A Logic for the Imprecision of Abstract Interpretations[*]

MARCO CAMPION, Inria Paris - ENS - Université PSL, France
MILA DALLA PREDA, University of Verona, Italy
ROBERTO GIACOBAZZI, University of Arizona, USA
CATERINA URBAN, Inria Paris - ENS - Université PSL, France

In numerical analysis, error propagation refers to how small inaccuracies in input data or intermediate computations accumulate and affect the final result, typically governed by the stability and sensitivity of the algorithm with respect to some perturbations. The definition of a similar concept in approximated program analysis is still a challenge. In abstract interpretation, inaccuracy arises from the abstraction itself, and the propagation of this error is dictated by the abstract interpreter. In most cases, such imprecision is inevitable. In this paper we introduce a logic for deriving (upper) bounds on the inaccuracy of an abstract interpretation. We are able to derive a function that bounds the imprecision of the result of an abstract interpreter from the imprecision of its input data. When this holds we have what we call partial local completeness of the abstract interpreter, a weaker form of completeness known in the literature. To this end, we introduce the notion of a *generator* for a property represented in the abstract domain. Generators allow us to restrict the search space when verifying whether the bounding function holds for a given program and input. We then introduce a program logic, called *Error Propagation Logic* (*EPL*), for propagating the error bounds produced by an abstract interpretation. This logic is a combination of correctness and incorrectness logics and a logic for *program ω-continuity* that is also introduced in this paper.

Additional Key Words and Phrases: Abstract Interpretation, Program Analysis, Program Logic, Galois Connection, Generator, Partial Completeness, Program Continuity.

1 INTRODUCTION

Abstract interpretation [Cousot and Cousot 1977, 1979] is a general theory for the approximation of program semantics. The approximation is achieved by interpreting programs in a simplified domain, called the abstract domain. The object of approximation are properties of programs, for instance the set of values stored in variables at some program point or the set of reachable program points. In its classical formulation, abstract interpretation considers a domain of concrete (or exact) properties $\boldsymbol{C}$ and a domain of abstract (or approximated) properties $\boldsymbol{\mathcal{A}}$, both assumed to be partially ordered sets, related by a Galois connection, namely by a pair (α,γ) where the abstraction function $\alpha\colon \boldsymbol{C} \to \boldsymbol{\mathcal{A}}$ maps any concrete property into an approximated abstract one, and the concretization function $\gamma\colon \boldsymbol{\mathcal{A}} \to \boldsymbol{C}$ gives the meaning to any abstract property as a corresponding concrete one.

The precision of an abstract interpretation has been the subject of an extensive study over the past decades (see, e.g., [Bruni et al. 2022, 2023; Campion et al. 2022; Giacobazzi et al. 2015]). It is well-known that interpreting a program in an abstract domain typically differs from abstracting its concrete semantics. This difference arises because *abstract interpreters approximate intermediate program computation states, introducing and propagating imprecision, which cumulates.* Although abstract interpretation guarantees soundness, thus preventing false negatives, its imprecision may result in false alarms, commonly referred to as false positives. This phenomenon is known as *incompleteness* or *imprecision* of the static analysis.

Completeness is a desirable property that expresses the absence of imprecision in the abstract interpreter with respect to the abstraction of the concrete execution [Giacobazzi et al. 2000]. If

[*] This is the extended version of [Campion et al. 2026], containing all proofs and various corrections and improvements.

Authors' addresses: Marco Campion, Inria Paris - ENS - Université PSL, Paris, France, marco.campion@inria.fr; Mila Dalla Preda, University of Verona, , Italy, mila.dallapreda@univr.it; Roberto Giacobazzi, University of Arizona, Tucson, USA, giacobazzi@arizona.edu; Caterina Urban, Inria Paris - ENS - Université PSL, Paris, France, caterina.urban@inria.fr.

$[\![\mathsf{P}]\!]: \boldsymbol{C} \rightarrow \boldsymbol{C}$ is the concrete semantics of a program P, and $[\![\mathsf{P}]\!]^\sharp: \mathcal{A} \rightarrow \mathcal{A}$ is its abstract interpretation, completeness corresponds to the equation:

$$\forall c \in \boldsymbol{C}.\, \alpha([\![\mathsf{P}]\!]c) = [\![\mathsf{P}]\!]^\sharp \alpha(c) \tag{1}$$

When the universal quantification in (1) is restricted to a strict subset $S \subset \boldsymbol{C}$ of concrete properties, namely, $\forall c \in S$, condition (1) is referred to as *local completeness* [Bruni et al. 2021, 2023]. In the context of static verification, completeness means that no false positives are raised by the abstract interpretation-based static analysis when used to verify any abstract property on the program computation [Cousot 2021; Rival and Yi 2020]. Completeness, however, is a *very rare* condition to be satisfied in practice, even in its local form [Campion et al. 2022; Giacobazzi et al. 2015]. Abstract domains can be refined in order to achieve completeness (e.g., see [Giacobazzi et al. 2000]), but the refinement may result in a way too concrete abstract domain, making the abstract interpreter inefficient if not boiling down to the concrete interpretation [Giacobazzi et al. 2015]. The weaker notion of local completeness may produce a more complete abstract domain tailored on a specific program and (set of) input states by means of the so called Abstract Interpretation Repair (AIR) [Bruni et al. 2022], a technique that shares similarities with the principles of the well known Counter-Example Guided Abstraction Refinement (CEGAR) in abstract model-checking. However, local completeness can still be too strong as an assumption, as a certain degree of imprecision may be acceptable (or even unavoidable) in analysis or verification without compromising the overall quality of the results.

To address this, Campion et al. [2022] introduced a more permissive notion of standard local completeness, called ε*-partial local completeness*. Partial completeness relaxes the equality requirement between the abstraction of the concrete execution and the result of the abstract interpretation, by allowing a *bounded* level of imprecision by the constant $\varepsilon \in \mathbb{R}_{\geq 0}$. This imprecision is measured by a distance $\delta_{\mathcal{A}}: \mathcal{A} \times \mathcal{A} \rightarrow \mathbb{R}_{\geq 0}$ over the elements of the abstract domain, and it is formalized in [Campion et al. 2023] as a pre-metric compatible with the underlying ordering of the abstract domain. Formally, (1) in its local form is weakened by:

$$\forall c \in S.\, \delta_{\mathcal{A}}(\alpha([\![\mathsf{P}]\!]c), [\![\mathsf{P}]\!]^\sharp \alpha(c)) \leq \varepsilon \tag{2}$$

where $\varepsilon \in \mathbb{R}_{\geq 0}$ is the allowed amount of imprecision. Notably, when the pre-metric $\delta_{\mathcal{A}}$ is able to distinguish equal elements (formally, $\forall a_1, a_2 \in \mathcal{A}.\, a_1 = a_2 \Leftrightarrow \delta_{\mathcal{A}}(a_1, a_2) = 0$), then 0-partial local completeness boils down to standard local completeness. Note that both (1) and (2) involve quantification over elements of the concrete domain. Specifically, for (2), the condition must hold at *each individual input* in the set S, while for (1), it must hold over *the entire concrete domain*.

Understanding how this limited imprecision is preserved or amplified during program analysis is a key challenge. This brings us to the central question that guides our work:

Can we define a program logic that models error propagation in abstract interpretation?

1.1 Main Contribution

In this paper, we address the above question by introducing a novel program logic, called *Error Propagation Logic* (*EPL*), designed to provide a logical framework for reasoning about the imprecision of abstract interpreters. Specifically, it enables the derivation of upper bounds on the output imprecision of an abstract interpreter as a function of the imprecision in its input, where the imprecision is parametrized by the chosen pre-metric. By capturing how error propagates through program constructs, EPL facilitates compositional reasoning and supports the systematic verification of precision guarantees in abstract interpretations. To this end, we first characterize the minimal set of program properties that play a key role in the error propagation, which we call *generators*.

Building on this notion, we then design EPL by combining correctness and incorrectness logics with a program logic for the ω-*continuity*.

1.1.1 Generators of an Abstract Domain. We observe that certain abstract properties in $\mathcal{A}$ admit a *minimal* representation in the concrete domain, with respect to the concrete partial ordering $\sqsubseteq_C$. We refer to these minimal concrete properties as *generators*. Intuitively, for an abstract property $a \in \mathcal{A}$, the concrete property $g \in C$ qualifies as a generator of a when $\alpha(g) = a$, meaning that g maps to a via the abstraction function α, and there is no other concrete property $c \in C$ such that $c \sqsubset_C g$ and $\alpha(c) = a$. In other words, g captures the *least amount of concrete information* required to represent a in the abstract domain via α.

One of the simplest examples that naturally illustrates the notion of generators is the interval abstract domain, defined as follows:

$$\mathsf{Int} \overset{\text{def}}{=} \{[a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \leq b\} \cup \{\bot_{\mathsf{Int}}\}$$

The interval abstraction α_{Int}, defined by Cousot and Cousot [1977], abstracts a set of integers $S \in \wp(\mathbb{Z})$ to the smallest interval in Int containing it. Let $[a, b] \in \mathsf{Int}$ be a closed interval, namely $a, b \in \mathbb{Z}$. In this case, the set containing the extreme values of $[a, b]$, i.e., the set $\{a, b\}$, represents the minimal information in the concrete domain $\wp(\mathbb{Z})$, in terms of $\subseteq$, which is necessary to describe the interval. Clearly, the generator of every closed interval is unique. But there are also abstract properties not admitting generators. This is the case for all infinite intervals of the form $[a, +\infty]$, $[-\infty, b]$, $[-\infty, +\infty]$.

Generators play a central role in proving (1) and (2) as, for certain sets of concrete properties $S \subseteq C$, establishing (partial local) completeness on the generators alone is a *necessary and sufficient condition* to ensure the property on the whole set S. We illustrate this result through an example.

Example 1.1. Consider the following program computing the absolute value of an input:

$$\mathsf{ABS}: \ \textbf{if}\ x \geq 0\ \textbf{then}\ x := x\ \textbf{else}\ x := -x$$

Assume $[\![\mathsf{ABS}]\!] \colon \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$ is the standard collecting reachability semantics and consider the standard interval abstract interpretation $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}} \colon \mathsf{Int} \to \mathsf{Int}$. Suppose that we are interested in studying the local completeness of $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ over the set of inputs

$$S = \{N \in \wp(\mathbb{Z}) \mid \{1, 5\} \subseteq N \subseteq \{1, 2, 3, 4, 5\}\} \subset \wp(\mathbb{Z})$$

namely all the integer sets whose abstraction gives the interval $[1, 5]$. Then $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ is local complete at S *if and only if* it is complete at the integer set $\{1, 5\}$, namely at the generator of the abstract property $[1, 5] \in \mathsf{Int}$. This is indeed the case: $\alpha_{\mathsf{Int}}([\![\mathsf{ABS}]\!]\{1, 5\}) = [1, 5] = [\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}[1, 5]$, and so we can conclude that $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ is local complete at S without further checking the property on the other sets in S. Instead, if we consider the set of inputs

$$R = \{N \in \wp(\mathbb{Z}) \mid \{-4, 4\} \subseteq N \subseteq \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}\} \subset \wp(\mathbb{Z})$$

then a proof of local *incompleteness* at the generator $\{-4, 4\}$ also entails a proof of incompleteness of $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ over all integer sets in R. Indeed:

$$\alpha_{\mathsf{Int}}([\![\mathsf{ABS}]\!]\{-4, 4\}) = [4, 4] \sqsubset_{\mathsf{Int}} [0, 4] = [\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}[-4, 4]$$

We observe a similar result for the partial local completeness property with a slightly different consequence: a proof of ε-partial local completeness at the generator $\{-4, 4\}$ provides an *upper bound* ε *of imprecision* for all integer sets in R. For example, we can consider an error measure $\delta^{\sim}_{\mathsf{Int}}$ such that $\delta^{\sim}_{\mathsf{Int}}([4, 4], [0, 4]) = 4$, modeling the fact that the interval $[0, 4]$ has four more values than the interval $[4, 4]$. Then the distance $\delta^{\sim}_{\mathsf{Int}}(\alpha_{\mathsf{Int}}([\![\mathsf{ABS}]\!]\{-4, 4\}), [\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}[-4, 4])$ corresponds to 4, namely $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ is 4-partial local complete at $\{-4, 4\}$, and thus also at all the sets in R. ♦

More generally, instead of considering a constant bound ε, we introduce a *bounding function* $\boldsymbol{e} \colon \boldsymbol{C} \longrightarrow \mathbb{R}_{\geq 0}$ that assigns a specific bound to each concrete property. The partial local completeness property in (2) is then reformulated as:

$$\forall c \in S.\, \delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P} \rrbracket c), \llbracket \mathsf{P} \rrbracket^{\sharp} \alpha(c)) \leq \boldsymbol{e}(c)$$

thus enabling a more flexible and expressive framework for reasoning about incomplete analyses. In the example above, a suitable bounding function for any closed set $N \in \wp(\mathbb{Z})$—that is, a set whose abstraction is a closed interval—is defined as follows: $\boldsymbol{e}(N) = 0$ if N is empty or $max(N) < 0$; otherwise $\boldsymbol{e}(N) = max(N)$. This choice is motivated by generators of the form $\{n_1, n_2\}$ with $n_2 \in \mathbb{N}$ and $n_1 = -n_2$, for which we obtain:

$$\delta^{\sim}_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(\llbracket \mathsf{ABS} \rrbracket \{n_1, n_2\}), \llbracket \mathsf{ABS} \rrbracket^{\sharp}_{\mathsf{Int}} \alpha_{\mathsf{Int}}(\{n_1, n_2\})) = n_2$$

This represents a *worst-case scenario*: the concrete result is always the singleton n_2, while the abstract semantics includes 0 due to the guard, yielding an over-approximation of size n_2. Hence, $\llbracket \mathsf{ABS} \rrbracket^{\sharp}_{\mathsf{Int}}$ will never return an interval whose imprecision—measured in terms of the number of spurious elements—exceeds $\boldsymbol{e}(N)$ relative to the abstraction of the concrete execution.

At this point, a natural follow-up question we aim to address is the following:

> *Can generators be leveraged to propagate, inductively on the syntax of the program, a bounding function* $\boldsymbol{e}$ *that estimates the worst-case imprecision incurred by an abstract interpretation over a given set of concrete properties?*

1.1.2 An Error Propagation Logic. Building on the notion of generators, we introduce EPL that derives judgments of the form: $\boldsymbol{e}\text{-}\text{Bound}(\mathsf{P}, g)_{\mathcal{A}}$, where P is a program and g is a generator of an abstract property $a \in \mathcal{A}$. The goal of EPL is to soundly establish the *worst-case imprecision of an abstract interpretation* with respect to a concrete semantics and a chosen imprecision distance $\delta_{\mathcal{A}}$, over the *chain* of properties $\{c \in \boldsymbol{C} \mid g \sqsubseteq_C c \sqsubseteq_C \gamma(a)\}$, denoted by $[g, \gamma(a)]$. The proof system derives a bounding function $\boldsymbol{e}$ inductively on the syntax of P: it starts from basic commands (assignments and Boolean guards), and then *propagates and updates the bounding function compositionally through the program structure*. The result is twofold:

(1) if $\boldsymbol{e}(g) > 0$ for a given generator g, then $\boldsymbol{e}(g)$ provides *an upper bound on the imprecision* over all elements in $[g, \gamma(a)]$;
(2) if $\boldsymbol{e}(g) = 0$, then the abstract interpreter is able to *precisely analyze every element* in $[g, \gamma(a)]$.

The last scenario further implies that any representable specification in the abstract domain can be verified *without false positives on these inputs.*

The key challenge in designing EPL lies in defining the bounding function $\boldsymbol{e}$ for the composition $\mathsf{P}_1; \mathsf{P}_2$ of two programs in terms of the bounding functions $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ for the individual programs:

$$\frac{\boldsymbol{e}_1\text{-}\text{Bound}(\mathsf{P}_1, g)_{\mathcal{A}} \quad \boldsymbol{e}_2\text{-}\text{Bound}(\mathsf{P}_2, h)_{\mathcal{A}} \quad \omega\text{-}\text{Cont}(\mathsf{P}_2)_{\mathcal{A}} \quad \begin{array}{c}[g]\mathsf{P}_1[h] \\ \{g\}\mathsf{P}_1\{\gamma\alpha(h)\}\end{array}}{\boldsymbol{e}\text{-}\text{Bound}(\mathsf{P}_1; \mathsf{P}_2, g)_{\mathcal{A}}} \ (\textbf{Seq})$$

It turns out that, if the *abstract interpretation* of the second program P_2 satisfies ω*-continuity*, captured by the predicate $\omega\text{-}\text{Cont}(\mathsf{P}_2)_{\mathcal{A}}$, then the resulting bounding function $\boldsymbol{e}$ for the composition $\mathsf{P}_1; \mathsf{P}_2$ can be expressed as *the sum* $\boldsymbol{e}(c) = \boldsymbol{e}_2(h) + \omega(\boldsymbol{e}_1(g))$, for all $c \in [g, \gamma(a)]$. The function ω in $\omega\text{-}\text{Cont}(\mathsf{P}_2)_{\mathcal{A}}$, called the *modulus of continuity*, quantifies *how much the output of a program analysis can change in response to small changes in the input.* As we will see in Section 6.3, our proposed notion of ω-continuity is strictly weaker than uniform continuity (and, in fact, than continuity in general) allowing it to capture arbitrary forms of error propagation in abstract interpretation. By leveraging the pre-metric $\delta_{\mathcal{A}}$, the ω-continuity of an abstract interpretation $[\![\mathsf{P}]\!]^\sharp_{\mathcal{A}}$ of a program P is formally defined as the inequality:

$$\forall a_1, a_2 \in \mathcal{A}.\, \delta_{\mathcal{A}}([\![\mathsf{P}]\!]^\sharp_{\mathcal{A}}(a_1), [\![\mathsf{P}]\!]^\sharp_{\mathcal{A}}(a_2)) \leq \omega(\delta_{\mathcal{A}}(a_1, a_2))$$

In particular, when the modulus of continuity is linear, that is, $\omega(t) = Kt$ for some constant K, we obtain Lipschitz continuity, a well-known notion widely used to model bounded linear variations in program behavior under small input perturbations [Chaudhuri et al. 2011; de Amorim et al. 2017].

We introduce a simple sound proof system for deriving a modulus of continuity function ω of a program inductively on its syntax, i.e. for inferring the validity of $\omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{A}}$. Interestingly, EPL integrates *three* distinct logical frameworks: standard *Hoare-style correctness logic* [Hoare 1969], here encoded by the triple $\{g\}\mathsf{P}_1\{\gamma\alpha(h)\}$, *O'Hearn's incorrectness logic* [O'Hearn 2020], here encoded by the triple $[g]\mathsf{P}_1[h]$, and our own *modulus of continuity logic* for deriving $\omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{A}}$. The combination of the first two is essential to capture the nature of local completeness and, in fact, aligns with the LCL fragment introduced in [Bruni et al. 2021, 2023]. The modulus of continuity logic, in turn, ensures compositional reasoning in the presence of error-bounding functions. The entire EPL proof system is made parametric with respect to the generators of the abstract domain. This design choice allows the inference rules to operate over a minimal and representative set of abstract elements. Instead of reasoning over sets of concrete properties, one can focus on proving properties for the generators, which serve as a basis for the abstract domain's structure.

Our broader vision is to integrate error propagation into a Dijkstra-style discipline of programming, thereby ensuring that code design is inherently aligned with the analyzer responsible for verifying the generated code. One possible scenario is that, during the evaluation of a program analysis, the user proposes an error bound and the proof assistant—guided by our logic—verifies its validity online. Moreover, since program analysis is intensional—that is, its precision depends on how the code is written—our proof system can also serve as a guide in code construction. It makes explicit how the program analysis imprecision propagates through the program under analysis and helps ensure that a desired upper bound on imprecision is ultimately achieved. In this perspective, the new notions of ω-continuity and generators of an abstract domain are fundamental: The former is the key notion to let error-bounds propagate in our program logic, while the latter both prunes the search space in the proofs and propagates error-bounds to all elements belonging to the same chain as the generator.

1.2 Structure of the Paper

We summarize our contributions as follows:

- We generalize the notion of partial (local) completeness to use bounding functions (Section 3).
- We formally define the notion of generator for an abstract property (Section 4).
- We establish key results for reducing a proof of partial local completeness from a set of concrete properties $S \subset \boldsymbol{C}$ to the generators only (Section 5).
- We provide a logic for deriving ω-continuity inductively on the program syntax (Section 6.3), as well as the EPL, a logic for deriving a bounding function (Section 6.4).

All the presented results rely on minimal assumptions on the concrete and abstract semantics, the chosen distance function, and the structure of the concrete and abstract domains. For ease of exposition, we complement these results with examples on the interval domain, although they remain fully general and apply to any Galois connection. All proofs can be found in Appendix A.

2 BACKGROUND

We introduce preliminary concepts and notations on order theory, trace and reachability semantics (Section 2.1), and abstract interpretation (Section 2.2). Lastly, in Section 2.3 we recall the notion of pre-metric compatible with a partial ordering, which will be used to formalize the partial completeness property in abstract interpretation.

2.1 Program Semantics

Functions and Order Theory. Given two sets S and T, $\wp(S)$ denotes the powerset of S, the symbol $\varnothing$ corresponds to the empty set, $S \smallsetminus T$ denotes the set-difference, $|S|$ denotes the cardinality of S. Set inclusion is denoted by $S \subseteq T$ while $S \subset T$ denotes strict set inclusion. We denote with $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ the sets of all natural, integer and real numbers, respectively, and with $\mathbb{I} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ any one of the three number sets. We will use the notation $\mathbb{I}^{\infty}_{\geq 0}$ with the following meaning: $\mathbb{I}^{\infty}_{\geq 0} \overset{\text{def}}{=} \{n \in \mathbb{I} \mid n \geq 0\} \cup \{\infty\}$, where for all $n \in \mathbb{I}$, $n < \infty$. As calculation rules, any sum, difference or multiplication that involves the symbol ∞, returns ∞, except for $0 \cdot \infty = \infty \cdot 0 = 0$.

When a binary relation $\sqsubseteq \subseteq S \times S$ is defined over a set which differs from $\mathbb{N}, \mathbb{Z}$ and $\mathbb{R}$, we will use the subscript $\sqsubseteq_S$ except for well-known relations, like equality $=$ and set inclusion $\subseteq$. We will highlight sets that form a partially ordered set by using calligraphic font on letters, e.g. $\mathcal{L}$, instead of standard font for a generic set of objects, e.g. L. That is, $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}} \rangle$ is called a partially ordered set, or briefly *poset*, when $\sqsubseteq_{\mathcal{L}}$ is a partial order relation (i.e., reflexive, antisymmetric and transitive). Its strict version is denoted by the symbol $\sqsubset_{\mathcal{L}}$ such that, for all $x, y \in \mathcal{L}$, $x \sqsubset_{\mathcal{L}} y$ if $x \sqsubseteq_{\mathcal{L}} y$ and $x \neq y$.

A poset $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}} \rangle$ is called a join-semilattice if for all $\{a, b\} \subseteq \mathcal{L}$, their join (i.e. least upper bound, or simply lub), denoted by $a \vee_{\mathcal{L}} b$, is an element of $\mathcal{L}$, and is called a meet-semilattice if for all $\{a, b\} \subseteq \mathcal{L}$, their meet (i.e. greatest lower bound, or simply glb), denoted by $a \wedge_{\mathcal{L}} b$, is an element of $\mathcal{L}$. $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}} \rangle$ is called a lattice if it is both a join- and a meet-semilattice. A lattice is complete when all subsets $X \subseteq \mathcal{L}$ have lubs $\bigvee_{\mathcal{L}} X$ and glbs $\bigwedge_{\mathcal{L}} X$ in $\mathcal{L}$ (empty subset included). A *complete lattice* $\mathcal{L}$ with partial order $\sqsubseteq_{\mathcal{L}}$, lub $\vee_{\mathcal{L}}$, glb $\wedge_{\mathcal{L}}$, the greatest element (top) $\top_{\mathcal{L}}$, and the least element (bottom) $\bot_{\mathcal{L}}$ is denoted by $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}}, \vee_{\mathcal{L}}, \wedge_{\mathcal{L}}, \top_{\mathcal{L}}, \bot_{\mathcal{L}} \rangle$.

