Limits and Difficulties in the Design of Under-Approximation Abstract Domains

FLAVIO ASCARI, ROBERTO BRUNI, and ROBERTA GORI, Università di Pisa, Pisa, Italy

The main goal of most static analyses is to prove the *absence of bugs*: if the analysis reports no alarms, then the program will not exhibit any unwanted behaviours. For this reason, they are designed to over-approximate program behaviours and, consequently, they can report some *false* alarms. O'Hearn's recent work on incorrectness has renewed the interest in the use of under-approximations for *bug finding*, because they only report *true* alarms. In principle, Abstract Interpretation techniques can handle under-approximations as well as over-approximations, but, in practice, few attempts were developed for the former, notwithstanding the much wider literature on the latter. In this article, we investigate the possibility of exploiting under-approximation abstract domains for bug-finding analyses. First we restrict to consider concrete powerset domains and highlight some intuitive asymmetries between over- and under-approximations. Then, we prove that the effectiveness of abstract domains defined by under-approximation Galois connection is limited because the analysis is likely to return trivial results whenever common transfer functions are encoded in the program. To this aim, we introduce the original concepts of *non-emptying functions* and *highly surjective function family*, and we prove the nonexistence of abstract domains able to under-approximate such functions in a non-trivial way. We show many examples of finite and infinite numerical domains, as well as other generic domains. In all such cases, we prove the impossibility of performing non-trivial analyses via under-approximation Galois connections.

CCS Concepts: • **Theory of computation** → **Abstraction**; **Program analysis**;

Additional Key Words and Phrases: Abstract Interpretation, under-approximation, abstract domains, impossibility results

ACM Reference format:
Flavio Ascari, Roberto Bruni, and Roberta Gori. 2024. Limits and Difficulties in the Design of Under-Approximation Abstract Domains. *ACM Trans. Program. Lang. Syst.* 46, 3, Article 11 (October 2024), 31 pages.
https://doi.org/10.1145/3666014

1 Introduction

Static program analyses aim to infer useful program properties by inspecting the source code but without its runtime execution. Starting from the seminal works of Floyd and Hoare [21, 25], many formal approaches have been proposed and successfully applied for over 50 years [7, 10,

This research has been supported by the Italian Ministero dell'Università e della Ricerca under Grant No. P2022HXNSC, PRIN 2022 PNRR–RAP (Resource Awareness in Programming: Algebra, Rewriting, and Analysis) and by the 2023-24 INdAM–GNCS project CUP_E53C22001930001.
Authors' Contact Information: Flavio Ascari (Corresponding author), Dipartimento di Informatica, Università di Pisa, Pisa, Italy; e-mail: flavio.ascari@phd.unipi.it; Roberto Bruni, Dipartimento di Informatica, Università di Pisa, Pisa, Italy; e-mail: roberto.bruni@unipi.it; Roberta Gori, Dipartimento di Informatica, Università di Pisa, