values that a variable x may assume by the least interval $\mathsf{Int}(p) = [a, b]$ such that $p \subseteq [a, b]$, where $a \leq b$, $a \in \mathbb{Z} \cup \{-\infty\}$ and $b \in \mathbb{Z} \cup \{+\infty\}$ and the commands

$$\mathsf{c}_1 \triangleq \textbf{if } \mathit{even}(x) \textbf{ then } 0 \textbf{ else } 1 \quad \text{and} \quad \mathsf{c}_2 \triangleq \textbf{if } \mathit{even}(x) \textbf{ then } x \textbf{ else } x + 1\ .$$

In this case, it turns out that $[\![\mathsf{c}_2]\!]^{\mathsf{Int}}([0, 1]) = [0, 2]$ and $[\![\mathsf{c}_1]\!]^{\mathsf{Int}} \circ [\![\mathsf{c}_2]\!]^{\mathsf{Int}}([0, 1]) = [\![\mathsf{c}_1]\!]^{\mathsf{Int}}([0, 2]) = [0, 1]$, while $([\![\mathsf{c}_1]\!] \circ [\![\mathsf{c}_2]\!])^{\mathsf{Int}}([0, 1]) = [0, 0]$ and $[0, 0] \subsetneq [0, 1]$. The lack of compositionality is the main cause of the interdependence between the precision of an abstract interpretation and the way programs are written [Bruni et al. 2020]. While it is obvious that if $[\![\mathsf{c}]\!]^\sharp_A \alpha(p)$ satisfies *spec* (i.e., $\gamma([\![\mathsf{c}]\!]^\sharp_A \alpha(p)) \subseteq \mathit{spec}$ holds) then the program is correct, it may happen that $[\![\mathsf{c}]\!]^\sharp_A \alpha(p)$ does not satisfy *spec* even if the program is correct, hence producing a false alarm. In this case, if $\gamma([\![\mathsf{c}]\!]^\sharp_A \alpha(p)) \not\subseteq \mathit{spec}$, we cannot conclude that the Hoare triple $\{p\}\ \mathsf{c}\ \{\mathit{spec}\}$ is not valid, because any witness in $\gamma([\![\mathsf{c}]\!]^\sharp_A \alpha(p)) \smallsetminus \mathit{spec}$ could be a *false alarm*.

Complete abstract interpretations, instead, do not raise false alarms. Technically, this also requires the assumption that the specification *spec* is expressible in A, namely that $\mathit{spec} = A(\mathit{spec})$ holds. For example, 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 $\mathsf{Int}(x \neq 0) = \mathbb{Z} \supsetneq \mathbb{Z} \smallsetminus \{0\}$. If *spec* is expressible in A, then completeness guarantees that a Hoare triple $\{p\}\ \mathsf{c}\ \{\mathit{spec}\}$ is valid iff $\gamma([\![\mathsf{c}]\!]^\sharp_A \alpha(p)) \subseteq \mathit{spec}$ holds. Thus, any complete abstract verification of *spec* by means of the (complete) abstract semantics $[\![\mathsf{c}]\!]^\sharp_A$ is the same as verifying *spec* with respect to the concrete semantics. Notably, when the abstract interpretation is complete then $[\![\mathsf{c}]\!]^\sharp_A$ always coincides with the bca $[\![\mathsf{c}]\!]^A$ [Giacobazzi et al. 2000]. According to a well-established definition [Cousot and Cousot 1977, 1979], completeness is a *global notion*, meaning that it involves *all* possible pre-conditions. More precisely, A is complete for a program c when $[\![\mathsf{c}]\!]^\sharp_A \circ \alpha = \alpha \circ [\![\mathsf{c}]\!]$, namely:

$$\text{for all } p \in C,\ [\![\mathsf{c}]\!]^\sharp_A \alpha(p) = \alpha([\![\mathsf{c}]\!]p) \text{ holds.}$$

This form of completeness is extremely hard to achieve. Giacobazzi et al. [2015] proved that for *any* non-straightforward abstraction A there always exists a program c for which any sound abstract interpretation of c on A yields at least one false alarm. Later, Bruni et al. [2020] showed that any program equivalence induced by an abstract interpreter built over a non-straightforward abstraction violates extensionality. Here, non-straightforward abstractions correspond to those abstract domains A that are able to distinguish at least two programs, i.e., there exist two programs c_1 and c_2 such that $[\![\mathsf{c}_1]\!]^\sharp_A \neq [\![\mathsf{c}_2]\!]^\sharp_A$, and A does not coincide with the identical abstraction, i.e., $[\![\cdot]\!]^\sharp_A \neq [\![\cdot]\!]$. In particular, it is known [Giacobazzi et al. 2015] that the main sources of incompleteness lie in the abstract interpretation of store assignments and Boolean guards. The case of Boolean guards is striking. The semantics of a Boolean guard is a predicate transformer $[\![\mathsf{b}?]\!] : \wp(\Sigma) \to \wp(\Sigma)$, where Σ denotes the set of concrete stores. Completeness of a Boolean guard b? in the abstract domain A means that for all $p \in \wp(\Sigma)$, $A([\![\mathsf{b}?]\!]p) = A([\![\mathsf{b}?]\!]A(p))$. Of course, because both branches of a conditional or loop statement guarded by b must be taken into account, the same condition applies to the negative test ¬b?. For example, Int is complete for a simple rectifier program, known as ReLU in neural networks:

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

even if its Boolean guards are not complete in Int. In fact, for the pre-condition $p \triangleq x \in \{-1, 1\}$, we have that:

$$\mathsf{Int}([\![x \geq 0?]\!]p) = \mathsf{Int}(\{1\}) = [1, 1] \subsetneq [0, 1] = \mathsf{Int}([\![x \geq 0?]\!][-1, 1]) = \mathsf{Int}([\![x \geq 0?]\!]\mathsf{Int}(p)),$$

meaning that Int is not complete for the guard $x \geq 0?$. This happens even if both guards $x < 0?$ and $x \geq 0?$ are expressible in Int as $x \in [-\infty, -1]$ and $x \in [0, +\infty]$, respectively. On the other hand, Int is instead trivially complete for both commands $x := 0$ and **skip** of ReLU. Hence, it is the lack of completeness of the abstract interpretation of the two guards $x < 0?$ and $x \geq 0?$ that prevents us to *inductively prove completeness* for the simple program ReLU. Abstraction refinement does not help either. The complete shell of Int, as described in [Giacobazzi et al. 1998, 2000], for the guards $x < 0?$ and $x \geq 0?$ blows up to nearly the concrete domain of all program stores. In fact, we should refine the interval abstraction by adding denotations that describe any pair of intervals (I_1, I_2) such that $I_1 \cap I_2 = \varnothing$ and $I_1 \cup I_2$ is not an interval where $I_1 \subseteq \mathbb{Z}_{<0}$ and $I_2 \subseteq \mathbb{Z}_{\geq 0}$. This would give us a complete abstract domain for $x < 0?$ and $x \geq 0?$ which is barely equivalent to the concrete domain, and therefore useless for a practical program verification.

Roadmap to Main Contributions

The main theme of the paper is to combine under- and over-approximations via abstract interpretation and the novel notion of local completeness to define a program logic whose triples either prove correctness or incorrectness.

After some background on abstract interpretation in Section 2, in Section 3 we give necessary and sufficient conditions that guarantee completeness of Boolean guards on an abstract domain A. These conditions require that both b? and ¬b? are expressible in the abstract domain and the same has to apply to the join of any two concretizations of abstract points below $\alpha([\![\mathrm{b}?]\!])$ and $\alpha([\![\neg\mathrm{b}?]\!])$. This requirement turns out to be very strong: for example, this condition allows us to prove that any Boolean guard on the interval abstraction Int is incomplete (cf. Example 3.7).

Since completeness is rare, in Section 4 we introduce *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 for some given subset of possible inputs. In the case of a guard b? with input p, local completeness amounts to checking whether $[\![\mathrm{b}?]\!]^\sharp_A \alpha(p) = \alpha([\![\mathrm{b}?]\!]p)$ holds *for that particular* $p \in C$. For example, any guard is locally complete for any input p that is expressible in the abstract domain. In our example of 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 in Int, so that we can inductively infer that ReLU is locally complete in Int with respect to any such input.

By means of local completeness we can prove the absence of false alarms for programs which are globally incomplete on a given abstract domain. As a simple example, consider the following program AbsVal for computing the absolute value of integer variables:

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

AbsVal is globally incomplete on the abstract domain Int. For instance, for a pre-condition $p \triangleq x \in \{-7, 7\}$, we have that $\mathsf{Int}([\![\mathsf{AbsVal}]\!]p) = [7, 7]$ while $\mathsf{Int}([\![\mathsf{AbsVal}]\!]\mathsf{Int}(p)) = \mathsf{Int}([\![\mathsf{AbsVal}]\!][-7, 7]) = [0, 7]$. Therefore, even if the pre-condition p does not include zero, an interval analysis of AbsVal yields a false alarm when checking the expressible specification $\mathit{spec} \triangleq x > 0$. Similarly to ReLU, this is not the case when the inputs all have the same sign, no matter if positive or

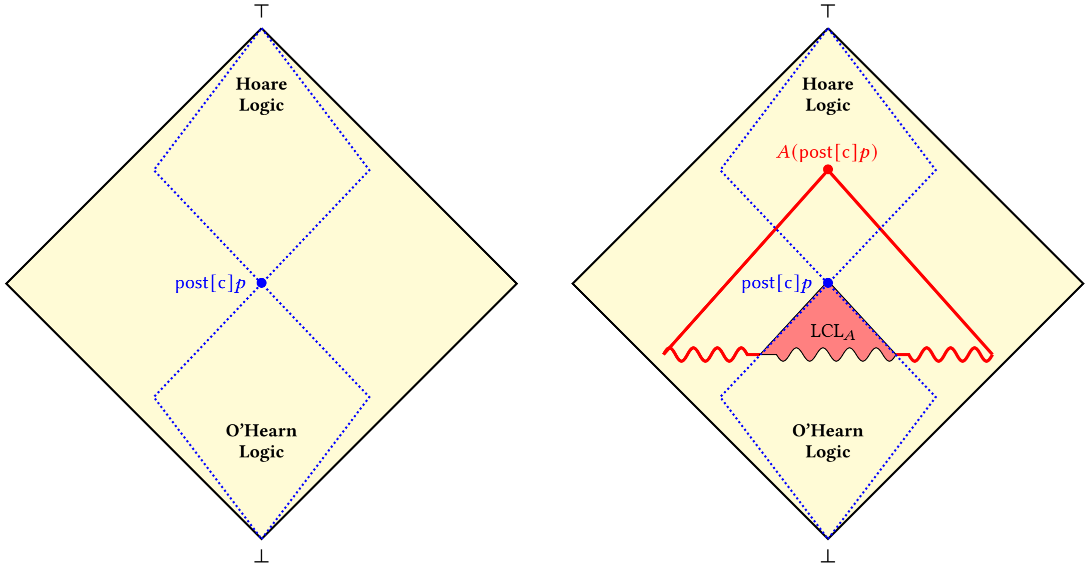


Fig. 1. Relation among LCL$_A$, O'Hearn's Incorrectness Logic and Hoare's partial correctness logic.

negative. Instead, a true alarm for *spec* can be raised only if the set of input values truly contains zero. This means that in order to use AbsVal in our code and not having false alarms in an interval analysis, e.g. within a loop, we need to make sure that AbsVal(x) will be called with x always having the same sign.

Our main contribution is in Section 5 where we present a logical proof system $\vdash_A$ called LCL$_A$ (Local Completeness Logic on A) for locally complete abstract interpretations parameterized on an arbitrary abstract domain A. Our assertions are Hoare-like triples $\vdash_A [p]\ \mathsf{c}\ [q]$ establishing that:

(i) q is an *under-approximation* of $\text{post}[\mathsf{c}]p$ (i.e., of $⟦\mathsf{c}⟧p$);
(ii) $\text{post}[\mathsf{c}]$ is *locally complete* for input p on A;
(iii) q and $\text{post}[\mathsf{c}]p$ 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 alarms raised by an abstract interpreter $⟦\mathsf{c}⟧^\sharp_A\alpha(p)$ for verifying any correctness specification *spec* that is expressible in A. The key rules in LCL$_A$ are:

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

$$\frac{p' \Rightarrow p \Rightarrow A(p') \quad \vdash_A [p']\ \mathsf{c}\ [q'] \quad q \Rightarrow q' \Rightarrow A(q)}{\vdash_A [p]\ \mathsf{c}\ [q]}\ (\text{relax})$$

The rule (transfer) checks that a basic expression e, such as a Boolean test b? or an assignment $x := \mathsf{a}$, is locally complete for p before inferring the output of post[e] on p as post-condition. The consequence rule (relax) is the key principle of LCL$_A$ and 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 $q \Rightarrow \text{post}[\mathsf{c}](p) \Rightarrow A(\text{post}[\mathsf{c}](p)) = A(q)$ holds. Likewise in the consequence rules of the reverse Hoare logic by de Vries and Koutavas [2011], incorrectness logic by O'Hearn [2020], and incorrectness separation logic by Raad et al. [2020], 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 classical Hoare logic, and this is needed because our post-conditions q are always under-approximations.

The key feature of (relax) is to constrain the under-approximating post-condition q to have *the same abstraction* as the strongest post-condition, which is needed for preserving local completeness. This twist is fundamental to guarantee that any triple derivable in LCL_A either proves correctness or incorrectness. Figure 1 illustrates the relations between different proof systems. On the left, it is shown that Hoare logic for correctness derives any over-approximation of $\mathrm{post}[\mathsf{c}](p)$ (i.e., any provable triple $\{p\}\ \mathsf{c}\ \{q\}$ is such that q belongs to the blue dotted upper diamond region) while O'Hearn logic for incorrectness is able to derive any under-approximation of $\mathrm{post}[\mathsf{c}](p)$ (i.e, any provable triple $\vdash_{\mathrm{IL}} [p]\ \mathsf{c}\ [q]$ is such that q belongs to the blue dotted lower diamond region). On the right, we show the interplay between approximations derived using locally complete abstract interpretation A (the red bordered region, comprising any q such that $A(q) = A(\mathrm{post}[\mathsf{c}]p))$ and LCL_A under-approximations (i.e., any provable triple $\vdash_A [p]\ \mathsf{c}\ [q]$ is such that q belongs to the red-filled region, still with $A(q) = A(\mathrm{post}[\mathsf{c}]p))$. We show that by the (relax) rule we can shrink the post-condition of any triples, up to some boundary that fringes the assertions of O'Hearn logic. This is made without much loss of precision: under mild conditions on *spec*, any approximation q in the red-filled region guarantees that if the over-approximation $A(q)$ reports some alarm, then q contains a *true* alarm and, conversely, any alarm in q is a true one. The key point is that any derivable 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 in Hoare logic: although $\{p\}\ \mathsf{c}\ \{spec\}$ is a valid triple, since post-conditions in Hoare triples can always be weakened, it is also possible to derive some triple $\{p\}\ \mathsf{c}\ \{q\}$ whose post-condition q includes some false alarm. Using IL, no true alarm 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 derivable triple* $\vdash_A [p]\ \mathsf{c}\ [q]$ of LCL_A will expose a true alarm witnessing that *spec* is violated. On the contrary, abstract interpretation in A, as well as Hoare logic, can also expose false alarms that cannot be distinguished from true ones. In IL, although it is possible to derive triples $\vdash_{\mathrm{IL}} [p]\ \mathsf{c}\ [q]$ where q exhibits some (true) alarms, since post-conditions can always be strengthened, other triples for p and c may have no alarm at all, e.g., $\vdash_{\mathrm{IL}} [p]\ \mathsf{c}\ [\mathbf{ff}]$.

We prove in Theorem 5.5 that LCL_A is sound with respect to the above properties (i–iii). To prove a result of logical completeness[2] of LCL_A for a program c, we add two more ingredients:

(1) as in incorrectness logic [O'Hearn 2020], an infinitary rule (limit) for iteration; and
(2) the assumption that all the basic instructions occurring in the program c of a provable triple $\vdash_A [p]\ \mathsf{c}\ [q]$ are globally complete on A.

In Section 6, we show that IL coincides with LCL_A when the straightforward abstraction A_{tr} which is unable to distinguish any two programs is considered. In fact A_{tr} is globally complete for every transfer function, hence for every program, and therefore the premises of the rule $(\mathsf{transfer})$ are always satisfied. We also prove that A_{tr} is the only abstraction A for which the proof system LCL_A can be logically complete for a Turing complete programming language.

[2] The term *logical* is used here to distinguish the standard notion of completeness for a proof system from the notion of completeness in abstract interpretation.

In Section 7, we observe that for iterative commands guarded by a Boolean guard b?, e.g., in while loops, it is not necessary to require the proof obligations of local completeness for b? at every iteration provided that local completeness is met when the loop invariant is reached. Therefore, to improve the expressiveness of the logical system in handling while programs, we introduce two additional sound rules that are specific to while loops.

In Section 8 we consider the possibility of refining the abstract domain to complete the proof when some proof obligations of local completeness are not met by the current abstraction. Proving a triple $\vdash_A [p]\ \mathsf{c}\ [q]$ in a refined domain can no longer guarantee the local completeness of $[\![\mathsf{c}]\!]^{\sharp}_{A}$ on input p, but we can still prove the local completeness of the best correct approximation $[\![\mathsf{c}]\!]^{A}$ so that $[\![\mathsf{c}]\!]^{A}\alpha(p) = \alpha([\![\mathsf{c}]\!]p) = \alpha(q)$ holds. From the viewpoint of program verification w.r.t. a correctness specification *spec* that is expressible in A, this means that we retain all the potential of LCL_A for finding bugs (i.e., any element in $q \setminus spec$) or proving correctness (when $q \subseteq spec$ holds) even if the abstract interpretation $[\![\mathsf{c}]\!]^{\sharp}_{A}\alpha(p)$ is not as precise as the bca $[\![\mathsf{c}]\!]^{A}\alpha(p)$.

Finally, Section 9 discusses some related work and Section 10 concludes by outlining some directions of future work expanding the ideas of this paper.

This paper is a full and revised version of the LICS 2021 paper [Bruni et al. 2021], extended to include all the technical proofs, further examples and some entirely novel contributions reported in: Section 4.1 concerning a local completeness characterization for Boolean guards; Section 7 concerning a relaxation of local completeness requirements for while loops; Section 8 concerning a program logic for best correct approximations.

2 BACKGROUND

As a matter of notation, given two sets X and Y, $X \smallsetminus Y$ denotes the set-difference between X and Y, while $X \subsetneq Y$ denotes strict inclusion. Given two functions $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ we denote with $g \circ f$, or simply gf, their composition. For $f : X \longrightarrow X$ and $n \in \mathbb{N}$ we let $f^n : X \longrightarrow X$ be defined inductively as usual: $f^0 \triangleq id_X$ and $f^{n+1} \triangleq f \circ f^n$, where id_X denotes the identity function on X (often abbreviated id) .

In ordered structures such as posets and complete lattices over a set of elements C, we typically use $\leq_C$ to denote a partial order relation, $\vee_C$ for least upper bound (lub), $\wedge_C$ for greatest lower bound (glb), $\top_C$ and $\bot_C$ for, respectively, greatest and least elements. We use $\wp(X)$ to denote the powerset complete lattice over a set X ordered by inclusion. If $f : X \longrightarrow Y$ then f is overloaded to denote its powerset lifting $f : \wp(X) \longrightarrow \wp(Y)$, where $f(S) \triangleq \{f(x) \mid x \in S\}$. If C is a complete lattice and $X \subseteq C$ then the Moore closure of X is defined as $\mathcal{M}(X) \triangleq \{\wedge_C Y \mid Y \subseteq X\}$, that is, $\mathcal{M}(X)$ is the least superset of X closed under glbs of its subsets (also called Moore closed); in particular, notice that $X \subseteq \mathcal{M}(X)$ and $\wedge_C \varnothing = \top_C \in \mathcal{M}(X)$. If C is a poset and $l, u \in C$ then $[l, u] \triangleq \{x \in C \mid l \leq_C x \leq_C u\}$ denotes the segment between l and u in C. For $f, g : C_1 \longrightarrow C_2$ between posets, $f \leq g$ denotes that for all $x \in C_1$, $f(x) \leq_{C_2} g(x)$, while f is monotone when $x \leq_{C_1} y$ implies $f(x) \leq_{C_2} f(y)$. A function f between complete lattices is additive (respectively co-additive) when f preserves arbitrary lubs (respectively glbs). The least fixpoint of a function $f : C \longrightarrow C$ on a poset C is denoted, when it exists, by $\mathrm{lfp}(f)$. Let us recall that if f is (Scott) continuous on a complete lattice then $\mathrm{lfp}(f) = \bigvee_C \{f^n(\bot) \mid n \in \mathbb{N}\}$.

2.1 Abstract Interpretation

2.1.1 Abstract Domains. If the concrete semantics of our programming language is specified on a given *concrete domain* of properties C, abstract interpretation [Cousot 2021; Cousot and Cousot 1977] is *the* method to specify abstract semantics, namely approximate semantics defined on an *abstract domain* of approximate program properties A. Concrete and abstract domains are typically complete lattices. This guarantees the existence of the basic lattice operators of join and meet used in the definition of concrete and abstract semantics. Since several 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 $\alpha : C \longrightarrow A$ and $\gamma : A \longrightarrow C$ forms a Galois connection (GC, a special case of an adjunction) when for all $c \in C$, $a \in A$, $\alpha(c) \leq_A a \Leftrightarrow c \leq \gamma(a)$ holds. In a GC the lattices C and A are called, respectively, concrete and abstract domain, and α and γ are called, respectively, abstraction and concretization maps. We only consider GCs such that $\alpha\gamma = id_A$, called Galois insertions (GIs), where α is surjective and γ is injective. Let us recall that α is additive, γ is co-additive, $\gamma\alpha$ is an (upper) closure operator, that is, $\gamma\alpha : C \longrightarrow C$ is a monotone, idempotent and extensive (i.e., $id_C \leq_C \gamma\alpha$ holds) function, and $\gamma(A) \subseteq C$ is Moore closed. Moreover, if $X \subseteq C$ is Moore closed then X can be viewed as an abstraction of C through the maps $\alpha = \lambda c. \wedge_C \{x \in X \mid c \leq_C x\}$ and $\gamma = id_X$. The class of abstract domains of C is given by $\mathrm{Abs}(C) \triangleq \{\langle A, \leq_A, \alpha, \gamma\rangle \mid \alpha : C \longrightarrow A,\ \gamma : A \longrightarrow C \text{ is a GI}\}$, and we write $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ to mean that A is an abstract domain related to C by the abstraction and concretization maps α and γ. When convenient, we simply use A in place of the function $\gamma\alpha : C \rightarrow C$, e.g., $\mathsf{Int}(\{-7, 7\}) = [-7, 7]$. For example, since γ is injective, a condition such as $\alpha(c) \leq_A \alpha(d)$ can be written as (and is equivalent to) $A(c) \leq A(d)$. An abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is called *strict* when $\gamma(\bot_A) = \bot$, and a concrete value $c \in C$ is *expressible* in A when $A(c) = c$, while if $c < A(c)$ 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, respectively, expressible and approximated in A. Given two abstract domains $A_{\alpha_A,\gamma_A}, B_{\alpha_B,\gamma_B} \in \mathrm{Abs}(C)$, B is a *refinement* of A, denoted by $B \preceq A$, when $\gamma_A(A) \subseteq \gamma_B(B)$ holds, i.e., when B is at least as expressive as A. An abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is *trivial* if: either (a) $\gamma\alpha = id_C$ holds, i.e., all the concrete values are expressible in A; or (b) $\gamma\alpha = \lambda x.\top_C$ holds, i.e., A is a singleton domain which can express the greatest element $\top_C$ only. A is called the *identity abstraction* in the case (a), and the *top abstraction* in the case (b).