A function $f\colon \mathcal{L} \longrightarrow \mathcal{L}$ over a poset $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}} \rangle$ is *order-preserving* (resp. *order-reversing*) if, $\forall x, y \in \mathcal{L}$ such that $x \sqsubseteq_{\mathcal{L}} y$, f *preserves* (resp. *reverses*) the order, i.e., $f(x) \sqsubseteq_{\mathcal{L}} f(y)$ (resp. $f(x) \sqsupseteq_{\mathcal{L}} f(y)$). A function is said to be *injective* when it satisfies: $\forall x, y \in \mathcal{L}.\, f(x) = f(y) \Rightarrow x = y$. It is *surjective* when $\forall y.\, \exists x.\, f(x) = y$. The composition of two functions $f_1\colon L_1 \longrightarrow L_2$, $f_2\colon L_2 \longrightarrow L_3$ is denoted by $f_2 \circ f_1\colon L_1 \longrightarrow L_3$. We also denote by f^n the function obtained by applying f n times, where $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$. A function $f\colon \mathcal{L}_1 \longrightarrow \mathcal{L}_2$ between complete lattices is *additive* if, for all $Y \subseteq \mathcal{L}_1$, $f(\vee_{\mathcal{L}_1} Y) = \vee_{\mathcal{L}_2} f(Y)$. We will abuse notation by writing $f_1 \sqsubseteq_{\mathcal{L}} f_2$ for functions $f_1, f_2\colon \mathcal{L} \longrightarrow \mathcal{L}$, to denote their pointwise ordering: $\forall x \in \mathcal{L}.\, f_1(x) \sqsubseteq_{\mathcal{L}} f_2(x)$.

Trace and Reachability Program Semantics. Let $\langle \Sigma, \tau \rangle$ be a transition system, where Σ is a (potentially infinite) set of program states and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the feasible transitions between states [Cousot 2021]. Let $\Sigma^n \overset{\text{def}}{=} \{s_0 \dots s_{n-1} \mid \forall i < n.\, s_i \in \Sigma\}$ be the set of all sequences of exactly n program states, where ϵ denotes the empty sequence, i.e., $\Sigma^0 \overset{\text{def}}{=} \{\epsilon\}$. We define $\Sigma^\star \overset{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \Sigma^n$ as the set of all finite sequences, $\Sigma^+ \overset{\text{def}}{=} \Sigma^\star \smallsetminus \Sigma^0$ as the set of all non-empty finite sequences, $\Sigma^\infty \overset{\text{def}}{=} \{s_0 \dots \mid \forall i \in \mathbb{N}.\, s_i \in \Sigma\}$ as the set of all infinite sequences, and $\Sigma^{+\infty} \overset{\text{def}}{=} \Sigma^+ \cup \Sigma^\infty$ as the set of all non-empty finite or infinite sequences. Given $\sigma \in \Sigma^{+\infty}$, we write $\sigma_0 \in \Sigma$ to denote the initial state of σ and $\sigma_\omega \in \Sigma$ to denote the final state when $\sigma \in \Sigma^+$. The sequence $\sigma \in \Sigma^{+\infty}$ is called a *trace* whenever it respects the transition relation τ, i.e., for every pair of consecutive states $s, s' \in \sigma$, it holds that $(s, s') \in \tau$. The *trace semantics* generated by a transition system $\langle \Sigma, \tau \rangle$ is the union of all finite traces that are terminating in a final state, and all non-terminating infinite traces [Cousot 2021]. Let Prog be the set of all syntactically well-defined programs in some Turing complete programming language. Given a program $\mathsf{P} \in \mathrm{Prog}$, we write $[\![\mathsf{P}]\!]^{\mathcal{T}}$ to denote the trace semantics of the specific program P.

The trace semantics fully describes the behavior of a program. However, reasoning about a particular property of a program is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. For instance, the *reachability semantics* of a program focuses on the final states reachable from a starting set of input states. It is defined by the function $[\![\mathsf{P}]\!] \colon \wp(\Sigma) \to \wp(\Sigma)$. Given an initial set of states $S \in \wp(\Sigma)$, $[\![\mathsf{P}]\!]S$ collects all the final program states starting from S and reaching the end of program P. It is the standard predicate transformer semantics (also called strongest post-condition semantics) since $[\![\mathsf{P}]\!]S \in \wp(\Sigma)$ turns out to be the strongest state predicate for the state precondition $S \in \wp(\Sigma)$. For all $\mathsf{P} \in \mathrm{Prog}$, $[\![\mathsf{P}]\!]$ is an additive function on the complete lattice $\langle \wp(\Sigma), \subseteq, \cup, \cap, \Sigma, \varnothing \rangle$, so that $[\![\mathsf{P}]\!]S = \bigcup_{s \in S} [\![\mathsf{P}]\!]\{s\}$ holds. When $[\![\mathsf{P}]\!]$ is applied to a singleton $\{s\}$, we use the simpler notation $[\![\mathsf{P}]\!]s$ in place of $[\![\mathsf{P}]\!]\{s\}$.

2.2 Abstract Interpretation

We recall some background on the standard Galois connection-based abstract interpretation framework as defined by Cousot and Cousot [1977, 1979] and based on the correspondence between a domain of concrete (or exact) properties and a domain of abstract (or approximate) properties.

Abstractions. Galois connections (sometimes called Galois adjunctions) formalize the correspondence between concrete elements, also called *concrete properties* (e.g., sets of traces), and abstract elements, also called *abstract properties* (e.g., sets of reachable states) in case there is always a most precise abstract property over-approximating any concrete property.

*Definition 2.1 (**Galois connection**).* Given posets $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle$, called the *concrete domain*, and $\langle \boldsymbol{\mathcal{A}}, \sqsubseteq_{\boldsymbol{\mathcal{A}}} \rangle$, called the *abstract domain*, the pair of order-preserving functions $\alpha : \boldsymbol{C} \to \boldsymbol{\mathcal{A}}$ (the *abstraction* or lower adjoint) and $\gamma : \boldsymbol{\mathcal{A}} \to \boldsymbol{C}$ (the *concretization* or upper adjoint) is a *Galois Connection* (GC) when the following holds:

$$\forall c \in \boldsymbol{C}. \forall a \in \boldsymbol{\mathcal{A}}.\ \alpha(c) \sqsubseteq_{\boldsymbol{\mathcal{A}}} a \ \Leftrightarrow\ c \sqsubseteq_{\boldsymbol{C}} \gamma(a)$$

which will be denoted by $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \boldsymbol{\mathcal{A}}, \sqsubseteq_{\boldsymbol{\mathcal{A}}} \rangle$. ■

The concretization γ provides the concrete meaning $\gamma(a) \in \boldsymbol{C}$ of abstract properties $a \in \boldsymbol{\mathcal{A}}$. $\gamma(a)$ is the least precise element of $\boldsymbol{C}$ (according to $\sqsubseteq_{\boldsymbol{C}}$) that can be over-approximated by a. We say that an abstract property $a \in \boldsymbol{\mathcal{A}}$ is a *sound (over-)approximation* of a concrete property $c \in \boldsymbol{C}$ whenever $c \sqsubseteq_{\boldsymbol{C}} \gamma(a)$. The abstraction $\alpha(c)$ of a concrete element c is the best (most precise in terms of $\sqsubseteq_{\boldsymbol{\mathcal{A}}}$) sound over-approximation of c in the abstract domain $\boldsymbol{\mathcal{A}}$. We say that a concrete element $c \in \boldsymbol{C}$ is *representable* in $\boldsymbol{\mathcal{A}}$ whenever $\gamma(\alpha(c)) = c$. The following two properties are satisfied by all GCs:

(i) $\gamma \circ \alpha : \boldsymbol{C} \to \boldsymbol{C}$ (which will be simply denoted by $\gamma\alpha$) is an upper closure operator, namely, it is order-preserving, idempotent ($\gamma\alpha \circ \gamma\alpha = \gamma\alpha$) and extensive ($\forall c \in \boldsymbol{C}.\ c \sqsubseteq_{\boldsymbol{C}} \gamma\alpha(c)$);

(ii) $\alpha \circ \gamma : \boldsymbol{\mathcal{A}} \to \boldsymbol{\mathcal{A}}$ (which will be simply denoted by $\alpha\gamma$) is a lower closure operator, namely, it is order-preserving, idempotent and reductive ($\forall a \in \boldsymbol{\mathcal{A}}.\ \alpha\gamma(a) \sqsubseteq_{\boldsymbol{\mathcal{A}}} a$).

A GC is a *Galois Insertion* (GI for short, sometimes also called Galois retraction) whenever one of the following conditions is satisfied: (i) $\alpha\gamma = id$ (where $id \overset{\text{def}}{=} \lambda x.\, x$), (ii) α is surjective, (iii) γ is injective. Every GC can be transformed into a GI by removing all these "spurious" abstract elements from $\boldsymbol{\mathcal{A}}$, namely, all $a \in \boldsymbol{\mathcal{A}}$ for which no $c \in \boldsymbol{C}$ abstracts into a. That is, GIs contain non-useless abstract properties, namely, for every abstract element $a \in \boldsymbol{\mathcal{A}}$ there exists a concrete element $c \in \boldsymbol{C}$ such that $\alpha(c) = a$. The following proposition is a direct result from the previous reasoning.

Proposition 2.2. *Let $a \in \boldsymbol{\mathcal{A}}$, then:* $(\exists c \in \boldsymbol{C}.\, \alpha(c) = a) \ \Rightarrow\ \alpha\gamma(a) = a$. □

Example 2.3 (Interval abstraction). A classic example of GC is $\langle \wp(\mathbb{I}), \subseteq \rangle \underset{\alpha_{\mathsf{Int}}}{\overset{\gamma_{\mathsf{Int}}}{\leftrightarrows}} \langle \mathsf{Int}, \sqsubseteq_{\mathsf{Int}} \rangle$, where Int is the *interval* abstract domain [Cousot and Cousot 1976] (Section 1.1.1) commonly used for verifying the absence of arithmetic overflows or out-of-bounds array accesses. It is endowed with the standard ordering $\sqsubseteq_{\mathsf{Int}}$ induced by interval containment. Consider the function $min \colon \wp(\mathbb{I}) \to \mathbb{I} \cup \{-\infty\}$ defined as $min(S) \overset{\text{def}}{=} x$ if there exists $x \in S$ such that for all $y \in S, x \leq y$, while $min(S) \overset{\text{def}}{=} -\infty$ otherwise, and the function $max \colon \wp(\mathbb{I}) \to \mathbb{I} \cup \{+\infty\}$ dually defined. The abstraction $\alpha_{\mathsf{Int}} \colon \wp(\mathbb{I}) \to \mathsf{Int}$ is defined by: $\alpha_{\mathsf{Int}}(S) \overset{\text{def}}{=} \bot_{\mathsf{Int}}$ if $S = \varnothing$; otherwise $\alpha_{\mathsf{Int}}(S) \overset{\text{def}}{=} [min(S), max(S)]$. The concretization $\gamma_{\mathsf{Int}} \colon \mathsf{Int} \to \wp(\mathbb{I})$ is defined by: $\gamma_{\mathsf{Int}}(\bot_{\mathsf{Int}}) \overset{\text{def}}{=} \varnothing$, and $\gamma_{\mathsf{Int}}([a, b]) \overset{\text{def}}{=} \{v \mid v \in [a, b]\}$. It is easy to note that α_{Int} is surjective and therefore gives rise to a GI. ♦

Abstract Interpretation. Consider a GC $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \boldsymbol{\mathcal{A}}, \sqsubseteq_{\boldsymbol{\mathcal{A}}} \rangle$, let $f \colon \boldsymbol{C} \to \boldsymbol{C}$ be a concrete order-preserving function and let $f^\sharp \colon \boldsymbol{\mathcal{A}} \to \boldsymbol{\mathcal{A}}$ be a corresponding abstract (not necessarily order-preserving) function. Then $f^\sharp$ is a *sound* (or *correct*) approximation of f on $\boldsymbol{\mathcal{A}}$ when

$$f \circ \gamma \sqsubseteq_{\boldsymbol{C}} \gamma \circ f^\sharp$$

holds. If $f^\sharp$ is correct for f then abstract least fixpoint correctness holds, that is $\alpha(\mathsf{lfp}(f)) \sqsubseteq_{\boldsymbol{\mathcal{A}}} \mathsf{lfp}(f^\sharp)$ holds, where lfp is the least fixpoint operator. When dealing with GCs, between all abstract functions that approximate a concrete one, we can define the most precise one.

*Definition 2.4 (**Best correct approximation**).* The abstract function $f^\alpha \colon \boldsymbol{\mathcal{A}} \to \boldsymbol{\mathcal{A}}$ defined as

$$f^\alpha \overset{\text{def}}{=} \alpha \circ f \circ \gamma$$

is called the *best correct approximation* (*bca* for short) of f on $\boldsymbol{\mathcal{A}}$. ■

It turns out that any abstract function $f^\sharp$ is a correct approximation of f if and only if $f^\alpha \sqsubseteq_{\boldsymbol{\mathcal{A}}} f^\sharp$ [Cousot 2021]. An abstract function $f^\sharp$ is precise when it is complete.

*Definition 2.5 (**(Local) Completeness**).* Given a set of inputs $S \subseteq \boldsymbol{C}$, a function $f^\sharp \colon \boldsymbol{\mathcal{A}} \to \boldsymbol{\mathcal{A}}$ is said to be a *local complete approximation* (or simply *local complete* [Bruni et al. 2021, 2023]) of $f \colon \boldsymbol{C} \to \boldsymbol{C}$ at $S \subset \boldsymbol{C}$, denoted by the predicate $\mathbb{C}_S(f, f^\sharp)$, when the following holds:

$$\forall c \in S.\, \alpha(f(c)) = f^\sharp(\alpha(c))$$

It is a *complete approximation* [Cousot 2021] if $f^\sharp$ is local complete for the set $S = \boldsymbol{C}$, namely when the equation

$$\alpha \circ f = f^\sharp \circ \alpha$$

holds. Completeness will be denoted by the predicate $\mathbb{C}(f, f^\sharp)$. ■

For the singleton set $\{c\}$, we use the simpler notation $\mathbb{C}_c(f, f^\sharp)$ in place of $\mathbb{C}_{\{c\}}(f, f^\sharp)$. This completeness notion is sometimes referred to as α-completeness [Cousot 2021] or backward-completeness [Giacobazzi and Quintarelli 2001]. Intuitively, local completeness encodes an optimal precision for $f^\sharp$ at the inputs in S, meaning that the abstract behavior of $f^\sharp$ on $\boldsymbol{\mathcal{A}}$ *exactly* matches the abstraction in $\boldsymbol{\mathcal{A}}$ of the concrete behavior of f on all inputs in S. When this holds for all elements of the concrete domain $\boldsymbol{C}$, then $f^\sharp$ is complete. It turns out that, when dealing with GIs, such as the interval abstraction of Example 2.3, the possibility of defining a complete approximation $f^\sharp$ of f only depends upon the concrete function f and the abstraction $\boldsymbol{\mathcal{A}}$, that is, f^α is *the only possible option as complete approximation* of f [Cousot 2021; Giacobazzi et al. 2000]. The same holds for the local version [Bruni et al. 2021, 2023].

Verification. Establishing whether a concrete order-preserving, possibly uncomputable, operator $f\colon \boldsymbol{C} \to \boldsymbol{C}$ satisfies a given property $p \in \boldsymbol{C}$, means checking the inequality $\forall c \in \boldsymbol{C}.\, f(c) \sqsubseteq_C p$. The idea of abstract interpretation is to leverage an abstract sound, possibly computable, computation $f^\sharp\colon \mathcal{A} \to \mathcal{A}$ with respect to f, over a Galois connection $\langle \boldsymbol{C}, \sqsubseteq_C\rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}}\rangle$, to prove the (abstract) inequality $f^\sharp(\alpha(c)) \sqsubseteq_{\mathcal{A}} \alpha(p)$, as shown by the following theorem.

Theorem 2.6. *Consider the GC $\langle \boldsymbol{C}, \sqsubseteq_C\rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}}\rangle$. Let $f^\sharp\colon \mathcal{A} \to \mathcal{A}$ be a sound approximation of $f\colon \boldsymbol{C} \to \boldsymbol{C}$. Let $p \in \boldsymbol{C}$ be a property representable in $\mathcal{A}$, namely $\gamma\alpha(p) = p$. Then the following hold for all $c \in \boldsymbol{C}$:*

(1) $f^\sharp(\alpha(c)) \sqsubseteq_{\mathcal{A}} \alpha(p) \;\Rightarrow\; f(c) \sqsubseteq_C p$;
(2) if $\mathbb{C}(f, f^\sharp)$ holds then: $f^\sharp(\alpha(c)) \sqsubseteq_{\mathcal{A}} \alpha(p) \;\Leftrightarrow\; f(c) \sqsubseteq_C p$. □

The abstract interpretation $f^\sharp$ raises an alarm when $f^\sharp(\alpha(c)) \not\sqsubseteq_{\mathcal{A}} \alpha(p)$, and this alarm is a false positive when $f(c) \sqsubseteq_C p$ holds, true alarm otherwise. When $f^\sharp$ is proved to be complete, then, for representable properties $p \in \boldsymbol{C}$, proving $f(c) \sqsubseteq_C p$ is *equivalent* to checking whether $f^\sharp(\alpha(c)) \sqsubseteq_{\mathcal{A}} \alpha(p)$ holds, i.e. no false positives can arise from checking the specification through the abstract computation. Giacobazzi et al. [2015] proved that, when $f^\sharp$ represents an always terminating computation (e.g., a static program analyzer [Miné 2017; Rival and Yi 2020]) and $\mathcal{A}$ is not trivial (i.e., $\mathcal{A} \neq \boldsymbol{C}$ and $\mathcal{A} \neq \{\top_{\mathcal{A}}\}$), the presence of false positives is unavoidable due to the need of $f^\sharp$ to over-approximates the concrete computation f.

2.3 Pre-metrics on Posets

It is known that (local) completeness is hard to achieve [Campion et al. 2022; Giacobazzi et al. 2015]. For this reason, Campion et al. [2022, 2023] introduced a weaker notion of local completeness, called *partial local completeness*, by allowing a *limited* amount of imprecision introduced by $f^\sharp(\alpha(c))$ with respect to $\alpha(f(c))$. The distance between $\alpha(f(c))$ and $f^\sharp(\alpha(c))$ can be measured by *pre-metrics* that manifest a form of compatibility with the underlying partial order of the abstract domain $\mathcal{A}$. This compatibility is formalized by the (*chain-order*) axiom [Campion et al. 2023].

*Definition 2.7 (**Order-compatible pre-metric**).* Let $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}}\rangle$ be a poset. $\delta_{\mathcal{L}} : \mathcal{L} \times \mathcal{L} \to \mathbb{I}^{\infty}_{\geq 0}$ is a *pre-metric compatible with the partial ordering* ($\sqsubseteq_{\mathcal{L}}$*-compatible* for short) if the following hold:

$$(\textit{if-identity})\;\; \forall x, y \in \mathcal{L}\colon x = y \;\Rightarrow\; \delta_{\mathcal{L}}(x, y) = 0\,;$$
$$(\textit{chain-order})\;\; \forall x, y, z \in \mathcal{L}\colon x \sqsubseteq_{\mathcal{L}} y \sqsubseteq_{\mathcal{L}} z \;\Rightarrow\; \delta_{\mathcal{L}}(x, y) \leq \delta_{\mathcal{L}}(x, z) \,\wedge\, \delta_{\mathcal{L}}(y, z) \leq \delta_{\mathcal{L}}(x, z)\,. \;\blacksquare$$

The first axiom states that when $\delta_{\mathcal{L}}$ calculates the distance between two equal elements, it must return 0. However, the converse may not hold: $\delta_{\mathcal{L}}(x, y)$ could be 0 even if $x \neq y$. This may happen, for instance, when $\delta_{\mathcal{L}}$ is computing the distance with some level of approximation, leading to recognizing two elements as “close” even if they are not the same element. Axiom (*if-identity*) alone defines $\delta_{\mathcal{L}}$ to be a *pre-metric* [Deza and Laurent 1997], a weakening of the standard metric definition. The second axiom asks for a compatibility with the ordering $\sqsubseteq_{\mathcal{L}}$. More specifically, the distance between the two extremes of a chain $x \sqsubseteq_{\mathcal{L}} y \sqsubseteq_{\mathcal{L}} z$, must always be greater or equal than the distance between intermediate elements. For instance, let $f_1^\sharp$ and $f_2^\sharp$ be two sound abstract computations of a concrete order-preserving function f. If $f_1^\sharp$ is more precise than $f_2^\sharp$, i.e., $f_1^\sharp \sqsubseteq_{\mathcal{A}} f_2^\sharp$, we expect a decrease in the imprecision (distance) with respect to the concrete computation when using $f_1^\sharp$ rather than $f_2^\sharp$, i.e., $\forall c \in \boldsymbol{C}.\, \delta_{\mathcal{A}}(\alpha(f(c)), f_1^\sharp(\alpha(c))) \leq \delta_{\mathcal{A}}(\alpha(f(c)), f_2^\sharp(\alpha(c)))$, and, furthermore, the distance between the two abstract computations $\delta_{\mathcal{A}}(f_1^\sharp(\alpha(c)), f_2^\sharp(\alpha(c)))$ should never exceed $\delta_{\mathcal{A}}(\alpha(f(c)), f_2^\sharp(\alpha(c)))$ since $\alpha \circ f \sqsubseteq_{\mathcal{A}} f_1^\sharp \sqsubseteq_{\mathcal{A}} f_2^\sharp$.

Definition 2.7 is general enough to be instantiated with distances used in the literature of abstract interpretation (see, e.g., [Campion et al. 2025, 2022; Casso et al. 2019; Di Pierro and Wiklicky 2000; Liew et al. 2024; Logozzo 2009; Sotin 2010]). We briefly recall some of these, as they will be used extensively in the examples throughout the paper.

Example 2.8 (Equality distance). The following distance between any element $x, y \in \mathcal{L}$

$$\delta^{=}_{\mathcal{L}}(x, y) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } x = y, \\ \infty & \text{otherwise} \end{cases}$$

satisfies both axioms of Definition 2.7, therefore it is a $\sqsubseteq_{\mathcal{L}}$-compatible pre-metric, and it will be called the *equality distance*. In fact, $\delta^{=}_{\mathcal{L}}$ is only able to distinguish equal elements in $\mathcal{L}$. ⧫

Example 2.9 (Volume distance). Let us consider the GC $\langle \wp(\mathbb{R}^n), \subseteq \rangle \underset{\alpha_{\boldsymbol{N}}}{\overset{\gamma_{\boldsymbol{N}}}{\leftrightarrows}} \langle \boldsymbol{N}, \sqsubseteq_{\boldsymbol{N}} \rangle$ where $\boldsymbol{N}$ contains a specific class of convex numerical polytopes. For instance, $\boldsymbol{N}$ could be the domain of hyperrectangles ($\boldsymbol{N} = \mathsf{Int}^n$), or zonotopes [Gehr et al. 2018; Girard 2005] ($\boldsymbol{N} = \mathsf{Zone}$), or octagons [Miné 2006] ($\boldsymbol{N} = \mathsf{Oct}$). The abstraction $\alpha_{\boldsymbol{N}}$ abstracts a set of points $S \in \wp(\mathbb{R}^n)$ into the smallest n-dimensional convex polytope in $\boldsymbol{N}$ containing them. We define the $\sqsubseteq_{\boldsymbol{N}}$-compatible pre-metric $\delta^{\mathit{Vol}}_{\boldsymbol{N}} : \boldsymbol{N} \times \boldsymbol{N} \longrightarrow \mathbb{R}^{\infty}_{\geq 0}$ as follows:

$$\delta^{\mathit{Vol}}_{\boldsymbol{N}}(N_1, N_2) \overset{\text{def}}{=} \mathit{Av}(\mathit{Vol}(N_2) - \mathit{Vol}(N_1))$$

calculating the absolute value (Av) of the difference between the volume of two polytope $N_1, N_2 \in \boldsymbol{N}$. The volume function $\mathit{Vol} : \boldsymbol{N} \longrightarrow \mathbb{R}^{\infty}_{\geq 0}$ is assumed to be order-preserving on $\gamma_{\boldsymbol{N}}$ (i.e., if $\gamma_{\boldsymbol{N}}(N_1) \subseteq \gamma_{\boldsymbol{N}}(N_2)$ then $\mathit{Vol}(N_1) \leq \mathit{Vol}(N_2)$) and it could be an over-approximation of the exact volume computation, thus satisfying (*if-identity*) of Definition 2.7. The order-compatible pre-metric $\delta^{\mathit{Vol}}_{\boldsymbol{N}}$ could be used to compare the volume between numerical invariants generated by two program analysis or between a program analysis and the actual strongest invariant from the concrete computation. ⧫

Example 2.10 (Counting distance over Int*).* We define the following distance $\delta^{\sim}_{\mathsf{Int}} : \mathsf{Int} \times \mathsf{Int} \longrightarrow \mathbb{N}^{\infty}$ over the domain of integer intervals Int:

$$\delta^{\sim}_{\mathsf{Int}}([a, b], [c, d]) \overset{\text{def}}{=} \begin{cases} 0 & \text{if } [a, b] = [c, d], \\ \mathit{Av}(|\gamma_{\mathsf{Int}}[c, d]| - |\gamma_{\mathsf{Int}}([a, b])|) & \text{otherwise.} \end{cases}$$

Intuitively, $\delta^{\sim}_{\mathsf{Int}}$ *counts* how many more integer values one interval has compared to the other: if $\delta^{\sim}_{\mathsf{Int}}([a, b], [c, d]) = k$ for some $k \in \mathbb{N}$, then the interval $[c, d]$ contains exactly k more values than the interval $[a, b]$. The distance $\delta^{\sim}_{\mathsf{Int}}$ is clearly a $\sqsubseteq_{\mathsf{Int}}$-compatible pre-metric. For instance, $\delta^{\sim}_{\mathsf{Int}}([0, 0], [-1, 2]) = 3$ as the interval $[-1, 2]$ has 3 more elements than the singleton $[0, 0]$, namely: $-1, 1, 2$; $\delta^{\sim}_{\mathsf{Int}}([0, 10], [0, +\infty]) = \infty$ as $[0, +\infty]$ has an infinite number of more values than $[0, 10]$. This distance can be used to evaluate the results of interval-based program analysis by counting the number of spurious elements that one invariant includes with respect to another. ⧫

When an order-compatible pre-metric $\delta_{\mathcal{L}}$ also satisfies the axiom

$$(\textit{chain iff-identity}) \quad x \sqsubseteq_{\mathcal{L}} y \implies (\delta_{\mathcal{L}}(x, y) = 0 \implies x = y)$$

then it is precise enough to distinguish between comparable elements that are not equal—that is, it assigns distance zero only when the two elements coincide. For instance, $\delta^{=}_{\mathcal{L}}$ and $\delta^{\sim}_{\mathsf{Int}}$ satisfy (*chain iff-identity*), while $\delta^{\mathit{Vol}}_{\boldsymbol{N}}$ only when Vol calculates the exact volume.

