Logics for Extensional, Locally Complete Analysis via Domain Refinements [⋆]

Flavio Ascari(✉), Roberto Bruni, and Roberta Gori

Dipartimento di Informatica, Università di Pisa, Largo B. Pontecorvo 3, Pisa, Italy,
flavio.ascari@phd.unipi.it, {roberto.bruni,roberta.gori}@unipi.it

Abstract. Abstract interpretation is a framework to design sound static analyses by over-approximating the set of program behaviours. While over-approximations can prove correctness, they cannot witness incorrectness because false alarms may arise. An ideal, but uncommon, situation is completeness of the abstraction that can ensure no false alarm is introduced by the abstract interpreter. Local Completeness Logic is a proof system that can decide both correctness and incorrectness of a program: any provable triple $\vdash_A [P]\ \mathsf{c}\ [Q]$ in the logic implies completeness of an *intensional* abstraction of program c on input P and is such that Q can be used to decide (in)correctness. However, completeness itself is an *extensional* property of the function computed by the program, while the above intensional analysis depends on the way the program is written and therefore not all valid triples can be derived in the proof system. Our main contribution is the study of new inference rules which allow one to perform part of the intensional analysis in a more precise abstract domain, and then to transfer the result back to the coarser domain. With these new rules, all (extensionally) valid triples can be derived in the proof system, thus untying the set of provable properties from the way the program is written.

Keywords: Abstract interpretation, Completeness in abstract interpretation, Hoare logic, Abstract domain refinement, Extensionality

1 Introduction

Static program analysis has been widely used to help developers produce valid software. Among static analysis techniques, abstract interpretation [6,7] is a general formalism to define sound-by-construction over-approximations that has been successfully applied in many fields, such as model checking, security and optimization [8]. Static analyses are often defined as *over-approximations*, that is the analysis computes a superset of the behaviors. This leads to no false negatives, that is all issues of the software are identified by the analysis, but it can cause false alarms: an incorrect behavior may be an artifact of the analysis, added by the over-approximation. While the absence of false negatives allowed a wide applicability of abstract interpretation techniques, it also make tools less

[⋆] Research supported by MIUR PRIN Project 201784YSZ5 *ASPRA–Analysis of Program Analyses*.

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 1–27, 2023.
https://doi.org/10.1007/978-3-031-30044-8_1

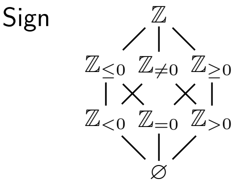
reliable to identify bugs. In fact, in many industrial applications any false alarm reported by the analysis to the developers diminishes its credibility, making it less effective in practice. This argument has recently led to the development of a logic of under-approximations, called incorrectness logic [16,17].

The Problem. In abstract interpretation, an ideal situation is *completeness.* Given an *expressible* specification, that is, one represented exactly in the abstract domain, a complete abstraction reports no false alarms. In its most widespread formulation [7], completeness is a *global* property: a program c is complete in the abstraction A if a condition holds for all possible inputs. Let C be the concrete domain and $[\![\mathsf{c}]\!] : C \to C$ be the (collecting) denotational semantics of c. Given an abstract domain A, a concretization function $\gamma : A \to C$ and an abstraction function $\alpha : C \to A$, an abstract interpreter $[\![\mathsf{c}]\!]^\sharp_A : A \to A$ is complete in A if for all possible inputs P we have $[\![\mathsf{c}]\!]^\sharp_A \alpha(P) = \alpha([\![\mathsf{c}]\!] P)$. Unfortunately, because of universal quantification over the possible inputs, this condition is difficult to meet in practice. Moreover, in most cases completeness is checked on an intensional abstraction of $[\![\mathsf{c}]\!]$ computed inductively on the syntax, through inductive reasoning by an abstract interpreter $[\![\mathsf{c}]\!]^\sharp_A$ making completeness an *intensional* property dependent on the program syntax [10]. However, in principle completeness is an *extensional* property, that only depends on the best correct abstraction $[\![\mathsf{c}]\!]^A$ of $[\![\mathsf{c}]\!]$ in A, defined by $[\![\mathsf{c}]\!]^A \triangleq \alpha [\![\mathsf{c}]\!] \gamma$. We sum up what we may call intensional (on the left) and extensional (on the right) completeness in the following equations:

$$[\![\mathsf{c}]\!]^\sharp_A \alpha = \alpha [\![\mathsf{c}]\!] \qquad\qquad [\![\mathsf{c}]\!]^A \alpha = \alpha [\![\mathsf{c}]\!] \gamma \alpha = \alpha [\![\mathsf{c}]\!] \tag{1}$$

We show the difference between $[\![\mathsf{c}]\!]^A$ and $[\![\mathsf{c}]\!]^\sharp_A$ in the following example.

Example 1 (Extensional and intensional properties). Consider the concrete domain of sets of integers and the abstract domain of signs:



The meaning of the abstract elements of Sign is to represent concrete values that satisfy the respective property. So for instance, denoting with the function γ the "meaning" of an abstract element, we have $\gamma(\mathbb{Z}_{<0}) = \{n \in \mathbb{Z} \,|\, n < 0\}$. Conversely, α "abstracts" a concrete set of values to the least abstract property describing it, for instance $\alpha(\{0; 1; 100\}) = \mathbb{Z}_{\geq 0}$.

Consider the simple program fragment $\mathsf{c} \triangleq$ `x := x + 1; x := x - 1`. Its denotational semantics $[\![\mathsf{c}]\!]$ is the identity function $\mathrm{id}_\mathbb{Z}$, so its best correct abstraction is the abstract identity $\mathrm{id}_{\mathsf{Sign}} = \alpha \; \mathrm{id}_\mathbb{Z} \; \gamma$. This is an *extensional* property of the program because it only depends on the function it computes, i.e., its denotational semantics. However, an analyzer does not know the semantics of c, so it has to analyze the program syntactically, breaking it down in elementary pieces and gluing the results together. So for instance, starting from the concrete point $P = \{1\}$ the analysis first abstracts it to the property $\alpha(P) = \mathbb{Z}_{>0}$, then it computes

$$\begin{aligned}[\![\mathsf{c}]\!]^\sharp_{\mathsf{Sign}}(\mathbb{Z}_{>0}) &= [\![\mathtt{x\ :=\ x\ -\ 1}]\!]^\sharp_{\mathsf{Sign}}[\![\mathtt{x\ :=\ x\ +\ 1}]\!]^\sharp_{\mathsf{Sign}}(\mathbb{Z}_{>0}) \\ &= [\![\mathtt{x\ :=\ x\ -\ 1}]\!]^\sharp_{\mathsf{Sign}}(\mathbb{Z}_{>0}) = \mathbb{Z}_{\geq 0}.\end{aligned}$$

Analogous calculations for all properties in Sign yields the abstraction

$$[\![\mathsf{c}]\!]^\sharp_{\mathsf{Sign}}(a) = \begin{cases} \bot & \text{if } a = \bot \\ \mathbb{Z}_{\geq 0} & \text{if } a \in \{ \colorbox{lightgray}{$\mathbb{Z}_{=0}$}, \colorbox{lightgray}{$\mathbb{Z}_{>0}$}, \mathbb{Z}_{\geq 0}\} \\ \mathbb{Z}_{<0} & \text{if } a = \mathbb{Z}_{<0} \\ \top & \text{if } a \in \{ \colorbox{lightgray}{$\mathbb{Z}_{\leq 0}$}, \colorbox{lightgray}{$\mathbb{Z}_{\neq 0}$}, \top\} \end{cases}$$

that, albeit sound, is less precise than $\mathrm{id}_{\mathsf{Sign}}$ (we highlight with a gray background all inputs on which $[\![\mathsf{c}]\!]^\sharp_{\mathsf{Sign}}$ loses accuracy). If instead the program were written as $\mathsf{c}' \triangleq \mathtt{skip}$, the analysis in Sign would yield the best correct abstraction $[\![\mathsf{c}']\!]^\sharp_{\mathsf{Sign}} = \mathrm{id}_{\mathsf{Sign}}$. Therefore, the abstraction depends on how the program is written and not only on its semantics: it is what it is called an *intensional* property (see e.g. [1] for more about intensional and extensional abstract properties). □

To overcome the former limitation of "global" completeness, the concept of *local* completeness [2] has been recently proposed that is related to some specific input. While this condition is much more common in practice, it is also much more complex to prove. In order to do so, the authors of [2] introduce a Local Completeness Logic parametric with respect to an abstraction A (LCL_A for short), that is able to prove triples $\vdash_A [P]\ \mathsf{c}\ [Q]$ with the following meaning

1. Q is an under-approximation of the concrete semantics $[\![\mathsf{c}]\!]P$,
2. Q and $[\![\mathsf{c}]\!]P$ have the same over-approximation in A,
3. A is locally complete for the intensional abstraction $[\![\mathsf{c}]\!]^\sharp_A$ on input P.

The important consequence of the previous points is the fact that a triple in LCL_A is able to prove *both correctness and incorrectness* of a program with respect to a specification *Spec* expressible in A. By point (2), if the abstract analysis reports no errors in Q then there are none because of the over-approximation. However, if the analysis does report an issue, this must be present in the abstraction of $[\![\mathsf{c}]\!]P$ as well, that is the same as the abstraction of Q: this means that Q contains a witness of the violation of *Spec*, and this witness must be in $[\![\mathsf{c}]\!]P$ because of the under-approximation ensured by point (1). While local completeness of point (3) is a key property to prove point (1-2), it would be enough to guarantee that (3) holds for the extensional best correct approximation $[\![\mathsf{c}]\!]^A$ of $[\![\mathsf{c}]\!]$ rather than for the intensional abstract interpreter $[\![\mathsf{c}]\!]^\sharp_A$: this suggests that it is possible to weaken the hypothesis (3) in order to make the proof system able to derive more valid triples.

Main Contributions. Building on the proof system of LCL_A, we add new rules to relax point (3) to local completeness of the extensional abstraction $[\![\mathsf{c}]\!]^A$. This way, while the proof system itself remains intensional as it deduces program properties by working inductively on the syntax, the information it produces is more precise. Specifically, since the property associated with triples is extensional no precision is lost because of the intensional abstract interpreter, and in the end allows us to prove more triples. In order to achieve this goal, we introduce new rules to *dynamically refine the abstract domain* during the analysis. While in general an analysis in a more concrete domain is more precise, LCL_A requires local completeness, which is not necessarily preserved by domain refinement [11]. For instance, a common way to combine two different abstract domains is their reduced product [7], but it is not always the case that the analysis in the reduced product is (locally) complete, even when it is such in the two domains.