2.1.2 *Correctness.*

Given an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ and a concrete operation $f : C \longrightarrow C$ (a generalization to n-ary functions of type $C^n \longrightarrow C$ can be easily done pointwise), an abstract function $f^\sharp : A \longrightarrow A$ is a *correct* (or *sound*) approximation (or abstract interpretation) of f when $\alpha f \leq f^\sharp \alpha$ 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., $\alpha(\mathrm{lfp}(f)) \leq \mathrm{lfp}(f^\sharp)$ holds. The *best correct approximation* (bca) of f in A is defined as the abstract function $f^A \triangleq \alpha f \gamma : A \longrightarrow A$. The term “best” is justified by the well-known fact [Cousot and Cousot 1979] that an abstract function $f^\sharp : A \longrightarrow A$ is a correct approximation of f iff $f^A \leq f^\sharp$ holds, so that f^A is the most precise, w.r.t. the pointwise ordering $\leq_A$, among the correct approximations of f on A.

2.1.3 *Completeness.*

The abstract function $f^\sharp$ is a *complete* approximation of f (or just complete) if $\alpha f = f^\sharp \alpha$ holds. The abstract domain A is called a *complete abstraction* for f if there exists a complete approximation $f^\sharp : A \longrightarrow A$ of f on the abstract domain A. Completeness intuitively encodes the greatest possible precision for an abstract function $f^\sharp$ defined on A, meaning that the abstract behaviour of $f^\sharp$ on A, i.e. $f^\sharp\alpha$, exactly matches the abstraction in A of the concrete behaviour of f, i.e. αf. In a complete approximation $f^\sharp$ the only loss of precision is due to the abstract domain and not to the definition of 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 $f^\sharp : A \longrightarrow A$ such that completeness $\alpha f = f^\sharp \alpha$ holds iff $\alpha f = \alpha f \gamma\alpha$ iff $(\gamma\alpha) f = (\gamma\alpha) f (\gamma\alpha)$ iff $Af = AfA$ iff $\gamma\alpha f = \gamma f^A \alpha$. Hence, the chance of defining a complete approximation $f^\sharp$ of f on some abstract domain A only depends upon the bca f^A of f in A, i.e., completeness is a property of the abstract domain only. Moreover, let us observe that any trivial abstract domain is always complete for any f. In the following, we say both “A is complete for

f" and "f is complete on A", and we write $\mathbb{C}^A(f)$ to denote that A is complete for f:

$$\mathbb{C}^A(f) \overset{\triangle}{\Leftrightarrow} A \circ f = A \circ f \circ A\,. \tag{1}$$

2.2 Regular Commands

Following O'Hearn [2020] for incorrectness logic, 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. The language is parametric on the syntax of basic expressions $\mathsf{e} \in \mathsf{Exp}$, 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, local variable primitives, etc. The term $\mathsf{r}_1;\mathsf{r}_2$ represents sequential composition, the term $\mathsf{r}_1 \oplus \mathsf{r}_2$ represents a nondeterministic choice command, and the term r^* is the Kleene iteration of r where r can be executed 0 or any finite number of times. As a shorthand, we write r^n for the sequence $\mathsf{r};\ldots;\mathsf{r}$ of n instances of r and let $\mathsf{Exp}(\mathsf{r})$ denote the set of basic expressions occurring in $\mathsf{r} \in \mathsf{Reg}$.

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

$$\begin{aligned}
[\![\mathsf{e}]\!]c &\triangleq (\!|\mathsf{e}|\!)c \\
[\![\mathsf{r}_1;\mathsf{r}_2]\!]c &\triangleq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]c) \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]c &\triangleq [\![\mathsf{r}_1]\!]c \vee [\![\mathsf{r}_2]\!]c \\
[\![\mathsf{r}^*]\!]c &\triangleq \bigvee\{[\![\mathsf{r}]\!]^n c \mid n \in \mathbb{N}\}
\end{aligned} \tag{2}$$

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

$$\begin{aligned}
[\![\mathsf{e}]\!]^\sharp_A a &\triangleq [\![\mathsf{e}]\!]^A a \\
[\![\mathsf{r}_1;\mathsf{r}_2]\!]^\sharp_A a &\triangleq [\![\mathsf{r}_2]\!]^\sharp_A([\![\mathsf{r}_1]\!]^\sharp_A a) \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^\sharp_A a &\triangleq [\![\mathsf{r}_1]\!]^\sharp_A a \vee_A [\![\mathsf{r}_2]\!]^\sharp_A a \\
[\![\mathsf{r}^*]\!]^\sharp_A a &\triangleq \bigvee_A\{([\![\mathsf{r}]\!]^\sharp_A)^n a \mid n \in \mathbb{N}\}
\end{aligned} \tag{3}$$

where it is worth emphasizing that as abstract semantics $[\![\mathsf{e}]\!]^\sharp_A$ of basic expressions we consider their bcas on A, namely $[\![\mathsf{e}]\!]^A \triangleq \alpha \circ [\![\mathsf{e}]\!] \circ \gamma$, i.e., we assume that no additional loss of precision is due to their interpretation. It is easy to check by structural induction that the abstract semantics in (3) is monotonic (provided that $[\![\mathsf{e}]\!]$ are monotone functions) and correct, i.e.,

$$\alpha[\![\mathsf{r}]\!] \leq [\![\mathsf{r}]\!]^\sharp_A \alpha. \tag{4}$$

Let us remark that (3) 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

$$\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^A a = \llbracket \mathsf{r}_1 \rrbracket^A a \vee_A \llbracket \mathsf{r}_2 \rrbracket^A a. \tag{5}$$

As observed above in Section 1, this property of preserving bcas, in general, does not hold for sequential composition and Kleene iteration: for example, $\llbracket \mathsf{r}_2 \rrbracket^A \circ \llbracket \mathsf{r}_1 \rrbracket^A$ is not guaranteed to be the bca $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^A$. On the other hand, it can be easily seen, by structural induction, that all the definitions in (3) preserve the property of being complete, meaning that if $\llbracket \mathsf{r}_1 \rrbracket^\sharp_A$, $\llbracket \mathsf{r}_2 \rrbracket^\sharp_A$, $\llbracket \mathsf{r} \rrbracket^\sharp_A$ are complete, then $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^\sharp_A$, $\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^\sharp_A$ and $\llbracket \mathsf{r}^* \rrbracket^\sharp_A$ are complete as well. In the following, as a shorthand, we write $\mathbb{C}^A(\mathsf{r})$ instead of $\mathbb{C}^A(\llbracket \mathsf{r} \rrbracket)$ to denote that A is complete for $\llbracket \mathsf{r} \rrbracket$.

In our proofs, we will exploit some standard properties of abstract and concrete semantics, as summarized by the following lemma.

Lemma 2.1. *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$*. For all* $\mathsf{r} \in \mathrm{Reg}$ *and* $c, d \in C$*:*

(a) $\llbracket \mathsf{r} \rrbracket^A \alpha(c) \leq_A \llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(c)$;
(b) $\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(c) = \alpha(\llbracket \mathsf{r} \rrbracket c) \Rightarrow \alpha(\llbracket \mathsf{r} \rrbracket c) = \alpha(\llbracket \mathsf{r} \rrbracket \gamma\alpha(c))$;
(c) $\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(c) \leq_A \alpha(c) \Rightarrow \llbracket \mathsf{r}^* \rrbracket^\sharp_A \alpha(c) = \alpha(c)$;
(d) $\llbracket \mathsf{r} \rrbracket^A \alpha(c) \leq_A \alpha(c) \Rightarrow \llbracket \mathsf{r}^* \rrbracket^A \alpha(c) = \alpha(c)$;
(e) $d \leq \llbracket \mathsf{r} \rrbracket c \Rightarrow \llbracket \mathsf{r}^* \rrbracket (c \vee d) = \llbracket \mathsf{r}^* \rrbracket c$.

Proof. Let us recall that $A = \gamma\alpha$.
(a): By correctness (4), and by definition of bca $\llbracket \mathsf{r} \rrbracket^A$.
(b): By (a), $\alpha(\llbracket \mathsf{r} \rrbracket c) \leq_A \alpha(\llbracket \mathsf{r} \rrbracket \gamma\alpha(c)) = \llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(c) \leq_A \llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(c) = \alpha(\llbracket \mathsf{r} \rrbracket c)$.
(c): Let us assume that $\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(c) \leq \alpha(c)$. The following property can be proved by an easy induction on $n \geq 1$:

$$\forall n \geq 1.\ \llbracket \mathsf{r} \rrbracket^{\sharp\,n}_A \alpha(c) \leq_A \alpha(c). \tag{‡}$$

Thus,

$$\begin{aligned} \llbracket \mathsf{r}^* \rrbracket^\sharp_A \alpha(c) = {} & \quad [\text{by definition}] \\ \textstyle\bigvee_A \{ \llbracket \mathsf{r} \rrbracket^{\sharp\,n}_A \alpha(c) \mid n \in \mathbb{N} \} = {} & \quad [\text{by (‡) and } \llbracket \mathsf{r} \rrbracket^{\sharp\,0}_A \alpha(c) = \alpha(c)] \\ \alpha(c). & \end{aligned}$$

(d): The proof is analogous to that of (c) and therefore omitted.
(e): Let us assume that $d \leq \llbracket \mathsf{r} \rrbracket c$. Then:

$$\begin{aligned} \llbracket \mathsf{r}^* \rrbracket c \leq {} & \quad [\text{by monotonicity of } \llbracket \mathsf{r}^* \rrbracket] \\ \llbracket \mathsf{r}^* \rrbracket (c \vee d) \leq {} & \quad [\text{by hyp. } d \leq \llbracket \mathsf{r} \rrbracket c] \\ \llbracket \mathsf{r}^* \rrbracket (c \vee \llbracket \mathsf{r} \rrbracket c) \leq {} & \quad [\text{as } \llbracket \mathsf{r} \rrbracket c \leq \llbracket \mathsf{r}^* \rrbracket c] \\ \llbracket \mathsf{r}^* \rrbracket (c \vee \llbracket \mathsf{r}^* \rrbracket c) = {} & \quad [\text{as } c \leq \llbracket \mathsf{r}^* \rrbracket c] \\ \llbracket \mathsf{r}^* \rrbracket (\llbracket \mathsf{r}^* \rrbracket c) = {} & \quad [\text{by idempotency of } \llbracket \mathsf{r}^* \rrbracket] \\ \llbracket \mathsf{r}^* \rrbracket c. & \end{aligned}$$

□

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

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

where a ranges over arithmetic expressions on integer values in $\mathbb{Z}$ and variables $x \in \mathit{Var}$, and b ranges over Boolean expressions in BExp including negation. Hence, a standard deterministic imperative language Imp (cf. [Winskel 1993]) can be defined using guarded branching and loop commands as syntactic sugar as follows (cf. [Kozen 1997, Section 2.2]):

$$\begin{aligned} \textbf{if } \mathsf{b} \textbf{ then } \mathsf{c}_1 \textbf{ else } \mathsf{c}_2 &\triangleq (\mathsf{b}?; \mathsf{c}_1) \oplus (\neg\mathsf{b}?; \mathsf{c}_2) \\ \textbf{while } \mathsf{b} \textbf{ do } \mathsf{c} &\triangleq (\mathsf{b}?; \mathsf{c})^* \,; \neg\mathsf{b}? \end{aligned} \tag{6}$$

The syntax of Imp commands is defined by the grammar below

$$\mathsf{Imp} \ni \mathsf{c} ::= \textbf{skip} \mid x := \mathsf{a} \mid \mathsf{c}; \mathsf{c} \mid \textbf{if } \mathsf{b} \textbf{ then } \mathsf{c} \textbf{ else } \mathsf{c} \mid \textbf{while } \mathsf{b} \textbf{ do } \mathsf{c}$$

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

A program store $\sigma : V \longrightarrow \mathbb{Z}$ is a total function from a finite set of variables of interest $V \subseteq \mathit{Var}$ to values and $\Sigma \triangleq V \longrightarrow \mathbb{Z}$ denotes the set of stores on the variables ranging in a set V that, for simplicity, is left implicit. This definition of stores allows us to compose logical proofs for regular commands by choosing a finite set of variables V which is large enough to include all the variables occurring in a finite set of commands. The concrete domain is $\mathbb{S} \triangleq \wp(\Sigma)$, ordered by inclusion.

The semantics $(\!|\cdot|\!) : \mathsf{Exp} \longrightarrow \mathbb{S} \longrightarrow \mathbb{S}$ of basic expressions is defined as follows:

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

where store update $\sigma[x \mapsto v]$ and the semantics of arithmetic expressions $\{\!|\mathsf{a}|\!\} : \Sigma \longrightarrow \mathbb{Z}$ and Boolean expressions $\{\!|\mathsf{b}|\!\} : \Sigma \longrightarrow \{\textbf{tt}, \textbf{ff}\}$ are 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 $(\!|\mathsf{b}?|\!)p = p \cap \mathsf{b}$ filters the concrete stores in p making b true. The usual strongest post-condition for r for a pre-condition $p \in \mathbb{S}$ is therefore $\mathrm{post}[\mathsf{r}]p \triangleq [\![\mathsf{r}]\!]p$. Analogously, we define $\mathrm{post}_A[\mathsf{r}]\alpha(p) \triangleq [\![\mathsf{r}]\!]^\sharp_A \alpha(p)$.

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., $p \in \wp(\mathbb{Z})$ represents the set $\{\sigma \in \Sigma \mid \sigma(x) \in p\} \in \mathbb{S}$. Accordingly, $\mathrm{Abs}(\wp(\mathbb{Z}))$ will represent $\mathrm{Abs}(\mathbb{S})$. For example, $\{-2, 2\}$ will be used to represent a verbose expression such as $x = -2 \vee x = 2$.

3 ON THE LIMITS OF (GLOBAL) COMPLETENESS

Giacobazzi et al. [2015, Theorem 4.5] proved 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. Giacobazzi et al. [2015] 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

$$\frac{}{\Vdash_A \mathbf{skip}} \qquad \frac{\mathbb{C}^A(x := \mathrm{a})}{\Vdash_A x := \mathrm{a}} \qquad \frac{\Vdash_A \mathrm{c}_1 \quad \Vdash_A \mathrm{c}_2}{\Vdash_A \mathrm{c}_1;\mathrm{c}_2} \qquad \frac{\mathbb{C}^A(\mathrm{b}) \quad \mathbb{C}^A(\neg\mathrm{b}) \quad \Vdash_A \mathrm{c}_1 \quad \Vdash_A \mathrm{c}_2}{\Vdash_A \mathbf{if}\ \mathrm{b}\ \mathbf{then}\ \mathrm{c}_1\ \mathbf{else}\ \mathrm{c}_2} \qquad \frac{\mathbb{C}^A(\mathrm{b}) \quad \mathbb{C}^A(\neg\mathrm{b}) \quad \Vdash_A \mathrm{c}}{\Vdash_A \mathbf{while}\ \mathrm{b}\ \mathbf{do}\ \mathrm{c}}$$

Fig. 2. A Basic Proof System $\Vdash_A$ for Global Completeness [Giacobazzi et al. 2015, Figure 5].

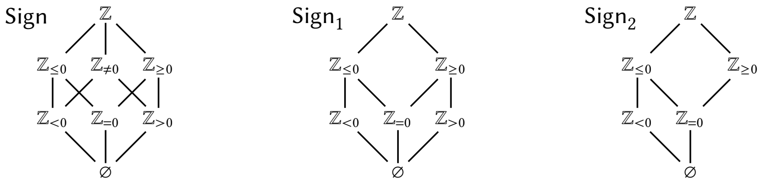


Fig. 3. Abstract Domains for Sign Analysis.

can prove the completeness of specific programs by structural induction on their syntax, through the basic proof system by Giacobazzi et al. [2015], recalled in Figure 2, where a proof of $\Vdash_A$ c entails the completeness of a program c in an abstract domain A. Hence, 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.

Example 3.1. As a simple example, consider the abstract domain Sign for sign analysis depicted in Figure 3 and the Imp program

$$\mathrm{c} \triangleq \mathbf{if}\ x < 0\ \mathbf{then}\ x := x * 2\ \mathbf{else}\ x := x * 3\ .$$

It turns out that all the guards and assignments occurring in c are complete on Sign, i.e., $\mathbb{C}^{\mathrm{Sign}}(x < 0?)$, $\mathbb{C}^{\mathrm{Sign}}(x \geq 0?)$, $\mathbb{C}^{\mathrm{Sign}}(x := x * 2)$ and $\mathbb{C}^{\mathrm{Sign}}(x := x * 3)$ hold. Thus, one can easily prove $\Vdash_{\mathrm{Sign}}$ c, entailing that c is complete on Sign. □

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 [Giacobazzi et al. 2015; Miné 2017; Ranzato 2020]), 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 $\mathbb{C}^A(\mathrm{b}?)$ and $\mathbb{C}^A(\neg\mathrm{b}?)$:

$$\forall p \in \mathbb{S}\,.\quad A(\llparenthesis \mathrm{b}? \rrparenthesis p) = A(\llparenthesis \mathrm{b}? \rrparenthesis A(p)) \quad \wedge \quad A(\llparenthesis \neg\mathrm{b}? \rrparenthesis p) = A(\llparenthesis \neg\mathrm{b}? \rrparenthesis A(p))\ ,$$

that is,

$$\forall p \in \mathbb{S}\,.\quad A(p \cap \mathrm{b}) = A(A(p) \cap \mathrm{b}) \quad \wedge \quad A(p \cap \neg\mathrm{b}) = A(A(p) \cap \neg\mathrm{b})\ . \tag{7}$$

The term *global* in the section title refers to the universal quantification over any possible set S of stores in (7), 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 $\mathbb{C}^A(\mathrm{b}?)$ and $\mathbb{C}^A(\neg\mathrm{b}?)$. We first observe that when the functions $[\![\mathrm{b}?]\!]$ and $[\![\neg\mathrm{b}?]\!]$ are complete in a strict abstract domain A, then b and ¬b are expressible in A, that is, $A(\mathrm{b}) = \mathrm{b}$ and $A(\neg\mathrm{b}) = \neg\mathrm{b}$ hold.

Lemma 3.2. *Let $A \in \mathrm{Abs}(\mathbb{S})$ be strict. If $\mathbb{C}^A(\mathrm{b}?)$ and $\mathbb{C}^A(\neg\mathrm{b}?)$ hold, then* b *and* ¬b *are expressible in A.*

Proof. Let us show the contrapositive, so we assume that either $A(\mathrm{b}) \neq \mathrm{b}$ or $A(\neg\mathrm{b}) \neq \neg\mathrm{b}$. Since, for all $p \in \mathbb{S}$, $p \subseteq A(p)$ holds, we have that $\mathrm{b} \subsetneq A(\mathrm{b})$ or $\neg\mathrm{b} \subsetneq A(\neg\mathrm{b})$, so that either $A(\mathrm{b}) \cap \neg\mathrm{b} \neq \varnothing$ or $A(\neg\mathrm{b}) \cap \mathrm{b} \neq \varnothing$. Assume that

$A(\mathrm{b}) \cap \neg\mathrm{b} \neq \varnothing$ holds. Then $A(\mathrm{b} \cap \neg\mathrm{b}) = A(\varnothing) = \varnothing$ while $\varnothing \subsetneq A(\mathrm{b}) \cap \neg\mathrm{b} \subseteq A(A(\mathrm{b}) \cap \neg\mathrm{b})$, so that $\mathbb{C}^A(\neg\mathrm{b})$ does not hold. The case $A(\neg\mathrm{b}) \cap \mathrm{b} \neq \varnothing$ is symmetric. □

Furthermore, when b and ¬b are both expressible in A, it turns out that the completeness of $[\![\mathrm{b}?]\!]$ and $[\![\neg\mathrm{b}?]\!]$ boils down to a co-additivity condition for the abstraction map α, or, equivalently, an additivity condition for the concretization map γ. This is clearly a way too strong requirement in abstract interpretation as co-additive abstraction maps imply that the abstract domain is (isomorphic to) a complete sublattice of the concrete domain, namely it is a disjunctive abstraction [Giacobazzi and Ranzato 1996, 1998].

Lemma 3.3. *Let* b *and* ¬b *be expressible in* $A_{\alpha,\gamma} \in \mathrm{Abs}(\mathbb{S})$. *Then,* $\mathbb{C}^A(\mathrm{b}?)$ *and* $\mathbb{C}^A(\neg\mathrm{b}?)$ *hold iff*

$$\forall p \in \mathbb{S}\,.\quad \alpha(p \cap \mathrm{b}) = \alpha(p) \wedge_A \alpha(\mathrm{b}) \quad \wedge \quad \alpha(p \cap \neg\mathrm{b}) = \alpha(p) \wedge_A \alpha(\neg\mathrm{b}). \tag{8}$$

Proof. We show that (7) is equivalent to (8). Observe that for all $p \in \mathbb{S}$,

$$\begin{aligned} \alpha(\gamma\alpha(p) \cap \mathrm{b}) = {} & \quad [\text{because b is expressible in } A] \\ \alpha(\gamma\alpha(p) \cap \gamma\alpha(\mathrm{b})) = {} & \quad [\text{by GI, } \gamma \text{ is co-additive}] \\ \alpha\gamma(\alpha(p) \wedge_A \alpha(\mathrm{b})) = {} & \quad [\text{by GI, } \alpha\gamma = id] \\ \alpha(p) \wedge_A \alpha(\mathrm{b}). & \end{aligned}$$

Thus, when b and ¬b are expressible, it turns out that $\alpha(\gamma\alpha(p) \cap \mathrm{b}) = \alpha(p \cap \mathrm{b})$ iff $\alpha(p) \wedge_A \alpha(\mathrm{b}) = \alpha(p \cap \mathrm{b})$. Symmetrically, $\alpha(\gamma\alpha(p) \cap \neg\mathrm{b}) = \alpha(p \cap \neg\mathrm{b})$ iff $\alpha(p) \wedge_A \alpha(\neg\mathrm{b}) = \alpha(p \cap \neg\mathrm{b})$. We have therefore proved (7) ⇔ (8). □

The next characterization result provides an effective way to check whether an abstract domain A is complete w.r.t. a Boolean guard b. It amounts to check that b and ¬b are both expressible in A and that the union of the concretizations of any two abstract points in A below, respectively, $\alpha(\mathrm{b})$ and $\alpha(\neg\mathrm{b})$, is also expressible in A (see Example 3.5).

Theorem 3.4 (Complete Guards). *Let* b *and* ¬b *be expressible in* $A_{\alpha,\gamma} \in \mathrm{Abs}(\mathbb{S})$. *Then,* $\mathbb{C}^A(\mathrm{b}?)$ *and* $\mathbb{C}^A(\neg\mathrm{b}?)$ *hold iff*

$$\forall a_1, a_2 \in A.\ \big(\, a_1 \leq_A \alpha(\mathrm{b}) \ \wedge\ a_2 \leq_A \alpha(\neg\mathrm{b}) \ \Rightarrow\ \gamma(a_1 \vee_A a_2) = \gamma(a_1) \cup \gamma(a_2)\,\big). \tag{9}$$

Proof. Let us first show that $\mathbb{C}^A(\mathrm{b}) \wedge \mathbb{C}^A(\neg\mathrm{b})$ is equivalent to condition (9). We know that $\gamma\alpha(\mathrm{b}) = \mathrm{b}$ and $\gamma\alpha(\neg\mathrm{b}) = \neg\mathrm{b}$. By co-additivity of γ, for any $p \in \mathbb{S}$, we have that

$$\gamma(\alpha(p) \wedge_A \alpha(\mathrm{b})) = \gamma\alpha(p) \cap \mathrm{b} \quad \text{and} \quad \gamma(\alpha(p) \wedge_A \alpha(\neg\mathrm{b})) = \gamma\alpha(p) \cap \neg\mathrm{b}. \tag{§}$$

By Lemma 3.3, to prove our statement we show that (8) ⇔ (9) holds.
(⇒) By contraposition we prove that if (9) is not satisfied then (8) does not hold. Let us assume that there exist $a_1 \leq_A \alpha(\mathrm{b})$ and $a_2 \leq_A \alpha(\neg\mathrm{b})$ such that $\gamma(a_1 \vee_A a_2) \supsetneq \gamma(a_1) \cup \gamma(a_2)$, so that there exists $\sigma \in \gamma(a_1 \vee_A a_2)$ such that $\sigma \notin \gamma(a_1) \cup \gamma(a_2)$. Clearly, either $\sigma \in \mathrm{b}$ or $\sigma \in \neg\mathrm{b}$. Without loss of generality we assume that $\sigma \in \mathrm{b}$ (the case $\sigma \in \neg\mathrm{b}$ is

symmetric). Let $p \triangleq \gamma(a_1) \cup \gamma(a_2)$ so that:

$$\begin{aligned}
\gamma\alpha(p \cap \mathrm{b}) = {} & \quad [\text{by definition of } p] \\
\gamma\alpha((\gamma(a_1) \cup \gamma(a_2)) \cap \mathrm{b}) = {} & \quad [\text{by distributivity}] \\
\gamma\alpha((\gamma(a_1) \cap \mathrm{b}) \cup (\gamma(a_2) \cap \mathrm{b})) = {} & \quad [\text{by } \gamma(a_1) \subseteq \gamma\alpha(\mathrm{b}) = \mathrm{b} \text{ and } \gamma(a_2) \subseteq \gamma\alpha(\neg\mathrm{b}) = \neg\mathrm{b}] \\
\gamma\alpha\gamma(a_1) = {} & \quad [\text{by GI}, \alpha\gamma = id] \\
\gamma(a_1). &
\end{aligned}$$

Thus, $\sigma \notin \gamma\alpha(p \cap \mathrm{b}) = \gamma(a_1)$ because $\sigma \notin \gamma(a_1) \cup \gamma(a_2)$. Moreover,

$$\begin{aligned}
\gamma(\alpha(p) \wedge_A \alpha(\mathrm{b})) = {} & \quad [\text{by } (\S)] \\
\gamma\alpha(p) \cap \mathrm{b} = {} & \quad [\text{by definition of } p] \\
\gamma(\alpha(\gamma(a_1) \cup \gamma(a_2))) \cap \mathrm{b} = {} & \quad [\text{by GI}, \alpha \text{ is additive}] \\
\gamma(\alpha\gamma(a_1) \vee_A \alpha\gamma(a_2)) \cap \mathrm{b} = {} & \quad [\text{by GI}, \alpha\gamma = id] \\
\gamma(a_1 \vee_A a_2) \cap \mathrm{b}, &
\end{aligned}$$

so that $\sigma \in \gamma(\alpha(p) \wedge_A \alpha(\mathrm{b}))$ because we have assumed $\sigma \in \gamma(a_1 \vee_A a_2)$ and $\sigma \in \mathrm{b}$. Thus, $\gamma\alpha(p \cap \mathrm{b}) \neq \gamma(\alpha(p) \wedge_A \alpha(\mathrm{b}))$. Since γ is injective, $\alpha(p \cap \mathrm{b}) \neq \alpha(p) \wedge_A \alpha(\mathrm{b})$, i.e., (8) does not hold.
($\Leftarrow$) By contraposition, we prove that if (8) fails then (9) does not hold. If (8) does not hold, there must exist some $p \in \mathbb{S}$ such that

$$\alpha(p \cap \mathrm{b}) <_A \alpha(p) \wedge_A \alpha(\mathrm{b}) \text{ or } \alpha(p \cap \neg\mathrm{b}) <_A \alpha(p) \wedge_A \alpha(\neg\mathrm{b}).$$

Thus, by applying γ which, by GI, is injective, $\gamma\alpha(p \cap \mathrm{b}) \subsetneq \gamma(\alpha(p) \wedge_A \alpha(\mathrm{b}))$ or $\gamma\alpha(p \cap \neg\mathrm{b}) \subsetneq \gamma(\alpha(p) \wedge_A \alpha(\neg\mathrm{b}))$. In turn, since γ is co-additive and b, ¬b are expressible in A, $\gamma\alpha(p \cap \mathrm{b}) \subsetneq \gamma(\alpha(p)) \cap \mathrm{b}$ or $\gamma\alpha(p \cap \neg\mathrm{b}) \subsetneq \gamma(\alpha(p)) \cap \neg\mathrm{b}$. Hence, we obtain that

$$\gamma\alpha(p \cap \mathrm{b}) \cup \gamma\alpha(p \cap \neg\mathrm{b}) \subsetneq \gamma(\alpha(p)). \tag{†}$$

Let us now consider $a_1 \triangleq \alpha(p \cap \mathrm{b})$ and $a_2 \triangleq \alpha(p \cap \neg\mathrm{b})$. Then,

$$\begin{aligned}
\gamma(a_1 \vee_A a_2) = {} & \quad [\text{by definition}] \\
\gamma(\alpha(p \cap \mathrm{b}) \vee_A \alpha(p \cap \neg\mathrm{b})) = {} & \quad [\text{by GI}, \alpha \text{ is additive}] \\
\gamma(\alpha((p \cap \mathrm{b}) \cup (p \cap \neg\mathrm{b}))) = {} & \quad [\text{by distributivity}] \\
\gamma(\alpha(p)) \supsetneq {} & \quad [\text{by } (\dagger)] \\
\gamma\alpha(p \cap \mathrm{b}) \cup \gamma\alpha(p \cap \neg\mathrm{b}) = {} & \quad [\text{by definition}] \\
\gamma(a_1) \cup \gamma(a_2), &
\end{aligned}$$

thus proving that (9) does not hold. □

Let us illustrate through some examples how Theorem 3.4 can be applied to check the completeness of some guards.

Example 3.5. Let $\mathrm{Sign}, \mathrm{Sign}_1, \mathrm{Sign}_2 \in \mathrm{Abs}(\wp(\mathbb{Z}))$ be the abstract domains depicted in Figure 3 and consider the programs AbsVal and ReLU defined in Section 1.
In Sign, all expressible Boolean guards are complete and the completeness of both the programs AbsVal and ReLU can be proved inductively.

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 (9). 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 $\mathbb{Z}_{\neq 0} \notin \mathsf{Sign}_1$. 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 AbsVal are complete in Sign_2, only ReLU can be inductively proved to be complete, because all its basic expressions are complete in Sign_2. In the case of AbsVal instead, the assignment $x := -x$ is not complete, e.g. for $p \triangleq x > 0$ (the fact that such input will never be provided to that branch of code is irrelevant). □

It is worth remarking that condition (9) of Theorem 3.4 suggests a strategy to make a given abstract domain A complete for b and ¬b: We can either add to A the concrete values $\gamma(a_1) \cup \gamma(a_2)$ that are missing in A or remove either a_1 or a_2 from A when $\gamma(a_1) \cup \gamma(a_2)$ is not in A.

Example 3.6. Consider the domain $\mathsf{Int}_{\mathbb{N}} \in \mathrm{Abs}(\wp(\mathbb{N}))$ of (possibly infinite) intervals on natural numbers $\mathbb{N}$ [Cousot and Cousot 1977] and assume we are interested in testing equality to 0. The Boolean guards $x = 0?$ and $x \neq 0?$ are expressible in $\mathsf{Int}_{\mathbb{N}}$ as the intervals $[0, 0]$ and $[1, \infty]$, respectively. However, $\varnothing$ is the only (trivial) interval below $[0, 0]$ and, besides $\varnothing$, any element of the form $[a, b]$, with $1 < a \leq b$, is below $[1, \infty]$. Theorem 3.4 requires that any element in the set $\{[0, 0] \cup [a, b] \mid 1 < a \leq b\}$ is also exactly represented. Therefore, an abstract domain which is complete for the guards $x = 0?$ and $x \neq 0?$ can be obtained by adding to $\mathsf{Int}_{\mathbb{N}}$ all the elements in $\{[0, 0] \cup [a, b] \mid 1 < a \leq b\}$. Observe that $\mathsf{Int}_{\mathbb{N}} \cup \{[0, 0] \cup [a, b] \mid 1 < a \leq b\}$ is Moore closed (see Section 2.1.1) and therefore it is an abstract domain of $\wp(\mathbb{N})$ (cf. Section 2.1.1). Note that only the guards $x = 0?$ and $x \neq 0?$ are complete on this refined abstract domain. □

Because all interesting programs include 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, similar to Example 3.6, shows that this phenomenon can be a major drawback of refining an abstract domain in order to achieve completeness for the Boolean guards occurring in a given program.

Example 3.7. Consider the abstract domain $\mathsf{Int} \in \mathrm{Abs}(\wp(\mathbb{Z}))$ of integer intervals [Cousot and Cousot 1977]. The only Boolean guards b such that both b and ¬b are expressible in Int are the infinite intervals $[-\infty, k]$ and $[k, \infty]$, for some $k \in \mathbb{Z}$, 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 (9) of Theorem 3.4 is not satisfied. As an example, let us fix $k = -1$, i.e. $\mathsf{b} = [-\infty, -1]$ and, correspondingly, $\neg\mathsf{b} = [0, \infty]$. Condition (9) of Theorem 3.4 would require the presence of all the concrete joins $[n_1, n_2] \cup [m_1, m_2]$ with $n_1 \leq n_2 < 0 \leq m_1 \leq m_2$, because $[n_1, n_2] \leq_{\mathsf{Int}} [-\infty, -1]$ and $[m_1, m_2] \leq_{\mathsf{Int}} [0, \infty]$, but these joins are not intervals, unless $n_2 = 0$ and $m_1 = 1$. If we add all such joins $[n_1, n_2] \cup [m_1, m_2]$ to Int, we obtain a Moore closed subset of $\wp(\mathbb{Z})$ and therefore we achieve a complete abstraction for the guards $x < 0?$ and $x \geq 0?$.
Even a basic guard such as $\mathsf{b} \triangleq \mathbb{Z}_{=0} = [0, 0]$ would need its complement $\neg\mathsf{b} = \mathbb{Z}_{\neq 0} = [-\infty, -1] \cup [1, \infty]$ 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, notice that the intersection of ¬b with any interval $[n, m]$ with $n < 0 < m$ is $[n, -1] \cup [1, m]$, and since abstract domains are Moore closed, all these sets must be included as well. □

4 LOCAL COMPLETENESS

Section 3 shows that the standard notion of (global) completeness (1) 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 considering completeness only along certain program executions.

Definition 4.1 (Local Completeness). An abstract domain $A \in \mathrm{Abs}(C)$ is *locally complete* for $f : C \to C$ at a concrete value $c \in C$, written $\mathbb{C}^A_c(f)$, if the following condition holds:

$$\mathbb{C}^A_c(f) \overset{\triangle}{\Leftrightarrow} Af(c) = AfA(c). \qquad \square$$

Accordingly, ordinary completeness as defined by (1) is also referred to as *global completeness*. Note that global completeness amounts to the universal quantification of local completeness, in the sense that $\mathbb{C}^A(f) \Leftrightarrow \forall c \in C.\ \mathbb{C}^A_c(f)$. Let us also 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 1, in program analysis it may well happen that $⟦\mathsf{r}⟧$ is not globally complete w.r.t. the abstract domain A but it is locally complete for a particular class of inputs p (that is a value in the concrete domain of powerset of stores). In the following, we write $\mathbb{C}^A_p(\mathsf{r})$ instead of the more verbose $\mathbb{C}^A_p(⟦\mathsf{r}⟧)$.

Example 4.2. Consider the following 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 $⟦0 < x?⟧$ is not globally complete (see Example 3.7), it is locally complete for any set $p \in \wp(\mathbb{Z})$ satisfying one of the following conditions:

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

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), meaning that its abstract interpretation in Int will not lose precision. For example, if $p = \{0, 1, 4\}$, condition (3) holds and we have that $\mathsf{Int}(⟦\mathsf{c}⟧p) = \mathsf{Int}(\{-1, 0, 2\}) = [-1, 2]$ and $\mathsf{Int}(⟦\mathsf{c}⟧\mathsf{Int}(p)) = \mathsf{Int}(⟦\mathsf{c}⟧[0, 4]) = \mathsf{Int}(\{-1, 0, 1, 2\}) = [-1, 2]$. As an example of local incompleteness, if $p = \{0, 4\}$ we have that $\mathsf{Int}(⟦\mathsf{c}⟧p) = \mathsf{Int}(\{0, 2\}) = [0, 2]$, but $\mathsf{Int}(⟦\mathsf{c}⟧\mathsf{Int}(p)) = [-1, 2]$. □

Let us remark that, with respect to 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 4.3. Consider a composition $\mathsf{c};\mathsf{c}$, where c is defined in Example 4.2. Int is locally complete for c on the input property $p = \{2, 6\}$, because condition (1) of Example 4.2 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 $\mathsf{Int}(⟦\mathsf{c};\mathsf{c}⟧\mathsf{Int}(\{2, 6\})) = \mathsf{Int}(⟦\mathsf{c};\mathsf{c}⟧[2, 6]) = \mathsf{Int}(⟦\mathsf{c}⟧[0, 4]) = \mathsf{Int}(\{-1, 0, 1, 2\}) = [-1, 2]$. □

4.1 Locally Complete Boolean Guards

By focussing on Boolean guards, we can characterize the local completeness of both positive b? and negative ¬b? branches of a conditional statement $\mathsf{b} \in \mathsf{BExp}$ for a given input store property.

Theorem 4.4 (Locally Complete Guards). *Let* $\mathsf{b} \in \mathsf{BExp}$, $A \in \mathrm{Abs}(\mathbb{S})$ *and* $p \in \mathbb{S}$. *Then, A is locally complete for both* $⟦\mathsf{b}?⟧$ *and* $⟦\neg\mathsf{b}?⟧$ *on p iff* $(A(p \cap \mathsf{b}) \cap \mathsf{b}) \cup (A(p \cap \neg\mathsf{b}) \cap \neg\mathsf{b}) \in A$.

Proof. ($\Rightarrow$): We assume that $\mathbb{C}^A_p(\mathrm{b})$ and $\mathbb{C}^A_p(\neg\mathrm{b})$ and prove that $(A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})$ is expressible in A. We have that

$$\begin{aligned}
A((A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})) \subseteq &\quad \text{[by monotonicity]}\\
A(A(p \cap \mathrm{b}) \cup A(p \cap \neg\mathrm{b})) = &\quad \text{[by GI, } \alpha \text{ is additive and } A(A(q) \cup A(w)) = A(q \cup w) \text{ for any } q, w]\\
A((p \cap \mathrm{b}) \cup (p \cap \neg\mathrm{b})) = &\quad \text{[by set theoretic properties]}\\
A(p), &
\end{aligned}$$

so that, by exploiting the hypothesis of local completeness of $⟦\mathrm{b}?⟧$ on p,

$$A((A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})) \cap \mathrm{b} \quad \subseteq \quad A(p) \cap \mathrm{b} \quad \subseteq \quad A(A(p) \cap \mathrm{b}) \cap \mathrm{b} \quad = \quad A(p \cap \mathrm{b}) \cap \mathrm{b}.$$

Likewise,

$$A((A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})) \cap \neg\mathrm{b} \quad \subseteq \quad A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b}.$$

Thus,

$$\begin{aligned}
A((A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})) = &\quad \text{[by set properties]}\\
\big(A((A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})) \cap \mathrm{b}\big) \cup \big(A((A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})) \cap \neg\mathrm{b}\big) \subseteq &\quad \text{[by above inclusions]}\\
(A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b}), &
\end{aligned}$$

meaning that $(A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b}) \in A$.
($\Leftarrow$): For the converse implication, we assume that $(A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b}) \in A$ and prove that both $\mathbb{C}^A_p(\mathrm{b})$ and $\mathbb{C}^A_p(\neg\mathrm{b})$ hold. It turns out that $p = (p \cap \mathrm{b}) \cup (p \cap \neg\mathrm{b}) \subseteq (A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})$.
Therefore, by exploiting the hypothesis, we obtain that

$$A(p) \quad \subseteq \quad A((A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b})) \quad = \quad (A(p \cap \mathrm{b}) \cap \mathrm{b}) \cup (A(p \cap \neg\mathrm{b}) \cap \neg\mathrm{b}).$$

Thus, $A(p) \cap \mathrm{b} \subseteq A(p \cap \mathrm{b}) \cap \mathrm{b} \subseteq A(p \cap \mathrm{b})$, in turn entailing that local completeness $A(A(p) \cap \mathrm{b}) = A(p \cap \mathrm{b})$ holds. Analogously, we obtain that $A(A(p) \cap \neg\mathrm{b}) = A(p \cap \neg\mathrm{b})$. □

Example 4.5. Let us apply Theorem 4.4 to show that for the interval abstraction Int, the Boolean guard $0 < x$ is locally complete for any $p \in \wp(\mathbb{Z})$ such that: (1) $p \subseteq \mathbb{Z}_{>0}$, or (2) $p \subseteq \mathbb{Z}_{\leq 0}$, or (3) $\{0, 1\} \subseteq p$. In fact, it turns out that positive and negative branches $⟦x > 0?⟧$ and $⟦x \leq 0?⟧$ are both locally complete on p iff

$$(\mathsf{Int}(p \cap x > 0) \cap x > 0) \cup (\mathsf{Int}(p \cap x \leq 0) \cap x \leq 0) \in \mathsf{Int}. \tag{$*$}$$

Since $x > 0$ and $x \leq 0$ both belong to Int, condition ($*$) boils down to

$$\mathsf{Int}(p \cap x > 0) \cup \mathsf{Int}(p \cap x \leq 0) \in \mathsf{Int}. \tag{$\star$}$$

Now, it is just a matter of noticing that ($\star$) holds iff either $\mathsf{Int}(p \cap x \leq 0)$ is empty (condition (1)), or $\mathsf{Int}(p \cap x > 0)$ is empty (condition (2)) or $\mathsf{Int}(p \cap x \leq 0)$ and $\mathsf{Int}(p \cap x > 0)$ are contiguous nonempty intervals, and this happens iff p includes both cut points 0 and 1 of the guard $0 < x$. □

Likewise Theorem 3.4 can be used to construct domains that are complete for some Boolean guards, Theorem 4.4 gives a way to construct domains that are locally complete for some Boolean guards. The next example shows that, as

$$\frac{\mathbb{C}^A_p(\mathsf{e})}{\vdash_A [p]\ \mathsf{e}\ [[\![\mathsf{e}]\!]p]}\ (\text{transfer}) \qquad \frac{p' \leq p \leq A(p') \quad \vdash_A [p']\ \mathsf{r}\ [q'] \quad q \leq q' \leq A(q)}{\vdash_A [p]\ \mathsf{r}\ [q]}\ (\text{relax})$$

$$\frac{\vdash_A [p]\ \mathsf{r}_1\ [w] \quad \vdash_A [w]\ \mathsf{r}_2\ [q]}{\vdash_A [p]\ \mathsf{r}_1; \mathsf{r}_2\ [q]}\ (\text{seq}) \qquad \frac{\vdash_A [p]\ \mathsf{r}_1\ [q_1] \quad \vdash_A [p]\ \mathsf{r}_2\ [q_2]}{\vdash_A [p]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [q_1 \vee q_2]}\ (\text{join})$$

$$\frac{\vdash_A [p]\ \mathsf{r}\ [w] \quad \vdash_A [p \vee w]\ \mathsf{r}^*\ [q]}{\vdash_A [p]\ \mathsf{r}^*\ [q]}\ (\text{rec}) \qquad \frac{\vdash_A [p]\ \mathsf{r}\ [q] \quad q \leq A(p)}{\vdash_A [p]\ \mathsf{r}^*\ [p \vee q]}\ (\text{iterate})$$

Fig. 4. The Proof System LCL_A.

expected, making an abstract domain locally complete for some guards is less demanding than making it (globally) complete for the same guards.

Example 4.6. In Example 3.7 we have shown that in order to make the domain Int globally complete for the guards $\mathsf{b} \triangleq x < 0 = x \in [-\infty, -1]$ and $\neg\mathsf{b} \triangleq x \geq 0 = [0, \infty]$ would require the addition of all the concrete joins $[n_1, n_2] \cup [m_1, m_2]$ with $n_1 \leq n_2 < 0 \leq m_1 \leq m_2$.

Given any input p, Theorem 4.4 shows that Int is made locally complete for the guards b and $\neg\mathsf{b}$ by adding the unique element $(\mathsf{Int}(p \cap \mathsf{b}) \cap \mathsf{b}) \cup (\mathsf{Int}(p \cap \neg\mathsf{b}) \cap \neg\mathsf{b})$ and taking the Moore closure. For example, when $p \triangleq \{n \mid n \text{ odd}\}$ then $(\mathsf{Int}(p \cap \mathsf{b}) \cap \mathsf{b}) \cup (\mathsf{Int}(p \cap \neg\mathsf{b}) \cap \neg\mathsf{b}) = [-\infty, -1] \cup [1, \infty] = \mathbb{Z}_{\neq 0}$ is the unique element to add (and, by Moore closure, any concrete join of the form $[n, -1] \cup [1, m]$ with $n \leq -1$ and $m \geq 1$). Thus, we get the domain that contains all the intervals, possibly with a hole in 0. □

5 LOCAL COMPLETENESS LOGIC

We define a 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 $[\![\mathsf{r}]\!]p$ (i.e. $q \subseteq [\![\mathsf{r}]\!]p$);
(ii) $[\![\mathsf{r}]\!]$ is *locally complete* for input p and abstraction A (i.e. $\mathbb{C}^A_p(\mathsf{r})$ holds);
(iii) q and $[\![\mathsf{r}]\!]p$ have the same *over-approximation* in A (i.e. $A(q) = A([\![\mathsf{r}]\!]p)$).

Given a correctness specification *spec*, we recall that abstract interpretation raises an alarm when $\gamma([\![\mathsf{r}]\!]^\sharp_A \alpha(p)) \not\subseteq \mathit{spec}$: such alarm is false if $[\![\mathsf{r}]\!]p \subseteq \mathit{spec}$ and true otherwise. It turns out that the above three properties of any provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ allow us to distinguish between true and false alarms, as described below:

Case 1: If the over-approximation $[\![\mathsf{r}]\!]^\sharp_A \alpha(p)$ does not raise alarms, 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 alarms because $\gamma([\![\mathsf{r}]\!]^\sharp_A \alpha(p)) \not\subseteq$ *spec*, then, by local completeness, any provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ is such that $q \smallsetminus \mathit{spec} \neq \varnothing$ and all the stores in $q \smallsetminus \mathit{spec}$ are true alarms. As discussed in the Introduction, let us recall that by relying on generic, thus possibly incomplete, abstract interpretation we could not distinguish which alarms 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 alarm is raised because $\gamma(\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(p)) \not\subseteq$ *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 alarms 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 5.9 and also Section 8).

The logical proof system $\vdash_A$ is defined in Figure 4 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 depend directly on the underlying abstraction A. For the readers' convenience, we copy rule (relax) below:

$$\frac{p' \le p \le A(p') \quad \vdash_A [p']\ \mathsf{r}\ [q'] \quad q \le q' \le A(q)}{\vdash_A [p]\ \mathsf{r}\ [q]}\ (\mathsf{relax})$$

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 $q \subseteq \llbracket \mathsf{r} \rrbracket p \subseteq A(\llbracket \mathsf{r} \rrbracket p) = A(q)$ holds. Likewise in the consequence rules of de Vries and Koutavas [2011], O'Hearn [2020], and Raad et al. [2020], the logical ordering between pre-conditions $p' \le p$ and post-conditions $q \le q'$ 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 5.10 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 6 for showing how to recover O'Hearn [2020]'s IL as an instance of our proof system. Moreover, in Section 5.3 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", meaning that if A is locally complete for some $c \in C$, then A is locally complete for all $d \in C$ such that $c \le d \le A(c)$ holds.

Lemma 5.1. *If* $\mathbb{C}^A_c(f)$ *and* $d \in [c, A(c)]$, *then* $\mathbb{C}^A_d(f)$.

Proof. Since $c \le d \le A(c)$, by monotonicity of f and A, we obtain that $Af(c) \le Af(d) \le AfA(c)$ and $AfA(d) \le AfAA(c) = AfA(c)$. Since $AfA(c) = Af(c)$ we have that $Af(c) = Af(d)$. Finally, $Af(d) \le AfA(d) \le Af(c) = Af(d)$. □

The rule (transfer) checks that the basic expressions e are locally complete on p and, in that case, provides the output of the corresponding transfer function $\llbracket \mathsf{e} \rrbracket$ on p as post-condition.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}}_{p_1}(\mathsf{b}_1?)}{\vdash_{\mathsf{Int}} [p_1]\ \mathsf{b}_1?\ [\{1,999,1000\}]}\ (\text{tr.}) \quad \dfrac{\mathbb{C}^{\mathsf{Int}}_{\{1,999,1000\}}(\mathsf{e}_1)}{\vdash_{\mathsf{Int}} [\{1,999,1000\}]\ \mathsf{e}_1?\ [\{0,998,999\}]}\ (\text{tr.})}{\vdash_{\mathsf{Int}} [p_1]\ \mathsf{r}_1\ [\{0,998,999\}]}\ (\text{seq}) \quad \dfrac{\dfrac{\mathbb{C}^{\mathsf{Int}}_{p_1}(\mathsf{b}_2?)}{\vdash_{\mathsf{Int}} [p_1]\ \mathsf{b}_2?\ [\{0,1,999\}]}\ (\text{tr.}) \quad \dfrac{\mathbb{C}^{\mathsf{Int}}_{\{0,1,999\}}(\mathsf{e}_2)}{\vdash_{\mathsf{Int}} [\{0,1,999\}]\ \mathsf{e}_2\ [\{1,2,1000\}]}\ (\text{tr.})}{\vdash_{\mathsf{Int}} [p_1]\ \mathsf{r}_2\ [\{1,2,1000\}]}\ (\text{seq})}{\vdash_{\mathsf{Int}} [p_1]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [\{0,1,2,998,999,1000\}]}\ (\text{join})}{(\star)}\ (\text{iterate})$$

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

Legenda: $\mathsf{b}_1 \triangleq 0 < x$ $\quad \mathsf{e}_1 \triangleq x := x - 1$ $\quad \mathsf{b}_2 \triangleq x < 1000$ $\quad \mathsf{e}_2 \triangleq x := x + 1$ $\quad p_1 \triangleq \{0, 1, 999, 1000\}$

Fig. 5. Derivation of $\vdash_{\mathsf{Int}} [p = \{1, 999\}]\ \mathsf{r}\ [\{0, 2, 1000\}]$ for Example 5.2, where the label (tr.) stands for (transfer).

Of course, for no-ops, Boolean guards and assignments, the rule (transfer) can be equivalently stated in symbolic form as follows:

$$\dfrac{}{\vdash_A [p]\ \mathbf{skip}\ [p]}\ (\text{skip}) \qquad \dfrac{\mathbb{C}^A_p(\mathsf{b}?)}{\vdash_A [p]\ \mathsf{b}?\ [p \wedge \mathsf{b}]}\ (\text{assume})$$

$$\dfrac{\mathbb{C}^A_p(x := \mathsf{a})}{\vdash_A [p]\ x := \mathsf{a}\ [\exists v.(p[^v/_x] \wedge x = \mathsf{a}[^v/_x])]}\ (\text{assign})$$

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.

The rule (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 $(x := x + 1)^*$), using (iterate) we can exploit the abstraction A to stop the proof when the abstraction of a finitary input p is already an infinitary abstract invariant[3] (cf. Lemma 5.4), 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 alarm 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 5.1 to show how LCL_A can help in program analysis.

Example 5.2. 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:

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

The triple $\vdash_{\mathsf{Int}} [p]\ \mathsf{r}\ [\{0, 2, 1000\}]$ can be derived as shown in Figure 5, where for brevity we let:

$$\begin{array}{lll} \mathsf{b}_1 \triangleq 0 < x & \mathsf{e}_1 \triangleq x := x - 1 & p_1 \triangleq \{0, 1, 999, 1000\} \\ \mathsf{b}_2 \triangleq x < 1000 & \mathsf{e}_2 \triangleq x := x + 1 & \end{array}$$

[3] A maybe non-obvious consequence of the condition $q \leq A(p)$.

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

Fig. 6. Derivation of $\vdash_{\mathsf{Sign}} [p = \{-10, -1, 100\}]\ \mathsf{c}\ [\{100\}]$ for Example 5.3.

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 command. This proof needs just one application of (rec) to compute an under-approximation of $\mathrm{post}[\mathsf{r}]p = [\![\mathsf{r}]\!]p$, 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 5.3. Let us consider the domain Sign, the pre-condition $p \triangleq \{-10, -1, 100\}$ and the following Imp program:

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

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 Figure 6 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 5.5 (1)), we conclude that $100 \notin \mathit{spec}$ is a true alarm. 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 that:

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

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:

$$\dfrac{\vdash_{A} [p]\ \mathsf{r}\ [q] \quad q \le p}{\vdash_{A} [p]\ \mathsf{r}^{*}\ [p]}\ (\text{invariant}) \qquad\qquad \dfrac{\vdash_{A} [p]\ \mathsf{r}\ [q] \quad A(p) = A(q)}{\vdash_{A} [p]\ \mathsf{r}^{*}\ [q]}\ (\text{abs-fix})$$

Lemma 5.4. *The rules* (invariant) *and* (abs-fix) *can be derived in* LCL_A.

Proof. We first show that (invariant) is a derived rule. If $q \le p$ we have $A(p \vee q) = A(p)$. Since $\vdash_{A} [p]\ \mathsf{r}\ [q]$, by (iterate) it follows $\vdash_{A} [p]\ \mathsf{r}^{*}\ [p \vee q]$, i.e. $\vdash_{A} [p]\ \mathsf{r}^{*}\ [p]$.

For (abs-fix), since γ is 1-1, if $A(q) = A(p)$, then $\alpha(p) = \alpha(q)$. Therefore, $A(p \vee q) = A(p)$, because α is additive and thus $A(p \vee q) = \gamma(\alpha(p \vee q)) = \gamma(\alpha(p) \vee_A \alpha(q)) = \gamma\alpha(q) = A(q)$. Since $\vdash_{A} [p]\ \mathsf{r}\ [q]$ and $q \le A(q) = A(p)$, by (iterate) it follows $\vdash_{A} [p]\ \mathsf{r}^{*}\ [p \vee q]$. Since $q \le (p \vee q) \le A(p \vee q) = A(q)$, by (relax), we conclude $\vdash_{A} [p]\ \mathsf{r}^{*}\ [q]$. □

The 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.

It is worth remarking that indeed (invariant) and (iterate) are equivalent rules within LCL_A. In fact, the proof of Lemma 5.4 shows that the rule (invariant) can be derived from the rule (iterate), and, vice versa, (iterate) can be derived using the rules (invariant), (rec) and (relax), as shown by the following inferences:

$$\cfrac{\vdash_A [p]\ \mathsf{r}\ [q] \qquad \cfrac{\cfrac{p \leq p \vee q \leq A(p) \quad \vdash_A [p]\ \mathsf{r}\ [q]}{\vdash_A [p \vee q]\ \mathsf{r}\ [q]}\ (\text{relax}) \qquad q \leq p \vee q}{\vdash_A [p \vee q]\ \mathsf{r}^*\ [p \vee q]}\ (\text{invariant})}{\vdash_A [p]\ \mathsf{r}^*\ [p \vee q]}\ (\text{rec})$$

where the proof obligation $q \leq p \vee q$ trivially holds, while $p \leq p \vee q \leq A(p)$ holds because the assumption $q \leq A(p)$ of the rule (iterate) implies that $p \leq p \vee q \leq p \vee A(p) = A(p)$.

5.1 Soundness

Our program logic LCL_A turns out to be sound for the target properties (i–iii) stated at the beginning of Section 5, as formalized by the following soundness result, where conditions (1) and (2) embody, respectively, items (i) and (ii)+(iii).

Theorem 5.5 (Soundness of LCL_A). *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) *$q \leq \llbracket \mathsf{r} \rrbracket p$, and*
(2) *$\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(p) = \alpha(q) = \alpha(\llbracket \mathsf{r} \rrbracket p)$.*

Proof. For the sake of simplicity, let us refer to the equality $\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(p) = \alpha(q)$ as (2a) and use (2b) for the equality $\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(p) = \alpha(\llbracket \mathsf{r} \rrbracket p)$. Then we note that (2b) follows immediately from (1) and (2a) because, by monotonicity of α and correctness (4), we have that:

$$\alpha(q) \leq_A \alpha(\llbracket \mathsf{r} \rrbracket p) \leq_A \llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(p) = \alpha(q).$$

Therefore, in some cases we just prove (1) and (2a) and leave the proof of (2b) implicit. We proceed by induction on the derivation tree of $\vdash_A [p]\ \mathsf{r}\ [q]$, by distinguishing the various cases on the basis of the last rule that is applied.

(transfer) : If rule (transfer) is applied as last rule, then it must be $\mathsf{r} \equiv \mathsf{e} \in \mathsf{Exp}$ such that $\mathbb{C}^A_p(\mathsf{e})$ and $q = \llbracket \mathsf{e} \rrbracket p$. We have that: (1) $\llbracket \mathsf{e} \rrbracket p \leq \llbracket \mathsf{e} \rrbracket p$ trivially holds, while $\llbracket \mathsf{e} \rrbracket^\sharp_A \alpha(p) = \llbracket \mathsf{e} \rrbracket^A \alpha(p) = \alpha(\llbracket \mathsf{e} \rrbracket \gamma\alpha(p)) = \alpha(\llbracket \mathsf{e} \rrbracket p)$ holds by $\mathbb{C}^A_p(\mathsf{e})$, so that we also have (2).

(relax) : If the last rule applied is (relax), we assume by induction that $\vdash_A [p']\ \mathsf{r}\ [q']$ can be derived and let p, q such that $p' \leq p \leq A(p')$ and $q \leq q' \leq A(q)$.
(1) By inductive hypothesis (1) and monotonicity of $\llbracket \mathsf{r} \rrbracket$ we have:

$$q \leq q' \leq \llbracket \mathsf{r} \rrbracket p' \leq \llbracket \mathsf{r} \rrbracket p.$$

(2) Let us observe that from $p \in [p', A(p')]$ and $q' \in [q, A(q)]$, we derive $\alpha(p') = \alpha(p)$ and $\alpha(q') = \alpha(q)$. Therefore:

$$\begin{aligned}
\alpha(q) \leq_A &\quad \text{[by monotonicity of } \alpha \text{ and (1)]}\\
\alpha(\llbracket \mathsf{r} \rrbracket p) \leq_A &\quad \text{[by correctness (4)]}\\
\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(p) = &\quad \text{[by hyp.]}\\
\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(p') = &\quad \text{[by ind. hyp. (2a)]}\\
\alpha(q') = &\quad \text{[by hyp.]}\\
\alpha(q), &
\end{aligned}$$

so that, $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(q) = \alpha(\llbracket \mathsf{r} \rrbracket p)$ follows.

(seq) : If the last rule applied is (seq), it must be $\mathsf{r} \equiv \mathsf{r}_1; \mathsf{r}_2$ and we can assume by induction that $\vdash_A [p]\ \mathsf{r}_1\ [w]$ and $\vdash_A [w]\ \mathsf{r}_2\ [q]$, can be derived for some w. We need to prove the thesis for the conclusion $\vdash_A [p]\ \mathsf{r}_1; \mathsf{r}_2\ [q]$.
(1) By inductive hypotheses (1) and monotonicity of $\llbracket \mathsf{r}_2 \rrbracket$:

$$q \leq \llbracket \mathsf{r}_2 \rrbracket w \leq \llbracket \mathsf{r}_2 \rrbracket (\llbracket \mathsf{r}_1 \rrbracket p) = \llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket p.$$

(2) We have that:

$$\begin{aligned}
\alpha(q) \leq_A &\quad \text{[by (1) and monotonicity of } \alpha \text{]}\\
\alpha(\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket p) \leq_A &\quad \text{[by correctness (4)]}\\
\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(p) = &\quad \text{[by definition]}\\
\llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{A} (\llbracket \mathsf{r}_1 \rrbracket^{\sharp}_{A} \alpha(p)) = &\quad \text{[by ind.hyp. (2a), } \llbracket \mathsf{r}_1 \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(w) \text{]}\\
\llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(w) = &\quad \text{[by ind.hyp. (2a)]}\\
\alpha(q), &
\end{aligned}$$

so that, $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(q) = \alpha(\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket p)$ follows.

(join) : If the last rule applied is (join), it must be $\mathsf{r} \equiv \mathsf{r}_1 \oplus \mathsf{r}_2$ and we assume by induction that $\vdash_A [p]\ \mathsf{r}_1\ [q_1]$ and $\vdash_A [p]\ \mathsf{r}_2\ [q_2]$ can be derived for some suitable q_1 and q_2 such that $q = q_1 \vee q_2$. We need to prove the thesis for the conclusion $\vdash_A [p]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [q_1 \vee q_2]$.
(1) By inductive hypothesis, $q_i \leq \llbracket \mathsf{r}_i \rrbracket p$, therefore:

$$q_1 \vee q_2 \leq \llbracket \mathsf{r}_1 \rrbracket p \vee \llbracket \mathsf{r}_2 \rrbracket p = \llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket p.$$

(2a) We have that:

$$\begin{aligned}
\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(p) = &\quad \text{[by definition]}\\
\llbracket \mathsf{r}_1 \rrbracket^{\sharp}_{A} \alpha(p) \vee_A \llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(p) = &\quad \text{[by ind. hyp. (2a)]}\\
\alpha(q_1) \vee_A \alpha(q_2) = &\quad \text{[by additivity of } \alpha \text{]}\\
\alpha(q_1 \vee q_2), &
\end{aligned}$$

so that, $\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(q_1 \vee q_2) = \alpha(\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket p)$ follows by (1) and (2a).

(iterate) : If the last rule applied is (iterate), it must be $\mathsf{r} \equiv \mathsf{r}_1^*$ and we assume by induction that $\vdash_A [p]\ \mathsf{r}_1\ [q_1]$ for some q_1 such that $q_1 \leq A(p)$ and $q = p \vee q_1$. Note that $\alpha(q_1) \leq_A \alpha(A(p)) = \alpha(p)$. We need to prove the thesis for the conclusion $\vdash_A [p]\ \mathsf{r}_1^*\ [p \vee q_1]$.

(1) By inductive hypothesis (1), we have that:

$$p \vee q_1 \leq p \vee [\![\mathsf{r}_1]\!]p = [\![\mathsf{r}_1]\!]^0 p \vee [\![\mathsf{r}_1]\!]p \leq \bigvee_{n\in\mathbb{N}}[\![\mathsf{r}_1]\!]^n p = [\![\mathsf{r}_1^*]\!]p.$$

(2a) By inductive hypothesis (2) and hypothesis $q_1 \leq A(p)$, we have $[\![\mathsf{r}_1]\!]^\sharp_A \alpha(p) = \alpha(q_1) \leq_A \alpha(p)$, so that Lemma 2.1 (c) is applicable. Therefore:

$$\begin{aligned} [\![\mathsf{r}_1^*]\!]^\sharp_A \alpha(p) = \quad & [\text{by Lemma 2.1 (c)}]\\ \alpha(p) = \quad & [\text{by hyp. } q_1 \leq A(p)]\\ \alpha(p) \vee_A \alpha(q_1) = \quad & [\text{by additivity of } \alpha]\\ \alpha(p \vee q_1), \quad & \end{aligned}$$

so that, $[\![\mathsf{r}_1^*]\!]^\sharp_A \alpha(p) = \alpha(p \vee q_1) = \alpha([\![\mathsf{r}_1^*]\!]p)$ follows by (1) and (2a).

(rec) : If the last rule applied is (rec), it must be $\mathsf{r} \equiv \mathsf{r}_1^*$ and we assume by induction that $\vdash_A [p]\ \mathsf{r}_1\ [w]$ and $\vdash_A [p\vee w]\ \mathsf{r}_1^*\ [q]$ can be derived for some w. We need to prove the thesis for the conclusion $\vdash_A [p]\ \mathsf{r}_1^*\ [q]$.

(1) By inductive hypothesis (1) and Lemma 2.1 (e) we have:

$$q \leq [\![\mathsf{r}_1^*]\!](p \vee w) = [\![\mathsf{r}_1^*]\!]p.$$

(2) We have that:

$$\begin{aligned} \alpha(q) \leq_A \quad & [\text{by monotonicity of } \alpha \text{ and (1)}]\\ \alpha([\![\mathsf{r}_1^*]\!]p) \leq_A \quad & [\text{by correctness (4)}]\\ [\![\mathsf{r}_1^*]\!]^\sharp_A \alpha(p) \leq_A \quad & [\text{by monotonicity}]\\ [\![\mathsf{r}_1^*]\!]^\sharp_A \alpha(p \vee w) = \quad & [\text{by ind.hyp. (2a)}]\\ \alpha(q), \quad & \end{aligned}$$

so that $[\![\mathsf{r}_1^*]\!]^\sharp_A \alpha(p) = \alpha(q) = \alpha([\![\mathsf{r}_1^*]\!]p)$ follows. □

As a consequence, if a correctness specification *spec* is expressible in A, i.e., if $spec = \gamma(a)$ for some abstract value $a \in A$, then any provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ allows us to use q to witness *either the correctness or the incorrectness* of r for the pre-condition p, as stated by the following result.

Corollary 5.6 (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:*

$$\forall a \in A.\ [\![\mathsf{r}]\!]p \leq \gamma(a) \textit{ iff } q \leq \gamma(a).$$

Example 5.7. In Example 5.3 we already noticed that, by Theorem 5.5 (1), $[\![\mathsf{c}]\!]p$ does not satisfy *spec*. Note that for $p' \triangleq \{-10, -1, 5, 100\}$ we could also prove, e.g., $\vdash_{\mathrm{Sign}} [p']\ \mathsf{c}\ [\{5\}]$ where the post-condition $\{5\}$ has no alarm, even if *spec* is not satisfied: since *spec* is not expressible in Sign, then Corollary 5.6 is not applicable. □

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Sign}}_{\{10,20\}}(\mathsf{b}_1?)}{\vdash_{\mathsf{Sign}} [10,20]\ \mathsf{b}_1?\ [\{10,20\}]}\,(\text{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Sign}}_{\{10,20\}}(\mathsf{e}_1)}{\vdash_{\mathsf{Sign}} [\{10,20\}]\ \mathsf{e}_1\ [\{20,30\}]}\,(\text{transfer})}{\vdash_{\mathsf{Sign}} [10,20]\ \mathsf{r}_1\ [\{20,30\}]}\,(\text{seq}) \qquad \{20,30\} \subseteq \mathsf{Sign}(\{10,20\}) = \mathbb{Z}_{>0}}{\dfrac{\vdash_{\mathsf{Sign}} [\{10,20\}]\ \mathsf{r}_1^*\ [\{10,20,30\}]}{(*)}\,(\text{rec})}\,(\text{iterate})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Sign}}_{p_1}(\mathsf{b}_1?)}{\vdash_{\mathsf{Sign}} [p_1]\ \mathsf{b}_1?\ [p_1]}\,(\text{tr.}) \quad \dfrac{\mathbb{C}^{\mathsf{Sign}}_{p_1}(\mathsf{e}_1)}{\vdash_{\mathsf{Sign}} [p_1]\ \mathsf{e}_1\ [\{20\}]}\,(\text{tr.})}{\vdash_{\mathsf{Sign}} [p_1]\ \mathsf{r}_1\ [\{20\}]}\,(\text{seq}) \quad (*)}{\vdash_{\mathsf{Sign}} [p_1]\ \mathsf{r}_1^*\ [\{10,20,30\}]}\,(\text{rec}) \quad \{10,30\} \subseteq \{10,20,30\} \subseteq \mathbb{Z}_{>0}}{\vdash_{\mathsf{Sign}} [p_1]\ \mathsf{r}_1^*\ [\{10,30\}]}\,(\text{relax}) \quad \dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Sign}}_{\{10,30\}}(\mathsf{b}_2?)}{\vdash_{\mathsf{Sign}} [\{10,30\}]\ \mathsf{b}_2?\ [\{30\}]}\,(\text{tr.}) \quad \dfrac{\mathbb{C}^{\mathsf{Sign}}_{\{30\}}(\mathsf{e}_2)}{\vdash_{\mathsf{Sign}} [\{30\}]\ \mathsf{e}_2\ [\{-1\}]}\,(\text{tr.})}{\vdash_{\mathsf{Sign}} [10,30]\ \mathsf{r}_3\ [\{-1\}]}\,(\text{seq}) \quad \dfrac{\mathbb{C}^{\mathsf{Sign}}_{\{10,30\}}(\neg\mathsf{b}_2)}{\vdash_{\mathsf{Sign}} [\{10,30\}]\ \neg\mathsf{b}_2?\ [\{10\}]}\,(\text{tr.})}{\vdash_{\mathsf{Sign}} [\{10,30\}]\ \mathsf{r}_2\ [\{-1,10\}]}\,(\text{join})}{\vdash_{\mathsf{Sign}} [p_1]\ \mathsf{r}\ [\{-1,10\}]}\,(\text{seq})$$

Legenda:

$$\begin{array}{llll} \mathsf{r} \triangleq \mathsf{r}_1^*;\mathsf{r}_2 & \mathsf{r}_1 \triangleq \mathsf{b}_1?;\mathsf{e}_1 & \mathsf{r}_2 \triangleq (\mathsf{b}_2?;\mathsf{e}_2) \oplus (\neg\mathsf{b}_2?) & \mathsf{r}_3 \triangleq \mathsf{b}_2?;\mathsf{e}_2 \\ \mathsf{b}_1 \triangleq x < 1000 & \mathsf{e}_1 \triangleq x := x + 10 & \mathsf{b}_2 \triangleq x = 30 & \mathsf{e}_2 \triangleq x := -1 \end{array}$$

Fig. 7. Derivation of $\vdash_{\mathsf{Sign}} [p_1 = \{10\}]\ \mathsf{r}\ [\{-1, 10\}]$ for Example 5.9.