3 GENERALIZING PARTIAL COMPLETENESS

Local completeness for an input $c \in \boldsymbol{C}$ is defined by the equality $\alpha(f(c)) = f^\sharp(\alpha(c))$ between the abstraction of the concrete computation and the (sound) abstract computation applied to the abstraction of the input (Definition 2.5). This property was first weakened by Campion et al. [2022, 2023] by allowing a *bounded* discrepancy between the concrete and abstract computations. This discrepancy is measured using an order-compatible pre-metric $\delta_{\mathcal{A}}$, and must not exceed a fixed bound $\varepsilon \in \mathbb{I}^{\infty}_{\geq 0}$. In [Campion et al. 2022, 2023], the authors define a sound abstract function $f^\sharp\colon \mathcal{A} \longrightarrow \mathcal{A}$ to be an ε-partial local complete approximation of $f\colon \boldsymbol{C} \longrightarrow \boldsymbol{C}$ at an input $c \in \boldsymbol{C}$ if the following inequality holds: $\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \varepsilon$.

In this section, we generalize the previously defined notion of ε-partial local completeness to a more flexible property in which the bound is no longer a fixed constant $\varepsilon \in \mathbb{I}^{\infty}_{\geq 0}$, but instead a *function* $\boldsymbol{e}\colon \boldsymbol{C} \longrightarrow \mathbb{I}^{\infty}_{\geq 0}$ whose output depends on the concrete input element. We refer to this function $\boldsymbol{e}$ as a *bounding function*. The idea is that the bounding function $\boldsymbol{e}$ maps the imprecision of an input abstraction to an upper bound on the imprecision of the corresponding abstract semantics applied to that input. The resulting generalized property, called *$\boldsymbol{e}$-partial (local) completeness*, supports reasoning at different levels of granularity: locally, over a single input or a set of inputs, and globally, over the entire concrete domain.

*Definition 3.1 (**$\boldsymbol{e}$-Partial (Local) completeness**).* Consider a GC $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$, a $\sqsubseteq_{\mathcal{A}}$-compatible pre-metric $\delta_{\mathcal{A}}$ and a bounding function $\boldsymbol{e}\colon \boldsymbol{C} \longrightarrow \mathbb{I}^{\infty}_{\geq 0}$. A sound abstract function $f^\sharp\colon \mathcal{A} \longrightarrow \mathcal{A}$ is said to be an *$\boldsymbol{e}$-partial local complete* approximation of $f\colon \boldsymbol{C} \longrightarrow \boldsymbol{C}$ at the set of inputs $S \subset \boldsymbol{C}$, denoted by the predicate $\mathbb{C}^{\boldsymbol{e}}_S(f, f^\sharp)$, when:

$$\mathbb{C}^{\boldsymbol{e}}_S(f, f^\sharp) \overset{def}{\Leftrightarrow} \forall c \in S.\, \delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}(c)$$

It is an *$\boldsymbol{e}$-partial complete* approximation whenever it holds for the entire concrete domain:

$$\mathbb{C}^{\boldsymbol{e}}(f, f^\sharp) \overset{def}{\Leftrightarrow} \forall c \in \boldsymbol{C}.\;\; \delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}(c) \qquad \blacksquare$$

Intuitively, when evaluating $\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c)))$, the $\sqsubseteq_{\mathcal{A}}$-compatible pre-metric $\delta_{\mathcal{A}}$ quantifies the *imprecision of interest* between $\alpha(f(c))$ and $f^\sharp(\alpha(c))$. For instance, when the abstract function is a sound abstract reachability semantics $f^\sharp = [\![\mathsf{P}]\!]^\sharp_{\mathcal{A}}$ approximating the (concrete) reachability semantics $f = [\![\mathsf{P}]\!]$ of a program $\mathsf{P} \in \mathsf{Prog}$, then $\delta_{\mathcal{A}}$ may measure the number of spurious elements (e.g., states) added by $[\![\mathsf{P}]\!]^\sharp_{\mathcal{A}}$, or the difference in volume of the numerical invariant generated by $[\![\mathsf{P}]\!]^\sharp_{\mathcal{A}}$ respect to $\alpha \circ [\![\mathsf{P}]\!]$.

When $\mathbb{C}^{\boldsymbol{e}}_S(f, f^\sharp)$ holds, the imprecision for inputs in S generated by $f^\sharp$ is guaranteed to be bounded by the bounding function $\boldsymbol{e}$. When the global property $\mathbb{C}^{\boldsymbol{e}}(f, f^\sharp)$ holds, $\boldsymbol{e}$ provides an upper bound on the imprecision introduced by $f^\sharp$ with respect to f for all inputs in $\boldsymbol{C}$. In other words, *$\boldsymbol{e}$ characterizes the input-dependent imprecision behavior of $f^\sharp$.*

Example 3.2. Consider the following program

$$\mathsf{ABS}: \;\; \textbf{if } x \geq 0 \textbf{ then } x := x \textbf{ else } x := -x$$

which computes the absolute value of an integer input. Let $[\![\mathsf{ABS}]\!]\colon \wp(\mathbb{Z}) \longrightarrow \wp(\mathbb{Z})$ be the standard (collecting) reachability semantics mapping a set of integers $S \in \wp(\mathbb{Z})$ to the union of outputs produced by running ABS on each $s \in S$, i.e., $[\![\mathsf{ABS}]\!]S = \bigcup_{s \in S} [\![\mathsf{ABS}]\!]s$. As abstract semantics, we consider the standard interval semantics over Int (Example 2.3), denoted $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}\colon \mathsf{Int} \longrightarrow \mathsf{Int}$. Our goal is to define a bounding function $\boldsymbol{e}$ proving that $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ is $\boldsymbol{e}$-partially complete with respect to the distance $\delta^{\sim}_{\mathsf{Int}}$ (Example 2.10). Note that the imprecision measured by $\delta^{\sim}_{\mathsf{Int}}$ could not be bounded by a constant function $\boldsymbol{e}(S) = \varepsilon$ for any $S \in \wp(\mathbb{Z})$, where $\varepsilon \in \mathbb{N}$. For example, consider the input $\{-2, 2\}$. The abstraction of the concrete semantics and the abstract semantics returns, respectively:

$$\alpha_{\mathsf{Int}}(\llbracket \mathsf{ABS} \rrbracket \{-2, 2\}) = \alpha(\{2\}) = [2, 2]$$

$$\llbracket \mathsf{ABS} \rrbracket^{\sharp}_{\mathsf{Int}} \alpha_{\mathsf{Int}}(\{-2, 2\}) = [0, 2]$$

thus $\delta^{\sim}_{\mathsf{Int}}([2, 2], [0, 2]) = 2$. More generally, for any symmetric pair $\{n_1, n_2\}$ with $n_2 \in \mathbb{N}$ and $n_1 = -n_2$, we obtain

$$\delta^{\sim}_{\mathsf{Int}}(\alpha_{\mathsf{Int}}(\llbracket \mathsf{ABS} \rrbracket \{n_1, n_2\}), \llbracket \mathsf{ABS} \rrbracket^{\sharp}_{\mathsf{Int}} \alpha_{\mathsf{Int}}(\{n_1, n_2\})) = n_2$$

This corresponds to a worst-case scenario: the concrete result is always a singleton n_2, while the abstract semantics includes 0 due to the guard, resulting in a spurious over-approximation of size n_2. To bound this imprecision, we define the following bounding function: $\boldsymbol{e}(S) = 0$ if $S = \varnothing$ or $max(S) < 0$, $\boldsymbol{e}(S) = max(S)$ otherwise. Thus the predicate $\mathbb{C}^{\boldsymbol{e}}(\llbracket \mathsf{ABS} \rrbracket, \llbracket \mathsf{ABS} \rrbracket^{\sharp}_{\mathsf{Int}})$ holds. ♦

From now on, we use red bold letters (e.g., $\boldsymbol{e}$, $\boldsymbol{u}$), to indicate bounding functions, and red bold numbers (e.g., $\boldsymbol{0}$, $\boldsymbol{1}$) to indicate their respective constant functions. For instance, $\boldsymbol{0} \overset{\text{def}}{=} \lambda x.0$ denote the constant function returning 0, $\boldsymbol{1} \overset{\text{def}}{=} \lambda x.1$ the one returning 1, and so on. The following proposition is a direct consequence of Definition 3.1.

Proposition 3.3. *The following statements hold:*

(i) If $\boldsymbol{u} \leq \boldsymbol{e}$ then: $\mathbb{C}^{\boldsymbol{u}}_S(f, f^{\sharp}) \Rightarrow \mathbb{C}^{\boldsymbol{e}}_S(f, f^{\sharp})$;

(ii) If $S \subseteq S'$ then: $\mathbb{C}^{\boldsymbol{e}}_{S'}(f, f^{\sharp}) \Rightarrow \mathbb{C}^{\boldsymbol{e}}_S(f, f^{\sharp})$;

(iii) If $\delta_{\mathcal{A}}$ satisfies (chain iff-identity), then: $\mathbb{C}^{\boldsymbol{0}}_S(f, f^{\sharp}) \Leftrightarrow \mathbb{C}_S(f, f^{\sharp})$. □

When $f^{\sharp}$ is $\boldsymbol{u}$-partial local complete, then any pointwise increase of the bounding function, i.e., $\boldsymbol{u} \leq \boldsymbol{e}$, ensures the validity of the predicate $\mathbb{C}^{\boldsymbol{e}}_S(f, f^{\sharp})$. Conversely, if $\mathbb{C}^{\boldsymbol{e}}_{S'}(f, f^{\sharp})$ holds and the input set is restricted to $S \subseteq S'$, then $f^{\sharp}$ is $\boldsymbol{e}$-partial local complete on S. The left-to-right implication of point (*iii*) always holds due to the (*if-identity*) axiom of a pre-metric. However, the reverse implication does not generally hold unless $\delta_{\mathcal{A}}$ satisfies (*chain iff-identity*). In this latter case, stating that $f^{\sharp}$ is $\boldsymbol{0}$-partial local complete on S becomes equivalent to stating that $f^{\sharp}$ is locally complete on S since, by (*chain iff-identity*) and $f^{\sharp}$ sound, a zero distance implies equality.

Naturally, one may ask how such a bounding function $\boldsymbol{e}$ can be determined in order to establish the validity of the predicate $\mathbb{C}^{\boldsymbol{e}}(f, f^{\sharp})$ or its local variant $\mathbb{C}^{\boldsymbol{e}}_S(f, f^{\sharp})$. This question is particularly crucial in the context of program analysis, where understanding how the imprecision varies depending on the input is fundamental for assessing the reliability and precision of the analysis results, identifying sources of imprecision and guiding the refinement or design of abstract domains and transfer functions. In Section 6 we will propose a dedicated proof system for deriving a bounding function inductively from the program syntax.

Before moving on to the next section, let us note that all four properties introduced in Definitions 2.5 and 3.1, and summarized in Table 1, rely on universal quantification—either over a subset of inputs ($\forall c \in S$, in the local case) or over the entire input space ($\forall c \in C$, in the global case). Starting from the next section, we will show that there are certain distinguished elements in the concrete domain $\langle C, \sqsubseteq_C \rangle$ that can characterize the overall precision and imprecision behavior of an abstract interpretation $f^{\sharp}$. This allows us to avoid checking the predicate over the full quantification range. These key elements are referred to as *generators*.

	Completeness	$\boldsymbol{e}$-Partial Completeness
Local	$\mathbb{C}_S(f, f^\sharp)$ $\Leftrightarrow$ $\forall c \in S.\, \alpha(f(c)) = f^\sharp(\alpha(c))$	$\mathbb{C}^{\boldsymbol{e}}_S(f, f^\sharp)$ $\Leftrightarrow$ $\forall c \in S.\, \delta_{\boldsymbol{\mathcal{A}}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}(c)$
Global	$\mathbb{C}(f, f^\sharp)$ $\Leftrightarrow$ $\alpha \circ f = f^\sharp \circ \alpha$	$\mathbb{C}^{\boldsymbol{e}}(f, f^\sharp)$ $\Leftrightarrow$ $\forall c \in \boldsymbol{C}.\, \delta_{\boldsymbol{\mathcal{A}}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}(c)$

Table 1. Completeness and partial completeness definitions.

4 GENERATORS OF AN ABSTRACT PROPERTY

In this section, we formally define the notion of generators of an abstract property. To achieve this, the minimum structure required in both the concrete and abstract domains is a partial order that models the amount of information (with larger properties exposing more information) and a GC between them that enables the existence of an abstraction function α. Thus, from now on, we assume the following three components:

(1) a poset $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle$ representing the concrete properties,

(2) a poset $\langle \boldsymbol{\mathcal{A}}, \sqsubseteq_{\boldsymbol{\mathcal{A}}} \rangle$ representing the abstract properties, and

(3) a GC $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \boldsymbol{\mathcal{A}}, \sqsubseteq_{\boldsymbol{\mathcal{A}}} \rangle$ between the two domains.

*Definition 4.1 (**Generator**).* The generator function $\mathscr{G} \colon \boldsymbol{\mathcal{A}} \longrightarrow \wp(\boldsymbol{C})$ is a map that associates to each abstract property $a \in \boldsymbol{\mathcal{A}}$ the set $\mathscr{G}(a) \in \wp(\boldsymbol{C})$ of its *generators*:

$$\mathscr{G}(a) \overset{\text{def}}{=} \left\{ g \in \boldsymbol{C} \;\middle|\; \begin{array}{l} (i)\;\; \alpha(g) = a \;\;\wedge \\ (ii)\;\; \forall c \in \boldsymbol{C}.\; c \sqsubset_{\boldsymbol{C}} g \;\Rightarrow\; \alpha(c) \sqsubset_{\boldsymbol{\mathcal{A}}} \alpha(g) \end{array} \right\} \qquad \blacksquare$$

Intuitively, for a concrete property $g \in \boldsymbol{C}$ to be a generator of the abstract property $a \in \boldsymbol{\mathcal{A}}$, g must be represented by a in the abstract domain through the abstraction function α (condition (i)). Additionally, g must contain the *minimal* information, in terms of the ordering $\sqsubseteq_{\boldsymbol{C}}$, necessary to be represented by a via α (condition (ii)). This means that, if we consider any other concrete property $c \in \boldsymbol{C}$ that contains less information than g according to the strict ordering $\sqsubset_{\boldsymbol{C}}$, then the abstract representation of c is strictly lower than the abstract representation of g in the abstract domain according to the abstract ordering $\sqsubset_{\boldsymbol{\mathcal{A}}}$. Note that α is order-preserving for the partial ordering $\sqsubseteq_{\boldsymbol{C}}$ but not for its strict version $\sqsubset_{\boldsymbol{C}}$. For instance, if we consider the abstraction α_{Int} defined in Example 2.3, the two sets $\{1, 3\} \subset \{1, 2, 3\}$ do not maintain the strict subset ordering after applying α_{Int} because $\alpha_{\mathsf{Int}}(\{1, 3\}) = [1, 3] \not\sqsubset_{\mathsf{Int}} [1, 3] = \alpha_{\mathsf{Int}}(\{1, 2, 3\})$.

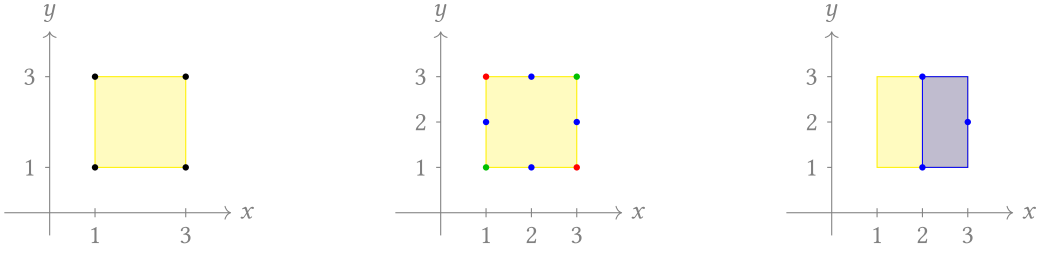
When $a \in \boldsymbol{\mathcal{A}}$ has at least one generator, i.e., $\mathscr{G}(a) \neq \emptyset$, we say that a is *generable*. Many properties in the hierarchy of semantics given by Cousot [2002] admit generators and are therefore generable.

Example 4.2 (Abstraction of a trace property into a reachability property). The trace to reachability property abstraction $\alpha_{\mathcal{I}}$ abstracts a trace property $P \in \wp(\Sigma^{+\infty})$ to an invariant $S \in \wp(\Sigma)$ collecting the possible values of the variables on the last state of each finite trace. The GC

$$\langle \wp(\Sigma^{+\infty}), \subseteq \rangle \underset{\alpha_{\mathcal{I}}}{\overset{\gamma_{\mathcal{I}}}{\leftrightarrows}} \langle \wp(\Sigma), \subseteq \rangle$$

is defined as:

$$\alpha_{\mathcal{I}}(P) \overset{\text{def}}{=} \{\sigma_\omega \mid \sigma \in P \cap \Sigma^+\} \qquad \gamma_{\mathcal{I}}(S) \overset{\text{def}}{=} \{\sigma \mid \sigma_\omega \in S \vee \sigma \in \Sigma^\infty\}$$



(a) (b) (c)

Fig. 1. The square Q of Example 4.3.

Given a set of states $S \in \wp(\Sigma)$, the set of generators of S is

$$\mathscr{G}(S) = \{G \in \wp(\Sigma^+) \mid \forall \sigma \in G.\, \sigma_\omega \in S \;\wedge\; \forall s \in S.\, \exists! \sigma \in G.\, \sigma_\omega = s\}$$

where $\exists!$ is the unique existential quantification. That is, the generator $G \in \mathscr{G}(S)$ is a set of traces containing only one trace that ends with a final state in S, for all possible $s \in S$. Removing a trace σ' from G implies that there is a state $s' \in S$ not reachable from any $\sigma \in G$, namely $\alpha_{\mathcal{I}}(G \smallsetminus \{\sigma'\}) \subset \alpha_{\mathcal{I}}(G)$. Therefore, G satisfies condition (ii). Moreover, for every set of states $S \in \wp(\Sigma)$, S is generable and there is exactly one generator: $|\mathscr{G}(S)| = 1$. Note that, as opposed to the previous example where the generator is always unique, here $|\mathscr{G}(S)| \geq 1$, namely we can have more than one generator describing S (there could be more than one trace leading to a final state). ♦

Example 4.3 (Abstraction of a reachability property into an interval property). Let us consider $\Sigma = \mathbb{Z}$. The interval abstraction α_{Int}, defined in Example 2.3, gives rise to the GC $\langle \wp(\mathbb{Z}), \subseteq \rangle \underset{\alpha_{\mathsf{Int}}}{\overset{\gamma_{\mathsf{Int}}}{\leftrightarrows}} \langle \mathsf{Int}, \sqsubseteq_{\mathsf{Int}} \rangle$. Let $[a, b] \in \mathsf{Int}$ be a closed interval, namely $a, b \in \mathbb{Z}$ and $a \leq b$. Then the generator of $[a, b]$ is

$$\mathscr{G}([a, b]) = \{\{a, b\}\}$$

In fact, a and b represent the minimal information which is necessary to describe the interval $[a, b]$. A similar reasoning can be applied to hyperrectangles Int^n over n-dimensions. For instance, let us consider $\Sigma = \mathbb{Z}^2$ and the GC $\langle \wp(\mathbb{Z}^2), \subseteq \rangle \underset{\alpha_{\mathsf{Int}}}{\overset{\gamma_{\mathsf{Int}}}{\leftrightarrows}} \langle \mathsf{Int}^2, \sqsubseteq_{\mathsf{Int}} \rangle$, where α_{Int}, γ_{Int} and $\sqsubseteq_{\mathsf{Int}}$ are extended componentwise. The square $Q \in \mathsf{Int}^2$ depicted in Figure 1a has four angles located at the coordinates: $(1, 1), (1, 3), (3, 1), (3, 3)$. The generators of Q are (see Figure 1b):

$$\begin{aligned} \mathscr{G}(Q) = \{\, &\{(1,1), (3,3)\}, \{(1,3), (3,1)\}, \{(1,2), (2,3), (3,2), (2,1)\}, \{(1,1), (1,3), (2,3)\}, \\ &\{(3,1), (3,3), (1,2)\}, \{(3,3), (1,3), (2,1)\}, \{(1,3), (1,1), (3,2)\} \,\} \end{aligned}$$

For every generator $G \in \mathscr{G}(Q)$, removing a coordinate would generate a new rectangle. Consider $\{(1, 2), (2, 3), (3, 2), (2, 1)\}$. Then, by removing the coordinate $(1, 2)$, $\alpha_{\mathsf{Int}}(\{(2, 3), (3, 2), (2, 1)\})$ corresponds to the blue rectangle in Figure 1c, which is clearly strictly included in $\alpha_{\mathsf{Int}}(Q)$:

$$\alpha_{\mathsf{Int}}(\{(2,3), (3,2), (2,1)\}) \sqsubset_{\mathsf{Int}} \alpha_{\mathsf{Int}}(\{(1,2), (2,3), (3,2), (2,1)\}) = \alpha_{\mathsf{Int}}(Q) \qquad ♦$$

The following proposition outlines some basic properties that directly derive from Definition 2.1 of GC and Definition 4.1 of generators.

Proposition 4.4. *The following statements hold:*

(i) $\bot_C \in \boldsymbol{C} \;\Rightarrow\; \mathscr{G}(\alpha(\bot_C)) = \{\bot_C\}$;

(ii) $a \in \boldsymbol{\mathcal{A}}$ *generable* $\;\Rightarrow\; \alpha\gamma(a) = a$;

(iii) $\exists! c \in \boldsymbol{C}.\, \alpha(c) = a \;\Rightarrow\; \mathscr{G}(a) = \{c\}$.

The first point (i) states that the generators of the abstraction of the bottom element of $\boldsymbol{C}$ (in case it exists) is exactly $\bot_{C}$ itself as, by definition, $\bot_{C}$ is the minimal element of $\boldsymbol{C}$. The next point (ii) states that for all generable abstract properties (i.e., $\mathscr{G}(a) \neq \varnothing$) going back to the concrete domain through γ and then abstracting again the result through α, gives as result the same property. This means that for all generable abstract properties a there is at least one concrete property c such that $\alpha(c) = a$. Finally, (iii) simply specifies that when there is only one concrete property represented by a in the abstract domain then for sure that concrete property is a generator of a.

However, not all abstract properties admit generators. This means that, in the abstract domain $\mathcal{A}$, there could exist *non-generable* abstract properties, i.e., those for which $\mathscr{G}(a) = \varnothing$. There are two possible reasons for this. The first one is when condition (i) of Definition 4.1 cannot be satisfied. This happens when the GC is not a GI, i.e. α is not surjective. In this case, all points $a \in \mathcal{A}$ which are not the abstraction of any property of $\boldsymbol{C}$, are not generable. The second reason occurs when condition (ii) could not be satisfied, namely, it is not possible to find a minimal quantity of information in $\boldsymbol{C}$ that generates a through α. For example, in the interval abstraction $\langle \wp(\mathbb{Z}), \subseteq \rangle \underset{\alpha_{\mathsf{Int}}}{\overset{\gamma_{\mathsf{Int}}}{\longleftrightarrows}} \langle \mathsf{Int}, \sqsubseteq_{\mathsf{Int}} \rangle$ all infinite intervals have no generators. These are all the intervals of the form $[a, +\infty], [-\infty, b], [-\infty, +\infty] \in \mathsf{Int}$ where $a, b \in \mathbb{Z}$. This is because there is no minimal elements in $\wp(\mathbb{Z})$ able to generate, e.g., $[0, +\infty]$.