To preserve local completeness, we introduce several rules for domain refinement in LCL_A and compare their expressiveness and usability. All of them provide extensional guarantees, in the sense that point (3) is replaced with local completeness of the best correct abstraction $[\![\mathsf{c}]\!]^A$ on input P. The first one is called (refine-ext). LCL_A extended with (refine-ext) turns out to be *logically complete*: any triple satisfying the above conditions (1–3) can be proved in our proof system. This is a theoretical improvement with respect to LCL_A, that instead was intrinsically incomplete as a logic, i.e., for all abstractions A there exists a sound triple that cannot be proved. While (refine-ext) is theoretically interesting, one of its hypothesis is unfeasible to check in practice. To improve applicability, we propose two derived rules, (refine-int) and (refine-pre), whose premises can be checked effectively and imply the hypotheses of the more general (refine-ext). Surprisingly, it turns out that (refine-int) enjoys a logical completeness result too, while (refine-pre) is strictly weaker (in terms of strength of the logic, see Example 6). Despite this, the latter is much simpler and preferable to use in practice whenever possible (see Example 5), while the former can be used in more situations and is at times the best choice.

We present a pictorial comparison among the expressiveness of the various proof systems in Fig. 1. Each node represent the proof system LCL_A extended with one rule (the bottom one being plain LCL_A). An arrow in the picture means a more powerful proof system, i.e., a proof system that can prove more triples, with its label pointing out the result justifying the claim. The two arrows between the two topmost nodes are because the two proof systems are logically equivalent, i.e., they can prove the same triples.

Structure of the paper. In Section 2 we explain the notation used in the paper and recall the basics of abstract interpretation. In Section 3 we present LCL_A, mostly summarizing the content of [2], with a focus on what is used in the following sections. In Section 4 we present and compare our new rules to refine the abstract domain, namely (refine-ext) and the two derived rules (refine-int) and (refine-pre). We conclude in Section 5. Some proofs and technical examples are in Appendix A.

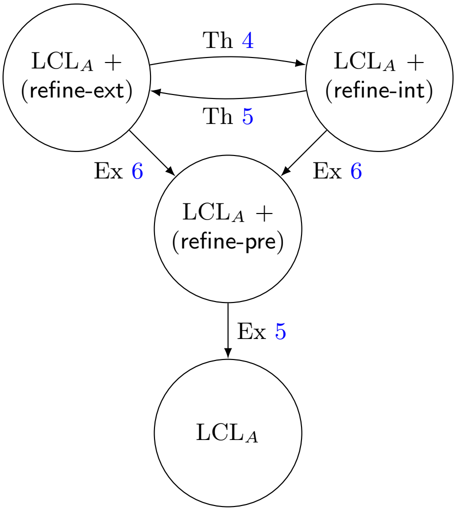


Fig. 1: Relations between the new proof systems

2 Background

Notation. We write $\mathcal{P}(S)$ for the powerset of S and $\mathrm{id}_S : S \to S$ for the identity function on a set S, with subscripts omitted when obvious from the context. If $f : S \to T$ is a function, we overload the symbol f to denote also its lifting $f : \mathcal{P}(S) \to \mathcal{P}(T)$ defined as $f(X) = \{f(x) \,|\, x \in X\}$ for any $X \subseteq S$. Given two functions $f : S \to T$ and $g : T \to V$ we denote their composition as $g \circ f$ or simply gf. For a function $f : S \to S$, we denote $f^n : S \to S$ the composition of f with itself n times, i.e. $f^0 = \mathrm{id}_S$ and $f^{n+1} = f \circ f^n$.

In ordered structures, such as posets and lattices, with carrier set C, we denote the ordering with $\leq_C$, least upper bounds (lubs) with $\sqcup_C$, greatest lower bounds (glbs) with $\sqcap_C$, least element with $\bot_C$ and greatest element with $\top_C$. For all these, we omit the subscript when evident from the context. Any powerset is a complete lattice ordered by set inclusion. In this case, we use standard symbols $\subseteq$, $\cup$, etc. Given a poset T and two functions $f, g : S \to T$, the notation $f \leq g$ means that, for all $s \in S$, $f(s) \leq_T g(s)$. A function f between complete lattices is additive (resp. co-additive) if it preserves arbitrary lubs (resp. glbs).

2.1 Abstract Interpretation

Abstract interpretation [6,7,5] is a general framework to define static analyses that are sound by construction. The main idea is to approximate the program semantics on some abstract domain A instead of working on the concrete domain C. The main tool used to study abstract interpretations are Galois connections. Given two complete lattices C and A, a pair of monotone functions $\alpha : C \to A$

and $\gamma : A \to C$ define a Galois connection (GC) when

$$\forall c \in C, a \in A. \quad \alpha(c) \leq_A a \iff c \leq_C \gamma(a).$$

We call C and A the concrete and the abstract domain respectively, α the abstraction function and γ the concretization function. The functions α and γ are also called adjoints. For any GC, it holds $\mathrm{id}_C \leq \gamma\alpha$, $\alpha\gamma \leq \mathrm{id}_A$, γ is co-additive and α is additive. A concrete value $c \in C$ is called *expressible* in A if $\gamma\alpha(c) = c$. We only consider GCs in which $\alpha\gamma = \mathrm{id}_A$, called Galois insertions (GIs). In a GI α is onto and γ is injective. A GI is said to be trivial if A is isomorphic to the concrete domain or if it is the singleton $\{\top_A\}$.

We overload the symbol A to denote also the function $\gamma\alpha : C \to C$: this is always a closure operator, that is a monotone, increasing (i.e. $c \leq A(c)$ for all c) and idempotent function. In the following, we use closure operators as much as possible to simplify the notation. Particularly, they are useful to denote domain refinements, as exemplified in the next paragraph. Note that they are still very expressive because γ is injective: for instance $A(c) = A(c')$ if and only if $\alpha(c) = \alpha(c')$. Nonetheless, the use of closure operators is only a matter of notation and it is always possible to rewrite them using the adjoints.

We use $\mathrm{Abs}(C)$ to denote the set of abstract domains over C, and we write $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ when we need to make the two maps α and γ explicit (we omit them when not needed). Given two abstract domains $A_{\alpha,\gamma}, A'_{\alpha',\gamma'} \in \mathrm{Abs}(C)$ over C, we say A' is a *refinement* of A, written $A' \preceq A$, when $\gamma(A) \subseteq \gamma'(A')$. When this happens, the abstract domain A' is more expressive than A, and in particular for all concrete elements $c \in C$ the inequality $A'(c) \leq_C A(c)$ holds.

Abstracting Functions. Given a monotone function $f : C \to C$ and an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$, a function $f^\sharp : A \to A$ is a sound approximation (or abstraction) of f if $\alpha f \leq f^\sharp \alpha$. Its *best correct approximation* (bca) is $f^A = \alpha f \gamma$, and it is the most precise of all the sound approximations of f: a function $f^\sharp$ is a sound approximation of f if and only if $f^A \leq f^\sharp$.

A sound abstraction $f^\sharp$ of f is *complete* if $\alpha f = f^\sharp \alpha$. It turns out that there exists a complete abstraction $f^\sharp$ if and only if the bca f^A is complete. If this is the case, we say that the abstract domain A is complete for f and denote it with $\mathbb{C}^A(f)$. Intuitively, completeness means that the abstract function $f^\sharp$ is as precise as possible in the given abstract domain A, and in program analysis this allows to have greater confidence in the alarms raised. We remark that A is complete for f if and only if $\alpha f = f^A \alpha = \alpha f \gamma \alpha$. Since γ is injective, this is true if and only if $\gamma\alpha f = \gamma\alpha f \gamma\alpha$, so that we define the (global) completeness property $\mathbb{C}^A(f)$ as follows:

$$\mathbb{C}^A(f) \iff Af = AfA.$$

2.2 Regular Commands.

Following [2] (see also [16]) we consider a language of *regular commands*:

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

This is a general language and can be instantiated differently changing the set Exp of basic transfer expressions e. These determines the kind of operations allowed in the language, and in our examples we assume to have deterministic assignments and boolean guards. Using standard definitions for arithmetic and boolean expressions $\mathsf{a} \in \mathrm{AExp}$ and $\mathsf{b} \in \mathrm{BExp}$, we consider

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

$\mathtt{skip}$ does nothing, $\mathtt{x\ :=\ a}$ is a standard deterministic assignment. The semantics of $\mathtt{b?}$ is that of an "assume" statement: if its input satisfies $\mathtt{b}$ it does nothing, otherwise it diverges. The term $\mathsf{r};\mathsf{r}$ represent the usual sequential composition, and $\mathsf{r}\oplus\mathsf{r}$ is nondeterministic choice. The Kleene star r^* denote a nondeterministic iteration, where r can be executed any number of time (possibly 0) before exiting. It can be thought as the solution of the recursive equation $\mathsf{r}^* \equiv \mathtt{skip} \oplus (\mathsf{r};\mathsf{r}^*)$. We write r^n to denote sequential composition of r with itself n times, analogously to how we use f^n for function composition.