Example 5.8. Let us consider again Example 5.2 and the triple $\vdash_{\mathsf{Int}} [p]\ \mathsf{r}\ [\{0, 2, 1000\}]$ (see Figure 5.2) to discuss the cases of three different correctness specifications: $spec_1 \triangleq x \neq 2$, $spec_2 \triangleq x \leq 1000$ and $spec_3 \triangleq 100 \leq x$.

Despite that $spec_1$ is not expressible in Int, the triple $\vdash_{\mathsf{Int}} [p]\ \mathsf{r}\ [\{0, 2, 1000\}]$ exposes the true alarm $2 \notin spec_1$, since $2 \in \{0, 2, 1000\} \subseteq [\![\mathsf{r}]\!]p$ (cf. Theorem 5.5 (1)).

By Theorem 5.5 (2), we know that $[\![\mathsf{r}]\!]p$ satisfies $spec_2$, because $\mathsf{Int}([\![\mathsf{r}]\!]p) = \mathsf{Int}(\{0, 2, 1000\}) = [0, 1000] \subseteq spec_2$. Since $spec_2$ is expressible in Int, by Corollary 5.6, 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 alarms $0, 2 \notin spec_3$. Likewise $spec_2$, by Corollary 5.6, the post-condition q of any provable triple $\vdash_{\mathsf{Int}} [p]\ \mathsf{r}\ [q]$ will contain a true alarm for $spec_3$. In particular, any such triple $\vdash_{\mathsf{Int}} [p]\ \mathsf{r}\ [q]$, is such that $0 \in q$, therefore contradicting the assertion $spec_3$, because it must be $\mathsf{Int}(q) = [0, 1000] = \mathsf{Int}([\![\mathsf{r}]\!]p)$. □

Example 5.9. Let us consider the program $\mathsf{r} \triangleq \mathsf{r}_1^*; \mathsf{r}_2$ where:

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

Consider the correctness specification $spec \triangleq x \neq -1$, and two pre-conditions $p_1 \triangleq \{10\}$ 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 alarm 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 alarm -1 violating *spec*. The whole derivation is in Figure 7.

Consider now the pre-condition p_2. Here, we apply (rec), (iterate) and (relax) so as to derive $\vdash_{\mathsf{Sign}}$ $[\{11\}]$ 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 $\vdash_{\mathsf{Sign}_{30}}$ $[p_2]$ r $[\{11\}]$ could be derived to witness that r satisfies *spec* (by Corollary 5.6). □

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 5.5.

Example 5.10. Consider the abstract domain Int, the absolute value program from the Introduction

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

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 $\vdash_{\mathsf{Int}}$ $[p']$ $\mathsf{AbsVal}(x)$ $[q']$ with $q' = \{0, 1, 5\}$. In fact, Int is locally complete for $[\![\mathsf{AbsVal}(x)]\!]$ on p', because $\mathsf{Int}([\![\mathsf{AbsVal}(x)]\!]\mathsf{Int}\ (p')) = [0, 5] = \mathsf{Int}(q')$.

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., $\vdash_{\mathsf{Int}}$ $[p]$ $\mathsf{AbsVal}(x)$ $[q]$. However, Int is not locally complete for $[\![\mathsf{AbsVal}(x)]\!]$ on p, because $\mathsf{Int}([\![\mathsf{AbsVal}(x)]\!]p) = [5, 5] \neq [0, 5] = \mathsf{Int}([\![\mathsf{AbsVal}(x)]\!]\mathsf{Int}(p))$. □

5.2 On the Logical Completeness of LCL_A

We now study the completeness of LCL_A as a proof system. In this context, completeness is intended in the logical sense, that is the ability of the proof system to derive any valid triple. To avoid the clash of terminology with completeness in abstract interpretation, we call it *logical* (in)completeness. Our logic LCL_A is not logically complete in general, i.e., the converse of Theorem 5.5 does not hold, as shown by the following example.

Example 5.11 (Logical Incompleteness). 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 $\vdash_{\mathsf{Sign}}$ $[p]$ e_1 $[[\![\mathsf{e}_1]\!]p]$ by rule (transfer), the proof obligation $\mathbb{C}^{\mathsf{Sign}}_p(\mathsf{e}_1)$ fails: $\alpha([\![\mathsf{e}_1]\!]p) = \alpha(\mathbb{Z}_{\geq 1}) = \mathbb{Z}_{>0} \neq \mathbb{Z}_{\geq 0} = \alpha(\mathbb{Z}_{\geq 0}) = \alpha([\![\mathsf{e}_1]\!]\mathsf{Sign}(p))$. 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 $\vdash_{\mathsf{Sign}}$ $[p']$ 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 $\alpha([\![\mathsf{e}_1]\!]p') \leq \alpha(\mathbb{Z}_{\geq 1}) = \mathbb{Z}_{>0}$ 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 $\vdash_{\mathsf{Sign}}$ $[p']$ e_1 $[[\![\mathsf{e}_1]\!]p']$ is provable. □

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

(1) to add the infinitary rule (limit) for Kleene star, whose soundness is shown by Lemma 5.12:

$$\frac{\forall n \in \mathbb{N}.\ \vdash_A [p_n]\ \mathsf{r}\ [p_{n+1}]}{\vdash_A [p_0]\ \mathsf{r}^*\ [\bigvee_{i\in\mathbb{N}} p_i]}\ (\mathsf{limit})$$

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

Lemma 5.12. *The proof system extended with rule* (limit) *is sound.*

Proof. We extend the inductive proof of Theorem 5.5 by considering the case where the last rule applied to derive $\vdash_A [p]\ \mathsf{r}\ [q]$ is (limit). As in the proof of Theorem 5.5, we refer to the equality $[\![\mathsf{r}]\!]^\sharp_A \alpha(p) = \alpha(q)$ as (2a) and use (2b) for the equality $[\![\mathsf{r}]\!]^\sharp_A \alpha(p) = \alpha([\![\mathsf{r}]\!]p)$. Then we recall that (2b) follows immediately by (1) and (2a).

(limit) : If the last rule applied is (limit), it must be $\mathsf{r} \equiv \mathsf{r}_1^*$ and we can assume by induction that $\forall n \in \mathbb{N}.\ \vdash_A [p_n]\ \mathsf{r}_1\ [p_{n+1}]$. We need to prove the thesis for the conclusion $\vdash_A [p_0]\ \mathsf{r}_1^*\ [\bigvee_{i\in\mathbb{N}} p_i]$.

(1) We have that for every $n \in \mathbb{N}$, $p_n \leq [\![\mathsf{r}_1]\!]^n p_0$: obvious for $n = 0$; then, by inductive hypothesis, $p_{n+1} \leq [\![\mathsf{r}_1]\!] p_n \leq [\![\mathsf{r}_1]\!]([\![\mathsf{r}_1]\!]^n p_0) = [\![\mathsf{r}_1]\!]^{n+1} p_0$. Thus, $\bigvee_{n\in\mathbb{N}} p_n \leq \bigvee_{n\in\mathbb{N}} [\![\mathsf{r}_1]\!]^n p_0 = [\![\mathsf{r}_1^*]\!] p_0$.

(2a) We have that for every $n \in \mathbb{N}$, $[\![\mathsf{r}_1]\!]^{\sharp\, n}_A \alpha(p_0) = \alpha(p_n)$. In fact, by induction on n, this is obvious for the base case $n = 0$. Then, for the inductive case:

$$\begin{aligned}
[\![\mathsf{r}_1]\!]^{\sharp\, n+1}_A \alpha(p_0) = {} & \quad \text{[by definition]}\\
[\![\mathsf{r}_1]\!]^{\sharp}_A ([\![\mathsf{r}_1]\!]^{\sharp\, n}_A \alpha(p_0)) = {} & \quad \text{[by ind. hyp. on } n\text{]}\\
[\![\mathsf{r}_1]\!]^{\sharp}_A \alpha(p_n) = {} & \quad \text{[by ind. hyp. (2a)]}\\
\alpha(p_{n+1}). &
\end{aligned}$$

Consequently,

$$\begin{aligned}
[\![\mathsf{r}_1^*]\!]^{\sharp}_A \alpha(p_0) = {} & \quad \text{[by definition]}\\
\textstyle\bigvee_A \{[\![\mathsf{r}_1]\!]^{\sharp\, n}_A \alpha(p_0) \mid n \in \mathbb{N}\} = {} & \quad \text{[by property above]}\\
\textstyle\bigvee_A \{\alpha(p_n) \mid n \in \mathbb{N}\} = {} & \quad \text{[by additivity of } \alpha\text{]}\\
\alpha(\vee_{n\in\mathbb{N}} p_n). &
\end{aligned}$$

□

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 expressions occurring in the program. Let us remark that incorrectness logic [O'Hearn 2020] also includes this infinitary rule (limit), there called *backwards variant* rule. Likewise (limit), the backwards variant rule of IL allows us to derive other finitary rules (e.g., *Iterate zero*) and plays a crucial role for proving the logical completeness of IL [O'Hearn 2020, Theorem 6].

Lemma 5.13. *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ *and* $\mathsf{r} \in \mathsf{Reg}$. *Then:*

$$\big(\forall \mathsf{e} \in \mathsf{Exp}(\mathsf{r}).\ \mathbb{C}^A(\mathsf{e})\big) \Rightarrow [\![\mathsf{r}]\!]^\sharp_A \alpha = \alpha[\![\mathsf{r}]\!].$$

Proof. We proceed by structural induction on $\mathsf{r} \in \mathsf{Reg}$.

($\mathsf{r} \equiv \mathsf{e}$):

$$\begin{aligned} \alpha \circ [\![\mathsf{e}]\!] = {} & [\text{by } \mathbb{C}^A(\mathsf{e})] \\ \alpha \circ [\![\mathsf{e}]\!] \circ \gamma \circ \alpha = {} & [\text{by definition}] \\ [\![\mathsf{e}]\!]^\sharp_A \circ \alpha & \end{aligned}$$

($\mathsf{r} \equiv \mathsf{r}_1; \mathsf{r}_2$):

$$\begin{aligned} \alpha \circ [\![\mathsf{r}_1; \mathsf{r}_2]\!] = {} & [\text{by definition}] \\ \alpha \circ [\![\mathsf{r}_2]\!] \circ [\![\mathsf{r}_1]\!] = {} & [\text{by ind.hyp.}] \\ [\![\mathsf{r}_2]\!]^\sharp_A \circ \alpha \circ [\![\mathsf{r}_1]\!] = {} & [\text{by ind.hyp.}] \\ [\![\mathsf{r}_2]\!]^\sharp_A \circ [\![\mathsf{r}_1]\!]^\sharp_A \circ \alpha = {} & [\text{by definition}] \\ [\![\mathsf{r}_1; \mathsf{r}_2]\!]^\sharp_A \circ \alpha & \end{aligned}$$

($\mathsf{r} \equiv \mathsf{r}_1 \oplus \mathsf{r}_2$): Let us take an arbitrary $p \in C$.

$$\begin{aligned} \alpha([\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]p) = {} & [\text{by definition}] \\ \alpha([\![\mathsf{r}_1]\!]p \vee [\![\mathsf{r}_2]\!]p) = {} & [\text{by additivity of } \alpha] \\ \alpha([\![\mathsf{r}_1]\!]p) \vee_A \alpha([\![\mathsf{r}_2]\!]p) = {} & [\text{by ind.hyp.}] \\ [\![\mathsf{r}_1]\!]^\sharp_A \alpha(p) \vee_A [\![\mathsf{r}_2]\!]^\sharp_A \alpha(p) = {} & [\text{by definition}] \\ [\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^\sharp_A \alpha(p) & \end{aligned}$$

($\mathsf{r} \equiv \mathsf{r}_1^*$): Let us first prove the following property:

$$\forall n \in \mathbb{N}.\ \alpha [\![\mathsf{r}_1]\!]^n = [\![\mathsf{r}_1]\!]^{\sharp\, n}_A \alpha. \qquad (*)$$

For $n = 0$: $\alpha \circ [\![\mathsf{r}_1]\!]^0 = \alpha = [\![\mathsf{r}_1]\!]^{\sharp\, 0}_A \circ \alpha$.
For $n + 1$,

$$\begin{aligned} \alpha \circ [\![\mathsf{r}_1]\!]^{n+1} = {} & [\text{by definition}] \\ \alpha \circ [\![\mathsf{r}_1]\!] \circ [\![\mathsf{r}_1]\!]^n = {} & [\text{by ind.hyp.}] \\ [\![\mathsf{r}_1]\!]^\sharp_A \circ \alpha \circ [\![\mathsf{r}_1]\!]^n = {} & [\text{by ind.hyp.}] \\ [\![\mathsf{r}_1]\!]^\sharp_A \circ [\![\mathsf{r}_1]\!]^{\sharp\, n}_A \circ \alpha = {} & [\text{by definition}] \\ [\![\mathsf{r}_1]\!]^{\sharp\, n+1}_A \circ \alpha & \end{aligned}$$

Thus, for any $p \in C$,

$$\begin{aligned} \alpha([\![\mathsf{r}_1^*]\!]p) = {} & [\text{by definition}] \\ \alpha(\bigvee \{[\![\mathsf{r}_1]\!]^n p \mid n \in \mathbb{N}\}) = {} & [\text{by additivity of } \alpha] \\ \bigvee_A \{\alpha([\![\mathsf{r}_1]\!]^n p) \mid n \in \mathbb{N}\}) = {} & [\text{by } (*)] \\ \bigvee_A \{[\![\mathsf{r}_1]\!]^{\sharp\, n}_A \alpha(p) \mid n \in \mathbb{N}\}) = {} & [\text{by definition}] \\ [\![\mathsf{r}_1^*]\!]^\sharp_A \alpha(p) & \end{aligned}$$

This closes the proof. □

Theorem 5.14 (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* $q \leq \llbracket \mathsf{r} \rrbracket p$ *and* $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(q)$*, then the triple* $\vdash_A [p]\ \mathsf{r}\ [q]$ *can be derived in the proof system extended with the* (limit) *rule.*

Proof. We proceed by structural induction on r.

($\mathsf{r} \equiv \mathsf{e}$): By hypothesis, $\llbracket \mathsf{e} \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(\llbracket \mathsf{e} \rrbracket p)$, thus, by Lemma 2.1 (b), $\alpha(\llbracket \mathsf{e} \rrbracket p) = \alpha(\llbracket \mathsf{e} \rrbracket \gamma\alpha(p))$. Hence, $\mathbb{C}^{A}_{p}(\llbracket \mathsf{e} \rrbracket)$ holds, so that by (transfer), $\vdash_A [p]\ \mathsf{e}\ [\llbracket \mathsf{e} \rrbracket p]$.

($\mathsf{r} \equiv \mathsf{r}_1; \mathsf{r}_2$): Assume that $q \leq \llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket p$ and $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(q) = \alpha(\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket p)$. Let $w \triangleq \llbracket \mathsf{r}_1 \rrbracket p$. Then, we have that:

(1) $w \leq \llbracket \mathsf{r}_1 \rrbracket p$ and, by Lemma 5.13, $\llbracket \mathsf{r}_1 \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(\llbracket \mathsf{r}_1 \rrbracket p) = \alpha(w)$;
(2) $q \leq \llbracket \mathsf{r}_2 \rrbracket w$ and, by Lemma 5.13, $\llbracket \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(w) = \alpha(\llbracket \mathsf{r}_2 \rrbracket w) = \alpha(\llbracket \mathsf{r}_2 \rrbracket \llbracket \mathsf{r}_1 \rrbracket p) = \alpha(\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket p) = \alpha(q)$.

Thus, by induction, we derive $\vdash_A [p]\ \mathsf{r}_1\ [w]$ and $\vdash_A [w]\ \mathsf{r}_2\ [q]$, so that by (seq), $\vdash_A [p]\ \mathsf{r}_1; \mathsf{r}_2\ [q]$ follows.

($\mathsf{r} \equiv \mathsf{r}_1 \oplus \mathsf{r}_2$): Assume that $q \leq \llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket p$ and $\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(q) = \alpha(\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket p)$. Let $q_i \triangleq \llbracket \mathsf{r}_i \rrbracket p$. Then, we have that $q_i \leq \llbracket \mathsf{r}_i \rrbracket p$ and, by Lemma 5.13, $\llbracket \mathsf{r}_i \rrbracket^{\sharp}_{A} p = \alpha(\llbracket \mathsf{r}_i \rrbracket p) = \alpha(q_i)$. Thus, by induction, $\vdash_A [p]\ \mathsf{r}_i\ [q_i]$ so that by (join), $\vdash_A [p]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [q_1 \vee q_2]$. Then, because $q \leq \llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket p = \llbracket \mathsf{r}_1 \rrbracket p \vee \llbracket \mathsf{r}_2 \rrbracket p = (q_1 \vee q_2) \leq \gamma\alpha(\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket p) = \gamma\alpha(q)$, by (relax), $\vdash_A [p]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [q]$ follows.

($\mathsf{r} \equiv \mathsf{r}_1^{*}$): Assume that $q \leq \llbracket \mathsf{r}_1^{*} \rrbracket p$ and $\llbracket \mathsf{r}_1^{*} \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(q) = \alpha(\llbracket \mathsf{r}_1^{*} \rrbracket p)$. Let us consider the $\mathbb{N}$-indexed sequence $\{p_n\}_{n \in \mathbb{N}}$ of values in C, where $p_n \triangleq \llbracket \mathsf{r}_1 \rrbracket^{n} p$. We have that for all $n \in \mathbb{N}$, $p_{n+1} \leq \llbracket \mathsf{r}_1 \rrbracket p_n$ and, by Lemma 5.13, $\llbracket \mathsf{r}_1 \rrbracket^{\sharp}_{A} \alpha(p_n) = \alpha(\llbracket \mathsf{r}_1 \rrbracket p_n) = \alpha(p_{n+1})$. Thus, by induction, for all $n \in \mathbb{N}$, $\vdash_A [p_n]\ \mathsf{r}_1\ [p_{n+1}]$ is derivable. Hence, by (limit), we derive $\vdash_A [p]\ \mathsf{r}_1^{*}\ [\bigvee_{n \in \mathbb{N}} p_n]$. Notice that $\bigvee_{n \in \mathbb{N}} p_n = \bigvee \{\llbracket \mathsf{r}_1 \rrbracket^{n} p \mid n \in \mathbb{N}\} = \llbracket \mathsf{r}_1^{*} \rrbracket p$. Therefore, $q \leq \llbracket \mathsf{r}_1^{*} \rrbracket p \leq \gamma\alpha(\llbracket \mathsf{r}_1^{*} \rrbracket p) = \gamma\alpha(q)$, and by (relax) it follows $\vdash_A [p]\ \mathsf{r}_1^{*}\ [q]$. □

It is worth noting that if any $\mathsf{e} \in \mathsf{Exp}(\mathsf{r})$ is globally complete on A then, as proved by [Giacobazzi et al. 2015, Theorem 5.1], $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha = \alpha \llbracket \mathsf{r} \rrbracket$ also holds. Thus, the hypotheses of Theorem 5.14 imply that that properties (1–2) of the soundness Theorem 5.5 all hold, i.e. Theorem 5.14 provides a result of (limited) logical completeness for LCL_A. Vice versa, 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 5.5) but LCL_A is unable to prove it.