When studying an abstract property $a \in \mathcal{A}$, one might be interested only in the generators g of a that are approximated by a concrete property $c \in \boldsymbol{C}$ according to $\sqsubseteq_{C}$, that is those generators for which $g \sqsubseteq_{C} c$ holds. When $\alpha(c) = a$, these generators identify the minimal elements of c that are responsible of the fact that c gets abstracted in a.

*Definition 4.5 (**Generators approximated by a concrete property**).* The set of generators of $a \in \mathcal{A}$ that *are approximated* by a concrete property $c \in \boldsymbol{C}$ with respect to $\sqsubseteq_{C}$, written $\mathscr{G}(a)_{\sqsubseteq c}$, is defined as:

$$\mathscr{G}(a)_{\sqsubseteq c} \overset{\text{def}}{=} \{g \in \mathscr{G}(a) \mid g \sqsubseteq_{C} c\} \qquad \blacksquare$$

Example 4.6. The generators of Q (Figure 1b) contained in the set of coordinates $\{(1,1), (1,3), (3,3)\}$ are

$$\mathscr{G}(Q)_{\sqsubseteq\{(1,1),(1,3),(3,3)\}} = \{\{(1,1), (3,3)\}\}$$

because $\{(1,1), (3,3)\}$ is a generator of Q and it holds that $\{(1,1), (3,3)\} \subseteq \{(1,1), (1,3), (3,3)\}$. ♦

We will extensively use this last definition in the following sections.

5 GENERATORS AND PARTIAL LOCAL COMPLETENESS

In this section, we show that a proof of local completeness over a set of concrete properties whose abstractions are generable, can be reduced to proving local completeness on their generators. The elegance of these results lies in the fact that generators not only characterize the precision (i.e. completeness) of an abstract interpretation but also capture its *bounded imprecision* through a bounding function. Indeed, we show that the result also holds for the $\boldsymbol{e}$-partial local completeness property. We will consider minimal assumptions about the concrete and abstract operators, as well as the underlying structure of the concrete and abstract domains. To this end, building on the elements introduced at the beginning of Section 4, we now additionally consider:

(4) a concrete order-preserving function $f\colon \boldsymbol{C} \to \boldsymbol{C}$ over the concrete domain,

(5) an abstract (not necessarily order-preserving) function $f^{\sharp}\colon \mathcal{A} \to \mathcal{A}$ over the abstract domain which is a *sound* over-approximations of f, and

(6) a $\sqsubseteq_{\mathcal{A}}$-compatible pre-metric $\delta_{\mathcal{A}}\colon \mathcal{A} \times \mathcal{A} \to \mathbb{I}^{\infty}_{\geq 0}$ measuring the imprecision of interest, together with a bounding function $\boldsymbol{e}\colon \boldsymbol{C} \to \mathbb{I}^{\infty}_{\geq 0}$.

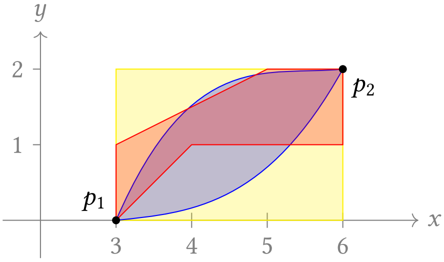


Fig. 2. Input abstractions of Example 5.3.

Let us begin with the first result, which shows what it means to prove ($\boldsymbol{e}$-partial) local completeness directly on generators. Given $l, r \in \boldsymbol{C}$ such that $l \sqsubseteq_C r$, we denote with the interval $[l, r]$ the set $[l, r] \overset{\text{def}}{=} \{c \in \boldsymbol{C} \mid l \sqsubseteq_C c \sqsubseteq_C r\}$ of all concrete properties in between l and r.

Theorem 5.1. *Let $a \in \mathcal{A}$ be a generable abstract property and $g \in \mathcal{G}(a)$ be a generator of a. Then:*

$$\mathbb{C}_g^{\boldsymbol{e}}(f, f^\sharp) \;\Rightarrow\; \mathbb{C}_{[g,\gamma(a)]}^{\tilde{\boldsymbol{e}}}(f, f^\sharp)$$

where $\tilde{\boldsymbol{e}}(c) \overset{\text{def}}{=} \boldsymbol{e}(g)$ if $c \in [g, \gamma(a)]$, $\tilde{\boldsymbol{e}}(c) \overset{\text{def}}{=} \boldsymbol{e}(c)$ otherwise. □

In simpler terms, if $f^\sharp(\alpha(g))$ has an imprecision bounded by $\boldsymbol{e}(g)$ at the generator g according to $\delta_{\mathcal{A}}$, then this bound is also an upper bound for the imprecision of $f^\sharp$ at *all inputs within the chain* $[g, \gamma(a)]$. As a corollary result, proving the local completeness of $f^\sharp$ at the generator g *is equivalent* to proving that $f^\sharp$ is actually precise at all inputs in $[g, \gamma(a)]$.

Corollary 5.2. $\mathbb{C}_g(f, f^\sharp) \;\Leftrightarrow\; \mathbb{C}_{[g,\gamma(a)]}(f, f^\sharp)$ □

Example 5.3. Consider the following while-loop program $\mathtt{W} \in$ Prog:

$$\textbf{while } x > y \textbf{ do } \{\ y := y + x\ ;\ x := x - 1\ \}$$

Suppose we want to analyze the interval invariant (over $\mathbb{R}$) at the end of $\mathtt{W}$ by using the standard interval analysis $[\![\mathtt{W}]\!]^\sharp_{\mathsf{Int}^2} : \mathsf{Int}^2 \longrightarrow \mathsf{Int}^2$ defined in [Cousot and Cousot 1976] (also in [Miné 2017, Chapter 4.5]), on the input $S = \{(3, 0), (6, 2)\}$ (points p_1 and p_2 in Figure 2). The abstraction of S through α_{Int}, gives rise to the rectangle $\alpha_{\mathsf{Int}}(S) = ([3, 6], [0, 2])$ (the yellow rectangle in Figure 2). We measure the imprecision of our abstract interpreter by using the $\sqsubseteq_{\mathsf{Int}}$-compatible pre-metric $\delta^{Vol}_{\mathsf{Int}^2}$ defined in Example 2.9. In particular, since the objects in Int^2 are 2-dimensional, $\delta^{Vol}_{\mathsf{Int}^2}((x_1, y_1), (x_2, y_2))$ calculates precisely (the absolute value of) the difference between the areas of the rectangle $(x_2, y_2) \in \mathsf{Int}^2$ and the rectangle $(x_1, y_1) \in \mathsf{Int}^2$, where $x_1, x_2, y_1, y_2 \in \mathsf{Int}$. The interval abstraction of the reachability semantics $[\![\mathtt{W}]\!]$ and the output of the interval analysis $[\![\mathtt{W}]\!]^\sharp_{\mathsf{Int}^2}$ are:

$$\alpha_{\mathsf{Int}}([\![\mathtt{W}]\!]S) = \alpha_{\mathsf{Int}}(\{(2, 3), (5, 8)\}) = ([2, 5], [3, 8])$$

$$[\![\mathtt{W}]\!]^\sharp_{\mathsf{Int}^2}\alpha_{\mathsf{Int}}(S) = [\![\mathtt{W}]\!]^\sharp_{\mathsf{Int}^2}([3, 6], [0, 2]) = ([0, 6], [0, 11])$$

Their distance is:

$$\delta^{Vol}_{\mathsf{Int}^2}(\alpha_{\mathsf{Int}}([\![\mathtt{W}]\!]S), [\![\mathtt{W}]\!]^\sharp_{\mathsf{Int}^2}\alpha_{\mathsf{Int}}(S)) = \delta^{Vol}_{\mathsf{Int}^2}(([2, 5], [3, 8]), ([0, 6], [0, 11])) = 51$$

Suppose the (constant) bounding function to verify is 55. This means that $\mathbb{C}^{55}_S([\![\mathtt{W}]\!], [\![\mathtt{W}]\!]^\sharp_{\mathsf{Int}^2})$ holds. Moreover, since $S \in \mathcal{G}(\alpha(S))$, namely, the set of points $\{p_1, p_2\}$ is a generator of the rectangle $\alpha(\{p_1, p_2\})$, Theorem 5.1 guarantees that $[\![\mathtt{W}]\!]^\sharp_{\mathsf{Int}^2}$ will satisfy the same imprecision bound 55 when considering all the (concrete) sets of points in $\{I \subset \mathbb{R}^2 \mid S \subseteq I \subseteq \gamma_{\mathsf{Int}}\alpha_{\mathsf{Int}}(S)\}$, namely all the possible forms that are contained within the yellow rectangle of Figure 2 and that also contain the set S. ♦

It is worth noting that the implication $\mathbb{C}^{\boldsymbol{e}}_{[g,\gamma(a)]}(f, f^\sharp) \;\Rightarrow\; \mathbb{C}^{\boldsymbol{e}}_{g}(f, f^\sharp)$ is trivially valid by $g \in [g, \gamma(a)]$ and Proposition 3.3. However, deriving an upper bound of imprecision $\boldsymbol{e}(c)$ for some c in the chain $(g, \gamma(a)]$ where g is excluded, does not imply that $f^\sharp$ satisfies the same bound for the generator g. The following example illustrates this point.

Example 5.4. Let us consider the GC $\langle \wp(\mathbb{Z}), \subseteq\rangle \underset{\alpha_{\mathsf{Int}}}{\overset{\gamma_{\mathsf{Int}}}{\leftrightarrows}} \langle \mathsf{Int}, \sqsubseteq_{\mathsf{Int}}\rangle$ and, as abstract function, $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ of the program $\mathsf{ABS} \in \mathsf{Prog}$ of Example 3.2. Suppose we are interested in measuring the "spurious" points added by $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$, by employing the $\sqsubseteq_{\mathsf{Int}}$-compatible pre-metric $\delta^{\sim}_{\mathsf{Int}}$, defined in Example 2.10, and we are tolerating a bound of imprecision quantified by $\boldsymbol{e} = \mathbf{1}$ over the input $\{-3, -1, 2\}$. The abstract semantics $[\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}$ is $\mathbf{1}$-partial local complete at $\{-3, -1, 2\}$ since:

$$\alpha_{\mathsf{Int}}([\![\mathsf{ABS}]\!]\{-3, -1, 2\}) = [1, 3] \sqsubset_{\mathsf{Int}} [0, 3] = [\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}[-3, 2]$$

and $\delta^{\sim}_{\mathsf{Int}}([1, 3], [0, 3]) = 1$. Thus, the predicate $\mathbb{C}^{\mathbf{1}}_{\{-3,-1,2\}}([\![\mathsf{ABS}]\!], [\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}})$ holds. However, the predicate $\mathbb{C}^{\mathbf{1}}_{\{-3,2\}}([\![\mathsf{ABS}]\!], [\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}})$ does not hold on the generator $\{-3, 2\}$ because:

$$\alpha_{\mathsf{Int}}([\![\mathsf{ABS}]\!]\{-3, 2\}) = [2, 3] \sqsubset_{\mathsf{Int}} [0, 3] = [\![\mathsf{ABS}]\!]^\sharp_{\mathsf{Int}}[-3, 2]$$

and $\delta^{\sim}_{\mathsf{Int}}([2, 3], [0, 3]) = 2 > 1$. ♦

We now want to further generalize the result of Theorem 5.1. To this end, we define two sets of concrete properties denoted by C_{gen} , $\mathcal{G} \in \wp(\boldsymbol{C})$. The former is defined as

$$C_{gen} \overset{\text{def}}{=} \{c \in \boldsymbol{C} \mid \mathscr{G}(\alpha(c))_{\sqsubseteq c} \neq \varnothing\}$$

and corresponds to the set of all concrete properties that approximate at least one generator of $\alpha(c)$. The latter

$$\mathcal{G} \overset{\text{def}}{=} \bigcup_{a \in \mathcal{A}} \mathscr{G}(a)$$

is the set of all generators of generable abstract properties of $\mathcal{A}$. Clearly, $\mathcal{G} \subseteq C_{gen}$ since if $g \in \mathcal{G}$ then $g \in \mathscr{G}(\alpha(g))_{\sqsubseteq g}$, which means that $g \in C_{gen}$.

Example 5.5. Consider the GC $\langle \wp(\mathbb{Z}^n), \subseteq\rangle \underset{\alpha_{\mathsf{Int}}}{\overset{\gamma_{\mathsf{Int}}}{\leftrightarrows}} \langle \mathsf{Int}^n, \sqsubseteq_{\mathsf{Int}}\rangle$. We have seen that all the generable elements of Int are the closed intervals $[a, b]$ with $a, b \in \mathbb{Z}$. A similar reasoning applies over n-dimensional intervals Int^n. This implies that $C_{gen} \subset \mathbb{Z}^n$ is the set of all closed sets, namely, for all $S \in C_{gen}$: $max(S(i)) \in \mathbb{Z}$ and $min(S(i)) \in \mathbb{Z}$ both exist with $i \in [1, n]$, and $\alpha_{\mathsf{Int}}(S) \in \mathsf{Int}^n$ represents an n-dimensional closed hyperrectangle. ♦

Theorem 5.6. *Suppose* $\delta_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathbb{N}^\infty$, *then*

$$\mathbb{C}^{\boldsymbol{e}}_{\mathcal{G}}(f, f^\sharp) \;\Rightarrow\; \mathbb{C}^{\hat{\boldsymbol{e}}}_{C_{gen}}(f, f^\sharp)$$

where $\hat{\boldsymbol{e}}(c) \overset{\text{def}}{=} min(\{\boldsymbol{e}(g) \mid g \in \mathscr{G}(\alpha(c))_{\sqsubseteq c}\})$ *if* $c \in C_{gen} \smallsetminus \mathcal{G}$, $\hat{\boldsymbol{e}}(c) \overset{\text{def}}{=} \boldsymbol{e}(c)$ *otherwise.* □

Note that Theorem 5.6 generalizes the result of Theorem 5.1, extending it from chains to the set of properties C_{gen}. Specifically, Theorem 5.6 shows that a proof of $\boldsymbol{e}$-partial local completeness over all possible generators $\mathcal{G}$ of the GC can be lifted to a proof of $\hat{\boldsymbol{e}}$-partial completeness over all concrete properties in C_{gen}, by redefining the bounding function $\boldsymbol{e}$ as a new function $\hat{\boldsymbol{e}}$. This new bounding function assigns to each $c \in C_{gen}$ the *minimum* imprecision bound among all generators approximated by c. For distances defined as $\delta_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathbb{R}^{\infty}_{\geq 0}$, a minimum may not exist. In such cases, $\hat{\boldsymbol{e}}$ can be defined by selecting an arbitrary bound from those associated with the approximated generators. This value still constitutes a valid upper bound on the imprecision over the set C_{gen}.

The local completeness property is a special case of Theorem 5.6.

Corollary 5.7. $\mathbb{C}_{\mathcal{G}}(f, f^\sharp) \iff \mathbb{C}_{C_{gen}}(f, f^\sharp)$ □

Theorem 5.6 and Corollary 5.7 both capture the key insight that analyzing the (im)precision of $f^\sharp$ over the set of inputs C_{gen}—those that approximate at least one generator—can be effectively reduced to analyzing the (im)precision over the generators $\mathcal{G}$ of the GC.

Example 5.8. Consider the program

$$\mathsf{ReLU}: \ \textbf{if } x < 0 \textbf{ then } x := 0 \textbf{ else } x := x$$

which corresponds to the well-known ReLU activation function used in neural networks to suppress negative inputs [Nair and Hinton 2010]. We consider the reachability semantics $[\![\mathsf{ReLU}]\!]$ and its sound approximation via $[\![\mathsf{ReLU}]\!]^\sharp_{\mathsf{Int}}$ over the interval abstract domain Int. We evaluate the imprecision introduced by the abstract interpretation using the pre-metric $\delta^{\sim}_{\mathsf{Int}}$. If we measure the imprecision for input sets of the form $\{i, j\}$, where $i, j \in \mathbb{Z}$, we observe:

$$\delta^{\sim}_{\mathsf{Int}}(\alpha_{\mathsf{Int}}([\![\mathsf{ReLU}]\!]\{i, j\}), [\![\mathsf{ReLU}]\!]^\sharp_{\mathsf{Int}}\alpha_{\mathsf{Int}}(\{i, j\})) = 0$$

Since each set $\{i, j\}$ is a generator of the closed interval $\alpha_{\mathsf{Int}}(\{i, j\})$, it follows that $[\![\mathsf{ReLU}]\!]^\sharp_{\mathsf{Int}}$ is $\mathbf{0}$-partial local complete at all inputs in $\mathcal{G}$. By Theorem 5.6, this implies that $\mathbf{0}$ is also a valid bounding function over the entire set C_{gen}, i.e., the predicate $\mathbb{C}^{\mathbf{0}}_{C_{gen}}([\![\mathsf{ReLU}]\!], [\![\mathsf{ReLU}]\!]^\sharp_{\mathsf{Int}})$ holds. Furthermore, since $\delta^{\sim}_{\mathsf{Int}}$ satisfies (*chain iff-identity*), we can invoke point (iii) of Proposition 3.3 to conclude that $[\![\mathsf{ReLU}]\!]^\sharp_{\mathsf{Int}}$ is also locally complete at all inputs in C_{gen}, that is $\mathbb{C}_{C_{gen}}([\![\mathsf{ReLU}]\!], [\![\mathsf{ReLU}]\!]^\sharp_{\mathsf{Int}})$ holds. ♦

6 A LOGIC FOR PROPAGATING ERROR BOUNDS

In this section, building on the notion of generators and the results from the previous section, we introduce a program logic designed to soundly establish the *worst-case imprecision of an abstract semantics with respect to a concrete semantics, across all inputs in the chain* $[g, \gamma(a)]$ between a generator g of an abstract property $a \in \mathcal{A}$. The proof system derives a bounding function $\boldsymbol{e}$ inductively from the program syntax: it starts from base cases (no-ops, assignments, Boolean guards), and then *propagates and updates* the bound *compositionally* through the program structure. In this sense, we are performing an *analysis of the analyzer* itself. The proof system will be introduced in Section 6.4. In Section 6.1, we fix the programming language and its semantics. We briefly recall the (in)correctness triples in Section 6.2. Section 6.3 introduces the notion of ω*-continuity*, which plays a fundamental role in propagating a bounding function when composing programs.

Before proceeding to the syntax of our programming language, let us remark that the results presented in Section 5 apply to any distance $\delta_{\mathcal{A}}$ satisfying Definition 2.7 of order-compatible pre-metric. This definition is based on a minimal and weak set of axioms sufficient to give meaning to the measurement of imprecision between the abstraction of a concrete operator and its sound abstract interpretation. However, the axiom (*chain-order*) alone does not provide guidance on *how* to compute an upper bound $\boldsymbol{e}$ between elements of a chain. To address this limitation, we introduce a stricter class of pre-metrics, which we call *strong pre-metrics.*

*Definition 6.1 (**Strong pre-metric**).* A $\sqsubseteq_{\mathcal{L}}$-compatible pre-metric $\delta_{\mathcal{L}}$ is said to be *strong* when the following auxiliary axioms hold $\forall x, y, z \in \mathcal{L}$:

$$(\textit{chain iff-identity}) \ \ x \sqsubseteq_{\mathcal{L}} y \ \Rightarrow \ (\delta_{\mathcal{L}}(x, y) = 0 \ \Rightarrow \ x = y)\ ;$$

$$(\textit{triangle-inequality})\ \ \delta_{\mathcal{L}}(x,z) \le \delta_{\mathcal{L}}(x,y) + \delta_{\mathcal{L}}(y,z)\ . \qquad \blacksquare$$

For example, the distance $\delta_{\mathcal{N}}^{Vol}$ introduced in Example 2.9 qualifies as strong when Vol computes the exact volume. The same holds for the order-compatible pre-metrics $\delta_{\mathcal{L}}^{=}$ and $\delta_{\mathsf{Int}}^{\sim}$, defined in Examples 2.8 and 2.10, respectively. Conversely, if Vol over-approximates the actual volume, then $\delta_{\mathcal{N}}^{Vol}$ may violate both axioms of Definition 6.1. This also applies, in general, to order-compatible pre-metrics that approximate, rather than compute exactly, the distance between elements [Campion et al. 2023]. Note that, within the theory of metric spaces, a strong pre-metric does not qualify as a metric, as it may lack symmetry and the iff-identity over the entire domain. Instead, it corresponds to a form of weak quasi-metric [Wilson 1931], where the iff-identity is required only along chains.

6.1 The Programming Language and its Semantics

In the following, the programming language Prog is assumed to be defined as follows:

$$\mathsf{Prog} \ni \mathsf{P} ::= \ \mathsf{c} \mid \mathsf{P};\mathsf{P} \mid \mathsf{P} \oplus \mathsf{P} \mid \mathsf{P}^{*}$$

which corresponds to the language of regular commands also used in, e.g., [Bruni et al. 2023; O'Hearn 2020]. The language Prog is general enough to cover deterministic imperative languages as well as nondeterministic and probabilistic programming. The term $\mathsf{P}_1;\mathsf{P}_2$ represents sequential composition, the term $\mathsf{P}_1 \oplus \mathsf{P}_2$ represents a nondeterministic choice command, and the term P^{*} represents the Kleene iteration of P, where P can be executed 0 or any finite number of times.

The language Prog is parametric on the syntax of basic commands $\mathsf{c} \in \mathsf{BCom}$ which can be instantiated with different kinds of instructions such as (deterministic or nondeterministic or parallel) assignments, (Boolean) guards or assumptions, etc. In the examples we will consider standard deterministic basic commands used in while programs, i.e., no-op, assignments and Boolean guards: $\mathsf{BCom} \ni \mathsf{c} ::= \textbf{skip} \mid x := a \mid \mathsf{b}?$ where a ranges over arithmetic expressions on integer values in $\mathbb{Z}$, variables $x \in Var$, and b ranges over Boolean expressions. A deterministic imperative while language can be defined using guarded branching and loop commands as syntactic sugar as follows [Kozen 1997]:

$$\textbf{if}\ \mathsf{b}\ \textbf{then}\ \mathsf{c}_1\ \textbf{else}\ \mathsf{c}_2 \overset{\text{def}}{=} (\mathsf{b}?;\mathsf{c}_1) \oplus (\neg\mathsf{b}?;\mathsf{c}_2)$$

$$\textbf{while}\ \mathsf{b}\ \textbf{do}\ \mathsf{c} \overset{\text{def}}{=} (\mathsf{b}?;\mathsf{c})^{*};\neg\mathsf{b}$$

We assume that basic commands have a semantics $(\!|\cdot|\!)_{\mathcal{L}} \colon \mathsf{BCom} \to \mathcal{L} \to \mathcal{L}$ on a complete lattice $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}}, \sqcup_{\mathcal{L}}, \sqcap_{\mathcal{L}}, \bot_{\mathcal{L}}, \top_{\mathcal{L}} \rangle$ such that, for all $\mathsf{c} \in \mathsf{BCom}$, $(\!|\mathsf{c}|\!)_{\mathcal{L}} \colon \mathcal{L} \to \mathcal{L}$ is order-preserving. $[\![\cdot]\!]_{\mathcal{L}} \colon \mathsf{Prog} \to \mathcal{L} \to \mathcal{L}$ is the semantics of Prog on the complete lattice $\mathcal{L}$ and it is inductively defined as follows:

$$[\![\mathsf{c}]\!]_{\mathcal{L}} l \overset{\text{def}}{=} (\!|\mathsf{c}|\!)_{\mathcal{L}} l \qquad\qquad [\![\mathsf{P}_1 \oplus \mathsf{P}_2]\!]_{\mathcal{L}} l \overset{\text{def}}{=} [\![\mathsf{P}_1]\!]_{\mathcal{L}} l \sqcup_{\mathcal{L}} [\![\mathsf{P}_2]\!]_{\mathcal{L}} l$$

$$[\![\mathsf{P}_1;\mathsf{P}_2]\!]_{\mathcal{L}} l \overset{\text{def}}{=} [\![\mathsf{P}_2]\!]_{\mathcal{L}} [\![\mathsf{P}_1]\!]_{\mathcal{L}} l \qquad\qquad [\![\mathsf{P}^{*}]\!]_{\mathcal{L}} l \overset{\text{def}}{=} \bigsqcup \{ [\![\mathsf{P}]\!]_{\mathcal{L}}^{n} l \mid n \in \mathbb{N} \}$$

It is easy to check, by structural induction, that the semantics above is order-preserving for $\sqsubseteq_{\mathcal{L}}$. In particular, given a GC $\langle C, \sqsubseteq_C \rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$ where both the concrete and abstract domains are complete lattices, it turns out that $[\![\mathsf{P}]\!]_C \circ \gamma \sqsubseteq_C \gamma \circ [\![\mathsf{P}]\!]_{\mathcal{A}}$ holds, namely the soundness of the abstract semantics $[\![\mathsf{P}]\!]_{\mathcal{A}}$ with respect to the concrete one $[\![\mathsf{P}]\!]_C$, provided that $(\!|\mathsf{c}|\!)_C \circ \gamma \sqsubseteq_C \gamma \circ (\!|\mathsf{c}|\!)_{\mathcal{A}}$ holds, namely the soundness on the basic commands. For instance, the concrete domain could be the complete lattice $\langle \wp(\mathbb{Z}^n), \subseteq, \cup, \cap, \varnothing, \mathbb{Z}^n \rangle$ of n-tuples of integers representing the program states of a program with n variables, and $[\![\mathsf{P}]\!]_{\wp(\mathbb{Z}^n)}$ (simply denoted as $[\![\mathsf{P}]\!]$ in the previous and following examples) is the standard collecting reachability semantics, where $(\!|\mathsf{b}?|\!) S$ filters all the program states in $S \in \wp(\mathbb{Z}^n)$ that make b true.