This formulation can accommodate for a standard imperative programming language [18] defining $\mathtt{if}$ and $\mathtt{while}$ statements as

$$\mathtt{if\ (b)\ then\ c_1\ else\ c_2} \triangleq (\mathtt{b?;\ c_1}) \oplus ((\neg\mathtt{b})\mathtt{?;\ c_2})$$
$$\mathtt{while\ (b)\ do\ c} \triangleq (\mathtt{b?;\ c})^*; (\neg\mathtt{b})\mathtt{?}$$

Concrete semantics. We assume the semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ of basic transfer expressions on a complete lattice C to be additive. We believe this assumption not to be restrictive, and is always satisfied in collecting semantics. For our instantiation of Exp, we consider a finite set of variables Var, then the set of stores $\Sigma = \mathrm{Var} \to \mathbb{Z}$ that are (total) functions σ from Var to integers. The complete lattice C is then defined simply as $\mathcal{P}(\Sigma)$ with the usual poset structure given by set inclusion. Given a store $\sigma \in \Sigma$, store update $\sigma[x \mapsto v]$ is defined as usual for $x \in \mathrm{Var}$ and $v \in \mathbb{Z}$. We consider standard, inductively defined semantics $(\!|\cdot|\!)$ for arithmetic and boolean expressions. The concrete semantics of regular commands $[\![\cdot]\!] : \mathsf{Reg} \to C \to C$ is defined inductively as in Fig. 2a, where the semantics of basic transfer expressions $\mathsf{e} \in \mathsf{Exp}$ is defined as follows:

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

Abstract Semantics. The (compositional) abstract semantics of regular commands $[\![\cdot]\!]^\sharp_A : \mathsf{Reg} \to A \to A$ on an abstract domain $A \in \mathrm{Abs}(C)$ is defined inductively as in Fig. 2b. As common for abstract interpreters, we assume the analyser knows the best correct abstraction of expression and thus is able to compute $[\![\mathsf{e}]\!]^A$. A straightforward proof by structural induction shows that the abstract semantics is sound w.r.t. $[\![\mathsf{r}]\!]$ (i.e., $\alpha[\![\mathsf{r}]\!] \leq [\![\mathsf{r}]\!]^\sharp_A\alpha$) and monotone. However, in general it is less precise than the bca, i.e., $[\![\mathsf{r}]\!]^\sharp_A \neq [\![\mathsf{r}]\!]^A = \alpha[\![\mathsf{r}]\!]\gamma$.

$$\begin{aligned}
\llbracket \mathsf{e} \rrbracket c &\triangleq (\!|\mathsf{e}|\!) c \\
\llbracket \mathsf{r}_1 ; \mathsf{r}_2 \rrbracket c &\triangleq \llbracket \mathsf{r}_2 \rrbracket \llbracket \mathsf{r}_1 \rrbracket (c) \\
\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket c &\triangleq \llbracket \mathsf{r}_1 \rrbracket c \sqcup_C \llbracket \mathsf{r}_2 \rrbracket c \\
\llbracket \mathsf{r}^* \rrbracket c &\triangleq \bigsqcup_{n \geq 0} \llbracket \mathsf{r} \rrbracket^n c
\end{aligned}$$

(a) Concrete semantics

$$\begin{aligned}
\llbracket \mathsf{e} \rrbracket^\sharp_A a &\triangleq \llbracket \mathsf{e} \rrbracket^A a = \alpha (\!|\mathsf{e}|\!) \gamma(a) \\
\llbracket \mathsf{r}_1 ; \mathsf{r}_2 \rrbracket^\sharp_A a &\triangleq \llbracket \mathsf{r}_2 \rrbracket^\sharp_A \llbracket \mathsf{r}_1 \rrbracket^\sharp_A (a) \\
\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^\sharp_A a &\triangleq \llbracket \mathsf{r}_1 \rrbracket^\sharp_A a \sqcup_A \llbracket \mathsf{r}_2 \rrbracket^\sharp_A a \\
\llbracket \mathsf{r}^* \rrbracket^\sharp_A a &\triangleq \bigsqcup_{n \geq 0} (\llbracket \mathsf{r} \rrbracket^\sharp_A)^n a
\end{aligned}$$

(b) Abstract semantics

Fig. 2: Concrete and abstract semantics of regular commands, side by side

Shorthands. Throughout the paper, we present some simple examples of program analysis. The programs discussed in the examples contain just one or two variables (usually `x` and `y`), so we denote their sets of stores just as $\Sigma = \mathbb{Z}$ or $\Sigma = \mathbb{Z}^2$. In these cases, the convention is that an element of $\mathbb{Z}$ is the value of the single variable in Var, and a pair $(n, m) \in \mathbb{Z}^2$ denote the store $\sigma(\mathtt{x}) = n$, $\sigma(\mathtt{y}) = m$. We also lift these conventions to sets of values in $\mathbb{Z}$ or $\mathbb{Z}^2$. At times, to improve readability, we use logical formulas such as $(y \in \{1; 2; 99\} \wedge x = y)$ possibly using intervals, like in $x \in [0; 5]$, to describe set of stores.

3 Local Completeness Logic

In this section we present the notion of local completeness and introduce the proof system LCL_A (Local Completeness Logic on A) as was defined in [2].

For a generic program and abstract domain, global completeness is a too strong requirement: for conditionals to be complete the abstract domain should basically contain a complete sublattice of the concrete domain. For this reason, the weaker notion of *local* completeness can be more convenient in many cases.

Definition 1 (Local completeness, cf. [2]). *Let $f : C \to C$ be a concrete function, $c \in C$ a concrete point and $A \in \mathrm{Abs}(C)$ and abstract domain for C. Then A is* locally complete *for f on c, written $\mathbb{C}^A_c(f)$, iff*

$$Af(c) = AfA(c).$$

A remarkable difference between global and local completeness is that, while the former can be proved compositionally irrespective of the input [10], the latter needs it. Consequently, to carry on a compositional proof of local completeness, information on the input to each subpart of the program is also required, i.e., all traversed states are important. However, local completeness enjoys an “abstract convexity” property, that is, local completeness on a concrete point c implies local completeness on any concrete point d between c and its abstraction $A(c)$. This observation has been exploited in the design of the proof system LCL_A. The system is able to prove triples $\vdash_A [P]\ \mathsf{r}\ [Q]$ ensuring that:

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

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

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

Fig. 3: The proof system LCL_A.

1. Q is an under-approximation of the concrete semantics $\llbracket \mathsf{r} \rrbracket P$,
2. Q and $\llbracket \mathsf{r} \rrbracket P$ have the same over-approximation in A,
3. A is locally complete for $\llbracket \mathsf{r} \rrbracket$ on input P.

The second point means that, given a specification *Spec* expressible in A, any provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ either proves correctness of r with respect to *Spec* or expose some alerts in $Q \setminus \mathit{Spec}$. These in turns correspond to true ones because of the first point, as spelled out by Corollary 1 below.

The proof system is defined in Fig. 3. The crux of the proof system is to constrain the under-approximation Q to have the same abstraction of the concrete semantics $\llbracket \mathsf{r} \rrbracket P$, as for instance explicitly required in rule (relax). This, by the abstract convexity property mentioned above, means that local completeness of $\llbracket \mathsf{r} \rrbracket$ on the *under-approximation* P of the concrete store is enough to prove local completeness.

The three key properties (1–3) listed above are formalized by the following (intensional) soundness result:

Theorem 1 (Soundness, cf. [2]). *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$*. If* $\vdash_A [P]\ \mathsf{r}\ [Q]$ *then:*

1. $Q \leq \llbracket \mathsf{r} \rrbracket P$,
2. $\alpha(\llbracket \mathsf{r} \rrbracket P) = \alpha(Q)$,
3. $\llbracket \mathsf{r} \rrbracket^\sharp_A \alpha(P) = \alpha(Q)$.

As a consequence of this theorem, given a specification expressible in the abstract domain A, a provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ can determine both correctness and incorrectness of the program r:

Corollary 1 (Proofs of Verification, cf. [2]). *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ *and* $a \in A$*. If* $\vdash_A [P]\ \mathsf{r}\ [Q]$ *then*

$$\llbracket \mathsf{r} \rrbracket P \leq \gamma(a) \iff Q \leq \gamma(a).$$

The corollary is useful in program analysis and verification because, given a specification a expressible in A and a provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$, it allows to distinguish two cases.

– If $Q \subseteq \gamma(a)$, then we have also $\llbracket \mathsf{r} \rrbracket P \subseteq \gamma(a)$, so that the program is correct with respect to the specification.

- If $Q \not\subseteq \gamma(a)$, then also $[\![\mathsf{r}]\!]P \not\subseteq \gamma(a)$, that means $[\![\mathsf{r}]\!]P \setminus \gamma(a)$ is not empty and thus contains a true alert of the program. Moreover, since $Q \subseteq [\![\mathsf{r}]\!]P$ we have that $Q \setminus \gamma(a) \subseteq [\![\mathsf{r}]\!]P \setminus \gamma(a)$, so that already Q pinpoints some issues.

To better show how this work, we briefly introduce the following example (discussed also in [2] where it is possible to find all details of the derivation).

Example 2. Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$, the abstract domain Int of intervals, the precondition $P = \{1; 999\}$ and the command $\mathsf{r} \triangleq (\mathsf{r}_1 \oplus \mathsf{r}_2)^*$, where

$$\mathsf{r}_1 \triangleq \texttt{(x > 0)?; x := x - 1}$$

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

In LCL$_A$ it is possible to prove the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$, whose postcondition is $Q = \{0; 2; 1000\}$. Consider the two specification $\mathit{Spec} = (x \leq 1000)$ and $\mathit{Spec}' = (x \geq 100)$. The triple is then able to prove correctness of Spec and incorrectness of Spec'. For the former, observe that $Q \subseteq \mathit{Spec}$. By Corollary 1 we then know $[\![\mathsf{r}]\!]P \subseteq \mathit{Spec}$, that is correctness. For the latter, Q exhibits two witnesses to the violation of Spec', that are $0, 2 \in Q \setminus \mathit{Spec}'$. By point (1) of soundness we then know that $0, 2 \in Q \subseteq [\![\mathsf{r}]\!]P$ are true alerts. □

Strictly speaking, the proof of Corollary 1 only relies on points (1-2) of Theorem 1. Point (3) is in turn needed to ensure the first two, but extensional completeness would suffice to this aim. This means that we can weaken the soundness theorem (logically speaking, that is we prove a stronger conclusion, so the theorem as an implication is weaker) while still preserving the validity of Corollary 1. To this end, we propose a new soundness result involving extensional completeness: the important difference is that in point (3) we use the best correct abstraction $[\![\mathsf{r}]\!]^A$ in place of the inductively defined $[\![\mathsf{r}]\!]^\sharp_A$. Since Theorem 1 involves $[\![\mathsf{r}]\!]^\sharp_A$, an *intensional* property of the program r that depends on how the program is written (see Example 1 or Example 1 in Section 5 of [13]), while the new statement we propose relies only on $[\![\mathsf{r}]\!]^A$, an *extensional* property of the computed function $[\![\mathsf{r}]\!]$ and not of r itself, for the rest of the paper we use the name *intensional soundness* for Theorem 1, and *extensional soundness* for the following Theorem 2.

Theorem 2 (Extensional soundness). *Let $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. If $\vdash_A [P]\ \mathsf{r}\ [Q]$ then:*

1. $Q \leq [\![\mathsf{r}]\!]P$,
2. $\alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$,
3. $[\![r]\!]^A\alpha(P) = \alpha(Q)$.

Lastly, we remark that the original LCL$_A$ is intrinsically logically incomplete ([2], cf. Theorem 5.12): for every non trivial abstraction A there exists a triple that is intensionally sound (satisfies points (1-3) of Theorem 1) but cannot be proved in LCL$_A$. We will discuss logical (in)completeness for our extensional framework in Section 4.1.

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A\llbracket \mathsf{r} \rrbracket^{A'} A(P) = A(Q)}{\vdash_{A} [P]\ \mathsf{r}\ [Q]} \ (\mathsf{refine\text{-}ext})$$

Fig. 4: Rule refine for LCL_A.

4 Refining Abstract Domain

LCL_A can prove a triple $[P]\ \mathsf{r}\ [Q]$ for some Q only when $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A}$ is locally complete, that is $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(P) = \alpha(\llbracket \mathsf{r} \rrbracket P)$ (see Theorem 1). Since $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A}$ is computed in a compositional way, the above condition strictly depends on how r is written: to prove the local completeness of $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A}$, we need to prove that all its syntactic components are locally complete, that is an intensional property. However, the goal of the analysis is to study the behaviour of the function $\llbracket \mathsf{r} \rrbracket$, not how it is encoded by r. Hence, our aim is to enhance the original proof system in order to be able to handle triples where the extensional abstraction $\llbracket \mathsf{r} \rrbracket^{A}$ is proved to be locally complete w.r.t. the given input, that is $\llbracket \mathsf{r} \rrbracket^{A} \alpha(P) = \alpha(\llbracket \mathsf{r} \rrbracket P)$. To this end, we extend the proof system with a new inference rule, that is shown in Fig. 4. It is named after "refine" because it allows to refine abstract domains A to some $A' \preceq A$ and "ext" since it involves the extensional bca $\llbracket \mathsf{r} \rrbracket^{A'}$ of $\llbracket \mathsf{r} \rrbracket$ in A' (to distinguish it from the rules we will introduce in Section 4.2).

Using (refine-ext) it is possible to construct a derivation that proves local completeness of portions of the whole program in a more precise abstract domain A' and then carries the result over to the global analysis in a coarser domain A. The only requirement for the application of the rule is that domain A' is chosen in such a way that $A\llbracket \mathsf{r} \rrbracket^{A'} A(P) = A(Q)$ is satisfied.

Formally, given the two abstract domains $A_{\alpha,\gamma}, A'_{\alpha',\gamma'} \in \mathrm{Abs}(C)$, this last premise of rule (refine-ext) should be written as $\alpha\gamma'\llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) = \alpha(Q)$ to match function domains and codomains. However we prefer the more concise, albeit a little imprecise, notation used in Fig. 4. That writing is justified by the following intuitive argument: since $A' \preceq A$ we can consider with a slight abuse of notation (seeing abstract domains as closures) $A \subseteq A' \subseteq C$, so that for any element $a \in A \subseteq C$ we have $\gamma(a) = \gamma'(a) = a$ and for any $c \in C$ we have $\alpha' A(c) = A(c)$. With these, it follows that

$$\alpha\gamma'\llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) = \alpha\llbracket \mathsf{r} \rrbracket^{A'} A(P) = A\llbracket \mathsf{r} \rrbracket^{A'} A(P).$$

With rule (refine-ext) we cannot prove intensional soundness (Theorem 1): since this rule allows to perform part of the analysis in a more concrete domain A', we do not get any information on $\llbracket \mathsf{r} \rrbracket^{\sharp}_{A}$. However, we can prove extensional soundness (Theorem 2) and get all the benefits of Corollary 1.

Theorem 3 (Extensional soundness of (refine-ext)). *The proof system in Fig. 3 with the addition of rule* (refine-ext) *(see Fig. 4) is extensionally sound (cf. Theorem 2).*

We also remark that a rule like (refine-ext), that allows to carry on part of the proof in a different abstract domain, cannot come unconstrained. We present an example showing that a similar inference rule only requiring the triple $[P]\ \mathsf{r}\ [Q]$ to be provable in an abstract domain $A' \preceq A$ without any other constraint would be unsound.

Example 3. Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the point $P = \{-5; -1\}$, the abstract domain Sign of Example 1 and the program

$$\mathsf{r} \triangleq \texttt{x := x + 10}.$$

Then $C \preceq \mathsf{Sign}$ and we can prove $\vdash_C [P]\ \mathsf{r}\ [\{5; 9\}]$ applying (transfer) since all assignments are locally complete in the concrete domain. However, if $f = [\![\mathsf{r}]\!] = (\!|\texttt{x := x + 10}|\!)$, it is not the case that $\mathbb{C}^{\mathsf{Sign}}_P(f)$: indeed

$$\mathsf{Sign}(f(\mathsf{Sign}(P))) = \mathsf{Sign}(f(\mathbb{Z}_{<0})) = \mathsf{Sign}(\{n \in \mathbb{Z} \mid n < 10\}) = \top$$

while

$$\mathsf{Sign}(f(P)) = \mathsf{Sign}(\{5, 9\}) = \mathbb{Z}_{>0}.$$

This means that a rule without any additional condition can prove a triple which is not locally complete, hence it is unsound. □

4.1 Logical Completeness

Among all the possible conditions that can be added to a rule like (refine-ext), we believe ours to be very general since, differently than the original LCL$_A$ proof system (see Section 5.2 of [2]), the introduction of (refine-ext) allows us to derive a logical completeness result, i.e. the ability to prove *any* triple satisfying the soundness properties guaranteed by the proof system.

However, to prove such a result, our extension need an additional rule to handle loops, just like the original LCL$_A$ and Incorrectness Logic [16]. The necessary infinitary rule, called (limit), allows the proof system to handle Kleene star, and is the same as LCL$_A$:

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

Theorem 4 (Logical completeness of (refine-ext)). *Consider the proof system of Fig. 3 with the addition of rules* (refine-ext) *and* (limit)*. If $Q \leq [\![\mathsf{r}]\!] P$ and $[\![\mathsf{r}]\!]^A \alpha(P) = \alpha(Q)$ then $\vdash_A [P]\ \mathsf{r}\ [Q]$.*

The previous theorem proves the logical completeness of our proof system with respect to the property of extensional soundness. Indeed, if $Q \leq [\![\mathsf{r}]\!] P$ and $[\![\mathsf{r}]\!]^A \alpha(P) = \alpha(Q)$ we also have:

$$\alpha(Q) \leq \alpha([\![\mathsf{r}]\!] P) \leq [\![\mathsf{r}]\!]^A \alpha(P) = \alpha(Q),$$

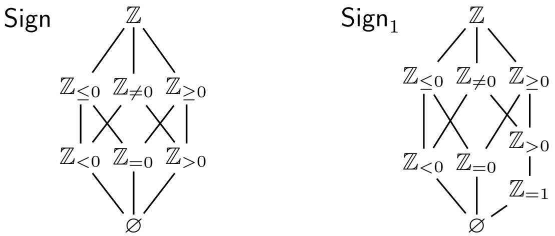
hence all three conditions of Theorem 2 are satisfied.

An interesting consequence of this result is the existence of a refinement A' in which it is possible to carry out the proof. In principle such a refinement could be the concrete domain C (as shown in the proof in Appendix A), that is not computable. However, it is worth nothing that for a sequential fragment (a portion of code without loops) the concrete domain can be actually used (for instance via first-order logic). This opens up the possibility, for instance, to infer a loop invariant on the body using C, and then prove it using an abstract domain. In Section 4.3 we discuss this issue further.

4.2 Derived Refinement Rules

The hypothesis $A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)$ is added to rule (refine-ext) in order to guarantee soundness: in general, the ability to prove a triple such as $[P]\ \mathsf{r}\ [Q]$ in a refined domain A' only gives information on $A[\![\mathsf{r}]\!]^{A'}A'(P)$ but not on $A[\![\mathsf{r}]\!]^{A'}A(P)$. In fact, the Example 4 shows that $A[\![\mathsf{r}]\!]^{A'}A'(P)$ and $A[\![\mathsf{r}]\!]^{A'}A(P)$ can be different.

Example 4. Consider the concrete domain $\mathcal{P}(\mathbb{Z})$, the abstract domain of signs $\mathsf{Sign}_{\alpha,\gamma} \in \mathrm{Abs}(\mathcal{P}(\mathbb{Z}))$ (introduced in Example 1) and its refinement Sign_1 below



For the command $\mathsf{r} \triangleq$ `x := x - 1` and the concrete point $P = \{1\}$ we have

$$\mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}\mathsf{Sign}_1(P) = \mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}(\mathbb{Z}_{=1}) = \mathbb{Z}_{=0}$$

but

$$\mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}\mathsf{Sign}(P) = \mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}(\mathbb{Z}_{>0}) = \mathbb{Z}_{\geq 0}. \qquad \square$$

Despite being necessary, the hypothesis of rule (refine-ext) cannot be checked in practice because the bca $[\![\mathsf{r}]\!]^{A'}$ of a composite command r is not known by the analyser. To mitigate this issue, we present two derived rules whose premises imply the premises of Rule (refine-ext), hence ensuring extensional soundness by means of Theorem 3.

The first rule we present replaces the requirement on the extensional bca $[\![\mathsf{r}]\!]^{A'}$ with requirements on the intensional compositional abstraction $[\![\mathsf{r}]\!]^{\sharp}_{A'}$ computed in A'. For this reason, we call this rule (refine-int).

Proposition 1. *The following rule* (refine-int) *is extensionally sound:*

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A[\![\mathsf{r}]\!]^{\sharp}_{A'}A(P) = A(Q)}{\vdash_{A} [P]\ \mathsf{r}\ [Q]}\ (\mathsf{refine\text{-}int})$$

It is worth noting that now the condition on the compositional abstraction $[\![\mathsf{r}]\!]^\sharp_{A'}$ can easily be checked by the analyser, possibly alongside the analysis of r with LCL or using a stand-alone abstract interpreter. Moreover, this rule is as powerful as the original (refine-ext) because it allows to prove a logical completeness result akin to Theorem 4.

Theorem 5 (Logical completeness of (refine-int)). *Consider the proof system of Fig. 3 with the addition of rules* (refine-int) *and* (limit)*. If $Q \leq [\![\mathsf{r}]\!]P$ and $[\![\mathsf{r}]\!]^A \alpha(P) = \alpha(Q)$ then $\vdash_A [P]\ \mathsf{r}\ [Q]$.*

Just like logical completeness for (refine-ext), this result implies the existence of a refinement A' in which it is possible to carry out the proof (possibly the concrete domain C). The discussion about how to find one is sketched in Section 4.3.

The second derived rule we propose is simpler than (refine-ext), as it just checks the abstractions $A(P)$ and $A'(P)$, with no reference to the regular command r nor to the postcondition Q. Since the premise is only on the precondition P, we call this rule (refine-pre).

Proposition 2. *The following rule* (refine-pre) *is extensionally sound:*

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A'(P) = A(P)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\mathsf{refine\text{-}pre})$$

Rule (refine-pre) only requires a simple check at the application site instead of an expensive analysis of the program r, so it can be preferred in practice.

We present an example to highlight the advantages of this rule (as well as (refine-int)), which allows us to use different domains in the proof derivation of different parts of the program.

Example 5 (The use of (refine-pre)*).* Consider the two program fragments

$$\begin{aligned}\mathsf{r}_1 &\triangleq \texttt{(y != 0)?; y := abs(y)}\\ \mathsf{r}_2 &\triangleq \texttt{x := y; while (x > 1) \{ y := y - 1; x := x - 1 \}}\\ &= \texttt{x := y; ((x > 1)?; y := y - 1; x := x - 1)}^*\texttt{; (x <= 1)?}\end{aligned}$$

and the program $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Here $\texttt{abs}$ is a function to compute the absolute value, and we assume, for the sake of simplicity, that the analyser knows its best abstraction. Consider the concrete domain $\mathcal{P}(\mathbb{Z}^2)$ where a pair (n, m) denote a state $\texttt{x} = n$, $\texttt{y} = m$, and the initial state $P = (\texttt{y} \in [-100; 100])$, a logical description of the concrete $\{(n, m) \mid m \in [-100; 100]\} \in \mathcal{P}(\mathbb{Z}^2)$. The bca $[\![\mathsf{r}]\!]^{\mathsf{Int}}$ in the abstract domain of intervals is locally complete on P (since P is expressible in Int), but the compositional abstraction $[\![\mathsf{r}]\!]^\sharp_{\mathsf{Int}}$ is not:

$$\begin{aligned}[\![\mathsf{r}]\!]^{\mathsf{Int}} \alpha(P) &= \mathsf{Int}([\![\mathsf{r}_2]\!][\![\mathsf{r}_1]\!](\{(n, m) \mid m \in [-100; 100]\}))\\ &= \mathsf{Int}([\![\mathsf{r}_2]\!](\{(n, m) \mid m \in [1; 100]\}))\\ &= \mathsf{Int}(\{(1, 1)\})\\ &= ([1; 1] \times [1; 1]),\end{aligned}$$

$$\dfrac{\dfrac{\mathbb{C}_P^{\mathsf{Int}_{\neq 0}}(\llbracket \texttt{y != 0?} \rrbracket)}{\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \texttt{y != 0?}\ [R_1]}\ (\mathsf{transfer}) \quad \dfrac{\dfrac{\mathbb{C}_{R_1}^{\mathsf{Int}_{\neq 0}}(\llbracket \texttt{y := abs(y)} \rrbracket)}{\vdash_{\mathsf{Int}_{\neq 0}} [R_1]\ \texttt{y := abs(y)}\ [\mathtt{y} \in [1;100]]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}_{\neq 0}} [R_1]\ \texttt{y := abs(y)}\ [R]}\ (\mathsf{relax})}{\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]}\ (\mathsf{seq})$$

Fig. 5: Derivation of $\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]$ for Example 5.

while

$$\begin{aligned}
\llbracket \mathsf{r} \rrbracket^\sharp_{\mathsf{Int}} \alpha(P) &= \llbracket \mathsf{r}_2 \rrbracket^\sharp_{\mathsf{Int}} \llbracket \mathsf{r}_1 \rrbracket^\sharp_{\mathsf{Int}} ([-\infty; +\infty] \times [-100; 100]) \\
&= \llbracket \mathsf{r}_2 \rrbracket^\sharp_{\mathsf{Int}} \llbracket \texttt{y := abs(y)} \rrbracket^{\mathsf{Int}} ([-\infty; +\infty] \times [-100; 100]) \\
&= \llbracket \mathsf{r}_2 \rrbracket^\sharp_{\mathsf{Int}} ([-\infty; +\infty] \times [0; 100]) \\
&= ([1; 1] \times [0; 100]) \neq ([1; 1] \times [1; 1]).
\end{aligned}$$

The issues are twofold. First, the analysis of r_1 in Int is incomplete, so we need a more concrete domain. For instance $\mathsf{Int}_{\neq 0}$, the Moore closure of Int with the addition of the element $\mathbb{Z}_{\neq 0}$ representing the property of being nonzero would work. Intuitively, $\mathsf{Int}_{\neq 0}$ contains all intervals, possibly having a "hole" in 0. Formally

$$\mathsf{Int}_{\neq 0} = \mathsf{Int} \cup \{ I_{\neq 0} \mid I \in \mathsf{Int} \}$$

with $\gamma'(I_{\neq 0}) = \gamma(I) \setminus \{0\}$. However, note that there is no need for a relational domain to analyze r_1 since variable $\mathtt{x}$ is never mentioned in it. On the contrary, the analysis of r_2 requires a relational domain to track the information that the value of variable $\mathtt{x}$ is equal to the value of variable $\mathtt{y}$. This suggests, for instance, to use the octagons domain Oct [15] to analyze r_2. It is worth noting that the domain of octagons Oct would not be able to perform a locally complete analysis of r_1 for the same reasons that the domain Int could not.

However, rule (refine-pre) allows us to combine these different proof derivations. Since the program state between r_1 and r_2 can be precisely represented in Int, we use this domain as a baseline and refine it in $\mathsf{Int}_{\neq 0}$ and Oct for the two parts respectively.

Let $R = (\mathtt{y} \in \{1; 2; 100\})$ that is an under-approximation of the concrete state in between r_1 and r_2 with the same abstraction in Int, so we can prove the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1\ [R]$. Note that the concrete point 2 was added to R in order to have local completeness for `(x > 1)?` in r_2. However, this triple cannot be proved in Int because $\llbracket \mathsf{r}_1 \rrbracket^\sharp_{\mathsf{Int}}$ is not locally complete on P, so we resort to (refine-pre) to change the domain to $\mathsf{Int}_{\neq 0}$. The full derivation in $\mathsf{Int}_{\neq 0}$ is shown in Fig. 5, where $R_1 = (\mathtt{y} \in [-100; 100] \wedge \mathtt{y} \neq 0)$ and we omitted for simplicity the additional hypothesis of (relax).

Again $\llbracket \mathsf{r}_2 \rrbracket$ is locally complete on R in Int, but the compositional analysis $\llbracket \mathsf{r}_2 \rrbracket^\sharp_{\mathsf{Int}}$ is not. Hence to perform the derivation we resort to (refine-pre) to introduce relational information in the abstract domain, using Oct instead of Int. Let

$Q = (\mathtt{x} = 1 \wedge \mathtt{y} = 1)$, that is the concrete output of the program, so that we can prove $\vdash_{\mathsf{Int}} [R]\ \mathsf{r}_2\ [Q]$. The derivation of this triple is only in Appendix A, Fig. 6. However, the proof is just a straightforward application of rules (seq), ($\mathsf{iterate}$) and ($\mathsf{transfer}$).

With those two derivation, the proof of the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$ is straightforward using ($\mathsf{refine\text{-}pre}$):

$$\dfrac{\dfrac{\vdash_{\mathsf{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1\ [R]}\ (\mathsf{refine\text{-}pre}) \qquad \dfrac{\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]}{\vdash_{\mathsf{Int}} [R]\ \mathsf{r}_2\ [Q]}\ (\mathsf{refine\text{-}pre})}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]}\ (\mathsf{seq})$$

For the derivation to fit the page, we write here the additional hypotheses of the rules. For the first application, $\mathsf{Int}_{\neq 0} \preceq \mathsf{Int}$ and $\mathsf{Int}_{\neq 0}(P) = P = \mathsf{Int}(P)$. For the second, $\mathsf{Oct} \preceq \mathsf{Int}$ and $\mathsf{Int}(R) = (\mathsf{y} \in [1; 100]) = \mathsf{Oct}(R)$.

It is worth noting that, in this example, all applications of ($\mathsf{refine\text{-}pre}$) can be replaced by ($\mathsf{refine\text{-}int}$). This means that also the latter is able to exploit $\mathsf{Int}_{\neq 0}$ and Oct to prove the triple in the very same way, but its application requires more expensive abstract analyses than the simple checks of ($\mathsf{refine\text{-}pre}$). □

While ($\mathsf{refine\text{-}pre}$) is simpler than ($\mathsf{refine\text{-}ext}$) and ($\mathsf{refine\text{-}int}$), it is also weaker in both a theoretical and practical sense. On the one hand, LCL_A extended with this rule does not admit a logical completeness result; on the other hand, there are situations in which, even though ($\mathsf{refine\text{-}pre}$) allows a derivation, the other rules are more effective. We show these two points by examples. For the first, we propose a sound triple that LCL_A extended with ($\mathsf{refine\text{-}pre}$) cannot prove. Since the example is quite technical, here we only sketch the idea, and leave the details only in Appendix A, Example 8.

Example 6 (Logical incompleteness of ($\mathsf{refine\text{-}pre}$)*).* Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the abstract domain Int of intervals, the concrete point $P = \{-1, 1\}$ and commands $\mathsf{r}_1 \triangleq \mathtt{x\ \ != \ 0?}$, $\mathsf{r}_2 \triangleq \mathtt{x\ >= \ 0?}$ and $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Then the triple $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [\{1\}]$ is sound but cannot be proved in LCL_A extended with ($\mathsf{refine\text{-}pre}$).