Theorem 5.15 (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* $q \leq \llbracket \mathsf{r} \rrbracket p$ *and* $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(p) = \alpha(\llbracket \mathsf{r} \rrbracket p) = \alpha(q)$*, but* $\nvdash_A [p]\ \mathsf{r}\ [q]$*.*

Proof. The assumption of Turing completeness of Reg implies the possibility of defining regular commands $\mathsf{r}_{=c?}, \mathsf{r}_{\neq c?} \in \mathsf{Reg}$, for any store $c \in \Sigma$, such that, for any $p \in \mathbb{S}$, $\llbracket \mathsf{r}_{=c?} \rrbracket p = p \cap \{c\}$ and $\llbracket \mathsf{r}_{\neq c?} \rrbracket p = p \cap \neg\{c\}$. Likewise, Turing completeness implies the possibility of defining regular commands $\mathsf{r}_c \in \mathsf{Reg}$, for any $c \in \Sigma$, such that, for any $\sigma \in \Sigma$ we have $\llbracket \mathsf{r}_c \rrbracket \{\sigma\} = \{c\}$. Finally, Turing completeness implies that there exists $\mathsf{w} \in \mathsf{Reg}$ such that for any $p \in \mathbb{S}$, $\llbracket \mathsf{w} \rrbracket p = \varnothing$ holds. For the sake of clarity, in Imp and assuming just one variable x so that $c \in \mathbb{Z}$, these programs would be:

$$\begin{aligned} \mathsf{w} &\triangleq \textbf{while}\ \ \textbf{tt}\ \ \textbf{do skip} \\ \mathsf{r}_{=c?} &\triangleq \textbf{if}\ (x = c)\ \textbf{then skip else}\ \mathsf{w} \\ \mathsf{r}_{\neq c?} &\triangleq \textbf{if}\ \neg(x = c)\ \textbf{then skip else}\ \mathsf{w} \\ \mathsf{r}_c &\triangleq x := c \end{aligned}$$

Consider now $A_{\alpha,\gamma} \in \mathrm{Abs}(\mathbb{S})$ such that $\gamma\alpha \neq id$ and $\gamma\alpha \neq \lambda x.\Sigma$. Then there exists $p \in \wp(\Sigma)$ such that $p \subsetneq A(p)$ (because $\gamma\alpha \neq id$) and $w \in \wp(\Sigma)$ such that $A(w) \subsetneq \Sigma$ (because $\gamma\alpha \neq \lambda x.\Sigma$). Let r_b, $\mathsf{r}_{=a?}$, and $\mathsf{r}_{\neq a?}$ be regular commands defined as above for some $a \in A(p) \setminus p$ and $b \in \Sigma \setminus A(w)$. Now, take the program $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{w}$ where

$$\mathsf{r}_1 \triangleq (\mathsf{r}_{=a?}; \mathsf{r}_b) \oplus (\mathsf{r}_{\neq a?}; \mathsf{w}).$$

We have that $[\![\mathsf{r}]\!]p = [\![\mathsf{w}]\!]([\![\mathsf{r}_1]\!]p) = \varnothing$ and $[\![\mathsf{r}]\!]A(p) = [\![\mathsf{w}]\!]([\![\mathsf{r}_1]\!]A(p)) = \varnothing$, so that local completeness $A([\![\mathsf{r}]\!]p) = A([\![\mathsf{r}]\!]A(p))$ holds. However, the triple $\vdash_A [p]\ \mathsf{r}\ [\varnothing]$ cannot be derived, because we cannot derive any triple $\vdash_A [p]\ \mathsf{r}_1\ [q]$ (to be used by the rule (seq) for $\mathsf{r}_1; \mathsf{w}$). This is a consequence of the fact that $[\![\mathsf{r}_1]\!]$ is not locally complete on p in A. In fact, $[\![\mathsf{r}_1]\!]p = \varnothing$ while $[\![\mathsf{r}_1]\!]A(p) = [\![(\mathsf{r}_{=a?}; \mathsf{r}_b) \oplus (\mathsf{r}_{\neq a?}; \mathsf{w})]\!]A(p) = [\![\mathsf{r}_{=a?}; \mathsf{r}_b]\!]A(p) \cup [\![\mathsf{r}_{\neq a?}; \mathsf{w}]\!]A(p) = \{b\} \cup \varnothing = \{b\}$. Hence, $\alpha([\![\mathsf{r}_1]\!]p) = \alpha(\varnothing)$ while $[\![\mathsf{r}_1]\!]^\sharp_A \alpha(p) \supseteq \alpha([\![\mathsf{r}_1]\!]A(p)) = \alpha(\{b\})$ (by soundness of $[\![\mathsf{r}_1]\!]^\sharp_A$ and monotonicity of α). We observe that $\alpha(\varnothing) \neq \alpha(\{b\})$, otherwise by monotonicity of γ and extensivity of $\gamma\alpha$, we would have $b \in \gamma\alpha(\{b\}) = \gamma\alpha(\varnothing) \subseteq \gamma\alpha(w) = A(w)$ which would be a contradiction. Now suppose that $\vdash_A [p]\ \mathsf{r}_1\ [q]$ for some q. By Theorem 5.5 (1), it must be $q \subseteq [\![\mathsf{r}_1]\!]p = \varnothing$, thus $q = \varnothing$ and by Theorem 5.5 (2), we would obtain $\alpha(\varnothing) = \alpha(q) = [\![\mathsf{r}_1]\!]^\sharp_A \alpha(p) \supseteq \alpha(\{b\})$ which is a contradiction. The rule (relax) cannot help here neither, because any strengthening of p, say p', such that $p' \subseteq p \subseteq A(p')$ makes $A(p) = A(p')$ therefore making $[\![\mathsf{r}_1]\!]$ incomplete also for p' in A. □

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 5.15 generalizes to LCL_A and to Turing complete regular commands the proofs in [Bruni et al. 2020] and [Giacobazzi et al. 2015]: the class of all programs for which an abstract interpretation on A is globally complete is the set of all programs if and only if A is trivial. As a consequence of Theorems 5.14 and 5.15, 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 $A = \lambda x.\Sigma$. In light of this, we may therefore observe that logical completeness of IL [O'Hearn 2020, Theorem 6] follows from the choice made in its consequence rule of letting pre-conditions and post-conditions be, respectively, arbitrarily weakened and strengthened, i.e., the choice of the abstraction $A = \lambda x.\Sigma$ in rule (relax). Section 6 provides additional details on the relationship with incorrectness logic.

5.3 A Backward Proof System

As pioneered by [Cousot and Cousot 1977, Section 5.2], it is known that abstract interpretation-based program analysis can be defined either backward or forward (see [Bourdoncle 1993; Miné 2017] for examples of backward abstract interpretations). 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 [1993] for abstract program debugging. An under-approximating backward program analysis by abstract interpretation has been put forward by Miné [2014]. In the following, we show how LCL_A can be easily dualized in order to accomodate backward analyses.

In backward analysis, the basic expressions $\mathsf{e} \in \mathsf{Exp}$ have a co-additive backward concrete semantics $(\!|\mathsf{e}|\!)_{\leftarrow} : C \to C$. Notably, $(\!|\mathsf{e}|\!)_{\leftarrow} Y \triangleq \{\sigma \mid (\sigma, \sigma') \in R_{\mathsf{e}} \Rightarrow \sigma' \in Y\}$, where $R_{\mathsf{e}} \triangleq \{(\sigma, (\!|\mathsf{e}|\!)\sigma) \mid \sigma \in \Sigma\}$ is the transition relation for e. For example, for the basic commands of Imp (Section 2.2.3) 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 (2):

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

This defines a standard backward semantics, e.g., as given in [Miné 2014, Section 2] for imperative programs.

By duality, 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 the partial order is the inverse relation $\geq \triangleq \leq^{-1}$. This means that for all $c \in C$, an under-approximation relation $\gamma\alpha(c) \leq c$ replaces an over-approximating relation $c \leq \gamma\alpha(c)$.

Example 5.16. The interval domain Int can be viewed as an under-approximating abstraction by dualizing its abstraction and concretization 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 from (3) by duality as follows:

$$\begin{aligned}
[\![\mathsf{e}]\!]^{\sharp}_{U} a &\triangleq \alpha((\!|\mathsf{e}|\!)_{\leftarrow} \gamma(a)) \\
[\![\mathsf{r}_1; \mathsf{r}_2]\!]^{\sharp}_{U} a &\triangleq [\![\mathsf{r}_1]\!]^{\sharp}_{U}([\![\mathsf{r}_2]\!]^{\sharp}_{A} a) \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^{\sharp}_{U} a &\triangleq [\![\mathsf{r}_1]\!]^{\sharp}_{U} a \wedge_U [\![\mathsf{r}_2]\!]^{\sharp}_{U} a \\
[\![\mathsf{r}^*]\!]^{\sharp}_{U} a &\triangleq \bigwedge_U \{([\![\mathsf{r}]\!]^{\sharp}_{U})^{n} a \mid n \in \mathbb{N}\}
\end{aligned}$$

Correspondingly, when our proof system $\vdash_U$ is instantiated to an under-approximating abstraction $U \in \mathrm{Abs}^{\leftarrow}(C)$, we need to replace $\leq$, which models logical implication, with $\geq$ and logical disjunction $\vee$ with conjunction $\wedge$. Hence, the fundamental (relax) rule becomes:

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

Fig. 8. Basic Transfer Functions for Additional Statements.

$$
\frac{U(p') \le p \le p' \quad \vdash_U [p']\ \mathsf{r}\ [q'] \quad U(q) \le q' \le q}{\vdash_U [p]\ \mathsf{r}\ [q]}
$$

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 5.5 and 5.14, 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: $[\![\mathsf{r}]\!]_{\leftarrow} q \le p$, 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.

6 RELATIONSHIP WITH INCORRECTNESS LOGIC

The idea to reverse the direction of implication in Hoare's consequence rule was investigated by de Vries and Koutavas [2011] reverse Hoare logic to study reachability specifications for randomized algorithms. They first put forward the following consequence rule for under-approximation triples:

$$
\frac{p' \le p \quad [p']\ \mathsf{r}\ [q'] \quad q \le q'}{[p]\ \mathsf{r}\ [q]}\ (\mathsf{cons})
$$

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 [2020], "*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 [O'Hearn 2020], 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 $x := \mathsf{nond}()$ also present in the setting of reverse Hoare logic. Thus, we set:

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

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 [O'Hearn 2020], 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 $\vdash_{\mathrm{IL}} [p]\ \mathsf{c}\ [\mathbf{ok}:q][\mathbf{er}:w]$ when $\vdash_{\mathrm{IL}} [p]\ \mathsf{c}\ [\mathbf{ok}:q]$ and $\vdash_{\mathrm{IL}} [p]\ \mathsf{c}\ [\mathbf{er}:w]$ 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 an 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 exploit the following four ingredients:

(i) A suitable concrete domain $C \triangleq \wp(\{\mathbf{ok}, \mathbf{er}\} \times \Sigma)$ 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 $S \in C$ takes the form $\mathbf{ok} : q \cup \mathbf{er} : w$ for suitable $q, w \in \wp(\Sigma)$, so we denote elements in C more concisely as $\mathbf{ok} : q, \mathbf{er} : w$.

(ii) Transfer functions $f : C \to C$ are additive functions such that

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

meaning that all the errors w in the argument 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 Figure 8, where we write $q[x \mapsto v]$ as a shorthand for $\{\sigma[x \mapsto v] \mid \sigma \in q\}$. For any $\mathsf{r} \in \mathsf{Reg}$ and any $q, w \in \wp(\Sigma)$, by structural induction on r, it follows that $[\![\mathsf{r}]\!](\mathbf{er} : w) = \mathbf{er} : w$ and $[\![\mathsf{r}]\!](\mathbf{ok} : q, \mathbf{er} : w) = [\![\mathsf{r}]\!](\mathbf{ok} : q) \cup \mathbf{er} : w$.

(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 $[\![\mathsf{r}]\!] : C \to C$ 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 [O'Hearn 2020, 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 6.1. *For any* $\mathsf{r} \in \mathsf{Reg}$ *and* $p \in \mathbb{S}$*, we have:*

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

Proof. The proof is by structural induction on $\mathsf{r} \in \mathsf{Reg}$.

($\mathsf{r} \equiv \mathbf{skip}$): By definition,

$$[\![\mathbf{skip}]\!](\mathbf{ok} : p) \quad = \quad \mathbf{ok} : p, \mathbf{er} : \varnothing \quad = \quad \mathbf{ok} : [\![\mathbf{skip}]\!]_{\mathbf{ok}} p, \mathbf{er} : [\![\mathbf{skip}]\!]_{\mathbf{er}} p.$$

($\mathsf{r} \equiv x := \mathsf{a}$): By definition,

$$[\![x := \mathsf{a}]\!](\mathbf{ok} : p) \quad = \quad \mathbf{ok} : \textstyle\bigcup_{\sigma \in p} \{\sigma[x \mapsto \{\!|\mathsf{a}|\!\} \sigma]\}, \mathbf{er} : \varnothing \quad = \quad \mathbf{ok} : [\![x := \mathsf{a}]\!]_{\mathbf{ok}} p, \mathbf{er} : [\![x := \mathsf{a}]\!]_{\mathbf{er}} p.$$

($\mathsf{r} \equiv \mathsf{error}()$): By definition,

$$[\![\mathsf{error}()]\!](\mathbf{ok} : p) \quad = \quad \mathbf{ok} : \varnothing, \mathbf{er} : p \quad = \quad \mathbf{ok} : [\![\mathsf{error}()]\!]_{\mathbf{ok}} p, \mathbf{er} : [\![\mathsf{error}()]\!]_{\mathbf{er}} p.$$

($\mathsf{r} \equiv \mathsf{assume}(\mathsf{b})$): By definition,

$$[\![\mathsf{assume}(\mathsf{b})]\!](\mathbf{ok} : p) \quad = \quad \mathbf{ok} : (p \cap \mathsf{b}), \mathbf{er} : \varnothing \quad = \quad \mathbf{ok} : [\![\mathsf{assume}(\mathsf{b})]\!]_{\mathbf{ok}} p, \mathbf{er} : [\![\mathsf{assume}(\mathsf{b})]\!]_{\mathbf{er}} p.$$

($\mathsf{r} \equiv x := \mathsf{nond}()$): By definition,

$$\llbracket x := \mathsf{nond}() \rrbracket(\mathbf{ok}:p) \quad = \quad \mathbf{ok}:\textstyle\bigcup_{v\in\mathbb{Z}} p[x \mapsto v], \mathbf{er}:\varnothing \quad = \quad \mathbf{ok}:\llbracket x := \mathsf{nond}() \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket x := \mathsf{nond}() \rrbracket_{\mathbf{er}} p.$$

($\mathsf{r} \equiv \mathsf{r}_1; \mathsf{r}_2$): Let us assume the inductive hypotheses

$$\forall p \in \mathbb{S}\,.\ \llbracket \mathsf{r}_1 \rrbracket(\mathbf{ok}:p) = \mathbf{ok}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p$$
$$\forall p' \in \mathbb{S}\,.\ \llbracket \mathsf{r}_2 \rrbracket(\mathbf{ok}:p') = \mathbf{ok}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{ok}} p', \mathbf{er}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{er}} p'.$$

Then, we have

$$\begin{aligned}
\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket(\mathbf{ok}:p) = \llbracket \mathsf{r}_2 \rrbracket(\llbracket \mathsf{r}_1 \rrbracket(\mathbf{ok}:p)) &= \\
\llbracket \mathsf{r}_2 \rrbracket(\mathbf{ok}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p) &= \\
\mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p \cup \llbracket \mathsf{r}_2 \rrbracket(\mathbf{ok}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p) &= \\
\mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p \cup \mathbf{ok}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{ok}}(\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p), \mathbf{er}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{er}}(\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p) &= \\
\mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p \cup \mathbf{ok}:(\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} \cdot \llbracket \mathsf{r}_2 \rrbracket_{\mathbf{ok}})p), \mathbf{er}:(\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} \cdot \llbracket \mathsf{r}_2 \rrbracket_{\mathbf{er}})p) &= \\
\mathbf{ok}:(\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket_{\mathbf{ok}} p), \mathbf{er}:(\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p \cup (\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} \cdot \llbracket \mathsf{r}_2 \rrbracket_{\mathbf{er}})p)) &= \\
\mathbf{ok}:\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket_{\mathbf{er}} p\ &.
\end{aligned}$$

($\mathsf{r} \equiv \mathsf{r}_1 \oplus \mathsf{r}_2$): Let us assume the inductive hypotheses

$$\forall p \in \mathbb{S}\,.\ \llbracket \mathsf{r}_1 \rrbracket(\mathbf{ok}:p) = \mathbf{ok}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p,$$
$$\forall p \in \mathbb{S}\,.\ \llbracket \mathsf{r}_2 \rrbracket(\mathbf{ok}:p) = \mathbf{ok}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{er}} p.$$

Then, we have

$$\begin{aligned}
\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket(\mathbf{ok}:p) = \llbracket \mathsf{r}_1 \rrbracket(\mathbf{ok}:p) \cup \llbracket \mathsf{r}_2 \rrbracket(\mathbf{ok}:p) &= \\
(\mathbf{ok}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p) \cup (\mathbf{ok}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_2 \rrbracket_{\mathbf{er}} p) &= \\
\mathbf{ok}:(\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p \cup \llbracket \mathsf{r}_2 \rrbracket_{\mathbf{ok}} p), \mathbf{er}:(\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p \cup \llbracket \mathsf{r}_2 \rrbracket_{\mathbf{er}} p) &= \\
\mathbf{ok}:\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket_{\mathbf{er}} p\ &.
\end{aligned}$$

($\mathsf{r} \equiv \mathsf{r}_1^*$): Let us assume the inductive hypotheses

$$\forall p \in \mathbb{S}\,.\ \llbracket \mathsf{r}_1 \rrbracket(\mathbf{ok}:p) = \mathbf{ok}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1 \rrbracket_{\mathbf{er}} p.$$

Then, we prove by induction on $n \in \mathbb{N}$ that

$$\forall n \in \mathbb{N}.\ \llbracket \mathsf{r}_1 \rrbracket^n(\mathbf{ok}:p) = \mathbf{ok}:\llbracket \mathsf{r}_1^n \rrbracket_{\mathbf{ok}} p, \mathbf{er}:\llbracket \mathsf{r}_1^n \rrbracket_{\mathbf{er}} p.$$

Finally, we have that

$$\begin{aligned}
[\![\mathsf{r}_1^*]\!](\mathbf{ok}:p) &= \\
\bigcup\{[\![\mathsf{r}_1]\!]^n(\mathbf{ok}:p) \mid n \in \mathbb{N}\} &= \\
\bigcup\{(\mathbf{ok}:[\![\mathsf{r}_1^n]\!]_{\mathbf{ok}}p, \mathbf{er}:[\![\mathsf{r}_1^n]\!]_{\mathbf{er}}p) \mid n \in \mathbb{N}\} &= \\
\mathbf{ok}:\bigcup\{[\![\mathsf{r}_1^n]\!]_{\mathbf{ok}}p \mid n \in \mathbb{N}\}, \mathbf{er}:\bigcup\{[\![\mathsf{r}_1^n]\!]_{\mathbf{er}}p \mid n \in \mathbb{N}\} &= \\
\mathbf{ok}:[\![\mathsf{r}_1^*]\!]_{\mathbf{ok}}p, \mathbf{er}:[\![\mathsf{r}_1^*]\!]_{\mathbf{er}}p &.
\end{aligned}$$

This therefore concludes the proof. □

Lemma 6.1 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 [O'Hearn 2020, Theorem 6] and our Theorem 5.14 of limited completeness, which guarantee that any under-approximation of the strongest post-condition $\mathrm{post}[\mathsf{r}]p$ is provable in both logics.

Corollary 6.2. *For any $p, q, w \in \mathbb{S}$ and $\mathsf{r} \in \mathsf{Reg}$:*

$$\vdash_{\mathrm{IL}} [p]\ \mathsf{r}\ [\mathbf{ok}:q][\mathbf{er}:w] \quad \textit{iff} \quad \vdash_{A_{tr}} [\mathbf{ok}:p]\ \mathsf{r}\ [\mathbf{ok}:q, \mathbf{er}:w].$$

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 5.15 that extends the impossibility results of [Bruni et al. 2020; Giacobazzi et al. 2015] to regular commands. Therefore, whenever $A \neq A_{tr}$ there will always exist some program r and some triple $\vdash_{\mathrm{IL}} [p]\ \mathsf{r}\ [\mathbf{ok}:q][\mathbf{er}:w]$ such that it will not be possible to derive $\vdash_A [\mathbf{ok}:p]\ \mathsf{r}\ [\mathbf{ok}:q, \mathbf{er}:w]$ because some proof obligation $\mathbb{C}_c^A(\mathsf{e})$ of local completeness will fail for some basic expression e appearing in r.

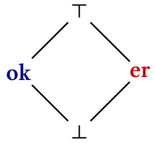
The correspondence provided by Corollary 6.2 is interesting, because although the rules of IL and ours share some similarities, they also display significant differences:

(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 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 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 [O'Hearn 2020, Figure 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 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.



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

Fig. 9. Abstract Domain Err

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 Figure 9.

Example 6.3. Let us revisit Example 5.8 (in turn using the regular command r of Example 5.2). To study $spec_2 \triangleq x \leq 1000$, in the context of $\vdash_{\mathrm{IL}}$ we focus on the command $\widehat{\mathsf{r}}_2 \triangleq \mathsf{r}; \mathsf{r}_{s2}$ where

$$\mathsf{r}_{s2} \triangleq \ \mathbf{if}\ x \leq 1000\ \mathbf{then\ skip\ else}\ \mathsf{error}() \ = \ (x \leq 1000?; \mathbf{skip}) \oplus (1000 < x?; \mathsf{error}()).$$

Using $\vdash_{\mathsf{Int}}$, it was shown in Example 5.8 that $\vdash_{\mathsf{Int}}$ is expressive enough to prove 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 [Cousot and Cousot 1979, 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. Actually, by Corollary 5.6, since $spec_2$ is expressible in Int, we are guaranteed that any provable triple $\vdash_{\mathsf{Int}} [p]\ \mathsf{r}\ [q]$ in our system will be able to prove that $spec_2$ holds.

To study $spec_3 \triangleq 100 \leq x$, we focus on $\widehat{\mathsf{r}}_3 \triangleq \mathsf{r}; \mathsf{r}_{s3}$ where

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

We know from Example 5.8, 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., $\vdash_{\mathrm{IL}} [p]\ \widehat{\mathsf{r}}_3\ [\mathbf{ok}\!:\{1000\}]$.

Let us consider once again the abstract domain Int^+. 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 5.6, 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 alarm $\mathbf{er}\!:\{0\}$ (as well as $\mathbf{ok}\!:\{1000\}$). □

$$\frac{\vdash_A [p \wedge \mathsf{b}]\ \mathsf{c}\ [w] \quad \vdash_A [p \vee w]\ \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}\ [q]}{\vdash_A [p]\ \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}\ [q]}\ (\mathsf{unroll}) \qquad \frac{\mathbb{C}^A_p(\mathsf{b}) \quad \mathbb{C}^A_p(\neg\mathsf{b}) \quad \vdash_A [p \wedge \mathsf{b}]\ \mathsf{c}\ [q] \quad q \le A(p)}{\vdash_A [p]\ \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}\ [(p \vee q) \wedge \neg\mathsf{b}]}\ (\mathsf{loopinv})$$

Fig. 10. The Rules for While Loops.

In order to develop some tool or extend existing ones to exploit LCL_A the idea is to start the analysis with some well-known abstract domain and then collect the local completeness proof obligations arising from the analysis: when they are not satisfied, then a new abstract domain must be synthesized to carry out the proof. As discussed in the conclusions, a successful strategy outlined in [Bruni et al. 2022] would be to refine the current abstract domain in a minimal way to recover local completeness of the analysis.

7 Imp PROGRAMS

The encoding (6) of while loops as regular commands is such that, due to the recursive rule (rec), proof obligations of local completeness for the Boolean guard of the loop are required at each iteration. This turns out to be advantageous to keep the rules of the logic as simple and straightforward as possible. However, this is not strictly necessary for achieving local completeness of the loop invariant. We therefore introduce in Figure 10 two additional rules for while loops that relax the local completeness requirements needed during their recursive applications. The key new rule is (unroll): it is analogous to the rule (rec) when applied to $\mathsf{r} = \mathsf{b}?; \mathsf{c}$, but the premise $\vdash_A [p \wedge \mathsf{b}]\ \mathsf{c}\ [w]$ of (unroll) requires local completeness exclusively on the body c of the loop and not on the Boolean guard b?. Moreover, the rule (loopinv) is analogous to (iterate) and can be used to stop a proof attempt as soon as an abstract fixpoint $A(p)$ is reached. By combining these rules (unroll) and (loopinv), the local completeness requirements for the Boolean guard b? of the loop, namely $\mathbb{C}^A_p(\mathsf{b}?)$ and $\mathbb{C}^A_p(\neg\mathsf{b}?)$, are therefore needed only at the last iteration and not at each iteration.

Lemma 7.1. *The proof system extended with rules* (unroll) *and* (loopinv) *is sound.*

Proof. We extend the proof by induction of Theorem 5.5 by considering the case where the last rule applied to derive $\vdash_A [p]\ \mathsf{r}\ [q]$ is either (unroll) or (loopinv). As in the proof of Theorem 5.5, we refer to the equality $[\![\mathsf{r}]\!]^\sharp_A \alpha(p) = \alpha(q)$ as (2a) and use (2b) for the equality $[\![\mathsf{r}]\!]^\sharp_A \alpha(p) = \alpha([\![\mathsf{r}]\!]p)$, and we recall that (2b) follows immediately by (1) and (2a).