6.2 Correctness and Incorrectness Triples

We briefly recall here two fundamental notions that will be used in our logic for propagating the imprecision bound: *Hoare correctness* and *O'Hearn incorrectness* triples.

A *correctness triple* is the central feature of the Hoare logic [Hoare 1969] for proving partial correctness of programs. Given a formal description of a program's behavior through the semantics $[\![\mathsf{P}]\!]_{\mathcal{L}} \colon \mathcal{L} \to \mathcal{L}$, a correctness triple is denoted by $\{pre\}\mathsf{P}\{post\}_{\mathcal{L}}$, where $\mathsf{P} \in \mathsf{Prog}$ is a program and $pre, post \in \mathcal{L}$ are the pre- and post-conditions, respectively. Formally, the validity of $\{pre\}\mathsf{P}\{post\}_{\mathcal{L}}$ is defined by the over-approximation condition:

$$\{pre\}\mathsf{P}\{post\}_{\mathcal{L}} \overset{def}{\Leftrightarrow} [\![\mathsf{P}]\!]_{\mathcal{L}} pre \sqsubseteq_{\mathcal{L}} post$$

In other words, the behavior of the program P with input pre, formalized by the semantics $[\![\cdot]\!]_{\mathcal{L}}$, satisfies the post-condition $post$.

Conversely, an *incorrectness triple* is the central feature of the O'Hearn logic [O'Hearn 2020] for proving program incorrectness, and it is denoted by $[pre]\mathsf{P}[post]_{\mathcal{L}}$. Formally, the validity of $[pre]\mathsf{P}[post]_{\mathcal{L}}$ is defined by the under-approximation condition:

$$[pre]\mathsf{P}[post]_{\mathcal{L}} \overset{def}{\Leftrightarrow} post \sqsubseteq_{\mathcal{L}} [\![\mathsf{P}]\!]_{\mathcal{L}} pre$$

In other words, the post-condition $post$ represents an under-approximation of the behavior of the program P with input pre. O'Hearn [2020] originally designed the program logic for bug detection: if $post$ describes error states and the triple $[pre]\mathsf{P}[post]_{\mathcal{L}}$ holds, then any error state appearing in the post-condition is guaranteed to be reachable from some input state satisfying the pre-condition.

In Section 6.4, the combination of Hoare correctness and O'Hearn incorrectness triples plays a key role in modeling rules (**Seq**) and (**Iterate**) of the proposed program logic for propagating error bounds.

6.3 ω-Continuity

The problem is to establish the validity of the predicate $\mathbb{C}_g^{\boldsymbol{e}}([\![\mathsf{P}_1;\mathsf{P}_2]\!]_{\mathcal{C}}, [\![\mathsf{P}_1;\mathsf{P}_2]\!]_{\mathcal{A}})$ for the sequential composition of two programs $\mathsf{P}_1, \mathsf{P}_2 \in \mathsf{Prog}$, given that both $\mathbb{C}_g^{\boldsymbol{e}_1}([\![\mathsf{P}_1]\!]_{\mathcal{C}}, [\![\mathsf{P}_1]\!]_{\mathcal{A}})$ and $\mathbb{C}_h^{\boldsymbol{e}_2}([\![\mathsf{P}_2]\!]_{\mathcal{C}}, [\![\mathsf{P}_2]\!]_{\mathcal{A}})$ hold for generators $g, h \in \boldsymbol{C}$. Our goal is to derive the bounding function $\boldsymbol{e}$ for the composition $\mathsf{P}_1;\mathsf{P}_2$ in terms of the individual bounding functions $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. We show that if the *abstract semantics* $[\![\mathsf{P}_2]\!]_{\mathcal{A}}$ of the second program P_2 satisfies a form of *quantitative* continuity, captured by a function ω, then the bounding function for the composition $\mathsf{P}_1;\mathsf{P}_2$ can be expressed as *the sum* between $\omega \circ \boldsymbol{e}_1$ and $\boldsymbol{e}_2$. We refer to this continuity property as *ω-continuity*.

*Definition 6.2 (ω-**Continuity**).* Let $f \colon \mathcal{L} \to \mathcal{L}$ be a function on a poset $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}} \rangle$, and $\delta_{\mathcal{L}}$ a strong $\sqsubseteq_{\mathcal{L}}$-compatible pre-metric. Let $\omega \colon \mathbb{R}_{\geq 0}^{\infty} \to \mathbb{R}_{\geq 0}^{\infty}$ be an order-preserving function satisfying $\omega(0) = 0$ which, from now on, will be referred to as the *modulus of continuity*. The function f is *ω-continuous* when the following condition holds:

$$\forall l_1, l_2 \in \mathcal{L}.\, \delta_{\mathcal{L}}(f(l_1), f(l_2)) \leq \omega(\delta_{\mathcal{L}}(l_1, l_2)) \qquad \blacksquare$$

For a function f that satisfies ω-continuity, applying ω to the distance between any pair of inputs always yields an upper bound on the distance between the corresponding outputs of f. Note that if ω_∞ is defined as $\omega_\infty(t) \overset{\text{def}}{=} \infty$ for all $t > 0$, then *any* function f is trivially ω_∞-continuous. Thus, ω_∞ represents the weakest (i.e., least informative) modulus of continuity.

It is worth remarking that Definition 6.2 is different from the standard notion of uniform continuity in calculus where only a *finite* modulus of continuity is allowed for every finite distance, namely $\omega(t) \neq \infty$ for all $t \in \mathbb{R}_{\geq 0}$ (see, e.g., [Rudin 1976]). In our setting, Definition 6.2 allows *every* function to admit an ω such that f is ω-continuous—including those that are not uniformly

$$\frac{[\![\mathsf{c}]\!]_{\mathcal{L}}\ \omega\text{-continuous}}{\omega\text{-}\textsc{Cont}(\mathsf{c})_{\mathcal{L}}}\ (\textbf{Base}_{\omega}) \qquad \frac{\omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}} \quad \omega \leq \omega'}{\omega'\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}}\ (\textbf{Weaken}_{\omega})$$

$$\frac{\omega_1\text{-}\textsc{Cont}(\mathsf{P}_1)_{\mathcal{L}} \quad \omega_2\text{-}\textsc{Cont}(\mathsf{P}_2)_{\mathcal{L}}}{\omega_2 \circ \omega_1\text{-}\textsc{Cont}(\mathsf{P}_1;\mathsf{P}_2)_{\mathcal{L}}}\ (\textbf{Seq}_{\omega}) \qquad \frac{\omega_1\text{-}\textsc{Cont}(\mathsf{P}_1)_{\mathcal{L}} \quad \omega_2\text{-}\textsc{Cont}(\mathsf{P}_2)_{\mathcal{L}}}{\omega_{\sqcup}\text{-}\textsc{Cont}(\mathsf{P}_1 \oplus \mathsf{P}_2)_{\mathcal{L}}}\ (\textbf{Join}_{\omega})$$

$$\frac{\omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}}{\omega_{*}\text{-}\textsc{Cont}(\mathsf{P}^{*})_{\mathcal{L}}}\ (\textbf{Iterate}_{\omega})$$

Fig. 3. A proof system for deriving a modulus of continuity of programs.

continuous or even continuous—thanks to the presence of ω_∞. Although this may seem unusual, our Definition 6.2 of ω-continuity is designed to model functions f representing any *abstract interpreter*, which are well known not to be uniformly continuous due to their intrinsic approximation process (e.g., through the use of widening operators to ensure termination [Cousot and Cousot 1977]).

Given a program $\mathsf{P} \in \mathsf{Prog}$ and the semantics $[\![\cdot]\!]_{\mathcal{L}}$ defined in Section 6.1, a modulus of continuity for $[\![\mathsf{P}]\!]_{\mathcal{L}}$ can be derived inductively from the syntax of P, as shown in Figure 3. The predicate ω-$\textsc{Cont}(\mathsf{P})_{\mathcal{L}}$ is defined as follows:

$$\omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}} \overset{def}{\Leftrightarrow} [\![\mathsf{P}]\!]_{\mathcal{L}}\text{is } \omega\text{-continuous.}$$

Rule $(\textbf{Base}_{\omega})$ provides the base cases for primitive commands. Once the ω moduli are derived for the base cases, they can be propagated inductively using the following rules.

$(\textbf{Weaken}_{\omega})$ allows to switch the modulus of continuity ω with a new function $\omega' \geq \omega$, where $\geq$ is assumed componentwise, which is still a modulus of continuity for P.

$(\textbf{Seq}_{\omega})$ formalizes that the modulus of continuity of a sequential composition is obtained by composing the moduli of its components, exactly as in calculus.

The rule $(\textbf{Join}_{\omega})$ is less straightforward and, as already observed by Campion et al. [2022], it requires a deeper analysis of both the underlying complete lattice $\mathcal{L}$ and the chosen strong pre-metric $\delta_{\mathcal{L}}$. The challenge arises from the fact that the resulting modulus function cannot, in general, be determined based only on the moduli associated with P_1 and P_2. Additional information is needed, specifically, a bound on the imprecision introduced by the lub operator $\sqcup_{\mathcal{L}}$ of the complete lattice $\mathcal{L}$ with respect to the distance $\delta_{\mathcal{L}}$. This additional information is captured by the function $\oplus_{\delta_{\mathcal{L}}}$, referred to as the *join-bound*, and defined as follows [Campion et al. 2022]:

*Definition 6.3 (**Join-bound**).* Given a complete lattice $\mathcal{L}$ and a strong $\sqsubseteq_{\mathcal{L}}$-compatible pre-metric $\delta_{\mathcal{L}}$, the function $\oplus_{\delta_{\mathcal{L}}} : \mathbb{R}^{\infty}_{\geq 0} \times \mathbb{R}^{\infty}_{\geq 0} \to \mathbb{R}^{\infty}_{\geq 0}$ is a *join-bound* if the following condition is satisfied for all $x, y, z, u \in \mathcal{L}$ and for all $\varepsilon, \beta \in \mathbb{R}^{\infty}_{\geq 0}$:

$$x \sqsubseteq_{\mathcal{L}} z \ \wedge\ y \sqsubseteq_{\mathcal{L}} u \ \wedge\ \delta_{\mathcal{L}}(x,z) \leq \varepsilon \ \wedge\ \delta_{\mathcal{L}}(y,u) \leq \beta \ \Rightarrow\ \delta_{\mathcal{L}}(x \sqcup_{\mathcal{L}} y, z \sqcup_{\mathcal{L}} u) \leq \oplus_{\delta_{\mathcal{L}}}(\varepsilon, \beta) \qquad \blacksquare$$

Every complete lattice equipped with a strong order-compatible pre-metric admits a join-bound: the constant function $\lambda x, y.\, \infty$ is the weakest such join-bound, although it provides no meaningful information about the possible bound on lub.

Example 6.4. Consider the complete lattice of one-dimensional intervals Int and the strong pre-metric $\delta^{\sim}_{\mathsf{Int}}$. In this setting, the addition operation $+$ (extended to also handle infinities) serves as a valid join-bound, namely $\oplus_{\delta^{\sim}_{\mathsf{Int}}} \overset{\text{def}}{=} +$. This is because the join operation merges each input interval into a single interval that contains both. Given intervals $i_1, i_2, j_1, j_2 \in \mathsf{Int}$ such that $i_1 \sqsubseteq_{\mathsf{Int}} i_2$ and $j_1 \sqsubseteq_{\mathsf{Int}} j_2$, if i_2 has ε more elements than i_1, and j_2 has β more elements than j_1, then the sum $\varepsilon + \beta$

provides an upper bound (though not necessarily optimal) on the number of elements introduced by $i_2 \sqcup_{\mathsf{Int}} j_2$ with respect to $i_1 \sqcup_{\mathsf{Int}} j_1$. A similar argument applies to n-dimensional convex numerical polytopes $\boldsymbol{\mathcal{N}}$ equipped with the strong pre-metric $\delta^{\mathit{Vol}}_{\boldsymbol{\mathcal{N}}}$, when the volume is computed exactly. ♦

We assume that the join-bound $\oplus_{\delta_{\mathcal{L}}}$ is a parameter of the proof system in Fig. 3, and that the same join-bound is also used in the EPL introduced in Section 6.4. Given that both predicates ω_1-$\textsc{Cont}(\mathsf{P}_1)_{\mathcal{L}}$ and ω_2-$\textsc{Cont}(\mathsf{P}_2)_{\mathcal{L}}$ hold, $\omega_{\sqcup}$ for rule $(\textbf{Join}_{\omega})$ is defined as:

$$\omega_{\sqcup}(t) \overset{\text{def}}{=} \oplus_{\delta_{\mathcal{L}}}(\omega_1(t), \omega_2(t))$$

Given the validity of ω-$\textsc{Cont}(\mathsf{P})_{\mathcal{L}}$, rule $(\textbf{Iterate}_{\omega})$ defines the modulus ω_* as the $\oplus_{\delta_{\mathcal{L}}}$*-limit*

$$\omega_*(t) \overset{\text{def}}{=} \bigoplus_{n=0}^{\infty} \omega^n(t)$$

where $\bigoplus_{n=0}^{0} \omega^n(t) = t$ and $\bigoplus_{n=0}^{i+1} \omega^n(t) = (\bigoplus_{n=0}^{i} \omega^n(t)) \oplus_{\delta_{\mathcal{L}}} \omega^{i+1}(t)$. Each iteration of $\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}$ corresponds to an application of ω. The join-bound operation accounts for the additional imprecision introduced by the join at each step.

Example 6.5. Let us consider the interval domain Int and the interval semantics $\llbracket \cdot \rrbracket^{\sharp}_{\mathsf{Int}}$. To simplify the calculations, from now on we assume that the abstract semantics for basic commands coincides with their bca (Definition 2.4) over the concrete collecting reachability semantics, that is, $\llbracket \mathsf{c} \rrbracket^{\sharp}_{\mathsf{Int}} = \llbracket \mathsf{c} \rrbracket^{\alpha}_{\mathsf{Int}}$. We use the counting distance $\delta^{\sim}_{\mathsf{Int}}$ introduced in Example 2.10, and the join-bound $\oplus_{\delta^{\sim}_{\mathsf{Int}}} = +$ of Example 6.4. We proceed to derive the modulus of continuity for the absolute-value program, written in our programming language

$$\mathsf{ABS}\ :\ (x \geq 0?; x := x) \oplus (x < 0?; x := -x)$$

by following the rules of Figure 3.

We observe that, in general, the abstract semantics $\llbracket x := kx + q \rrbracket^{\sharp}_{\mathsf{Int}}$ of a linear assignment, where $k, q \in \mathbb{Z}$, is ω-continuous with modulus $\omega(t) = Av(k)t$, where $Av(k)$ denotes the absolute value of k. Indeed, applying the abstract linear assignment $\llbracket x := kx + q \rrbracket^{\sharp}_{\mathsf{Int}}$ can increase the size of an interval by at most a factor of $Av(k)$, therefore, for any two input intervals whose distance is t, the distance between their abstract images is at most $Av(k)t$. This allows us to derive id-$\textsc{Cont}(x := x)_{\mathsf{Int}}$ and id-$\textsc{Cont}(x := -x)_{\mathsf{Int}}$ by rule $(\textbf{Base}_{\omega})$.

For Boolean guards of the form $x \geq 0?$ and $x < 0?$, the corresponding abstract transformers can only reduce the size of an interval or leave it unchanged. Therefore, $\omega = id$ is a valid modulus for both guards, and we can derive id-$\textsc{Cont}(x \geq 0)_{\mathsf{Int}}$ and id-$\textsc{Cont}(x < 0)_{\mathsf{Int}}$ by $(\textbf{Base}_{\omega})$.

We can now compose $x \geq 0?$ and $x := x$ by rule $(\textbf{Seq}_{\omega})$ obtaining id-$\textsc{Cont}(x \geq 0?; x := x)_{\mathsf{Int}}$, and $x < 0?$ with $x := -x$ obtaining id-$\textsc{Cont}(x < 0?; x := -x)_{\mathsf{Int}}$. Rule $(\textbf{Join}_{\omega})$ with $\oplus_{\delta^{\sim}_{\mathsf{Int}}} = +$ allows to derive $\omega_{\sqcup}$-$\textsc{Cont}(\mathsf{ABS})_{\mathsf{Int}}$ namely $\llbracket \mathsf{ABS} \rrbracket^{\sharp}_{\mathsf{Int}}$ is $\omega_{\sqcup}$-continuous with $\omega_{\sqcup}(t) = id(t) + id(t) = 2t$. ♦

Example 6.6. Consider the program R^* where

$$\mathsf{R}\ :\ x > 1?; x := x/2$$

and the semantics $\llbracket \mathsf{R}^* \rrbracket^{\sharp}_{\mathsf{Int}_{\mathbb{R}}}$ where $\mathsf{Int}_{\mathbb{R}}$ is the interval domain over the real numbers. We use the volume distance $\delta^{\mathit{Vol}}_{\mathsf{Int}_{\mathbb{R}}}$ where $\mathit{Vol}([a, b]) = b - a$, and the join-bound $\oplus_{\delta^{\mathit{Vol}}_{\mathsf{Int}_{\mathbb{R}}}} = +$ (Example 6.4). Then, $\llbracket x := x/2 \rrbracket^{\sharp}_{\mathsf{Int}_{\mathbb{R}}}$ is $\lambda t.\ \frac{t}{2}$-continuous, while $\llbracket x > 1 \rrbracket^{\sharp}_{\mathsf{Int}_{\mathbb{R}}}$ is id-continuous. Their sequential composition gives, by rule $(\textbf{Seq}_{\omega})$, $id \circ \lambda t.\ \frac{t}{2}$-$\textsc{Cont}(\mathsf{R})_{\mathsf{Int}_{\mathbb{R}}}$. Since $id \circ \lambda t.\ \frac{t}{2} = \lambda t.\ \frac{t}{2}$ and the series $\sum_{n=0}^{\infty}(\frac{1}{2})^n t$ is a geometric sum converging to $2t$, we can apply rule $(\textbf{Iterate}_{\omega})$ and conclude $\lambda t.2t$-$\textsc{Cont}(\mathsf{R}^*)_{\mathsf{Int}_{\mathbb{R}}}$. ♦

$$\frac{g \in \mathscr{G}(\gamma\alpha(g)) \quad \mathbb{C}^{\boldsymbol{e}}_{g}(\llbracket \mathsf{c} \rrbracket_{C}, \llbracket \mathsf{c} \rrbracket_{\mathcal{A}})}{\tilde{\boldsymbol{e}}\text{-}\textsc{Bound}(\mathsf{c}, g)_{\mathcal{A}}}\ (\textbf{Base}) \qquad \tilde{\boldsymbol{e}}(c) \overset{\text{def}}{=} \begin{cases} \boldsymbol{e}(g) & \text{if } c \in [g, \gamma\alpha(g)], \\ \boldsymbol{e}(c) & \text{otherwise.} \end{cases}$$

$$\frac{\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}} \quad \boldsymbol{e} \leq \boldsymbol{e}'}{\boldsymbol{e}'\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}}\ (\textbf{Weaken})$$

$$\frac{\begin{array}{cc} \boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}} & \alpha(g) = \alpha(h) \\ \boldsymbol{e}'\text{-}\textsc{Bound}(\mathsf{P}, h)_{\mathcal{A}} & \boldsymbol{e}(g) \leq \boldsymbol{e}'(h) \end{array}}{\boldsymbol{e}''\text{-}\textsc{Bound}(\mathsf{P}, h)_{\mathcal{A}}}\ (\textbf{Gen-Switch}) \qquad \boldsymbol{e}''(c) \overset{\text{def}}{=} \begin{cases} \boldsymbol{e}(g) & \text{if } c \in [g, \gamma\alpha(g)] \cap [h, \gamma\alpha(h)], \\ \boldsymbol{e}'(h) & \text{otherwise.} \end{cases}$$

$$\frac{\begin{array}{ccc} & \omega\text{-}\textsc{Cont}(\mathsf{P}_2)_{\mathcal{A}} & [g]\mathsf{P}_1[h]_{C} \\ \boldsymbol{e}_1\text{-}\textsc{Bound}(\mathsf{P}_1, g)_{\mathcal{A}} & \boldsymbol{e}_2\text{-}\textsc{Bound}(\mathsf{P}_2, h)_{\mathcal{A}} & \{g\}\mathsf{P}_1\{\gamma\alpha(h)\}_{C} \end{array}}{\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}_1; \mathsf{P}_2, g)_{\mathcal{A}}}\ (\textbf{Seq}) \qquad \boldsymbol{e}(c) \overset{\text{def}}{=} \begin{cases} \boldsymbol{e}_2(h) + \omega(\boldsymbol{e}_1(g)) & \text{if } c \in [g, \gamma\alpha(g)], \\ \boldsymbol{e}_2(c) & \text{otherwise.} \end{cases}$$

$$\frac{\boldsymbol{e}_1\text{-}\textsc{Bound}(\mathsf{P}_1, g)_{\mathcal{A}} \quad \boldsymbol{e}_2\text{-}\textsc{Bound}(\mathsf{P}_2, g)_{\mathcal{A}}}{\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}_1 \oplus \mathsf{P}_2, g)_{\mathcal{A}}}\ (\textbf{Join}) \qquad \boldsymbol{e}(c) \overset{\text{def}}{=} \begin{cases} \oplus_{\delta_{\mathcal{A}}}(\boldsymbol{e}_1(g), \boldsymbol{e}_2(g)) & \text{if } c \in [g, \gamma\alpha(g)], \\ \boldsymbol{e}_1(c) & \text{otherwise.} \end{cases}$$

$$\frac{\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}} \quad [g]\mathsf{P}^*[post]_{C} \quad \{\alpha(g) \sqcup_{\mathcal{A}} inv\}\mathsf{P}\{\alpha(g) \sqcup_{\mathcal{A}} inv\}_{\mathcal{A}}}{\bar{\boldsymbol{e}}\text{-}\textsc{Bound}(\mathsf{P}^*, g)_{\mathcal{A}}}\ (\textbf{Iterate}) \qquad \bar{\boldsymbol{e}}(c) \overset{\text{def}}{=} \begin{cases} \delta_{\mathcal{A}}(\alpha(post), \alpha(g) \sqcup_{\mathcal{A}} inv) & \text{if } c \in [g, \gamma\alpha(g)], \\ \boldsymbol{e}(c) & \text{otherwise.} \end{cases}$$

Fig. 4. Rules of EPL.

Let $\vdash \omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}$ denote a derivation of the ω-continuity predicate for program P according to rules of Figure 3. The following theorem states the soundness of all rules in Figure 3.

Theorem 6.7 (Soundness). $\vdash \omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}} \implies \omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}$ □

6.4 EPL: Error Propagation Logic

We now have all the necessary components to present our program logic for deriving a bounding function $\boldsymbol{e}$ inductively from the syntax of a program, in support of the partial local completeness property. Let us fix a GC $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$ and a strong order-compatible pre-metric $\delta_{\mathcal{A}}$. The goal of the *Error Propagation Logic* (*EPL*), defined in Figure 4, is to derive the predicate $\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$ defined as follows. For clarity, from this point on, we will use the color blue to highlight the generators in the concrete domain C.

*Definition 6.8 (**EPL predicate**).* Let $\mathsf{P} \in \mathsf{Prog}$, $g \in C$ and $\boldsymbol{e} \colon C \longrightarrow \mathbb{I}^{\infty}_{\geq 0}$. The predicate $\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$ holds when the following two conditions are satisfied:

$$\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}} \overset{def}{\Leftrightarrow} (i)\ g \in \mathscr{G}(\gamma\alpha(g)) \land (ii)\ \mathbb{C}^{\boldsymbol{e}}_{[g, \gamma\alpha(g)]}(\llbracket \mathsf{P} \rrbracket_{C}, \llbracket \mathsf{P} \rrbracket_{\mathcal{A}}) \qquad \blacksquare$$

Deriving a proof of $\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$ establishes that $\boldsymbol{e}(g)$ *is an upper bound on the imprecision* introduced by the abstract semantics $\llbracket \mathsf{P} \rrbracket_{\mathcal{A}}$ with respect to the abstraction of the concrete semantics $\llbracket \mathsf{P} \rrbracket_{C}$, measured by the distance $\delta_{\mathcal{A}}$, for all inputs in the set $[g, \gamma\alpha(g)]$. In particular, if we can derive $\boldsymbol{e}(g) = 0$, then two important consequences follow:

(1) $\llbracket \mathsf{P} \rrbracket_{\mathcal{A}}$ introduces *no imprecision* at any input in $[g, \gamma\alpha(g)]$;
(2) by Theorem 2.6, any specification $Spec \in \mathcal{A}$ can be *precisely* proved by $\llbracket \mathsf{P} \rrbracket_{\mathcal{A}}$ on this input set, i.e., with *no false positives*, by checking whether $\llbracket \mathsf{P} \rrbracket_{\mathcal{A}} \alpha(c) \sqsubseteq_{\mathcal{A}} Spec$ holds for all $c \in [g, \gamma\alpha(g)]$.