The key observations for this example are two. First, all strict subset $P' \subset P$ are such that $\mathsf{Int}(P') \subset \mathsf{Int}(P)$. Moreover, for all refinements $A' \preceq \mathsf{Int}$ such that $A'(P) = \mathsf{Int}(P)$ we have the same condition, namely if $P' \subset P$ then $A'(P') \subset A'(P)$. This is because for all $P' \subset P$ we have $A'(P') \subseteq \mathsf{Int}(P') \subset \mathsf{Int}(P) = A'(P)$. Second, $[\![\mathsf{r}_1]\!]P = P$. This means that all triples appearing in the derivation tree of $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [\{1\}]$ have the same precondition P. Since ($\mathsf{refine\text{-}pre}$) requires $A'(P) = \mathsf{Int}(P)$, all possible applications of this rule change the abstract domain to some A' satisfying the condition above. Since LCL_A computes under-approximations with the same abstraction of the strongest postcondition, these two observations make it impossible to under-approximate P further, both with (relax) and ($\mathsf{refine\text{-}pre}$). This in turn make the triple not provable because $[\![\mathsf{r}_2]\!]$ is not locally complete on P in Int or in any refinement satisfying

$A'(P) = \mathsf{Int}(P)$:

$$\begin{aligned} A'[\![\mathsf{r}_2]\!](P) &= A'(\{1\}) \subseteq \mathsf{Int}(\{1\}) = \{1\} \\ A'[\![\mathsf{r}_2]\!]A'(P) &\supseteq [\![\mathsf{r}_2]\!]A'(P) = [\![\mathsf{r}_2]\!](\mathsf{Int}(P)) = \{0, 1\}. \end{aligned}$$

Example 8 in Appendix A exhibits the formal argument showing that this triple cannot be proved. □

As a corollary, this example (and more in general logical incompleteness) shows that is not always possible to find a refinement A' to carry out the proof using (refine-pre). Another consequence of this incompleteness result is the fact that, even when a command is locally complete in an abstract domain A, we may need to reason about properties that are not expressible in A in order to prove it, as (refine-pre) may not be sufficient.

Second, we present an example to illustrate that there are situations in which (refine-int) is more practical than (refine-pre), even though they are both able to prove the same triple.

Example 7. Consider the two program fragments

$$\mathsf{r}_1 \triangleq \texttt{(y != 0)?; x := y; y := abs(y)}$$

$$\mathsf{r}_2 \triangleq \texttt{x := y; while (x > 1) \{ y := y - 1; x := x - 1 \}}$$

and the program $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Consider also the initial state $P = \mathtt{y} \in [-100; 100]$.

This example is a variation of Example 5: the difference is the introduction of the relational dependency `x := y` in r_1, that is partially stored in the postcondition R of r_1. Because of this, $\mathsf{Oct}(R)$ and $\mathsf{Int}(R)$ are different, so we cannot apply (refine-pre) to prove $[R]\ \mathsf{r}_2\ [Q]$ for some Q.

Following Example 5, the domain $\mathsf{Int}_{\neq 0}$ is able to infer on r_1 a subset R of the strongest postcondition $\mathtt{y} \in [1; 100] \wedge \mathtt{y} = \mathrm{abs}(\mathtt{x})$ with the same abstraction $\mathsf{Int}_{\neq 0}(R) = [-100; 100]_{\neq 0} \times [1; 100]$. However, for any such R we cannot use (refine-pre) to prove the triple $\vdash_{\mathsf{Int}} [R]\ \mathsf{r}_2\ [\mathtt{x} = 1 \wedge \mathtt{y} = 1]$ via Oct because $\mathsf{Int}(R) = \mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]$ while $\mathsf{Oct}(R) = 1 \leq \mathtt{y} \leq 100 \wedge -\mathtt{y} \leq \mathtt{x} \leq \mathtt{y}$. More in general, any subset of the strongest postcondition contains the relational information $\mathtt{y} = \mathrm{abs}(\mathtt{x})$, so relational domains like octagons and polyhedra [9] do not have the same abstraction as the non-relational Int, preventing the use of (refine-pre). However, we can apply (refine-int): considering $R = (\mathtt{y} \in \{1; 2; 100\} \wedge \mathtt{y} = \mathrm{abs}(\mathtt{x}))$, $Q = (\mathtt{x} = 1 \wedge \mathtt{y} = 1)$ and $\mathsf{r}_w \triangleq$ `while (x > 1) { y := y - 1; x := x - 1 }`, we have

$$\begin{aligned} \mathsf{Int}[\![\mathsf{r}_2]\!]^\sharp_{\mathsf{Oct}}\mathsf{Int}(R) &= \mathsf{Int}[\![\mathsf{r}_2]\!]^\sharp_{\mathsf{Oct}}(\mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]) \\ &= \mathsf{Int}[\![\mathsf{r}_w]\!]^\sharp_{\mathsf{Oct}}[\![\mathtt{x}\ \texttt{:=}\ \mathtt{y}]\!]^\sharp_{\mathsf{Oct}}(\mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]) \\ &= \mathsf{Int}[\![\mathsf{r}_w]\!]^\sharp_{\mathsf{Oct}}(1 \leq \mathtt{y} \leq 100, \mathtt{y} = \mathtt{x}) \\ &= \mathsf{Int}(\mathtt{x} = 1 \wedge \mathtt{y} = 1) \\ &= \mathsf{Int}(Q). \end{aligned}$$

In this example, rule (refine-pre) can be applied to prove the triple, but it requires to have relational information from the assignment `x := y` in r_1, hence forcing the use of a relational domain (eg. the reduced product [7] of Oct and $\mathsf{Int}_{\neq 0}$) for the whole r, making the analysis more expensive. □

4.3 Choosing The Refinement

All three new rules allow to combine different domains in the same derivation, but do not define an algorithm because of the choice of the right refinement to use is nondeterministic. A crucial point to their applicability is a strategy to select the refined abstract domain. While we have not addressed this problem yet, we believe there are some interesting starting points in the literature.

As already anticipated in previous sections, we settled the question from a theoretical point of view. Logical completeness results for (refine-ext) (Theorem 4) and (refine-int) (Theorem 5) implies the existence of a domain in which it is possible to complete the proof (if this were not the case, then the proof could not be completed in any domain, against the logical completeness). However, the proofs of those theorems exhibit the concrete domain C as an example, which is unfeasible in general. Dually, as (refine-pre) is logically incomplete (Example 6), there are triples that cannot be proved in any domain with it.

As more practical alternatives, we envisage some possibilities. First, we are studying relationships with counterexample-guided abstraction refinement (CEGAR) [4], which is a technique that exploits refinement in the context of abstract model checking. However, CEGAR and our approach seem complementary. On the one hand, our refinement rules allow a dynamic change of domain, during the analysis and only for a part of it, while CEGAR performs a static refinement and then a new analysis of the whole transition system in the new, more precise domain. On the other hand, our rules lack an instantiation technique, while for CEGAR there are effective algorithms available to pick a suitable refinement.

Second, local completeness shell [3] were proposed as an analogous of completeness shell [11] for local completeness. In the article, the authors proposed to use local completeness shells to perform abstract interpretation repair, a technique to refine the abstract domain depending on the program to analyse, just like CEGAR does for abstract model checking. Abstract interpretation repair works well with LCL_A, and could be a way to decide the best refinement for one of our rules in presence of a failed local completeness proof obligation. The advantage of combining repair with our new rules is given by the possibility of discarding the refined domain just after its use in a subderivation instead of using it to carry out the whole derivation. Investigations in this direction is ongoing.

Another related approach, which shares some common ground with CEGAR, is Lazy (Predicate) Abstraction [12,14]. Both ours and this approach exploits different abstract domains for different parts of the proof, refining it as needed. The key difference is that Lazy Abstraction unwinds the control flow graph (CFG) of the program (with techniques to handle loops) while we work inductively on the syntax. This means that, when Lazy Abstraction refines a domain, it must use it from that point onward (unless it finds a loop invariant). On the other

Proof system	Extensional	Logical completeness
Plain LCL_A	✗	✗
LCL_A + (refine-ext)	✓	✓
LCL_A + (refine-int)	✓	✓
LCL_A + (refine-pre)	✓	✗

Table 1: Comparison of the proof systems

hand, our method can change abstract domain even for different parts of sequential code. However, the technique used in Lazy Abstraction (basically to trace a counterexample back with a theorem prover until it is either found to be spurious or proved to be true) could be applicable to LCL_A: a failed local completeness proof obligation in (transfer) can be traced back with a theorem prover and the failed proof can be used to understand how to refine the abstract domain.

5 Conclusions

In this paper, we have proposed a logical framework to prove both correctness and incorrectness of a program exploiting locally complete abstractions. Indeed, from any provable triple $[P]$ r $[Q]$ we can either prove that r meets an expressible specification Spec or find a concrete counterexample in Q. Differently from the original LCL_A [2], that was proved to be *intensionally* sound, our framework is *extensionally* sound, meaning that is able to prove more properties about programs. To achieve this, our inference rules are based on the best correct abstraction of a program behaviour instead of a generic abstract interpreter. The key feature of our proof systems is the ability to exploit different abstract domains to analyse different portions of the whole program. In particular, the domains are selected among the refinements of a chosen abstract domain from which the analysis begins. The main advantage of our extensional approach is the possibility of proving many triples that could not be proved in LCL_A because of the way the program is written. More in details, we presented three new rules to refine the abstract domain, each of which can be added independently to the proof system with different complexity-precision trade-off.

Table 1 summarizes the properties LCL_A enjoys when extended with different rules, and Figure 1 from the Introduction graphically compare the logical strength of these proof systems. (refine-ext) is the most general rule, from which the other two (refine-int) and (refine-pre) are derived. The former turns out to be as strong as (refine-ext), since they are both logically complete, while the latter is simpler to use, although weaker.

Future work. In principle completeness could be achieved either refining or simplifying the abstract domain [11]. In this article we have only focused on refinement rules for local completeness, but we are investigating some simplification rules as well as their relation to the ones presented in this paper. To date, domain simplification seems theoretically weaker, but apparently it can accommodate for techniques useful in practice that are beyond the reach of refinement rules.

While the new rules we introduced are relevant from both a theoretical and practical point of view, they do not define an algorithm because of their nondeterminism: we need techniques to determine *when* a change of abstract domain is needed and *how* to choose the most convenient new domain. We believe these two issues are actually related. For instance, if the analysis is unable to satisfy a local completeness proof obligation to apply (transfer), an heuristics may determine both what additional information is needed to make it true (i.e., how to refine the abstract domain) and where that additional information came from (i.e., when to refine). We briefly discussed in Section 4.3 some possibilities to perform this choice. Ideally, one would systematically select an off-the- shelf abstract domain best suited to deal with each code fragment and the heuristic would inspect the proof obligations, and exploit some sort of catalog that can track suitable abstract domains that are locally complete for the code and input at hand or derive on-the-fly some convenient domain refinement as done, e.g., by partition refinement. To this aim, we intend to investigate a mutual exchange of ideas between CEGAR and our approach, and to integrate abstract interpretation repair into our framework.

Acknowledgments. We thank the anonymous referees for their helpful comments that helped us to improve the presentation and the discussion with related work.