(unroll): If the last rule applied is (unroll), it must be $\mathsf{r} \equiv \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}$ and we can assume by induction that $\vdash_A [p \wedge \mathsf{b}]\ \mathsf{c}\ [w]$ and $\vdash_A [p \vee w]\ \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}\ [q]$ can be derived for some w. We need to prove the thesis for the conclusion $\vdash_A [p]\ \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}\ [q]$.
(1) By inductive hypothesis (1), we have that $w \le [\![\mathsf{c}]\!](p \wedge \mathsf{b}) = [\![\mathsf{b}?; \mathsf{c}]\!]p$, so that, by Lemma 2.1 (e), it follows $[\![(\mathsf{b}?; \mathsf{c})^*]\!]p = [\![(\mathsf{b}?; \mathsf{c})^*]\!](p \vee w)$. Then, by inductive hypothesis (1), we derive:

$$q \le [\![\mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}]\!](p \vee w) = [\![\neg\mathsf{b}?]\!]([\![(\mathsf{b}?; \mathsf{c})^*]\!](p \vee w)) = [\![\neg\mathsf{b}?]\!]([\![(\mathsf{b}?; \mathsf{c})^*]\!]p) = [\![\mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}]\!]p.$$

(2) The proof is analogous to the case of rule (rec) in the proof of Theorem 5.5:

$$\begin{aligned} \alpha(q) \leq_A & \quad \text{[by monotonicity of } \alpha \text{ and (1)]} \\ \alpha(\llbracket \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \rrbracket p) \leq_A & \quad \text{[by correctness (4)]} \\ \llbracket \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \rrbracket^\sharp_A \alpha(p) \leq_A & \quad \text{[by monotonicity]} \\ \llbracket \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \rrbracket^\sharp_A \alpha(p \vee w) = & \quad \text{[by ind.hyp. (2a)]} \\ \alpha(q) & \end{aligned}$$

so that $\llbracket \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \rrbracket^\sharp_A \alpha(p) = \alpha(q) = \alpha(\llbracket \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \rrbracket p)$ follows.

(loopinv): If the last rule applied is (loopinv), it must be $\mathsf{r} \equiv \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}$ and we can assume by induction that $\vdash_A [p \wedge \mathsf{b}]\ \mathsf{c}\ [q_1]$ can be derived for some q_1 such that $q_1 \leq A(p)$ and $q = (p \vee q_1) \wedge \neg\mathsf{b}$, with $\mathbb{C}^A_p(\mathsf{b})$ and $\mathbb{C}^A_p(\neg\mathsf{b})$. We need to prove the thesis for the conclusion $\vdash_A [p]\ \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c}\ [(p \vee q_1) \wedge \neg\mathsf{b}]$.
(1) We first observe that, by inductive hypothesis (1), $q_1 \leq \llbracket \mathsf{c} \rrbracket (p \wedge \mathsf{b}) = \llbracket \mathsf{b}?;\mathsf{c} \rrbracket p$ and that $\llbracket \mathsf{b}?;\mathsf{c} \rrbracket^0 p = p$. Then:

$$p \vee q_1 \leq \llbracket \mathsf{b}?;\mathsf{c} \rrbracket^0 p \vee \llbracket \mathsf{b}?;\mathsf{c} \rrbracket^1 p \leq \bigvee_{n \in \mathbb{N}} \llbracket \mathsf{b}?;\mathsf{c} \rrbracket^n p = \llbracket (\mathsf{b}?;\mathsf{c})^* \rrbracket p.$$

From this, it immediately follows by monotonicity of $\llbracket \neg\mathsf{b} \rrbracket$:

$$(p \vee q_1) \wedge \neg\mathsf{b} = \llbracket \neg\mathsf{b} \rrbracket (p \vee q_1) \leq \llbracket \neg\mathsf{b} \rrbracket (\llbracket (\mathsf{b}?;\mathsf{c})^* \rrbracket p = \llbracket \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \rrbracket p.$$

(2a) By $\mathbb{C}^A_p(\mathsf{b})$, we get $\alpha(p \wedge \mathsf{b}) = \alpha(\llbracket \mathsf{b} \rrbracket p) = \llbracket \mathsf{b} \rrbracket^\sharp_A \alpha(p)$. Therefore, $\alpha(q_1) = \llbracket \mathsf{c} \rrbracket^\sharp_A \alpha(p \wedge \mathsf{b}) = \llbracket \mathsf{c} \rrbracket^\sharp_A \llbracket \mathsf{b} \rrbracket^\sharp_A \alpha(p) = \llbracket \mathsf{b}?;\mathsf{c} \rrbracket^\sharp_A \alpha(p)$. By monotonicity of α, we have that $\alpha(q_1) \leq_A \alpha(p)$, so that Lemma 2.1 (c) is applicable to derive $\llbracket (\mathsf{b}?;\mathsf{c})^* \rrbracket^\sharp_A \alpha(p) = \alpha(p)$. Since $\mathbb{C}^A_p(\neg\mathsf{b})$ and $p \leq (p \vee q_1) \leq A(p)$ hold, by convexity, we get that $\mathbb{C}^A_{(p \vee q_1)}(\neg\mathsf{b})$ holds. Therefore:

$$\begin{aligned} \alpha((p \vee q_1) \wedge \neg\mathsf{b}) = & \quad \text{[by definition]} \\ \alpha(\llbracket \neg\mathsf{b}? \rrbracket (p \vee q_1)) = & \quad \text{[by convexity } \mathbb{C}^A_{(p \vee q_1)}(\neg\mathsf{b})\text{]} \\ \llbracket \neg\mathsf{b}? \rrbracket^\sharp_A \alpha(p \vee q_1) = & \quad \text{[by additivity of } \alpha\text{]} \\ \llbracket \neg\mathsf{b}? \rrbracket^\sharp_A (\alpha(p) \vee_A \alpha(q_1)) = & \quad \text{[by hyp. } q_1 \leq A(p)\text{]} \\ \llbracket \neg\mathsf{b}? \rrbracket^\sharp_A \alpha(p) = & \quad \text{[by Lemma 2.1 (c), see above]} \\ \llbracket \neg\mathsf{b}? \rrbracket^\sharp_A (\llbracket (\mathsf{b}?;\mathsf{c})^* \rrbracket^\sharp_A \alpha(p)) = & \quad \text{[by definition]} \\ \llbracket \mathbf{while}\ \mathsf{b}\ \mathbf{do}\ \mathsf{c} \rrbracket^\sharp_A \alpha(p). & \end{aligned}$$

□

We refer to the next section for an example (cf. Example 8.2) showing the advantage of applying the rules (unroll) and (loopinv) because the local completeness requirements are not met at every iteration of the proof for a while loop but just at the last one.

8 A LOGIC FOR LOCALLY COMPLETE BEST CORRECT APPROXIMATIONS

When a provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ is used for program verification, we exploit the following property:

$$q \leq \llbracket \mathsf{r} \rrbracket p \leq \alpha(\llbracket \mathsf{r} \rrbracket p) = \alpha(q) \qquad (\S)$$

$$\frac{A' \leq A \quad A(p) = A'(p) \quad \vdash^{\lesssim}_{A'} [p]\ \mathsf{r}\ [q] \quad A'(q) = A(q)}{\vdash^{\lesssim}_{A} [p]\ \mathsf{r}\ [q]}\ (\mathsf{refine})$$

Fig. 11. The Refinement Rule of $\mathrm{LCL}^{\lesssim}_{A}$.

so that, by under-approximation $q \leq [\![\mathsf{r}]\!]p$, any alarm in q is a true alarm for p, and, by over-approximation $[\![\mathsf{r}]\!]p \leq \alpha(q)$, the lack of alarms in $\alpha(q)$ entails the correctness of p. In LCL_A, local completeness can be viewed as a technical assumption to infer (§), meaning that it is a necessary condition for deriving $\vdash_A [p]\ \mathsf{r}\ [q]$ within our proof system, since we are able to prove only triples that satisfy local completeness $[\![\mathsf{r}]\!]^{\sharp}_{A}\alpha(p) = \alpha([\![\mathsf{r}]\!]p)$ (cf. Theorem 5.5). In this section, we show that it is possible to relax our program logic so that local completeness is not required for the intensional and inductively defined abstract interpreter $[\![\mathsf{r}]\!]^{\sharp}_{A}$ but merely for the best correct approximation $[\![\mathsf{r}]\!]^{A} \triangleq \alpha \circ [\![\mathsf{r}]\!] \circ \gamma$ of the extensional concrete semantics $[\![\mathsf{r}]\!]$. This is achieved by allowing different abstract domains in different sub-derivations to increase the precision of the analysis whenever necessary. Without the extension proposed in this section, whenever it would be convenient to use different abstract domains for different parts of the proof, the only possibility would be to check if it is possible to complete the derivation in the abstract domain obtained as the reduced product of all domains involved in every sub-derivation: for example, if we are able to derive $\vdash_{A_1} [p]\ \mathsf{r}_1\ [w]$ and $\vdash_{A_2} [w]\ \mathsf{r}_2\ [q]$, we could try to derive $\vdash_{A_1 \sqcap A_2} [p]\ \mathsf{r}_1; \mathsf{r}_2\ [q]$ leveraging the reduced product $A_1 \sqcap A_2$. One remarkable advantage of using different abstractions within the same derivation will be that it is not necessary to consider their join at every step.

This extension of LCL_A is obtained by adding the rule (refine) in Figure 11, where we recall that $\preceq$ denotes the refinement relation between abstract domains and write $\vdash^{\lesssim}_{A} [p]\ \mathsf{r}\ [q]$ for a triple that can be derived in this extended proof system $\mathrm{LCL}^{\lesssim}_{A} \triangleq \mathrm{LCL}_A \cup \{(\mathsf{refine})\}$. When A is not locally complete for r on p, (refine) allows us to exploit an abstraction refinement A' of A which is locally complete provided that the over-approximations in A and A' of both p and q coincide. The soundness result for $\mathrm{LCL}^{\lesssim}_{A}$ shows that any triple $\vdash^{\lesssim}_{A} [p]\ \mathsf{r}\ [q]$ still ensures that $q \leq [\![\mathsf{r}]\!]p \leq \alpha(q)$ holds. Let us remark that the only difference in soundness of LCL_A and $\mathrm{LCL}^{\lesssim}_{A}$, as stated by Theorems 5.5 and 8.1, is that the intensional abstract semantics $[\![\mathsf{r}]\!]^{\sharp}_{A}$ of Theorem 5.5 is replaced by the bca $[\![\mathsf{r}]\!]^{A}$ of Theorem 8.1.

Theorem 8.1 (Soundness of $\mathrm{LCL}^{\lesssim}_{A}$**).** *Let $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. For all $\mathsf{r} \in \mathrm{Reg}$, $p, q \in C$, if $\vdash^{\lesssim}_{A} [p]\ \mathsf{r}\ [q]$ then:*

(1) *$q \leq [\![\mathsf{r}]\!]p$, and*
(2) *$[\![\mathsf{r}]\!]^{A}\alpha(p) = \alpha(q) = \alpha([\![\mathsf{r}]\!]p)$.*

Proof. As in the proof of Theorem 5.5, we refer to the equality $[\![\mathsf{r}]\!]^{A}\alpha(p) = \alpha(q)$ as (2a) and use (2b) for the equality $[\![\mathsf{r}]\!]^{A}\alpha(p) = \alpha([\![\mathsf{r}]\!]p)$. We then recall that (2b) follows immediately by (1) and (2a).

The proof is by induction on the derivation tree of $\vdash^{\lesssim}_{A} [p]\ \mathsf{r}\ [q]$. For the cases where the last used rule is in LCL_A (in Figure 4), the proof follows the same pattern of the proof of Theorem 5.5: for (1) there is nothing to change, while for (2) we just need to replace $[\![\cdot]\!]^{\sharp}_{A}$ with $[\![\cdot]\!]^{A}$ everywhere and slightly change the proof of (2a) for the sequence with the additional inequality $[\![\mathsf{r}_1;\mathsf{r}_2]\!]^A\alpha(p) \leq_A [\![\mathsf{r}_2]\!]^A([\![\mathsf{r}_1]\!]^A\alpha(p))$ as follows:

$$\begin{aligned}
\alpha(q) &\leq_A && \text{[by (1) and monotonicity of } \alpha\text{]}\\
\alpha([\![\mathsf{r}_1;\mathsf{r}_2]\!]p) &\leq_A && \text{[by correctness (4)]}\\
[\![\mathsf{r}_1;\mathsf{r}_2]\!]^A\alpha(p) &\leq_A && \text{[by definition]}\\
[\![\mathsf{r}_2]\!]^A([\![\mathsf{r}_1]\!]^A\alpha(p)) &= && \text{[by ind.hyp. (2a), } [\![\mathsf{r}_1]\!]^A\alpha(p) = \alpha(w)\text{]}\\
[\![\mathsf{r}_2]\!]^A\alpha(w) &= && \text{[by ind.hyp. (2a)]}\\
\alpha(q).
\end{aligned}$$

Analogously, we slightly change the proof of (2a) for (iterate) as follows:

$$\begin{aligned}
[\![\mathsf{r}_1^*]\!]^A\alpha(p) &= && \text{[by Lemma 2.1 (d)]}\\
\alpha(p) &= && \text{[by hyp. } q \leq A(p)\text{]}\\
\alpha(p) \vee_A \alpha(q) &= && \text{[by additivity of } \alpha\text{]}\\
\alpha(p \vee q)
\end{aligned}$$

Instead, no changes are necessary for the join, because the abstract semantics of the choice command preserves bcas (cf. (5)).

Therefore, to conclude, we just need to consider the case for the new rule (refine):

(refine) : If the last rule applied is (refine), we can assume by induction that $\vdash^{\precsim}_{A'} [p]\ \mathsf{r}\ [q]$ for some abstraction refinement $A' \leq A$ such that $A'(p) = A(p)$ and $A'(q) = A(q)$. We need to prove the thesis for the conclusion $\vdash^{\precsim}_{A} [p]\ \mathsf{r}\ [q]$.
(1) by inductive hypothesis (1) we already have that $q \leq [\![\mathsf{r}]\!]p$ holds.
(2) Let us recall that $[\![\mathsf{r}]\!]^A\alpha(p) = \alpha([\![\mathsf{r}]\!]A(p))$. We first note that, by inductive hypothesis (2a), $\alpha'([\![\mathsf{r}]\!]A'(p)) = \alpha'(q)$, so that $[\![\mathsf{r}]\!]A'(p) \leq A'(q)$ holds. Therefore:

$$\begin{aligned}
\alpha(q) &\leq_A && \text{[by monotonicity of } \alpha \text{ and (1)]}\\
\alpha([\![\mathsf{r}]\!]p) &\leq_A && \text{[by monotonicity of } \alpha \circ [\![\mathsf{r}]\!]\text{]}\\
\alpha([\![\mathsf{r}]\!]A(p)) &= && \text{[by hyp.]}\\
\alpha([\![\mathsf{r}]\!]A'(p)) &\leq_A && \text{[by monotonicity of } \alpha \text{ and above hyp.]}\\
\alpha(A'(q)) &= && \text{[by hyp.]}\\
\alpha(A(q)) &= && \text{[by GI]}\\
\alpha(q)
\end{aligned}$$

so that, $[\![\mathsf{r}]\!]^A\alpha(p) = \alpha(q) = \alpha([\![\mathsf{r}]\!]p)$ follows. □

The following example shows how the rule (refine) allows us to infer a triple $\vdash^{\precsim}_{A} [p]\ \mathsf{r}\ [q]$ when $[\![\mathsf{r}]\!]^{\sharp}_A$ is not locally complete for A on p — i.e., soundness as stated by Theorem 5.5 does not hold — while the best correct approximation $[\![\mathsf{r}]\!]^A$ turns out to be locally complete. Interestingly, the rule (refine) allows us to combine sub-derivations that are

$$\dfrac{\dfrac{\mathbb{C}^{A_1}_{w_1}(y<0?)}{\vdash_{A_1}\ [w_1]\ (y<0)?\ [y\in\{-201,-3,-1\}]}\;\text{(transfer)}\qquad \dfrac{\mathbb{C}^{A_1}_{(y\in\{-201,-3,-1\})}(y:=-y)}{\vdash_{A_1}\ [y\in\{-201,-3,-1\}]\ y:=-y\ [w]}\;\text{(transfer)}}{(*)}\;\text{(seq)}$$

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{A_1}_{p}(y:=2*y+1)}{\vdash_{A_1}\ [p]\ y:=2*y+1\ [y\in[-201,201]\wedge y\ \text{odd}]}\;\text{(transfer)}}{\vdash_{A_1}\ [p]\ y:=2*y+1\ [w_1]}\;\text{(relax)}\qquad \dfrac{\dfrac{(*)}{\vdash_{A_1}\ [w_1]\ (y<0)?;\,y:=-y\ [w]}\;\text{(seq)}\qquad \dfrac{\dfrac{\mathbb{C}^{A_1}_{w_1}(y\geq 0?)}{\vdash_{A_1}\ [w_1]\ (y\geq 0)?\ [w]}\;\text{(transfer)}\qquad \dfrac{\mathbb{C}^{A_1}_{w}(\mathbf{skip})}{\vdash_{A_1}\ [w]\ \mathbf{skip}\ [w]}\;\text{(transfer)}}{\vdash_{A_1}\ [w_1]\ (y\geq 0)?;\mathbf{skip}\ [w]}\;\text{(seq)}}{\vdash_{A_1}\ [w_1]\ \mathsf{AbsVal}(y)\ [w]}\;\text{(join)}}{\vdash_{A_1}\ [p]\ \mathsf{c}_1\ [w]}\;\text{(seq)}$$

Fig. 12. Derivation of $\vdash_{A_1}\ [p]\ \mathsf{c}_1\ [w]$ for Example 8.2, with $w_1 \triangleq y \in \{-201,-3,-1,1,3,201\}$.

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{A_2}_{(x=y\wedge y\in\{3,201\})}(x:=x-1)}{\vdash_{A_2}\ [x=y\wedge y\in\{3,201\}]\ x:=x-1\ [x=y-1\wedge y\in\{3,201\}]}\;\text{(transfer)}\qquad \dfrac{\mathbb{C}^{A_2}_{(x=y-1\wedge y\in\{3,201\})}(y:=\max(0,y-1))}{\vdash_{A_2}\ [x=y-1\wedge y\in\{3,201\}]\ y:=\max(0,y-1)\ [x=y\wedge y\in\{2,200\}]}\;\text{(transfer)}}{\vdash_{A_2}\ [x=y\wedge y\in\{3,201\}]\ \mathsf{c}_4\ [x=y\wedge y\in\{2,200\}]}\;\text{(seq)}}{(*)}\;\text{(unroll)}$$

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{A_2}_{(x=y\wedge w_3)}(x:=x-1)}{\vdash_{A_2}\ [x=y\wedge w_3]\ x:=x-1\ [x=y-1\wedge w_3]}\;\text{(transfer)}\qquad \dfrac{\mathbb{C}^{A_2}_{(x=y-1\wedge w_3)}(y:=\max(0,y-1))}{\vdash_{A_2}\ [x=y-1\wedge w_3]\ y:=\max(0,y-1)\ [x=y\wedge y\in\{1,2,199,200\}]}\;\text{(transfer)}}{\vdash_{A_2}\ [x=y\wedge w_3]\ \mathsf{c}_4\ [x=y\wedge y\in\{1,2,199,200\}]}\;\text{(seq)}}{(**)}\;\text{(loopinv)}$$

$$\dfrac{\dfrac{\mathbb{C}^{A_2}_{w}(x:=y)}{\vdash_{A_2}\ [w]\ x:=y\ [x=y\wedge w]}\;\text{(transfer)}\qquad \dfrac{(*)\qquad \dfrac{\mathbb{C}^{A_2}_{(x=y\wedge w_2)}(x>1?)\quad \mathbb{C}^{A_2}_{(x=y\wedge w_2)}(x\leq 1?)\quad (**)\quad (x=y\wedge y\in\{1,2,199,200\})\leq(x=y\wedge y\in[1,201])}{\vdash_{A_2}\ [x=y\wedge w_2]\ \mathsf{c}_3\ [q]}\;\text{(loopinv)}}{\vdash_{A_2}\ [x=y\wedge w]\ \mathsf{c}_3\ [q]}\;\text{(unroll)}}{\vdash_{A_2}\ [w]\ \mathsf{c}_2\ [q]}\;\text{(seq)}$$

Fig. 13. Derivation of $\vdash_{A_2}\ [w]\ x := y;\mathsf{c}_2\ [q]$ for Example 8.2, with $w_2 \triangleq y \in \{1,2,3,200,201\}$ and $w_3 \triangleq y \in \{2,3,200,201\}$.

computed in different abstractions without the need to resort to a common more precise abstract domain such as their reduced product.

Example 8.2 (Benefits of (refine)**).** Let us consider the Imp program $\mathsf{c} \triangleq \mathsf{c}_1;\mathsf{c}_2$, where:

$$\mathsf{c}_1 \triangleq y := (2*y)+1;\mathsf{AbsVal}(y), \qquad \mathsf{c}_2 \triangleq x := y;\mathsf{c}_3 \qquad \mathsf{c}_3 \triangleq \mathbf{while}\ x>1\ \mathbf{do}\ \{x := x-1; y := \max(0,y-1)\},$$

using the syntactic sugar

$$y := \max(\mathsf{a}_1,\mathsf{a}_2) \;\triangleq\; \mathbf{if}\ \mathsf{a}_2 < \mathsf{a}_1\ \mathbf{then}\ y := \mathsf{a}_1\ \mathbf{else}\ y := \mathsf{a}_2.$$

Consider the pre-condition $p \triangleq y \in [-101, 100]$. It turns out that the interval domain $A \triangleq \mathsf{Int}$ is not locally complete, because

$$\begin{aligned}
[\![\mathsf{c}]\!]^\sharp_A \alpha(p) &= [\![\mathsf{c}_2]\!]^\sharp_A([\![\mathsf{c}_1]\!]^\sharp_A(y \in [-101, 100]))\\
&= [\![\mathsf{c}_2]\!]^\sharp_A([\![\mathsf{AbsVal}(y)]\!]^\sharp_A([\![y := (2 * y) + 1]\!]^\sharp_A(y \in [-101, 100])))\\
&= [\![\mathsf{c}_2]\!]^\sharp_A([\![\mathsf{AbsVal}(y)]\!]^\sharp_A(y \in [-201, 201]))\\
&= [\![\mathsf{c}_3]\!]^\sharp_A([\![x := y]\!]^\sharp_A(y \in [0, 201]))\\
&= [\![\mathsf{c}_3]\!]^\sharp_A(x \in [0, 201] \wedge y \in [0, 201])\\
&= x \in [0, 1] \wedge y \in [0, 201],
\end{aligned}$$

while

$$\begin{aligned}
[\![\mathsf{c}]\!]\alpha(p) &= [\![\mathsf{c}_2]\!]([\![\mathsf{c}_1]\!](y \in [-101, 100]))\\
&= [\![\mathsf{c}_2]\!]([\![\mathsf{AbsVal}(y)]\!]([\![y := (2 * y) + 1]\!](y \in [-101, 100])))\\
&= [\![\mathsf{c}_2]\!]([\![\mathsf{AbsVal}(y)]\!](y \in [-201, 201] \wedge y\,\text{odd}))\\
&= [\![\mathsf{c}_3]\!]([\![x := y]\!](y \in [1, 201] \wedge y\,\text{odd}))\\
&= [\![\mathsf{c}_3]\!](x = y \wedge y \in [1, 201] \wedge y\,\text{odd})\\
&= [\![x \leq 1?]\!]([\![(x > 1?; x := x - 1; y := \max(0, y - 1))^*]\!](x = y \wedge y \in [1, 201] \wedge y\,\text{odd}))\\
&= [\![x \leq 1?]\!](x = y \wedge y \in [1, 201])\\
&= x \in [1, 1] \wedge y \in [1, 1],
\end{aligned}$$

and thus $[\![\mathsf{c}]\!]^A \alpha(p) = \alpha([\![\mathsf{c}]\!]A(p)) = \alpha([\![\mathsf{c}]\!]p) = x \in [1, 1] \wedge y \in [1, 1]$. Therefore, by Theorem 5.5 (2), it is not possible to derive in LCL_A a triple $\vdash_A [p]\ \mathsf{c}\ [q]$, for any q. However, we show that, in this case, we can successfully resort to the rule (refine) to infer the triple $\vdash^{\preceq}_A [p]\ \mathsf{c}\ [q]$ with $q \triangleq x = 1 \wedge y = 1$. To achieve this, we identify two suitable refinements of Int that are used in different subtrees of the logical derivation of $\vdash^{\preceq}_A [p]\ \mathsf{c}\ [q]$. The lack of local completeness in Int is due to two reasons:

(i) the presence of the value 0 introduced in the over-approximation of c_1;

(ii) the linear relationship $x = y$ between the variables x and y after the assignment $x := y$ cannot be expressed in a nonrelational abstraction such as Int.