We now provide an intuitive explanation of the EPL in Figure 4, where the deductive rules are shown on the left, and their corresponding effects on the bounding function on the right.

An inductive proof for P begins with its base commands. Rule (**Base**) asserts that if the abstract interpretation $[\![\mathsf{c}]\!]_{\mathcal{A}}$ of a base command $\mathsf{c} \in \mathsf{BCom}$ is $\boldsymbol{e}$-partial local complete at the generator g, then, by Theorem 5.1, it is also $\tilde{\boldsymbol{e}}$-partial local complete over the entire chain $[g, \gamma\alpha(g)]$. Here, $\tilde{\boldsymbol{e}}$ is a new bounding function with value $\boldsymbol{e}(g)$ for all elements in the chain.

(**Weaken**) simply states that we may safely replace a bounding function $\boldsymbol{e}$ with any pointwise greater function $\boldsymbol{e}' \geq \boldsymbol{e}$, without affecting the validity of the predicate.

By (**Gen-Switch**), suppose we have two derivations, $\boldsymbol{e}$-$\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$ and $\boldsymbol{e}'$-$\textsc{Bound}(\mathsf{P}, h)_{\mathcal{A}}$, corresponding to two different generators of the same abstract property (i.e., $\alpha(g) = \alpha(h)$), where $\boldsymbol{e}(g)$ provides a tighter (i.e., lower) bound than $\boldsymbol{e}'(h)$. In this case, it is possible to retain h in the predicate while *updating* the bounding function $\boldsymbol{e}'$ to a new function $\boldsymbol{e}''$ that incorporates the improved result obtained at g. Specifically, $\boldsymbol{e}''$ assigns the value $\boldsymbol{e}(g)$ to all elements in the intersection between the set $[g, \gamma\alpha(g)]$ and $[h, \gamma\alpha(h)]$, while preserving the original bounds elsewhere. This rule is particularly useful for two main purposes: first, to *refine* a bounding function by injecting better (i.e., lower) bounds for overlapping portions of chains; and second, to align EPL predicates at a common or suitable generator, which is required in order to apply rules (**Seq**) and (**Join**).

To apply (**Seq**) for the sequential composition of P_1 and P_2 under the premises $\boldsymbol{e}_1$-$\textsc{Bound}(\mathsf{P}_1, g)_{\mathcal{A}}$ and $\boldsymbol{e}_2$-$\textsc{Bound}(\mathsf{P}_2, h)_{\mathcal{A}}$, three additional conditions must be verified. First, the abstract semantics $[\![\mathsf{P}_2]\!]_{\mathcal{A}}$ must be *ω-continuous*, that is, the predicate ω-$\textsc{Cont}(\mathsf{P}_2)_{\mathcal{A}}$ must hold. This condition ensures that for any two inputs, thus including all the elements in the chain $[\alpha([\![\mathsf{P}_1]\!]_{C} g), [\![\mathsf{P}_1]\!]_{\mathcal{A}}\alpha(g)]$, the distance between their abstract outputs under $[\![\mathsf{P}_2]\!]_{\mathcal{A}}$ *is bounded above* by applying the modulus function ω to the distance between those inputs. In other words, ω provides a sound upper bound on *how the abstract semantics* of P_2 *propagates imprecision* through its inputs within the chain. Second, the triple $[g]\mathsf{P}_1[h]_C$ corresponds to the *incorrectness logic* of O'Hearn [2020], which requires that $h \sqsubseteq_C [\![\mathsf{P}_1]\!]_C g$, that is, h under-approximates the concrete behavior of P_1 on g. Third, the triple $\{g\}\mathsf{P}_1\{\gamma\alpha(h)\}_C$ encodes standard *Hoare partial correctness*, requiring that $\gamma\alpha(h)$ over-approximates $[\![\mathsf{P}_1]\!]_C g$. Together, these two triples ensure that the image of the set $[g, \gamma\alpha(g)]$ under the concrete semantics $[\![\mathsf{P}_1]\!]_C$ is contained within $[h, \gamma\alpha(h)]$, making the sequential composition well-defined.

(**Join**) leverages the join-bound function $\oplus_{\delta_{\mathcal{A}}}$ defined in Definition 6.3 to derive a new bounding function for the join of the two programs. Here the bound assigned to $\mathsf{P}_1 \oplus \mathsf{P}_2$ at the generator g is given by $\oplus_{\delta_{\mathcal{A}}}(\boldsymbol{e}_1(g), \boldsymbol{e}_2(g))$. This bound is then propagated to the entire chain $[g, \gamma\alpha(g)]$.

(**Iterate**) addresses the Kleene iteration of a program. The idea is to derive an upper bound on the distance between $\alpha([\![\mathsf{P}^*]\!]_C g)$ and $[\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(g)$ by relating them to, respectively, an abstracted under-approximation and an abstract over-approximation. Specifically, we consider an under-approximation *post* of $[\![\mathsf{P}^*]\!]_C g$, encoded by the (concrete) incorrectness triple $[g]\mathsf{P}^*[\mathit{post}]_C$, and an abstract invariant $\mathit{inv} \in \mathcal{A}$ of $[\![\mathsf{P}]\!]_{\mathcal{A}}$ containing $\alpha(g)$, encoded by the (abstract) correctness triple

$$\{\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}\}\mathsf{P}\{\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}\}_{\mathcal{A}}$$

By (*chain-order*) of $\delta_{\mathcal{A}}$, the distance $\delta_{\mathcal{A}}(\alpha(\mathit{post}), \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv})$ provides an upper bound on the distance $\delta_{\mathcal{A}}(\alpha([\![\mathsf{P}^*]\!]_C g), [\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(g))$. By Theorem 5.1, this bound is also valid for all input elements of the chain $[g, \gamma\alpha(g)]$. Clearly, the wider the under-approximation *post* is and the tighter the invariant *inv* is, the more precise the resulting bounding function will be. For instance, a sound choice of *post* could be $[\![\mathsf{P}]\!]_C g$ since $[\![\mathsf{P}]\!]_C g \sqsubseteq_C [\![\mathsf{P}^*]\!]_C g$. In this case, thanks to $\boldsymbol{e}$-$\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$ and (*chain-order*) of $\delta_{\mathcal{A}}$, the overall distance $\delta_{\mathcal{A}}(\alpha([\![\mathsf{P}^*]\!]_C g), [\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(g))$ can be bounded by the sum

$$\boldsymbol{e}(g) + \delta_{\mathcal{A}}([\![\mathsf{P}]\!]_{\mathcal{A}}\alpha(g), \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv})$$

For all elements of the concrete domain outside $[g, \gamma\alpha(g)]$, the new bounding function $\bar{\boldsymbol{e}}$ preserves the previous values $\boldsymbol{e}$ derived for P. This ensures that no upper bound obtained during the derivation of P is lost.

All the rules in Figure 4 are sound as stated by the following theorem.

Theorem 6.9 (Soundness of EPL). $\vdash \boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}} \;\implies\; \boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$ □

We conclude by presenting three examples that illustrate the application of EPL.

Example 6.10. We continue Example 6.5 by deriving a bounding function $\boldsymbol{e}$ for the partial completeness of the standard interval semantics $\llbracket\mathsf{ABS}\rrbracket^{\sharp}_{\mathsf{Int}} : \mathsf{Int} \to \mathsf{Int}$ with respect to the concrete reachability semantics $\llbracket\mathsf{ABS}\rrbracket : \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, for the generator $\{5, 10\}$ of the interval $[5, 10]$. By rule (**Base**), we can derive $\mathbf{0}\text{-}\textsc{Bound}(x \geq 0?, \{5, 10\})_{\mathsf{Int}}$ and $\mathbf{0}\text{-}\textsc{Bound}(x := x, \{5, 10\})_{\mathsf{Int}}$. Since both $x \geq 0?$ and $x := x$ are not modifying the input $\{5, 10\}$, the two triples

$$[\{5, 10\}]x \geq 0?[\{5, 10\}]_{\wp(\mathbb{Z})} \qquad \{\{5, 10\}\}x \geq 0?\{\{5, 6, 7, 8, 9, 10\}\}_{\wp(\mathbb{Z})}$$

hold. By rule (**Seq**) and the predicate $\mathit{id}\text{-}\textsc{Cont}(x := x)_{\mathsf{Int}}$, we derive $\mathbf{0}\text{-}\textsc{Bound}(x \geq 0?; x := x, \{5, 10\})_{\mathsf{Int}}$. From the composition on the right of the join we get, by (**Base**), $\mathbf{0}\text{-}\textsc{Bound}(x < 0?, \{5, 10\})_{\mathsf{Int}}$ and $\mathbf{0}\text{-}\textsc{Bound}(x := -x, \{5, 10\})_{\mathsf{Int}}$. The two triples

$$[\{5, 10\}]x < 0?[\varnothing]_{\wp(\mathbb{Z})} \qquad \{\{5, 10\}\}x < 0?\{\varnothing\}_{\wp(\mathbb{Z})}$$

trivially hold. By rule (**Seq**) and $\mathit{id}\text{-}\textsc{Cont}(x := -x)_{\mathsf{Int}}$, we derive $\mathbf{0}\text{-}\textsc{Bound}(x < 0?; x := -x, \{5, 10\})_{\mathsf{Int}}$. Thus by rule (**Join**), we can conclude $\mathbf{0}\text{-}\textsc{Bound}(\mathsf{ABS}, \{5, 10\})_{\mathsf{Int}}$. We have obtained an optimal bounding function that tells us that $\llbracket\mathsf{ABS}\rrbracket^{\sharp}_{\mathsf{Int}}$ is local complete on every input in $[\{5, 10\}, \gamma\alpha(\{5, 10\})]$.

Conversely, if we start from the generator $\{-5, 5\}$, by rule (**Base**), we can derive $\mathbf{5}\text{-}\textsc{Bound}(x \geq 0?, \{-5, 5\})_{\mathsf{Int}}$ and $\mathbf{0}\text{-}\textsc{Bound}(x := x, \{5\})_{\mathsf{Int}}$. The constant function $\mathbf{5}$ is due to both guards and the input $\{-5, 5\}$ not containing the value 0. Indeed, $\alpha_{\mathsf{Int}}(\llbracket\mathsf{ABS}\rrbracket\{-5, 5\}) = [5, 5]$, while $\llbracket\mathsf{ABS}\rrbracket^{\sharp}_{\mathsf{Int}}[-5, 5] = [0, 5]$. Since $\{5\} \subseteq \llbracket x \geq 0?\rrbracket\{-5, 5\} \subseteq \{5\}$, both triples

$$[\{-5, 5\}]x \geq 0?[\{5\}]_{\wp(\mathbb{Z})} \qquad \{\{-5, 5\}\}x \geq 0?\{\{5\}\}_{\wp(\mathbb{Z})}$$

hold. We can conclude with rule (**Seq**) $\mathbf{5}\text{-}\textsc{Bound}(x \geq 0?; x := x, \{-5, 5\})_{\mathsf{Int}}$ because $\mathbf{0}(\{5\}) + \mathit{id}(\mathbf{5}(\{-5, 5\})) = 0 + 5 = 5$. In a similar way, we can derive $\mathbf{4}\text{-}\textsc{Bound}(x < 0?; x := -x, \{-5, 5\})_{\mathsf{Int}}$ for the right part of the join. Finally, (**Join**) concludes $\mathbf{9}\text{-}\textsc{Bound}(\mathsf{ABS}, \{-5, 5\})_{\mathsf{Int}}$. Note that here the derived bound is not optimal because the chosen join-bound $\oplus_{\delta^{\sim}_{\mathsf{Int}}} = +$, which is a parameter of EPL, is not precise. ♦

Example 6.11. We now reconsider the program ABS, this time applied to two-dimensional inputs in $\mathbb{Z}^2$, such that, for instance, $\llbracket\mathsf{ABS}\rrbracket\{(-1, 4), (1, 0)\} = \{(1, 4), (1, 0)\}$ as the program filters only the x-component of the input. In this setting, we use the volume distance $\delta^{\mathit{Vol}}_{\mathsf{Int}^2}$ as strong order-compatible pre-metric, and the join-bound $\oplus_{\delta^{\mathit{Vol}}_{\mathsf{Int}^2}} = +$. Let us consider the rectangle $R = (x : [-1, 1], y : [0, 4])$ and one of its generators $H = \{(-1, 4), (1, 0)\}$. Following Example 6.10, we apply similar derivations on the left and on the right of the join to obtain $\mathbf{1}\text{-}\textsc{Bound}(x \geq 0?; x := x, H)_{\mathsf{Int}^2}$ and $\mathbf{1}\text{-}\textsc{Bound}(x < 0?; x := -x, H)_{\mathsf{Int}^2}$. Then (**Join**) concludes $\mathbf{2}\text{-}\textsc{Bound}(\mathsf{ABS}, H)_{\mathsf{Int}^2}$, thus assigning a constant bound of 2 to all the sets of points in $[H, \gamma_{\mathsf{Int}^2}(R)]$ including the rectangle R. This means that the difference in terms of volume between the concrete and abstract invariant on those points is at most 2.

In fact, we can improve this bound on some points in $[H, \gamma_{\mathsf{Int}^2}(R)]$. If we start another derivation from the generator $G = \{(1, 2), (0, 4), (0, 0), (-1, 2)\}$, we get the following derivations: $\mathbf{0}\text{-}\textsc{Bound}(x \geq 0?; x := x, G)_{\mathsf{Int}^2}$ and $\mathbf{0}\text{-}\textsc{Bound}(x < 0?; x := -x, G)_{\mathsf{Int}^2}$ by rule (**Seq**), then $\mathbf{0}\text{-}\textsc{Bound}(\mathsf{ABS}, G)_{\mathsf{Int}^2}$ by rule (**Join**). Note that this result guarantees the precision of the abstract interpreter at all inputs in $[G, \gamma_{\mathsf{Int}^2}(R)]$, due to the choice of the generator G for which the abstraction of the concrete semantics coincides with the abstract semantics. Since $\alpha_{\mathsf{Int}^2}(H) = R = \alpha_{\mathsf{Int}^2}(G)$ and $\mathbf{0}(G) \leq \mathbf{2}(H)$, we can apply rule (**Gen-Switch**) to refine the bounding function $\mathbf{2}$ for the generator H, obtaining $\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{ABS}, H)_{\mathsf{Int}^2}$. The updated bounding function $\boldsymbol{e}$ assigns $\boldsymbol{e}(c) = 0$ to all the elements in the intersection $[H, \gamma_{\mathsf{Int}^2}(R)] \cap [G, \gamma_{\mathsf{Int}^2}(R)]$—in particular, to the rectangle R—while preserving the original bound 2 elsewhere. Thus, $\boldsymbol{e}$ provides the tightest bound (which was not the case for 2). ♦

Example 6.12. Consider the following program

$$\mathsf{F} \;:\; x \geq 0 \land y \geq 2?;\; y := y/4;\; x := x - 1$$

and the 2-dimensional intervals Int^2 over reals $\mathbb{R}^2$ together with the volume distance $\delta^{Vol}_{\mathsf{Int}^2}$. We want to derive a bounding function for $\mathsf{ABS}; \mathsf{F}^*$, namely the composition between the program ABS and F^* over the set of inputs $[G, \gamma_{\mathsf{Int}^2}(R)]$, where R is the rectangle $R = (x : [-2, 2], y : [23, 24])$ and the generator $G = \{(-2, 24), (2, 23)\}$. In order to apply rule (**Seq**) of EPL between ABS and F^*, let us start by inductively deriving a modulus of continuity for $[\![\mathsf{F}^*]\!]^\sharp_{\mathsf{Int}^2}$ by following the rules in Figure 3. For the three base commands, by rule (**Base**$_\omega$), we derive id-$\textsc{Cont}(x \geq 0 \land y \geq 2?)_{\mathsf{Int}^2}$, $\lambda t.\, t/4$-$\textsc{Cont}(y := y/4)_{\mathsf{Int}^2}$ and id-$\textsc{Cont}(x := x - 1)_{\mathsf{Int}^2}$. Then by (**Seq**$_\omega$), $\lambda t.\, \frac{t}{4}$-$\textsc{Cont}(\mathsf{F})_{\mathsf{Int}^2}$. Finally, (**Iterate**$_\omega$) concludes with $\lambda t.\, \frac{4}{3}t$-$\textsc{Cont}(\mathsf{F}^*)_{\mathsf{Int}^2}$.

We now apply EPL to F^*. By using in sequence (**Base**) and (**Seq**), we derive 0-$\textsc{Bound}(\mathsf{F}, H)_{\mathsf{Int}^2}$ for $H = \{(2, 23), (2, 24)\}$, namely the abstract interpreter is precise when the first computational step is applied to the input $\alpha_{\mathsf{Int}^2}(H)$. In order to apply rule (**Iterate**), we consider as (non-optimal) *post* the result $[\![\mathsf{F}]\!]_C H$ since the incorrectness triple $[H]\mathsf{F}^*[[\![\mathsf{F}]\!]H]_{\wp(\mathbb{R}^2)}$ holds, while as *inv* the actual result $[\![\mathsf{F}^*]\!]^\sharp_{\mathsf{Int}^2}\alpha_{\mathsf{Int}^2}(H)$ of the abstract interpreter. We can then apply (**Iterate**) to derive $\boldsymbol{e}$-$\textsc{Bound}(\mathsf{F}^*, H)_{\mathsf{Int}^2}$ where:

$$\forall S \in [H, \gamma_{\mathsf{Int}^2}(H)].\, \boldsymbol{e}(S) = \mathbf{0}(H) + \delta^{Vol}_{\mathsf{Int}^2}([\![\mathsf{F}]\!]^\sharp_{\mathsf{Int}^2}\alpha_{\mathsf{Int}^2}(H), [\![\mathsf{F}^*]\!]^\sharp_{\mathsf{Int}^2}\alpha_{\mathsf{Int}^2}(H)) = 47$$

Finally, since 4-$\textsc{Bound}(\mathsf{ABS}, G)_{\mathsf{Int}^2}$ holds (derivations are similar to Example 6.11), by (**Seq**) we obtain $\bar{\boldsymbol{e}}$-$\textsc{Bound}(\mathsf{ABS}; \mathsf{F}^*, G)_{\mathsf{Int}^2}$, where:

$$\forall S \in [G, \gamma_{\mathsf{Int}^2}(G)].\, \bar{\boldsymbol{e}}(S) = 47 + \tfrac{4}{3}\mathbf{4}(G) = 52.34$$

Thus, for all input sets in $[G, \gamma_{\mathsf{Int}^2}(R)]$, the abstract semantics is guaranteed to produce an imprecision—measured as the volume difference between the abstract and concrete invariants—bounded by the value 52.34. This is a sound upper bound, albeit not optimal: the real distance is

$$\delta^{Vol}_{\mathsf{Int}^2}(\alpha_{\mathsf{Int}^2}([\![\mathsf{ABS}; \mathsf{F}^*]\!]G), [\![\mathsf{ABS}; \mathsf{F}^*]\!]^\sharp_{\mathsf{Int}^2}\alpha_{\mathsf{Int}^2}(G)) = 25.38$$

This non-optimality arises mainly from the choice of $post = [\![\mathsf{F}]\!]H$ in the incorrectness triple $[H]\mathsf{F}^*[post]$. By choosing a more precise under-approximation of $[\![\mathsf{F}^*]\!]H$, for instance by taking $post = [\![\mathsf{F}]\!][\![\mathsf{F}]\!]H$, i.e., two iterations of F, we would obtain a more precise result. This follows a reasoning analogous to loop unrolling, a technique commonly used in static analyzers based on the standard framework of abstract interpretation performing over-approximations [Miné 2017; Rival and Yi 2020]: the more a loop is unrolled, the more accurate the resulting static analysis becomes, since each unrolling exposes additional control-flow structure and reduces the loss of precision caused by merging abstract states at loop headers. In our setting, iterating F a few times plays an analogous role: it provides a more refined under-approximation of the reachable states, thus increasing the precision of the derived result. ♦

7 RELATED WORK

The idea of minimally representing abstract domains goes back to the notions of abstract domain compression by Filé et al. [1996]. For the case of disjunctive bases [Giacobazzi and Ranzato 1996], the compression corresponds to the set of join-irreducible elements of the abstract domain. This notion has found an order theoretic characterization by Giacobazzi and Ranzato [1998], where the authors introduced uniform closure operators corresponding precisely to those abstract domains that can be minimally compressed, namely that can be reduced to a minimal abstract domain whose refinement gives back the original domain. The notion of generator is strictly weaker and does not require that the original domain can be reconstructed from a unique set of generators. This makes the notion of generator more widely applicable.

The notion of generator of an abstract property in a GC takes inspiration from the generator representation of (closed) convex polytope, sometimes also called the vertex-representation (e.g., see [Grünbaum et al. 1967; Ziegler 2012]). A closed convex polytope can be defined as the convex hull of a finite set of points, where the finite set must contain the set of extreme points of the polytope, i.e., its vertices. Such a definition is also called a vertex-representation [Grünbaum et al. 1967]. For a closed convex polytope, the minimal vertex-representation is unique and it is given by the set of the vertices of the polytope. The existence of generable abstract properties (i.e. abstract properties that admit at least one generator) in an abstract domain is not always guaranteed and, to the best of our knowledge, has never been formally defined or studied in the context of (non necessarily numerical) abstract domains. In the polyhedra domain, introduced by Cousot and Halbwachs [1978], a key result is the Weyl-Minkowski Theorem, stating that polyhedra have dual representations: one using constraints, and one using generators. In this context, generators can consider additionally rays, namely a finite set of directions that can be followed. This is necessary in order to be able to represent an unbounded polyhedron. In fact, adding details to a property representation, such as rays, may help designing the minimal property representing it, i.e., the existence of a generator of that property.

While the validity of the predicates $\mathbb{C}_c(\llbracket \mathsf{P} \rrbracket, \llbracket \mathsf{P} \rrbracket^{\sharp}_{\mathcal{A}})$ and $\mathbb{C}(\llbracket \mathsf{P} \rrbracket, \llbracket \mathsf{P} \rrbracket^{\sharp}_{\mathcal{A}})$ for the local completeness at a singleton $c \in \boldsymbol{C}$, and completeness properties have already been investigated through a deductive system by Bruni et al. [2023] and Giacobazzi et al. [2015], respectively, the only work that proposes a similar approach using deductive rules for establishing ε-partial local completeness is [Campion et al. 2022]. However, in that work, ε is restricted to a constant, which is insufficient in scenarios where one aims to prove partial completeness across a subset of the concrete domain and the imprecision has no constant limit. Thanks to our settings, we are also able to improve the derived bounding function for specific elements through rule (**Gen-Switch**). Moreover, in their work Campion et al. [2022] did not exploit any notion of continuity for the composition of programs. Their rule (**seq**) requires a very strong assumption on the relation between the concrete behavior of P_1 with respect to P_2, which makes the proposed logic non-compositional. This is in contrast to EPL, where the ω-continuity logic is essential for making the EPL compositional and propagating the error bound inductively on the syntax of programs. Overall, to the best of our knowledge, the EPL system introduced in Section 6 is the only proof system that reasons solely on generators to identify inputs in the concrete domain that satisfy the partial completeness property, without requiring separate derivations for intermediate inputs lying between a generator and its abstraction.

In their work, Campion et al. [2024] and Johnson et al. [2024] show that, when the concrete semantics is monotone (i.e., either order-preserving or order-reversing) over a closed set (i.e., with minimum and maximum), then the bca over the interval domain turns out to be local complete. This reasoning is, in fact, a specific instance of a more general result concerning generators: when the concrete semantics f maps a generator of the input to a generator of the output of the abstract semantics $f^{\sharp}$, local completeness of $f^{\sharp}$ follows. Monotonicity ensures that the generators of a closed input set, namely, its minimum and maximum elements are mapped to the generators of the abstract output. Monotonicity is also exploited in practical applications to achieve greater precision. For instance, recent algorithms for computing classifier explanations [Hurault and Marques-Silva 2023; Marques-Silva et al. 2021] leverage this property to reduce the analysis to just the minimum and maximum values of the input features, that is, to the generators of the input domain.