Appendix A Proofs and Supplementary Material

A.1 Extensional Soundness (Theorem 2)

Proof (Proof of Theorem 2). First we remark that points (1) and (3) implies point (2):

$$\begin{aligned}\alpha(Q) &\leq \alpha([\![\mathsf{r}]\!]P) && [(1) \text{ and monotonocity of } \alpha]\\ &\leq [\![\mathsf{r}]\!]^{A}\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^{A}]\\ &= \alpha(Q) && [(3)]\end{aligned}$$

So all the lines are equal, in particular $\alpha(Q) = \alpha([\![\mathsf{r}]\!]P)$. The proof is then by induction on the derivation tree of $\vdash_A [P]\ \mathsf{r}\ [Q]$, but we only have to prove (1) and (3) because of the observation above. We only include one inductive case as an example, others are standard.

(seq): (1) $Q \leq [\![\mathsf{r}_2]\!]R \leq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]P) = [\![\mathsf{r1}; \mathsf{r}_2]\!]P$, where the inequalities follow from inductive hypotheses and monotonicity of $[\![\mathsf{r}_2]\!]$.

(3) We recall that $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^A \leq \llbracket \mathsf{r}_2 \rrbracket^A \llbracket \mathsf{r}_1 \rrbracket^A$.

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket P) && \text{[(1) and monotonicity of } \alpha\text{]} \\
&\leq \llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^A \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^A\text{]} \\
&\leq \llbracket \mathsf{r}_2 \rrbracket^A \llbracket \mathsf{r}_1 \rrbracket^A \alpha(P) && \text{[recalled above]} \\
&= \llbracket \mathsf{r}_2 \rrbracket^A \alpha(R) && \text{[inductive hp]} \\
&= \alpha(Q) && \text{[inductive hp]}
\end{aligned}$$

So all the lines are equal, in particular $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^A \alpha(P) = \alpha(Q)$.

□

A.2 Soundness and Completeness of (refine-ext)

This technical lemma is used in the following proofs.

Lemma 1. *If $A' \preceq A$ then $A = AA' = A'A$*

Proof. Fix a concrete element $c \in C$. Since $A' \preceq A$ we have $c \leq A'(c) \leq A(c)$. Applying A, by monotonicity we get $A(c) \leq AA'(c) \leq AA(c) = A(c)$, where the last equality is idempotency of A. This means $A = AA'$. Now consider $A'A(c)$. Since A is a closure operator $A'A(c) \leq A(A'A(c))$. But we just showed $AA'(A(c)) = A(A(c)) = A(c)$. Lastly, since A' is a closure operator too, $A(c) \leq A'A(c)$. Hence $A(c) \leq A'A(c) \leq A(c)$, so $A(c) = A'A(c)$.

We point out that, by injectivity of γ, this also means $\alpha\gamma'\alpha' = \alpha$.

Proof (Proof of Theorem 3). We recall that the intuitive premise $A\llbracket \mathsf{r} \rrbracket^{A'} A(P) = A(Q)$ of the rule formally is $\alpha\gamma'\llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) = \alpha(Q)$. Since the proof of Theorem 2 is by induction, we can extend it just proving the inductive case for (refine-ext).
(1) It's the same as point (1) of extensional soundness (Theorem 2) applied to $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$, since this conclusion does not depend on the abstract domain.
(2-3)

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r} \rrbracket P) && \text{[(1) and monotonicity of } \alpha\text{]} \\
&\leq \llbracket \mathsf{r} \rrbracket^A \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^A\text{]} \\
&= \alpha \llbracket \mathsf{r} \rrbracket \gamma \alpha(P) && \text{[definition]} \\
&= \alpha\gamma'\alpha' \llbracket \mathsf{r} \rrbracket \gamma'\alpha'\gamma\alpha(P) && \text{[Lemma 1]} \\
&= \alpha\gamma' \llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) && \text{[definition]} \\
&= \alpha(Q) && \text{[hypothesis of the rule]}
\end{aligned}$$

Hence all the lines are equal; in particular $\alpha(\llbracket r \rrbracket P) = \alpha(Q)$ and $\llbracket r \rrbracket^A \alpha(P) = \alpha(Q)$. □

Proof (Proof of Theorem 4). First, the hypotheses of the theorem implies $\mathbb{C}^A_P(\llbracket \mathsf{r} \rrbracket)$:

$$\begin{aligned} \llbracket \mathsf{r} \rrbracket^A \alpha(P) &= \alpha(Q) && \text{[hp of the theorem]} \\ &\leq \alpha(\llbracket \mathsf{r} \rrbracket P) && \text{[monotonicity of } \alpha \text{ and hp of the theorem } Q \leq \llbracket \mathsf{r} \rrbracket P\text{]} \\ &\leq \llbracket \mathsf{r} \rrbracket^A \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^A\text{]} \end{aligned}$$

Hence $\alpha(\llbracket \mathsf{r} \rrbracket P) = \llbracket \mathsf{r} \rrbracket^A \alpha(P) = \alpha \llbracket \mathsf{r} \rrbracket \gamma\alpha(P)$, that is local completeness. Moreover $\alpha(Q) = \alpha(\llbracket \mathsf{r} \rrbracket P)$.

Now consider a triple P, r, Q satisfying the hypotheses. If $Q < \llbracket \mathsf{r} \rrbracket P$, using (relax) we get

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

But the first condition is trivial, and the third one is made of $Q \leq \llbracket r \rrbracket P$ (the hypothesis) and $\llbracket r \rrbracket P \leq A(Q)$, that follows because $\alpha(\llbracket r \rrbracket P) = \alpha(Q)$ (shown above) and in a GC this implies $\llbracket r \rrbracket P \leq \gamma\alpha(Q) = A(Q)$. Hence without loss of generality we can assume $Q = \llbracket \mathsf{r} \rrbracket P$.

Now we want to apply (refine-ext) to move to the concrete domain C. Clearly $C \preceq A$. The last hypothesis of the rule can be readily verified recalling that $\llbracket r \rrbracket^C = \llbracket r \rrbracket$ and $\alpha' = \gamma' = \mathrm{id}_C$:

$$\begin{aligned} \alpha \llbracket r \rrbracket^C A(P) &= \alpha \llbracket r \rrbracket A(P) \\ &= \llbracket r \rrbracket^A \alpha(P) \\ &= \alpha(\llbracket \mathsf{r} \rrbracket P) \end{aligned}$$

so if we can show $\vdash_C [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]$ we can apply (refine-ext) to prove the triple $\vdash_A [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]$:

$$\frac{\vdash_C [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P] \quad C \preceq A \quad A \llbracket \mathsf{r} \rrbracket^C A(P) = A(\llbracket \mathsf{r} \rrbracket P)}{\vdash_A [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]}\ (\mathsf{refine\text{-}ext})$$

Lastly, we resort to logical completeness of LCL_A (cf. [2], Th 5.11) to say that the triple $\vdash_C [P]\ \mathsf{r}\ [\llbracket \mathsf{r} \rrbracket P]$ is provable. The hypothesis of that theorem are satisfied: all expressions are globally complete in the concrete domain C, $\llbracket r \rrbracket P \leq \llbracket r \rrbracket P$ and $\llbracket r \rrbracket^\sharp_C \mathrm{id}_C(P) = \llbracket \mathsf{r} \rrbracket P = \mathrm{id}_C(\llbracket r \rrbracket P)$, where we used $\alpha' = \mathrm{id}_C$ and $\llbracket r \rrbracket^\sharp_C = \llbracket r \rrbracket$. □

A.3 Derived Refinement Rules

Proof (Proof of Proposition 1). We show that the hypotheses of (refine-int) implies those of (refine-ext). This means than whenever we can apply the former we could also apply the latter, that in turn means Theorem 3 ensures extensional soundness.

The first two hypotheses $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ and $A' \preceq A$ are shared among the two rules, so we only have to show $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. We recall that $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ implies $Q \leq [\![\mathsf{r}]\!]P$ by extensional soundness.

$$\begin{aligned}
\alpha(Q) &\leq \alpha([\![\mathsf{r}]\!]P) && [Q \leq [\![\mathsf{r}]\!]P \text{ and monotonicity of } \alpha]\\
&\leq [\![\mathsf{r}]\!]^{A}\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^{A}]\\
&= \alpha[\![\mathsf{r}]\!]A(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha'[\![\mathsf{r}]\!]A'A(P) && [\text{Lemma 1}]\\
&= \alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) && [\text{definition}]\\
&\leq \alpha\gamma'[\![\mathsf{r}]\!]^{\sharp}_{A'}\alpha' A(P) && [[\![\mathsf{r}]\!]^{A'} \leq [\![\mathsf{r}]\!]^{\sharp}_{A'}]\\
&= \alpha(Q) && [\text{Last hypothesis of the rule}]
\end{aligned}$$

Hence all the lines are equal, and in particular $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. □

Proof (Proof of Theorem 5). The proof is the same as that of Theorem 4, the only difference being that to apply (refine-int) we need to show $A[\![\mathsf{r}]\!]^{\sharp}_{C}A(P) = A([\![\mathsf{r}]\!]P)$ instead of $A[\![\mathsf{r}]\!]^{C}A(P) = A([\![\mathsf{r}]\!]P)$. However, since in the concrete domain $[\![\mathsf{r}]\!]^{\sharp}_{C} = [\![\mathsf{r}]\!]^{C} = [\![\mathsf{r}]\!]$ the proof still holds. □

Proof (Proof of Proposition 2). As in the proof or Proposition 1 above, we show that the hypotheses of (refine-pre) implies those of (refine-ext).