To address (i), we enrich the interval abstraction by adding a new abstract value $\mathbb{Z}_{\neq 0}$ expressing $x \neq 0$. Let A_1 denote the Moore closure of Int with this additional point $\mathbb{Z}_{\neq 0}$, so that A_1 contains all the intervals possibly having a “hole” in 0 (see Example 4.6). Using $A_1 \preceq A$, we can infer the triple $\vdash_{A_1} [p]\ \mathsf{c}_1\ [w]$ in LCL_{A_1}, where the post-condition is $w \triangleq y \in \{1, 3, 201\}$, as shown by the proof in Figure 12, where the application of the rules is quite straightforward. In particular, note the use of rule (relax) to reduce the number of concrete points in the under-approximation and the inclusion of the value 3 in w, which will be useful for accelerating the convergence in the successive part of the proof carried out in a second abstraction refinement A_2.

To address (ii), we consider the well-known weakly relational abstract domain $A_2 \triangleq \mathsf{Oct}$ of octagons [Miné 2006]. Oct consists of octagonal constraints between two variables x, y of type $\pm x \pm y \leq k$ and interval constraints of type $\pm x \leq k$, where $k \in \mathbb{Z} \cup \{-\infty, +\infty\}$, while the representation of the abstract values in Oct relies on difference bound

matrices. $\langle \mathrm{Oct}, \subseteq \rangle$ is a complete lattice and is defined by a GI such that, for all $X \in \wp(\mathbb{Q}^n)$, $\mathrm{Oct}(X)$ is the least octagon containing X. Using A_2 it is possible to derive the triple $\vdash_{A_2} [w]\ \mathsf{c}_2\ [q]$ in LCL_{A_2}, as shown in Figure 13, where $\mathsf{c}_4 \triangleq x := x - 1; y := \max(0, y - 1)$ is the loop body. This derivation provides an example of use of the rules (unroll) and (loopinv), discussed in Section 7, in lieu of (rec) and (iterate): in fact, the local completeness for the Boolean guard $x > 1$ of c_2 is not satisfied at each iteration but only when the abstract invariant is computed.
Finally, these two proofs $\vdash_{A_1} [p]\ \mathsf{c}_1\ [w]$ and $\vdash_{A_2} [w]\ \mathsf{c}_2\ [q]$ can be combined using (refine) twice and then (seq) to infer $\vdash^{\preceq}_{A} [p]\ \mathsf{c}\ [q]$ in $\mathrm{LCL}^{\preceq}_{A}$ as follows:

$$\dfrac{\dfrac{A_1 \preceq A \quad A_1(p) = A(p) = p \quad \vdash^{\preceq}_{A_1} [p]\,\mathsf{c}_1\,[w] \quad A_1(w) = A(w) = y \in [1, 201]}{\vdash^{\preceq}_{A} [p]\,\mathsf{c}_1\,[w]}\,(\text{refine}) \quad \dfrac{A_2 \preceq A \quad A_2(w) = A(w) = y \in [1, 201] \quad \vdash^{\preceq}_{A_2} [w]\,\mathsf{c}_2\,[q] \quad A_2(q) = A(q) = q}{\vdash^{\preceq}_{A} [w]\,\mathsf{c}_2\,[q]}\,(\text{refine})}{\vdash^{\preceq}_{A} [p]\,\mathsf{c}\,[q]}\,(\text{seq})$$

Note that, while the extremes of the interval $p = [-101, 100]$ have been chosen so to ease the derivation, the same example would apply to any (finite) input p that contains at least a negative and a positive number. □

Let us remark how the above example shows the ability of rule (refine) of carrying out different parts of a proof in some abstraction refinements of a domain A in a situation where local completeness would fail in the original abstraction A. Here, the combined use of two refinements A_1 and A_2 turns out to be more convenient than carrying out the whole proof in a more concrete domain that could be designed, e.g., as reduced product of A_1 and A_2, because this latter domain would not necessarily be locally complete for the whole program.

9 RELATED WORK

As already mentioned, de Vries and Koutavas [2011] were the first to introduce under-approximation triples by proposing the backward consequence rule in reverse Hoare logic for the analysis of non-deterministic algorithms, e.g., for array shuffle. Later, the idea of defining a logic of under-approximation has been fully developed into the design of Incorrectness Logic [O'Hearn 2020], whose comparison with our work has been fully investigated in Section 6. Incorrectness logic attracted a lot of research interest and originated a recent strand of work that aims to exploit incorrectness triples for catching memory errors by moving under-approximating reasoning to separation logic for pointer analysis [Le et al. 2022; Maksimovic et al. 2022; Poskitt 2021; Poskitt and Plump 2023; Raad et al. 2020, 2022; Yan et al. 2022]. The ability of incorrectness triples for compositional bug finding strategies has been practically shown by [Le et al. 2022], where 15 new real bugs in OpenSSL were discovered and reported to OpenSSL maintainers thanks to Pulse-X, an automatic program analysis tool based on incorrectness separation logic (ISL) [Raad et al. 2020]. Of course, the main benefit of Pulse-X is that all reported errors are true positives. The work by Raad et al. [2022] extends the above bug catching theory to concurrent programs by defining a parametric framework able to deal with race detection, deadlock detection, and memory safety error detection. Exact separation logic (ESL) [Maksimovic et al. 2022] relies on similar ideas but defines exact triples, in the sense that the consequences are at the same time an under- and over-approximation of the semantics: while the ISL quadruple $[p]\ \mathsf{r}\ [\mathbf{ok}: q][\mathbf{er}: w]$ asserts that any state satisfying either the success post-condition or the error post-condition is the outcome of executing the command r from some state satisfying the pre-condition p, the ESL quadruple $(p)\ \mathsf{r}\ (\mathbf{ok}: q)(\mathbf{er}: w)$ additionally guarantees that any terminating execution of r starting from a state satisfying the pre-condition p either leads to a success state that satisfies q or raises some fault in a state that satisfies w. A major difference is therefore that valid ISL quadruples $[p]\ \mathsf{r}\ [\mathbf{ok}: q][\mathbf{er}: w]$ can always be split into valid triples of the form $[p]\ \mathsf{r}\ [\mathbf{ok}: q]$ and $[p]\ \mathsf{r}\ [\mathbf{er}: w]$, while ESL quadruples $(p)\ \mathsf{r}\ (\mathbf{ok}: q)(\mathbf{er}: w)$ are atomic because they must account for all behaviours. Our conjecture is that ESL accounts for the case of LCL_A where the trivial identity abstraction is considered, but to confirm this intuition we first plan to explore whether and how the relationship of LCL_A with IL studied in Section 6 could be extended to ISL [Raad et al. 2020]. The main challenge for achieving this extension will be to engineer a suitable frame rule for heap assertions that is able to preserve local completeness for the abstraction A.

Recently, Zilberstein et al. [2023] put forward Outcome Logic (OL), a generalization of Hoare Logic that is able to find true bugs while preserving the ability of proving programs correct. OL is parametric on a monoidal structure for predicates and on a so-called outcome assertion logic, and this enables correctness and incorrectness reasoning within the same program logic. OL does not consider the chance of abstracting the predicates as we do in LCL_A, so that we envisage that this challenge can be an appealing subject for future work.

Kleene algebras with tests (KAT) enriched by forward and backward box and diamond operators, so-called modal KATs, have been exploited to unify correctness and incorrectness logics in a single algebraic framework [Möller et al. 2021]. Möller and Struth [2006] already formalized the symmetries of forward and backward box and diamond operators as Galois connections, complementarities and dualities, and provided algebraic soundness and completeness proofs for Hoare logic. Later on, Möller et al. [2021] showed that modal KATs with countable joins of tests can also embed incorrectness logic and can be used to draw some links between correctness and incorrectness specifications as well as to deal with backward under-approximations. Recently, Zhang et al. [2022] have shown that O'Hearn incorrectness logic cannot be formulated within a conventional KAT, but, at the same time, a full fledged modal KAT is not needed. In fact, Zhang et al. [2022] prove that a KAT including a greatest element, so-called TopKAT, is capable to encode both Hoare and O'Hearn program logics in a purely equational fashion. Milanese and Ranzato [2022] gave a further contribution within this stream of works by showing how KATs extended either with a modal diamond operator or with a top element are able to encode our local completeness logic LCL_A. Thus, this latter result generalizes both the modal KAT [Möller et al. 2021] and TopKAT [Zhang et al. 2022] approaches for encoding Hoare correctness and O'Hearn incorrectness logics.

Regarding the use of under-approximation in abstract interpretation, Cousot and Cousot introduced the general framework that could be used to study both over- and under-approximations already in their seminal work [Cousot and Cousot 1977]. However, to the extent of our knowledge, there are no abstract domains thought from the ground up for under-approximation in forward program analysis. The main problems in designing significant abstract domains for under-approximation have been recently studied by Ascari et al. [2022].

10 CONCLUSION AND FUTURE WORK

We presented a program logic for locally complete abstract interpretations, called LCL_A and parametric on an abstract domain A. LCL_A can *prove the presence as well as the absence of true alarms*, meaning that proofs in LCL_A are potentially able to infer *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 programs based on abstract interpretation. We think that this work opens up many promising lines of research for the automatic verification of program correctness and incorrectness.

When some proof obligation $\mathbb{C}^A_p(f)$ about the local completeness in the abstract domain 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 $\mathbb{C}^{A'}_p(f)$ and conclude the derivation in $\vdash_{A'}$. In particular, following [Filé et al. 1996; Giacobazzi and Ranzato 1997; Giacobazzi et al. 2000], 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. This problem has been studied recently in [Bruni et al. 2022] where we proved that in general there is no unique minimal solution to the problem of abstract domain refinement for local completeness and a forward/backward strategy for optimally refining the abstract domain to achieve local completeness has been introduced. As sketched in Example 5.9, in program verification the strategy would be to iteratively transform the original abstract domain A_0 stepping through a sequences of abstract domains $A_1, ..., A_n$, until a derivation $\vdash_{A_n} [p]\ \mathsf{r}\ [q]$ can be completed in A_n. Each domain A_{i+1} can be designed by looking at the proof obligations of local completeness that fail in the attempt to prove $\vdash_{A_i} [p]\ \mathsf{r}\ [q]$ using the current abstract domain A_i. Notably, different abstraction refinements could be used by applying the rule (refine) of the extended proof system $\mathrm{LCL}_A^{\preceq}$, introduced in Section 8, in different subtrees of the same derivation.

We also aim at investigating extensions of our proof system with *non-compositional rules* for programs. Here, the incompleteness for a statement r may turn to completeness for an extensionally equivalent statement r' such that $[\![\mathsf{r}]\!] = [\![\mathsf{r}']\!]$. For example, a command such as $x := xy + 1; x := x - 1$ is incomplete for a simple sign analysis in $\mathrm{Sign}^{\pm} \triangleq \{\varnothing, \mathbb{Z}_{<0}, \mathbb{Z}_{=0}, \mathbb{Z}_{>0}, \mathbb{Z}\}$ because

$$[\![x := x - 1]\!]^{\sharp}_{\mathrm{Sign}^{\pm}}([\![x := xy + 1]\!]^{\sharp}_{\mathrm{Sign}^{\pm}})\langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle = [\![x := x - 1]\!]^{\sharp}_{\mathrm{Sign}^{\pm}}\langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle = \langle x/\mathbb{Z}, y/\mathbb{Z}_{>0}\rangle\ .$$

Nevertheless, when considering an equivalent statement such as $x := xy$, we achieve a complete sign analysis because $[\![x := xy]\!]^{\sharp}_{\mathrm{Sign}^{\pm}}\langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle = \langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle$. Patterns of this kind could be handled by a noncompositional rule for manipulating assignments such as

$$\frac{\vdash_A [p]\ x := (\mathsf{a}_2[{}^{\mathsf{a}_1}/_x])\ [q]}{\vdash_A [p]\ x := \mathsf{a}_1; x := \mathsf{a}_2\ [q]}$$

Because $x := (x - 1)[{}^{xy+1}/_x] \equiv x := xy + 1 - 1 \equiv x := xy$, from $\vdash_{\mathrm{Sign}^{\pm}} [\langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle]\ x := xy\ [\langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle]$, we would be able to derive $\vdash_{\mathrm{Sign}^{\pm}} [\langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle]\ x := xy + 1; x := x - 1\ [\langle x/\mathbb{Z}_{>0}, y/\mathbb{Z}_{>0}\rangle]$.

While completeness of abstract transfer functions is preserved by function composition, as encoded by the rule (seq) of LCL_A, one major issue of abstract interpretation is that best correct approximations are not compositional, i.e., the composition of abstract predicate transformers may not be as precise as the abstraction of their concrete composition [Reps et al. 2004; Yorsh et al. 2004]. 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.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their helpful comments. The authors have been funded by the *Italian MIUR*, under the PRIN2017 project no. 201784YSZ5 “AnalysiS of PRogram Analyses (ASPRA)” and by a *Meta research* gift. Roberto Giacobazzi and Francesco Ranzato have been partially funded by *Facebook Research*, under a “Probability and Programming Research Award”, by an *Amazon Research Award* for “AWS Automated Reasoning”, and by a *WhatsApp Research Award* on “Privacy-aware Program Analysis”.

REFERENCES

Flavio Ascari, Roberto Bruni, and Roberta Gori. 2022. Limits and difficulties in the design of under-approximation abstract domains. In *Proc. of 25th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2022 (Lecture Notes in Computer Science, Vol. 13242)*, Patricia Bouyer and Lutz Schröder (Eds.). Springer, 21–39. https://doi.org/10.1007/978-3-030-99253-8_2

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

François Bourdoncle. 1993. Abstract Debugging of Higher-Order Imperative Languages. In *Proc. ACM SIGPLAN PLDI'93* (Albuquerque, New Mexico, USA). ACM, 46–55. https://doi.org/10.1145/155090.155095

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

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

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2022. Abstract interpretation repair. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI'22*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 426–441. https://doi.org/10.1145/3519939.3523453

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *Proc. NFM'15 (LNCS, Vol. 9058)*. Springer, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

Patrick Cousot. 2021. *Principles of Abstract Interpretation.* MIT Press.

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

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

Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *J. Logic and Comput.* 2, 4 (1992), 511–547.

Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2018. Program analysis is harder than verification: A computability perspective. In *Proc. of the 30th International Conference on Computer Aided Verification, CAV 2018 (Lecture Notes in Computer Science, Vol. 10982)*. Springer, 75–95. https://doi.org/10.1007/978-3-319-96142-2_8

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Proc. SEFM'11*. Springer, 155–171. https://doi.org/10.1007/978-3-642-24690-6_12

Edsger W. Dijkstra. 1972a. *Chapter I: Notes on Structured Programming.* Academic Press Ltd., GBR, 1–82.

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

Edsger W. Dijkstra. 1972c. Turing Award Lecture. https://www.youtube.com/watch?v=6sIlKP2LzbA.

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

G. Filé, R. Giacobazzi, and F. Ranzato. 1996. A unifying view of abstract domain design. *ACM Comput. Surv.* 28, 2 (1996), 333–336. https://doi.org/10.1145/234528.234742

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32.

Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 261–273. https://doi.org/10.1145/2676726.2676987

Roberto Giacobazzi and Francesco Ranzato. 1996. Compositional Optimization of Disjunctive Abstract Interpretations. In *Proceedings of 6th European Symposium on Programming, ESOP'96, (Lecture Notes in Computer Science, Vol. 1058)*. Springer, 141–155. https://doi.org/10.1007/3-540-61055-3_34

Roberto Giacobazzi and Francesco Ranzato. 1997. Completeness in Abstract Interpretation: A Domain Perspective. In *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology, AMAST'97 (Lecture Notes in Computer Science, Vol. 1349)*. Springer, 231–245. https://doi.org/10.1007/BFb0000474

Roberto Giacobazzi and Francesco Ranzato. 1998. Optimal Domains for Disjunctive Abstract Intepretation. *Sci. Comput. Program.* 32, 1-3 (1998), 177–210. https://doi.org/10.1016/S0167-6423(97)00034-8

Roberto Giacobazzi and Francesco Ranzato. 2022. History of Abstract Interpretation. *IEEE Ann. Hist. Comput.* 44, 2 (2022), 33–43. https://doi.org/10.1109/MAHC.2021.3133136

Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. 1998. Complete Abstract Interpretations Made Constructive. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science, MFCS'98 (Lecture Notes in Computer Science, Vol. 1450)*. Springer, 366–377. https://doi.org/10.1007/BFb0055786

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

Charles A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Charles A. R. Hoare. 2003. The verifying compiler: A grand challenge for computing research. *J. ACM* 50, 1 (2003), 63–69. https://doi.org/10.1145/602382.602403

Cliff Jones, Peter O'Hearn, and Jim Woodcock. 2006. Verified Software: A Grand Challenge. *IEEE Computer* 39, 04 (2006), 93–95. https://doi.org/10.1109/MC.2006.145

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 247–259. https://doi.org/10.1145/2676726.2676966

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 427–443. https://doi.org/10.1145/256167.256195

Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–27. https://doi.org/10.1145/3527325

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006.* 42–54. https://doi.org/10.1145/1111037.1111042

Petar Maksimovic, Caroline Cronjäger, Julian Sutherland, Andreas Lööw, Sacha-Élie Ayoun, and Philippa Gardner. 2022. Exact Separation Logic. *CoRR* abs/2208.07200 (2022). https://doi.org/10.48550/arXiv.2208.07200 arXiv:2208.07200

John McCarthy. 1962. Towards a Mathematical Science of Computation. In *IFIP Congress.* 21–28.

Marco Milanese and Francesco Ranzato. 2022. Local Completeness Logic on Kleene Algebra with Tests. In *Proceedings of the 29th International Static Analysis Symposium (SAS'22) (Springer LNCS, Vol. 13790).* 350–371. https://doi.org/10.1007/978-3-031-22308-2_16

Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1

Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages* 4, 3-4 (2017), 120–372. https://doi.org/10.1561/2500000034

Antoine Miné. 2014. Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. *Sci. Comput. Program.* 93 (2014), 154 – 182. https://doi.org/10.1016/j.scico.2013.09.014

Bernhard Möller, Peter W. O'Hearn, and Charles A. R. Hoare. 2021. On Algebra of Program Correctness and Incorrectness. In *Relational and Algebraic Methods in Computer Science - 19th International Conference, RAMiCS 2021, Marseille, France, November 2-5, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13027)*, Uli Fahrenberg, Mai Gehrke, Luigi Santocanale, and Michael Winter (Eds.). Springer, 325–343. https://doi.org/10.1007/978-3-030-88701-8_20

Bernhard Möller and Georg Struth. 2006. Algebras of modal operators and partial correctness. *Theor. Comput. Sci.* 351, 2 (2006), 221–239. https://doi.org/10.1016/j.tcs.2005.09.069

Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the Impact of Formal Methods. In *Proc. LICS'18* (Oxford, United Kingdom). ACM, 13–25. https://doi.org/10.1145/3209108.3209109

Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. https://doi.org/10.1145/3371078

Benjamin Pierce. 2002. *Types and Programming Languages.* MIT Press.

Christopher M. Poskitt. 2021. Incorrectness Logic for Graph Programs. In *Proceedings of the 14th International Conference on Graph Transformation, ICGT 2021 (Lecture Notes in Computer Science, Vol. 12741).* Springer, 81–101. https://doi.org/10.1007/978-3-030-78946-6_5

Christopher M. Poskitt and Detlef Plump. 2023. Monadic second-order incorrectness logic for GP 2. *J. Log. Algebraic Methods Program.* 130 (2023), 100825. https://doi.org/10.1016/j.jlamp.2022.100825

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Proc. CAV'20, Part II (LNCS, Vol. 12225).* Springer, 225–252. https://doi.org/10.1007/978-3-030-53291-8_14

Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Concurrent incorrectness separation logic. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498695

Francesco Ranzato. 2020. Decidability and Synthesis of Abstract Inductive Invariants. In *Proceedings of the 31st International Conference on Concurrency Theory, CONCUR 2020 (LIPIcs, Vol. 171).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:21. https://doi.org/10.4230/LIPIcs.CONCUR.2020.30

Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Proc. VMCAI'04 (LNCS, Vol. 2937).* Springer, 252–266. https://doi.org/10.1007/978-3-540-24622-0_21

Henry G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366.

Xavier Rival and Kwang Yi. 2020. *Introduction to Static Analysis – An Abstract Interpretation Perspective.* MIT Press.

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

Alan M. Turing. 1989. Checking a Large Routine. In *The Early British Computer Conferences*, Martin Campbell-Kelly (Ed.). MIT Press, Cambridge, MA, USA, 70–72. http://dl.acm.org/citation.cfm?id=94938.94952

Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: an Introduction.* MIT press.

Peng Yan, Hanru Jiang, and Nengkun Yu. 2022. On incorrectness logic for Quantum programs. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–28. https://doi.org/10.1145/3527316

Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. 2004. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In *Proc. TACAS'04 (LNCS, Vol. 2988).* Springer, 530–545. https://doi.org/10.1007/978-3-540-24730-2_39

Cheng Zhang, Arthur Azevedo de Amorim, and Marco Gaboardi. 2022. On Incorrectness Logic and Kleene Algebra with Top and Tests. *Proc. ACM Program. Lang.* 6, POPL, Article 29 (jan 2022), 30 pages. https://doi.org/10.1145/3498690

Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* OOPSLA, To appear (2023).