Definition 6.2 of ω-continuity is different from the standard definition of uniform continuity and of moduli of continuity in calculus. A function is defined to be *uniform continuous* when it admits a *finite* modulus of continuity for every finite distance, namely $\omega(t) \neq \infty$ for all $t \in \mathbb{R}_{\geq 0}$. Uniform continuity thus requires a global finite bound on how much a function's outputs can change when its inputs vary, and it is a stronger notion than standard continuity, where a function is merely required to have no jumps or discontinuities at each input, but weaker than Lipschitz continuity, which imposes that perturbations to the function's input leads to at most linear changes to its output. The functions admitting a finite modulus of continuity are precisely the uniformly continuous ones [Rudin 1976]. In our setting, Definition 6.2 allows every function to admit an ω such that f is ω-continuous, including those that are not uniformly continuous or even continuous. This generalization allows modeling functions representing any abstract interpreter, which are well known not to be uniformly continuous due to their intrinsic approximation process.

A number of papers studied the problem of continuity in programming languages. In particular, Chaudhuri et al. [2010] observe that many well-known programs in computer science are continuous, while in [Chaudhuri et al. 2011] they develop a proof system for establishing program robustness via Lipschitz continuity. [de Amorim et al. 2017] use Lipschitz continuity to capture a notion of program sensitivity, while Reed and Pierce [2010] to control type sensitivity. Metrics have been extensively studied in program refinement and relational theories of programs [Dal Lago and Gavazzo 2022a,b], again with the goal of controlling forms of sensitivity. For the purpose of EPL, we require a generalized notion of program Lipschitz continuity—one that allows the bounding relation to be expressed as a function, rather than a fixed constant. This is precisely captured by ω-continuity. In the spirit of Reed and Pierce [2010], we can say that: *Distance makes the abstract interpreter sharper.*

Finally, we observe that program properties that deal indirectly with generators, have already found practical applications in the literature, even if not explicitly formalized this way. For instance, recently proposed algorithms for computing classifier explanations [Hurault and Marques-Silva 2023; Marques-Silva et al. 2021] leverage monotonicity to reduce the computation to only consider the minimum and maximum (i.e., the generators of the) possible values of the input features of the classifiers.

8 CONCLUSION

We introduced EPL, a program logic designed to formally bound error propagation in abstract interpretation. EPL combines elements of both correctness and incorrectness logics with a logic for capturing the modulus of continuity of programs. This allows one to express how the imprecision of abstract computations depends on the imprecision of inputs. We also characterized the minimal amount of information which is necessary in order to express proof obligations in EPL. This information is captured by the notion of generators of an abstract domain, minimal concrete elements that maps into the abstract properties. We proved that the entire EPL framework is sound and broadly applicable to abstract domains structured by Galois connections. EPL provides a principled and general approach to compositional reasoning about precision and error bounds in static program analysis.

Although EPL has been proved sound, it currently lacks completeness for two main reasons. First, it provides a worst-case estimate of the imprecision of an abstract interpretation by focusing on generators and propagating the derived bound along the chain. As a result, the inferred bound may be tight for the generators but over-approximated for other elements of the chain. Second, the join-bound is defined simply as a binary function over real numbers, without taking the specific programs being joined into account. Consequently, it may fail to be optimal and might not capture the tightest bound for all inputs. As future work, we plan to extend EPL with rules that guarantee the soundness and completeness of the proof system, allowing the user to choose the concrete elements from which to start the derivation in the input domain, without necessarily restricting the analysis to generators. Generators can still be used when the user wishes to obtain a worst-case estimation over all elements, without deriving the imprecision for each individual input. This could be done by also reformulating the logic in the form of Hoare-style triples.

Similarly, the proof system for deriving a modulus of continuity is still preliminary and requires further refinement. It is not complete for the same join-bound–related reason, and the definition of ω_* in rule (**Iterate**$_\omega$) may fail to converge to a (non-infinite) function unless specific program behaviors hold (e.g., when the program under analysis implements a contractive function). We believe that reformulating the ω-continuity notion as a local property together with its logic—thus bringing it closer to EPL, which is local to an input—would significantly improve its applicability. This is also connected to the previous future work consisting in reformulating EPL to allow using any input element of the concrete domain, without necessarily starting from a generator.

It would be also interesting to study the imprecision induced by different steps of the abstraction process, such as the distance between the concrete semantics and its bca, or the distance between the bca and a sound abstract semantics. In the former case, a tight proof of imprecision over generators would reveal the baseline imprecision introduced by the chosen abstraction. A notion of *partial best correct approximation* could be then defined and studied as a further weakening of the standard bca notion [Giacobazzi and Ranzato 2025]. Both imprecision bounds could be obtained as an instance of our proposed EPL: for instance, by setting the bca as the concrete semantics and the chosen abstract interpretation of the abstract semantics, EPL would then derive a bounding function defining an upper bound on their distance.

We note that the template semantics defined in Section 6.1 does not allow defining a widening-based abstract semantics. This design choice was made to simplify the exposition and examples without explicitly defining a widening operator. Although this may appear as a limitation, our definition of ω-continuity—which allows a non-finite modulus—ensures that the two proposed program logics can also handle abstract semantics defined with widening operators, without requiring major changes to the rules. From this perspective it is interesting to note that widening-based abstract interpretations naturally break the standard definition of uniform continuity, which requires a finite modulus of continuity. This observation opens an interesting theoretical question: *can widening-based abstract interpretations be characterized as non-uniformly continuous abstract interpreters?* This would provide a topological characterization of the widening operation, paving the road to the use of methods known in numerical analysis to widening-based abstract interpreters such as the notion of acceleration.

The notion of generator can also be generalized to the case where the abstraction α is partial, that is, for those pairs of concrete and abstract domains that do not admit a GC. However, (partial) completeness compares the result of the abstract interpreter with $\alpha(f(c))$, i.e., the abstraction of the concrete execution. Therefore, if α is partial, we must require that both c and $f(c)$ are in the domain of α, so that their abstraction is defined. In this setting, the notion of γ-completeness [Cousot 2021], could be explored as a future work to study the precision of abstract interpretations that lack a GC.

Finally, the reliance of EPL on generators may significantly simplify a future implementation of the logic within a proof assistant. Since proof obligations in EPL are formulated solely in terms of generators—rather than the full concrete domain—this restriction reduces the complexity of mechanized reasoning and proof construction. It enables a more modular and scalable encoding of the logic, potentially facilitating automation and easing the verification of local partial completeness properties within interactive theorem provers such as Coq, Isabelle, or Lean.

ACKNOWLEDGMENTS

This work was partially supported by the Air Force Office of Scientific Research under award number FA9550-23-1-0544 and by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. This work was also partially supported by the SAIF project, funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-23-PEIA-0006.

REFERENCES

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2021. A Logic for Locally Complete Abstract Interpretations. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021.* 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 *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 426–441. https://doi.org/10.1145/3519939.3523453

Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 2023. A Correctness and Incorrectness Program Logic. *J. ACM* 70, 2 (2023), 15:1–15:45. https://doi.org/10.1145/3582267

Marco Campion, Mila Dalla Preda, Roberto Giacobazzi, and Caterina Urban. 2024. Monotonicity and the Precision of Program Analysis. *Proc. ACM Program. Lang.* 8, POPL (2024), 1629–1662. https://doi.org/10.1145/3632897

Marco Campion, Mila Dalla Preda, Roberto Giacobazzi, and Caterina Urban. 2026. A Logic for the Imprecision of Abstract Interpretations. *Proc. ACM Program. Lang.* 10, POPL (2026). https://doi.org/10.1145/3776707

Marco Campion, Isabella Mastroeni, and Caterina Urban. 2025. Relating Distances and Abstractions: An Abstract Interpretation Perspective. In *Static Analysis - 32th International Symposium, SAS 2025, Singapore, October 13-14, 2025, Proceedings (Lecture Notes in Computer Science, Vol. 16100)*, Hakjoo Oh and Yulei Sui (Eds.). Springer, 249–277. https://doi.org/10.1007/978-3-032-07106-4_11

Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2022. Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498721

Marco Campion, Caterina Urban, Mila Dalla Preda, and Roberto Giacobazzi. 2023. A Formal Framework to Measure the Incompleteness of Abstract Interpretations. In *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14284)*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer, 114–138. https://doi.org/10.1007/978-3-031-44245-2_7

Ignacio Casso, José F Morales, Pedro López-García, Roberto Giacobazzi, and Manuel V. Hermenegildo. 2019. Computing abstract distances in logic programs. In *International Symposium on Logic-Based Program Synthesis and Transformation.* Springer, 57–72. https://doi.org/10.1007/978-3-030-45260-5_4

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2010. Continuity analysis of programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 57–70. https://doi.org/10.1145/1706299.1706308

Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. 2011. Proving programs robust. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 102–112. https://doi.org/10.1145/2025113.2025131

Patrick Cousot. 2002. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.* 277, 1-2 (2002), 47–103. https://doi.org/10.1016/S0304-3975(00)00313-3

Patrick Cousot. 2021. *Principles of Abstract Interpretation.* The MIT Press, Cambridge, Mass.

Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming.* Dunod, Paris, 106–130. https://doi.org/10.1145/390019.808314

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, New York, NY, Tucson, Arizona, 84–97. https://doi.org/10.1145/512760.512770

Ugo Dal Lago and Francesco Gavazzo. 2022a. Effectful program distancing. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498680

Ugo Dal Lago and Francesco Gavazzo. 2022b. A relational theory of effects and coeffects. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498692

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, Shin-ya Katsumata, and Ikram Cherigui. 2017. A semantic account of metric preservation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 545–556. https://doi.org/10.1145/3009837.3009890

Michel Marie Deza and Monique Laurent. 1997. *Geometry of cuts and metrics*. Algorithms and combinatorics, Vol. 15. Springer. https://doi.org/10.1007/978-3-642-04295-9

Alessandra Di Pierro and Herbert Wiklicky. 2000. Measuring the Precision of Abstract Interpretations. In *Logic Based Program Synthesis and Transformation, 10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Papers (Lecture Notes in Computer Science, Vol. 2042)*, Kung-Kiu Lau (Ed.). Springer, 147–164. https://doi.org/10.1007/3-540-45142-0_9

Gilberto Filé, Roberto Giacobazzi, and Francesco Ranzato. 1996. A Unifying View of Abstract Domain Design. *ACM Comput. Surv.* 28, 2 (1996), 333–336. https://doi.org/10.1145/234528.234742

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 3–18. https://doi.org/10.1109/SP.2018.00058

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*, Sriram K. Rajamani and David Walker (Eds.). ACM, 261–273. https://doi.org/10.1145/2676726.2676987

Roberto Giacobazzi and Elisa Quintarelli. 2001. Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2126)*, Patrick Cousot (Ed.). Springer, 356–373. https://doi.org/10.1007/3-540-47764-0_20

Roberto Giacobazzi and Francesco Ranzato. 1996. Complementing Logic Program Semantics. In *Algebraic and Logic Programming, 5th International Conference, ALP'96, Aachen, Germany, September 25-27, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1139)*, Michael Hanus and Mario Rodríguez-Artalejo (Eds.). Springer, 238–253. https://doi.org/10.1007/3-540-61735-3_16

Roberto Giacobazzi and Francesco Ranzato. 1998. Uniform Closures: Order-Theoretically Reconstructing Logic Program Semantics and Abstract Domain Refinements. *Inf. Comput.* 145, 2 (1998), 153–190. https://doi.org/10.1006/INCO.1998.2724

Roberto Giacobazzi and Francesco Ranzato. 2025. The Best of Abstract Interpretations. *Proc. ACM Program. Lang.* 9, POPL (2025), 1355–1385. https://doi.org/10.1145/3704882

Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. 2000. Making abstract interpretations complete. *J. ACM* 47, 2 (2000), 361–416. https://doi.org/10.1145/333979.333989

Antoine Girard. 2005. Reachability of Uncertain Linear Systems Using Zonotopes. In *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3414)*, Manfred Morari and Lothar Thiele (Eds.). Springer, 291–305. https://doi.org/10.1007/978-3-540-31954-2_19

Branko Grünbaum, Victor Klee, Micha A Perles, and Geoffrey Colin Shephard. 1967. *Convex polytopes*. Vol. 16. Springer.

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

Aurélie Hurault and João Marques-Silva. 2023. Certified Logic-Based Explainable AI - The Case of Monotonic Classifiers. In *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14066)*, Virgile Prevosto and Cristina Seceleanu (Eds.). Springer, 51–67. https://doi.org/10.1007/978-3-031-38828-6_4

Keith J. C. Johnson, Rahul Krishnan, Thomas W. Reps, and Loris D'Antoni. 2024. Automating Pruning in Top-Down Enumeration for Program Synthesis Problems with Monotonic Semantics. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 935–961. https://doi.org/10.1145/3689744

Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (1997), 427–443. https://doi.org/10.1145/256167.256195

Dennis Liew, Tiago Cogumbreiro, and Julien Lange. 2024. Sound and Partially-Complete Static Analysis of Data-Races in GPU Programs. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 2434–2461. https://doi.org/10.1145/3689797

Francesco Logozzo. 2009. Towards a Quantitative Estimation of Abstract Interpretations. In *Workshop on Quantitative Analysis of Software* (workshop on quantitative analysis of software ed.). Microsoft. https://www.microsoft.com/en-us/research/publication/towards-a-quantitative-estimation-of-abstract-interpretations/

João Marques-Silva, Thomas Gerspacher, Martin C. Cooper, Alexey Ignatiev, and Nina Narodytska. 2021. Explanations for Monotonic Classifiers. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 7469–7479. http://proceedings.mlr.press/v139/marques-silva21a.html

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

Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, Johannes Fürnkranz and Thorsten Joachims (Eds.). Omnipress, 807–814. https://icml.cc/Conferences/2010/papers/432.pdf

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

Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 157–168. https://doi.org/10.1145/1863543.1863568

Xavier Rival and Kwangkeun Yi. 2020. *Introduction to static analysis: an abstract interpretation perspective.* Mit Press.

Walter Rudin. 1976. *Principles of Mathematical Analysis* (3rd ed.). McGraw-Hill.

Pascal Sotin. 2010. *Quantifying the precision of numerical abstract domains.* Technical Report HAL Id: inria-00457324. INRIA. https://hal.inria.fr/inria-00457324

Wallace Alvin Wilson. 1931. On quasi-metric spaces. *American Journal of Mathematics* 53, 3 (1931), 675–684. https://doi.org/10.2307/2371174

Günter M Ziegler. 2012. *Lectures on polytopes.* Vol. 152. Springer Science & Business Media.

A PROOFS OMITTED FROM THE MAIN TEXT

Proof of Proposition 2.2.

$$
\begin{aligned}
\alpha(c) = a &\ \Rightarrow \\
\alpha(c) \sqsubseteq_{\mathcal{A}} a &\ \Leftrightarrow \text{[by Definition 2.1]} \\
c \sqsubseteq_{\boldsymbol{C}} \gamma(a) &\ \Rightarrow \text{[by } \alpha \text{ order-preserving]} \\
\alpha(c) \sqsubseteq_{\mathcal{A}} \alpha\gamma(a) &\ \Rightarrow \text{[by } \alpha(c) = a\text{]} \\
a \sqsubseteq_{\mathcal{A}} \alpha\gamma(a) &\ \Rightarrow \text{[by } \alpha\gamma \text{ reductive and } \sqsubseteq_{\mathcal{A}} \text{ anti-symmetric]} \\
\alpha\gamma(a) = a &
\end{aligned}
$$

□

Proof of Proposition 3.3. $(i) - (ii)$ immediate by Definition 3.1.
(iii) Assume $\delta_{\mathcal{A}}$ satisfies (*chain iff-identity*). It follows:

$$
\begin{aligned}
\mathbb{C}^{0}_{c}(f, f^{\sharp}) &\ \Leftrightarrow \text{[by Definition 3.1]} \\
\delta_{\mathcal{A}}(\alpha(f(c)), f^{\sharp}(\alpha(c))) \leq 0 &\ \Leftrightarrow \text{[by } f^{\sharp} \text{ sound and (\textit{chain iff-identity}) of } \delta_{\mathcal{A}}\text{]} \\
\alpha(f(c)) = f^{\sharp}(\alpha(c)) &\ \Leftrightarrow \text{[by Definition 3.1]} \\
\mathbb{C}_{c}(f, f^{\sharp}) &
\end{aligned}
$$

□

Proof of Proposition 4.4. (i) By definition of the bottom element, $\bot_{\boldsymbol{C}}$ is the minimum in $\boldsymbol{C}$ therefore a generator of $\alpha(\bot_{\boldsymbol{C}})$.
(ii) $a \in \mathcal{A}$ generable implies there exists $g \in \mathscr{G}(a)$ such that $\alpha(g) = a$, then, by Proposition 2.2, $\alpha\gamma(a) = a$.
(iii) $\exists! c \in \boldsymbol{C}$ means that c is the only concrete element such that $\alpha(c) = a$, therefore it is also (the only) generator of $\alpha(c)$. □

Lemma A.1. *Given a GC* $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$, *let* $f^{\sharp} \colon \mathcal{A} \longrightarrow \mathcal{A}$ *be a sound approximation of* $f \colon \boldsymbol{C} \longrightarrow \boldsymbol{C}$. *Then* $\alpha \circ f \sqsubseteq_{\mathcal{A}} f^{\sharp} \circ \alpha$ *holds.*

Proof. Starting from the soundness definition, we get:

$$
\begin{aligned}
\forall a \in \mathcal{A}.\, f(\gamma(a)) \sqsubseteq_{\mathcal{A}} \gamma(f^{\sharp}(a)) &\ \Rightarrow \text{[by Definition 2.1]} \\
\forall c \in \boldsymbol{C}.\, f(\gamma\alpha(c)) \sqsubseteq_{\boldsymbol{C}} \gamma(f^{\sharp}(\alpha(c))) &\ \Rightarrow \text{[by } \alpha \text{ order-preserving]} \\
\forall c \in \boldsymbol{C}.\, \alpha(f(\gamma\alpha(c))) \sqsubseteq_{\mathcal{A}} \alpha\gamma(f^{\sharp}(\alpha(c))) &\ \Rightarrow \text{[by } f \text{ order-preserving} \\
&\ \qquad \text{and } \gamma\alpha \text{ extensive]} \\
\forall c \in \boldsymbol{C}.\, \alpha(f(c)) \sqsubseteq_{\mathcal{A}} \alpha(f(\gamma\alpha(c))) \sqsubseteq_{\mathcal{A}} \alpha\gamma(f^{\sharp}(\alpha(c))) &\ \Rightarrow \text{[by } \alpha\gamma \text{ reductive]} \\
\forall c \in \boldsymbol{C}.\, \alpha(f(c)) \sqsubseteq_{\mathcal{A}} \alpha(f(\gamma\alpha(c))) \sqsubseteq_{\mathcal{A}} \alpha\gamma(f^{\sharp}(\alpha(c))) \sqsubseteq_{\mathcal{A}} f^{\sharp}(\alpha(c)) &\ \Rightarrow \text{[by transitivity of } \sqsubseteq_{\mathcal{A}}\text{]} \\
\forall c \in \boldsymbol{C}.\, \alpha(f(c)) \sqsubseteq_{\mathcal{A}} f^{\sharp}(\alpha(c)) &
\end{aligned}
$$

□

Proof of Theorem 2.6. Assume we have a GC $\langle \boldsymbol{C}, \sqsubseteq_{\boldsymbol{C}} \rangle \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$ and let $f^{\sharp} \colon \mathcal{A} \longrightarrow \mathcal{A}$ be a sound approximation of $f \colon \boldsymbol{C} \longrightarrow \boldsymbol{C}$. Consider $p \in \boldsymbol{C}$ such that $\gamma\alpha(p) = p$.

Proof of (1):

$$\begin{aligned}
\forall c \in \boldsymbol{C}.\, f^\sharp(\alpha(c)) \sqsubseteq_{\mathcal{A}} \alpha(p) &\Rightarrow \text{[by Lemma A.1]}\\
\forall c \in \boldsymbol{C}.\, \alpha(f(c)) \sqsubseteq_{\mathcal{A}} f^\sharp(\alpha(c)) \sqsubseteq_{\mathcal{A}} \alpha(p) &\Rightarrow \text{[by } \gamma \text{ order-preserving]}\\
\forall c \in \boldsymbol{C}.\, \gamma\alpha(f(c)) \sqsubseteq_{C} \gamma(f^\sharp(\alpha(c))) \sqsubseteq_{C} \gamma\alpha(p) &\Rightarrow \text{[by } \gamma\alpha \text{ extensive]}\\
\forall c \in \boldsymbol{C}.\, f(c) \sqsubseteq_{C} \gamma\alpha(f(c)) \sqsubseteq_{C} \gamma(f^\sharp(\alpha(c))) \sqsubseteq_{C} \gamma\alpha(p) &\Rightarrow \text{[by } \gamma\alpha(p) = p\text{]}\\
\forall c \in \boldsymbol{C}.\, f(c) \sqsubseteq_{C} \gamma\alpha(f(c)) \sqsubseteq_{C} \gamma(f^\sharp(\alpha(c))) \sqsubseteq_{C} p &\Rightarrow \text{[by transitivity of } \sqsubseteq_{\mathcal{A}}\text{]}\\
\forall c \in \boldsymbol{C}.\, f(c) \sqsubseteq_{C} p &
\end{aligned}$$

Assume the predicate $\mathbb{C}(f, f^\sharp)$ holds. Proof of (2):

$$\begin{aligned}
\forall c \in \boldsymbol{C}.\, f(c) \sqsubseteq_{C} p &\Rightarrow \text{[by } \alpha \text{ order-preserving]}\\
\forall c \in \boldsymbol{C}.\, \alpha(f(c)) \sqsubseteq_{\mathcal{A}} \alpha(p) &\Rightarrow \text{[by } \mathbb{C}(f, f^\sharp)\text{]}\\
\forall c \in \boldsymbol{C}.\, \alpha(f(c)) = f^\sharp(\alpha(c)) \sqsubseteq_{\mathcal{A}} \alpha(p) &
\end{aligned}$$

□

Proof of Theorem 5.1. Consider any input $c \in [g, \gamma(a)]$. Note that, by Proposition 4.4 and since a is generable by assumption, $\alpha\gamma(a) = a$. We prove first that if $f^\sharp$ is local $\boldsymbol{e}$-partial complete at g for the concrete function f (i.e., the predicate $\mathbb{C}^{\boldsymbol{e}}_g(f, f^\sharp)$ holds), then $f^\sharp$ is also local $\boldsymbol{e}$-partial complete at c for f, i.e., $\mathbb{C}^{\boldsymbol{e}}_c(f, f^\sharp)$. We get the following implications:

$$\begin{aligned}
c \in [g, \gamma(a)] &\Rightarrow \text{[by } [g, \gamma(a)] \overset{\text{def}}{=} \{ i \in \boldsymbol{C} \mid g \sqsubseteq_C i \sqsubseteq_C \gamma(a)\ \}\text{]}\\
g \sqsubseteq_C c &\Rightarrow \text{[by } f \text{ order-preserving]}\\
f(g) \sqsubseteq_C f(c) &\Rightarrow \text{[by } \alpha \text{ order-preserving]}\\
\alpha(f(g)) \sqsubseteq_{\mathcal{A}} \alpha(f(c)) &\Rightarrow \text{[by } f^\sharp \text{ sound]}\\
\alpha(f(g)) \sqsubseteq_{\mathcal{A}} \alpha(f(c)) \sqsubseteq_{\mathcal{A}} f^\sharp(\alpha(c)) &\Rightarrow \text{[by (}\textit{chains-order}\text{) of } \delta_{\mathcal{A}}\text{]}\\
\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \delta_{\mathcal{A}}(\alpha(f(g)), f^\sharp(\alpha(c))) &\Rightarrow \text{[by } \alpha(c) = \alpha(g)\text{]}\\
\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \delta_{\mathcal{A}}(\alpha(f(g)), f^\sharp(\alpha(g))) &\Rightarrow \text{[by } \mathbb{C}^{\boldsymbol{e}}_g(f, f^\sharp)\text{]}\\
\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}(g) &\Rightarrow \text{[by setting } \boldsymbol{e}(c) = \boldsymbol{e}(g) \text{ and Definition 3.1]}\\
\mathbb{C}^{\boldsymbol{e}}_c(f, f^\sharp) &
\end{aligned}$$

Since c is taken arbitrarily in the set $[g, \gamma(a)]$, the predicate $\mathbb{C}^{\tilde{\boldsymbol{e}}}_{[g,\gamma(a)]}(f, f^\sharp)$ also holds for the function $\tilde{\boldsymbol{e}}$ where $\tilde{\boldsymbol{e}}(c) \overset{\text{def}}{=} \boldsymbol{e}(g)$ if $c \in [g, \gamma(a)]$, $\tilde{\boldsymbol{e}}(c) \overset{\text{def}}{=} \boldsymbol{e}(c)$ otherwise. □

Proof of Corollary 5.2. The proof follows directly from Theorem 5.1 and Proposition 3.3 by setting $\boldsymbol{e} = \boldsymbol{0}$ and $\delta_{\mathcal{A}}(x, y) = \delta^{=}_{\mathcal{A}}(x, y)$ of Example 2.8. This is because $\delta^{=}_{\mathcal{A}}(x, y)$ is strong, implying that, when $\delta^{=}_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq 0$, the equality $\alpha(f(c)) = f^\sharp(\alpha(c))$ holds. □