The first two hypotheses $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ and $A' \preceq A$ are shared among the two rules, so we only have to show $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. We recall that $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ implies by extensional soundness (1) $Q \leq [\![\mathsf{r}]\!]P$ and (3) $[\![\mathsf{r}]\!]^{A'}\alpha'(P) = \alpha'(Q)$.

$$\begin{aligned}
\alpha(Q) &\leq \alpha([\![\mathsf{r}]\!]P) && [Q \leq [\![\mathsf{r}]\!]P \text{ and monotonicity of } \alpha]\\
&\leq [\![\mathsf{r}]\!]^{A}\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^{A}]\\
&= \alpha[\![\mathsf{r}]\!]A(P) && [\text{definition}]\\
&= \alpha[\![\mathsf{r}]\!]A'(P) && [\text{hp of the rule}]\\
&= \alpha\gamma'\alpha'[\![\mathsf{r}]\!]A'(P) && [\text{Lemma 1}]\\
&= \alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha'(Q) && [\text{extensional soundness (3)}]\\
&= \alpha(Q) && [\text{Lemma 1}]
\end{aligned}$$

Hence all the lines are equal, and in particular $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. □

Details about Example 5. The full derivation of the triple $\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]$ for Example 5 is shown in Fig. 6, rotated and split to fit the page. The command r_i = `(x > 1)?; y := y - 1; x := x - 1` is iterated with the Kleene star and we let $R_2 = (\mathtt{y} \in \{1; 2; 100\} \wedge \mathtt{x} = \mathtt{y})$. We also used the logical implication $R_2 \implies (\mathtt{y} \in \{1; 99\} \wedge \mathtt{x} = \mathtt{y})$, both explicitly and implicitly in the equivalence $R_2 \vee (\mathtt{y} \in \{1; 99\} \wedge \mathtt{x} = \mathtt{y}) = R_2$.

$$\dfrac{\dfrac{\mathbb{C}^{\mathsf{Oct}}_{\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}}(\llbracket \texttt{y := y - 1}\rrbracket)}{\vdash_{\mathsf{Oct}} [\mathtt{y} \in \{2;100\} \wedge \mathtt{x} = \mathtt{y}]\ \texttt{y := y - 1}\ [\mathtt{y} \in \{1;99\} \wedge \mathtt{x}-1=\mathtt{y}]}\,(\mathsf{transfer}) \quad \dfrac{\mathbb{C}^{\mathsf{Oct}}_{\mathtt{y}\in\{1;99\}\wedge\mathtt{x}-1=\mathtt{y}}(\llbracket \texttt{x := x - 1}\rrbracket)}{\vdash_{\mathsf{Oct}} [\mathtt{y} \in \{1;99\} \wedge \mathtt{x}-1 = \mathtt{y}]\ \texttt{x := x - 1}\ [\mathtt{y} \in \{1;99\} \wedge \mathtt{x}=\mathtt{y}]}\,(\mathsf{transfer})}{(**)}\,(\mathsf{seq})$$

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathsf{Oct}}_{R_2}(\llbracket \texttt{(x > 1)?}\rrbracket)}{\vdash_{\mathsf{Oct}} [R_2]\ \texttt{(x > 1)?}\ [\mathtt{y} \in \{2;100\} \wedge \mathtt{x} = \mathtt{y}]}\,(\mathsf{transfer}) \quad \dfrac{(**)}{\vdash_{\mathsf{Oct}} [\mathtt{y} \in \{2;100\} \wedge \mathtt{x} = \mathtt{y}]\ \texttt{y := y - 1; x := x - 1}\ [\mathtt{y} \in \{1;99\} \wedge \mathtt{x}=\mathtt{y}]}\,(\mathsf{seq})}{\vdash_{\mathsf{Oct}} [R_2]\ \mathsf{r}_i\ [\mathtt{y} \in \{1;99\} \wedge \mathtt{x} = \mathtt{y}]}\,(\mathsf{seq}) \qquad (\mathtt{y} \in \{1;99\} \wedge \mathtt{x} = \mathtt{y}) \leq A(R_2)}{(*)}\,(\mathsf{iterate})$$

$$\dfrac{\dfrac{\mathbb{C}^{\mathsf{Oct}}_{R}(\llbracket \texttt{x := y}\rrbracket)}{\vdash_{\mathsf{Oct}} [R]\ \texttt{x := y}\ [R_2]}\,(\mathsf{transfer}) \quad \dfrac{\dfrac{(*)}{\vdash_{\mathsf{Oct}} [R_2]\ \mathsf{r}_i^*\ [R_2 \vee (\mathtt{y} \in \{1;99\} \wedge \mathtt{x} = \mathtt{y})]}\,(\mathsf{iterate}) \quad \dfrac{\mathbb{C}^{\mathsf{Oct}}_{R_2}(\llbracket \texttt{(x <= 1)?}\rrbracket)}{\vdash_{\mathsf{Oct}} [R_2]\ \texttt{(x <= 1)?}\ [Q]}\,(\mathsf{transfer})}{\vdash_{\mathsf{Oct}} [R_2]\ \mathsf{r}_i^*\texttt{; (x <= 1)?}\ [Q]}\,(\mathsf{seq})}{\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]}\,(\mathsf{seq})$$

Fig. 6: Derivation of $\vdash_{\mathsf{Oct}} [R]\ \mathsf{r}_2\ [Q]$ for Example 5.

$$\dfrac{\dfrac{\dfrac{\mathbb{C}_P^{\mathsf{Int}_P}([\![\texttt{x != 0?}]\!])}{\vdash_{\mathsf{Int}_P} [P]\ \texttt{x != 0?}\ [P]}\ (\mathsf{transfer}) \qquad \dfrac{\mathbb{C}_P^{\mathsf{Int}_P}([\![\texttt{x >= 0?}]\!])}{\vdash_{\mathsf{Int}_P} [P]\ \texttt{x >= 0?}\ [Q]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Int}_P} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [Q]}\ (\mathsf{seq}) \quad \mathsf{Int}_P \preceq \mathsf{Int} \quad \mathsf{Int}([\![\mathsf{r}]\!]^{\sharp}_{\mathsf{Int}_P}(\mathsf{Int}(P))) = \mathsf{Int}(Q)}{\vdash_{\mathsf{Int}} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [Q]}\ (\mathsf{refine\text{-}int})$$

Fig. 7: Derivation of $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$ for Example 8.

Example 8 (Supplement to Example 6). Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the abstract domain Int of intervals, the concrete points $P = \{-1, 1\}$ and $Q = \{1\}$, commands $\mathsf{r}_1 \triangleq \texttt{x != 0?}$, $\mathsf{r}_2 \triangleq \texttt{x >= 0?}$ and $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Let $f_1 = [\![\mathsf{r}_1]\!]$, $f_2 = [\![\mathsf{r}_2]\!]$ and $f = [\![\mathsf{r}]\!] = f_2 \circ f_1$. Observe that in the concrete semantics $f_1(P) = P$ and $f(P) = f_2(P) = \{1\}$. Consider LCL$_A$ extended with ($\mathsf{refine\text{-}pre}$), and let us show that we cannot prove $\vdash_{\mathsf{Int}} [P]\ \mathsf{r}\ [Q]$. Inspecting the logic, we can only apply three rules to prove this triple: (relax), ($\mathsf{refine\text{-}pre}$) or ($\mathsf{seq}$). To apply rule ($\mathsf{relax}$) we would need either an under-approximation P' of P with the same abstraction, that does not exist, or an over-approximation of Q, that would be unsound since $Q = f(P)$. Hence we cannot apply (relax). Suppose to apply ($\mathsf{refine\text{-}pre}$): any A' used in the rule should satisfy $A' \preceq \mathsf{Int}$ and $A'(P) = \mathsf{Int}(P)$; as we remarked in Example 6 this means that $P' \subset P$ implies $A'(P') \subset A'(P)$. Again this means we cannot apply (relax) even after the domain refinement. The only rule that can be applied is then (seq): to do that, we must prove two triples $\vdash_{A'} [P]\ \mathsf{r}_1\ [R]$ and $\vdash_{A'} [R]\ \mathsf{r}_2\ [Q]$. Irrespective of how we prove the first triple, by soundness (Theorem 2) we have $R \subseteq f_1(P) = P$ and $A'(R) = A'(f_1(P)) = A'(P)$, so again $R = P$. Now we should prove a triple $\vdash_{A'} [P]\ \mathsf{r}_2\ [Q]$, but this is impossible since by soundness this would imply local completeness of $[\![\mathsf{r}_2]\!] = f_2$ on P in A', that does not hold:

$$\begin{aligned} A' f_2(P) &= A'(\{1\}) \subseteq \mathsf{Int}(\{1\}) = \{1\} \\ A' f_2 A'(P) &\supseteq f_2 A'(P) = f_2(\mathsf{Int}(P)) = \{0, 1\} \end{aligned}$$

Observe that, if we add ($\mathsf{refine\text{-}int}$) to the proof system, we can use it to change the domain to one where we can express P (for instance, the concrete domain $\mathcal{P}(\mathbb{Z})$ or the refinement $\mathsf{Int} \cup \{P\}$) to prove the triple applying (seq) and then ($\mathsf{transfer}$) on both subtrees, as shown in Fig. 7. □

References

1. Bruni, R., Giacobazzi, R., Gori, R., Garcia-Contreras, I., Pavlovic, D.: Abstract extensionality: On the properties of incomplete abstract interpretations. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371096
2. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A logic for locally complete abstract interpretations. In: Proc. of LICS'21. pp. 1–13. IEEE (2021). https://doi.org/10.1109/LICS52264.2021.9470608

3. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: Abstract interpretation repair. In: Jhala, R., Dillig, I. (eds.) Proc. of PLDI'22. pp. 426–441. ACM (2022). https://doi.org/10.1145/3519939.3523453
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Proc. of CAV'00. pp. 154–169. Springer (2000). https://doi.org/10.1007/10722167_15
5. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77. p. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of POPL'79. p. 269–282. ACM (1979). https://doi.org/10.1145/567752.567778
8. Cousot, P., Cousot, R.: Abstract interpretation: Past, present and future. In: Proc. of CSL-LICS'14. ACM (2014). https://doi.org/10.1145/2603088.2603165
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of POPL'78. p. 84–96. ACM (1978). https://doi.org/10.1145/512760.512770
10. Giacobazzi, R., Logozzo, F., Ranzato, F.: Analyzing program analyses. In: Rajamani, S.K., Walker, D. (eds.) Proc. of POPL'15. pp. 261–273. ACM (2015). https://doi.org/10.1145/2676726.2676987
11. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (mar 2000). https://doi.org/10.1145/333979.333989
12. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Launchbury, J., Mitchell, J.C. (eds.) Proc. of POPL'02. pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
13. Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: Hu, Z. (ed.) Proc. of APLAS'09. LNCS, vol. 5904, pp. 343–358. Springer (2009). https://doi.org/10.1007/978-3-642-10672-9_24
14. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Proc. of CAV'06. LNCS, vol. 4144, pp. 123–136. Springer (2006). https://doi.org/10.1007/11817963_14
15. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1
16. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371078
17. Raad, A., Berdine, J., Dang, H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Lahiri, S.K., Wang, C. (eds.) Proc. of CAV'20, Part II. LNCS, vol. 12225, pp. 225–252. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_14
18. Winskel, G.: The Formal Semantics of Programming Languages: an Introduction. MIT press (1993)