Lemma A.2. *Let $c \in \boldsymbol{C}$ and suppose $\mathscr{G}(\alpha(c))_{\sqsubseteq c} = \{g_1, g_2\}$. If both $\mathbb{C}^{\boldsymbol{e}}_{g_1}(f, f^\sharp)$ and $\mathbb{C}^{\boldsymbol{e}'}_{g_2}(f, f^\sharp)$ hold, then both inequalities $\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}(g_1)$ and $\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}'(g_2)$ hold.*

Proof. If the predicate $\mathbb{C}^{\boldsymbol{e}}_{g_1}(f, f^\sharp)$ holds, then by Theorem 5.1 and $\alpha(g_1)$ generable, the inequality $\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}(g_1)$ holds for all $c \in [g_1, \gamma\alpha(g_1)]$. Similarly, if the predicate $\mathbb{C}^{\boldsymbol{e}'}_{g_2}(f, f^\sharp)$ holds, then by Theorem 5.1 and $\alpha(g_2)$ generable, $\delta_{\mathcal{A}}(\alpha(f(c)), f^\sharp(\alpha(c))) \leq \boldsymbol{e}'(g_2)$ holds for all $c \in [g_2, \gamma\alpha(g_2)]$. □

Proof of Theorem 5.6. We start by proving (i). Consider $\delta_{\mathcal{A}}\colon \mathcal{A} \times \mathcal{A} \longrightarrow \mathbb{N}^{\infty}$, and assume $\mathbb{C}^{\boldsymbol{e}}_{\mathcal{G}}(f, f^{\sharp})$ holds, namely $f^{\sharp}$ is local $\boldsymbol{e}$-partial complete at all generators of each abstract property in $\mathcal{A}$. Let us consider a generator $g \in \mathcal{G}$. Then, by Theorem 5.1 and $\alpha(g)$ generable, $f^{\sharp}$ is also locally $\tilde{\boldsymbol{e}}$-partial complete at the set $[g, \gamma\alpha(g)]$ where $\tilde{\boldsymbol{e}}$ is defined as in Theorem 5.1. Consider now $c \in [g, \gamma\alpha(g)]$ and suppose $g' \in \mathscr{G}(\alpha(c))_{\sqsubseteq c}$ such that $g \neq g'$ and the predicate $\mathbb{C}^{\boldsymbol{e}'}_{g_2}(f, f^{\sharp})$ holds. Then by Lemma A.2, the inequality $\delta_{\mathcal{A}}(\alpha(f(c)), f^{\sharp}(\alpha(c))) \leq \boldsymbol{e}'(g_2)$ holds as well. This means that, given $c \in \boldsymbol{C}$ such that $\alpha(c)$ is generable, between all the generators of $\mathcal{G}$ that are approximated by c, that is $\mathscr{G}(\alpha(c))_{\sqsubseteq c} \subseteq \mathcal{G}$, we can select as upper bound of imprecision the generator $g \in \mathscr{G}(\alpha(c))_{\sqsubseteq c}$ that has the minimal bound according to $\boldsymbol{e}$, namely $min(\{\boldsymbol{e}(g) \mid g \in \mathscr{G}(\alpha(c))_{\sqsubseteq c}\})$. This minimal value exists since $\delta_{\mathcal{A}}$ is assumed to output only natural numbers. This process leads to the definition of $\hat{\boldsymbol{e}}$ which is then a bounding function for the set $\bigcup_{g \in \mathcal{G}}[g, \gamma\alpha(g)] = \bigcup_{g \in \mathcal{G}}\{c \in \boldsymbol{C} \mid g \leq_{\boldsymbol{C}} c \leq_{\boldsymbol{C}} \gamma\alpha(g)\}$. We show that $\bigcup_{g \in \mathcal{G}}[g, \gamma\alpha(g)] = C_{gen}$. Consider an element $c \in \bigcup_{g \in \mathcal{G}}[g, \gamma\alpha(g)]$. This means that there exists $\hat{g} \in \mathcal{G}$ such that $c \in [\hat{g}, \gamma\alpha(\hat{g})]$. But this implies that $\hat{g} \in \mathscr{G}(\alpha(c))_{\sqsubseteq c}$, namely, $c \in C_{gen}$. Now suppose $c \in C_{gen}$ and $\hat{g}$ is the generator such that $\hat{g} \in \mathscr{G}(\alpha(c))_{\sqsubseteq c}$. Then, by Definition 4.5, $\hat{g} \leq_{\boldsymbol{C}} c \leq_{\boldsymbol{C}} \gamma\alpha(c) = \gamma\alpha(\hat{g})$, namely, $c \in [\hat{g}, \gamma\alpha(\hat{g})]$, and therefore $c \in \bigcup_{g \in \mathcal{G}}[g, \gamma\alpha(g)]$. Thus $\hat{\boldsymbol{e}}$ is also a bounding function for C_{gen}.

Proof of (ii) follows immediately by Lemma A.2. □

Proof of Corollary 5.7. Similar to Corollary 5.2. □

Proof of Theorem 6.7. (**Base**$_\omega$): Follows directly by the definition of the predicate ω-Cont(c)$_{\mathcal{L}}$.
(**Weaken**$_\omega$):

$$\begin{aligned}
&\omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}} && \Leftrightarrow \text{[by Definition 6.2]}\\
\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(f(l_1), f(l_2)) \leq{}& \omega(\delta_{\mathcal{L}}(l_1, l_2)) && \leq \text{[by } \omega \leq \omega'\text{]}\\
&\omega'(\delta_{\mathcal{L}}(l_1, l_2)) && \Leftrightarrow \text{[by Definition 6.2]}\\
&\omega'\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}
\end{aligned}$$

(**Seq**$_\omega$):

$$\begin{aligned}
&\omega_1\text{-}\textsc{Cont}(\mathsf{P}_1)_{\mathcal{L}} && \Leftrightarrow \text{[by Definition 6.2]}\\
\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\llbracket \mathsf{P}_1 \rrbracket_{\mathcal{L}}(l_1), \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{L}}(l_2)) \leq{}& \omega_1(\delta_{\mathcal{L}}(l_1, l_2)) && \Rightarrow \text{[by } \omega_2\text{-}\textsc{Cont}(\mathsf{P}_2)_{\mathcal{L}}\text{]}\\
\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\llbracket \mathsf{P}_2 \rrbracket_{\mathcal{L}}\llbracket \mathsf{P}_1 \rrbracket_{\mathcal{L}}(l_1), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{L}}\llbracket \mathsf{P}_1 \rrbracket_{\mathcal{L}}(l_2)) \leq{}& \omega_2(\omega_1(\delta_{\mathcal{L}}(l_1, l_2))) && \Leftrightarrow \text{[by definition of } \llbracket \mathsf{P}_1; \mathsf{P}_2 \rrbracket_{\mathcal{L}}\text{]}\\
\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\llbracket \mathsf{P}_1; \mathsf{P}_2 \rrbracket_{\mathcal{L}}(l_1), \llbracket \mathsf{P}_1; \mathsf{P}_2 \rrbracket_{\mathcal{L}}(l_2)) \leq{}& \omega_2(\omega_1(\delta_{\mathcal{L}}(l_1, l_2))) && \Leftrightarrow \text{[by Definition 6.2]}\\
&\omega_1 \circ \omega_2\text{-}\textsc{Cont}(\mathsf{P}_1; \mathsf{P}_2)_{\mathcal{L}}
\end{aligned}$$

(**Join**$_\omega$):

$$\begin{aligned}
\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\llbracket \mathsf{P}_1 \oplus \mathsf{P}_2 \rrbracket_{\mathcal{L}}(l_1), \llbracket \mathsf{P}_1 \oplus \mathsf{P}_2 \rrbracket_{\mathcal{L}}(l_2)) && = \text{[by definition of } \llbracket \mathsf{P}_1 \oplus \mathsf{P}_2 \rrbracket_{\mathcal{L}}\text{]}\\
\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\llbracket \mathsf{P}_1 \rrbracket_{\mathcal{L}}(l_1) \sqcup_{\mathcal{L}} \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{L}}(l_1), \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{L}}(l_2) \sqcup_{\mathcal{L}} \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{L}}(l_2)) && \leq \text{[by } \oplus_{\delta_{\mathcal{L}}}, \omega_1\text{-}\textsc{Cont}(\mathsf{P}_1)_{\mathcal{L}}, \omega_2\text{-}\textsc{Cont}(\mathsf{P}_2)_{\mathcal{L}}\text{]}\\
\oplus_{\delta_{\mathcal{L}}}(\omega_1(\delta_{\mathcal{L}}(l_1, l_2)), \omega_2(\delta_{\mathcal{L}}(l_1, l_2)) && \Leftrightarrow \text{[by } \omega_{\sqcup}(t) = \oplus_{\delta_{\mathcal{L}}}(\omega_1(t), \omega_2(t))\\
&& \text{and Definition 6.2]}\\
\omega_{\sqcup}\text{-}\textsc{Cont}(\mathsf{P}_1 \oplus \mathsf{P}_2)_{\mathcal{L}}
\end{aligned}$$

(**Iterate**$_\omega$): By assuming ω-Cont(P)$_{\mathcal{L}}$, we want to prove that the following inequality holds:

$$\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\llbracket \mathsf{P}^* \rrbracket_{\mathcal{L}} l_1, \llbracket \mathsf{P}^* \rrbracket_{\mathcal{L}} l_2) \leq \omega_*(\delta_{\mathcal{L}}(l_1, l_2))$$

By definition of $\llbracket \mathsf{P}^* \rrbracket_{\mathcal{L}}$ and ω_*, this corresponds to prove that $\forall n \in \mathbb{N}$:

$$\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\bigsqcup_{i=0}^{n}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_1\}, \bigsqcup_{i=0}^{n}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_2\}) \leq \bigoplus_{i=0}^{n} \omega^i(\delta_{\mathcal{L}}(l_1, l_2)) \tag{3}$$

The proof is made by induction on the iteration steps n.

Base case $n = 0$: it follows that $\forall l_1, l_2 \in \mathcal{L}$:

$$\begin{aligned}
\delta_{\mathcal{L}}(\bigsqcup_{i=0}^{0}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_1\}, \bigsqcup_{i=0}^{0}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_2\}) &= \\
\delta_{\mathcal{L}}(\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{0} l_1, \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{0} l_2) &= [\text{by } \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{0} l = l] \\
\delta_{\mathcal{L}}(l_1, l_2) &= [\text{by } \omega^0(t) = t] \\
\omega^0(\delta_{\mathcal{L}}(l_1, l_2)) &= \\
\bigoplus_{i=0}^{0} \omega^i(\delta_{\mathcal{L}}(l_1, l_2))
\end{aligned}$$

thus the base case holds.

Inductive step: suppose (3) holds for an arbitrary natural number k, namely:

$$\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\bigsqcup_{i=0}^{k}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_1\}, \bigsqcup_{i=0}^{k}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_2\}) \leq \bigoplus_{i=0}^{k} \omega^i(\delta_{\mathcal{L}}(l_1, l_2)) \tag{IH}$$

We now prove that (3) also holds for $k + 1$. We first prove the following inequality

$$\forall l_1, l_2 \in \mathcal{L}.\ \delta_{\mathcal{L}}(\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k+1} l_1, \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k+1} l_2) \leq \omega^{k+1}(\delta_{\mathcal{L}}(l_1, l_2))) \tag{ST}$$

as follows:

$$\begin{aligned}
\delta_{\mathcal{L}}(\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k+1} l_1, \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k+1} l_2) &= [\text{by definition of } \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k+1}] \\
\delta_{\mathcal{L}}(\llbracket \mathsf{P} \rrbracket_{\mathcal{L}} \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k} l_1, \llbracket \mathsf{P} \rrbracket_{\mathcal{L}} \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k} l_2) &\leq [\text{by } \omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}] \\
\omega(\delta_{\mathcal{L}}(\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k} l_1, \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k} l_2)) &\leq [\text{by applying } \omega\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}\ k \text{ times}] \\
\omega(\omega^k(\delta_{\mathcal{L}}(l_1, l_2))) &= [\text{by definition of } \omega^{k+1}] \\
\omega^{k+1}(\delta_{\mathcal{L}}(l_1, l_2))
\end{aligned}$$

It follows that $\forall l_1, l_2 \in \mathcal{L}$:

$$\begin{aligned}
\delta_{\mathcal{L}}(\bigsqcup_{i=0}^{k+1}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_1\}, \bigsqcup_{i=0}^{k+1}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_2\}) &= [\text{by definition of } \bigsqcup_{i=0}^{k+1}] \\
\delta_{\mathcal{L}}(\bigsqcup_{i=0}^{k}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_1\} \sqcup_{\mathcal{L}} \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k+1} l_1, \bigsqcup_{i=0}^{k}\{\llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{i} l_2\} \sqcup_{\mathcal{L}} \llbracket \mathsf{P} \rrbracket_{\mathcal{L}}^{k+1} l_2) &\leq [\text{by (IH), (ST) and } \oplus_{\mathcal{L}} \text{ join-bound}] \\
(\bigoplus_{i=0}^{k} \omega^i(\delta_{\mathcal{L}}(l_1, l_2))) \oplus_{\mathcal{L}} \omega^{k+1}(\delta_{\mathcal{L}}(l_1, l_2)) &= [\text{by definition of } \bigoplus_{i=0}^{k+1}] \\
(\bigoplus_{i=0}^{k+1} \omega^i(\delta_{\mathcal{L}}(l_1, l_2)))
\end{aligned}$$

This concludes that (3) also holds for $k + 1$. Since both the base case and the inductive step have been proved as true, by mathematical induction the statement (3) holds for every natural number $n \in \mathbb{N}$, thus we can conclude $\forall l_1, l_2 \in \mathcal{L}.\, \delta_{\mathcal{L}}(\llbracket \mathsf{P}^* \rrbracket_{\mathcal{L}} l_1, \llbracket \mathsf{P}^* \rrbracket_{\mathcal{L}} l_2) \leq \omega_*(\delta_{\mathcal{L}}(l_1, l_2))$, namely $\omega_*\text{-}\textsc{Cont}(\mathsf{P})_{\mathcal{L}}$ holds.

□

Proof of Theorem 6.9. (**Base**): Follows directly by Theorem 5.1.

(**Weaken**): Follows directly by Proposition 3.3.

(**Gen-Switch**): Let $c \in [g, \gamma\alpha(g)] \cap [h, \gamma\alpha(h)]$. Then, by $\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$, $\boldsymbol{e}'\text{-}\textsc{Bound}(\mathsf{P}, h)_{\mathcal{A}}$ and $\alpha(g) = \alpha(h)$, both $\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P} \rrbracket_{C} c), \llbracket \mathsf{P} \rrbracket_{\mathcal{A}} \alpha(c)) \leq \boldsymbol{e}(g)$ and $\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P} \rrbracket_{C} c), \llbracket \mathsf{P} \rrbracket_{\mathcal{A}} \alpha(c)) \leq \boldsymbol{e}'(h)$ holds. Thus we can define $\boldsymbol{e}''(c) = \boldsymbol{e}(g)$ as bounding function without altering the validity of the predicate.

(**Seq**): Inequality (a):

$$\begin{aligned}
\omega\text{-}\textsc{Cont}(\mathsf{P}_2)_{\mathcal{A}} &\Leftrightarrow [\text{by Definition 6.2}] \\
\forall a_1, a_2 \in \mathcal{A}.\, \delta_{\mathcal{A}}(\llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} a_1, \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} a_2) \leq \omega(\delta_{\mathcal{A}}(a_1, a_2)) &\Rightarrow [\text{by } a_1 = \alpha(\llbracket \mathsf{P}_1 \rrbracket_{C} g) \text{ and } a_2 = \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{A}} \alpha(g)] \\
\delta_{\mathcal{A}}(\llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(\llbracket \mathsf{P}_1 \rrbracket_{C} g), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{A}} \alpha(g)) &\leq \\
\omega(\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P}_1 \rrbracket_{C} g), \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{A}} \alpha(g))) &\leq [\text{by } \boldsymbol{e}_1\text{-}\textsc{Bound}(\mathsf{P}_1, g)_{\mathcal{A}} \text{ and } \omega \text{ order-preserving}] \\
\omega(\boldsymbol{e}_1(g)) &
\end{aligned}$$

The two triples $[g]\mathsf{P}_1[h]$ and $\{g\}\mathsf{P}_1\{\gamma\alpha(h)\}$ correspond to the inequality $h \sqsubseteq_{C} \llbracket \mathsf{P}_1 \rrbracket_{C} g \sqsubseteq_{C} \gamma\alpha(h)$, thus $\llbracket \mathsf{P}_1 \rrbracket_{C} g \in [h, \gamma\alpha(h)]$. We then get the following inequality (b):

$$\begin{aligned}
\boldsymbol{e}_2\text{-}\textsc{Bound}(\mathsf{P}_2, h)_{\mathcal{A}} &\Leftrightarrow [\text{by Definitions 6.8 and 3.1}] \\
\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P}_2 \rrbracket_{C} h), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(h)) \leq \boldsymbol{e}_2(h) &\Rightarrow [\text{by Theorem 5.1}] \\
\forall c \in [h, \gamma\alpha(h)].\, \delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P}_2 \rrbracket_{C} c), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(c)) \leq \boldsymbol{e}_2(h) &\Rightarrow [\text{by } [g]\mathsf{P}_1[h] \text{ and } \{g\}\mathsf{P}_1\{\gamma\alpha(h)\}] \\
\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P}_2 \rrbracket_{C} \llbracket \mathsf{P}_1 \rrbracket_{C} g), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(\llbracket \mathsf{P}_1 \rrbracket_{C} g)) \leq \boldsymbol{e}_2(h) &
\end{aligned}$$

Finally, by considering the chain $\alpha(\llbracket \mathsf{P}_2 \rrbracket_{C} \llbracket \mathsf{P}_1 \rrbracket_{C} g) \sqsubseteq_{\mathcal{A}} \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(\llbracket \mathsf{P}_1 \rrbracket_{C} g) \sqsubseteq_{\mathcal{A}} \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{A}} \alpha(g)$ and by exploiting the (*triangle-inequality*) axiom of the strong pre-metric $\delta_{\mathcal{A}}$, we get the following derivations:

$$\begin{aligned}
\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P}_1; \mathsf{P}_2 \rrbracket_{C} g), \llbracket \mathsf{P}_1; \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(g)) &= [\text{by def of } \llbracket \mathsf{P}_1; \mathsf{P}_2 \rrbracket_{\mathcal{L}}] \\
\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P}_2 \rrbracket_{C} \llbracket \mathsf{P}_1 \rrbracket_{C} g), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{A}} \alpha(g)) &\leq [\text{by } \delta_{\mathcal{A}} \text{ strong}] \\
\delta_{\mathcal{A}}(\alpha(\llbracket \mathsf{P}_2 \rrbracket_{C} \llbracket \mathsf{P}_1 \rrbracket_{C} g), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(\llbracket \mathsf{P}_1 \rrbracket_{C} g)) + \delta_{\mathcal{A}}(\llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \alpha(\llbracket \mathsf{P}_1 \rrbracket_{C} g), \llbracket \mathsf{P}_2 \rrbracket_{\mathcal{A}} \llbracket \mathsf{P}_1 \rrbracket_{\mathcal{A}} \alpha(g)) &\leq [\text{by (b) and (a)}] \\
\boldsymbol{e}_2(h) + \omega(\boldsymbol{e}_1(g)) &\Leftrightarrow [\text{by Definition 6.8} \\
&\quad\ \text{and } \boldsymbol{e} \text{ as in Figure 4}] \\
\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}_1; \mathsf{P}_2, g)_{\mathcal{A}} &
\end{aligned}$$

(**Join**): The proof follows the same structure as that of proof of rule (**Join**$_{\omega}$) in Theorem 6.7.

(**Iterate**): Suppose the predicate $\boldsymbol{e}\text{-}\textsc{Bound}(\mathsf{P}, g)_{\mathcal{A}}$ holds as well as two triples $[g]\mathsf{P}^*[post]_{C}$ and $\{\alpha(g) \sqcup_{\mathcal{A}} inv\}\mathsf{P}\{\alpha(g) \sqcup_{\mathcal{A}} inv\}_{\mathcal{A}}$. We first prove that this last abstract correctness triple implies $\llbracket \mathsf{P}^* \rrbracket_{\mathcal{A}} \alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} inv$. This initial proof is made by induction on the iteration steps n by showing that $\forall n \in \mathbb{N}.\, \llbracket \mathsf{P} \rrbracket^{n}_{\mathcal{A}} \alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} inv$ which implies, by definition of $\sqcup_{\mathcal{A}}$, that also $\llbracket \mathsf{P}^* \rrbracket_{\mathcal{A}} \alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} inv$ holds.

Base case $n = 0$: $\llbracket \mathsf{P} \rrbracket^{0}_{\mathcal{A}} \alpha(g) = \alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} inv$ holds trivially.

Inductive step: suppose the statement holds for an arbitrary natural number k, namely (IH) $[\![\mathsf{P}]\!]^{k}_{\mathcal{A}}\alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}$. We prove that it also holds for $k+1$.

$$\begin{array}{rl}
[\![\mathsf{P}]\!]^{k+1}_{\mathcal{A}}\alpha(g) & = [\text{by definition of } [\![\mathsf{P}]\!]^{k+1}_{\mathcal{A}}] \\
[\![\mathsf{P}]\!]_{\mathcal{A}}[\![\mathsf{P}]\!]^{k}_{\mathcal{A}}\alpha(g) & \sqsubseteq_{\mathcal{A}} [\text{by (IH) and } [\![\mathsf{P}]\!]_{\mathcal{A}} \text{ order-preserving}] \\
[\![\mathsf{P}]\!]_{\mathcal{A}}(\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}) & \sqsubseteq_{\mathcal{A}} [\text{by } \{\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}\}\mathsf{P}\{\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}\}_{\mathcal{A}}] \\
\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv} &
\end{array}$$

Since both the base case and the inductive step have been proved as true, by mathematical induction the statement $\forall n \in \mathbb{N}.\ [\![\mathsf{P}]\!]^{n}_{\mathcal{A}}\alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}$ holds for every natural number $n \in \mathbb{N}$, thus we can conclude $[\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}$.

It follows that:

$$\begin{array}{rl}
[g]\mathsf{P}^*[\mathit{post}]_{\mathcal{C}} & \Leftrightarrow [\text{by definition of } [g]\mathsf{P}^*[\mathit{post}]_{\mathcal{C}}] \\
\mathit{post} \sqsubseteq_{\mathcal{C}} [\![\mathsf{P}^*]\!]_{\mathcal{C}}g & \Rightarrow [\text{by } \alpha \text{ order-preserving}] \\
\alpha(\mathit{post}) \sqsubseteq_{\mathcal{A}} \alpha([\![\mathsf{P}^*]\!]_{\mathcal{C}}g) & \Rightarrow [\text{by } [\![\cdot]\!]_{\mathcal{A}} \text{ sound}] \\
\alpha(\mathit{post}) \sqsubseteq_{\mathcal{A}} \alpha([\![\mathsf{P}^*]\!]_{\mathcal{C}}g) \sqsubseteq_{\mathcal{A}} [\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(g) & \Rightarrow [\text{by } \{\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}\}\mathsf{P}\{\alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}\}_{\mathcal{A}}] \\
\alpha(\mathit{post}) \sqsubseteq_{\mathcal{A}} \alpha([\![\mathsf{P}^*]\!]_{\mathcal{C}}g) \sqsubseteq_{\mathcal{A}} [\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(g) \sqsubseteq_{\mathcal{A}} \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv} & \Rightarrow [\text{by } (\textit{chain-order}) \text{ of } \delta_{\mathcal{A}}] \\
\delta_{\mathcal{A}}(\alpha([\![\mathsf{P}^*]\!]_{\mathcal{C}}g), [\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(g)) \leq \delta_{\mathcal{A}}(\alpha(\mathit{post}), \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}) & \Leftrightarrow [\text{by Theorem 5.1}] \\
\forall c \in [g, \gamma\alpha(g)].\ \delta_{\mathcal{A}}(\alpha([\![\mathsf{P}^*]\!]_{\mathcal{C}}c), [\![\mathsf{P}^*]\!]_{\mathcal{A}}\alpha(c)) \leq \delta_{\mathcal{A}}(\alpha(\mathit{post}), \alpha(g) \sqcup_{\mathcal{A}} \mathit{inv}) & \Leftrightarrow [\text{by } \boldsymbol{e}\text{-Bound}(\mathsf{P}, g)_{\mathcal{A}} \text{ and} \\
& \quad \bar{e} \text{ as in Figure 4 and Definition 3.1}] \\
\mathbb{C}^{\bar{e}}_{[g,\gamma\alpha(g)]}([\![\mathsf{P}^*]\!]_{\mathcal{C}}, [\![\mathsf{P}^*]\!]_{\mathcal{A}}) & \Leftrightarrow [\text{by Definition 6.8}] \\
\bar{e}\text{-Bound}(\mathsf{P}^*, g)_{\mathcal{A}} &
\end{array}$$

□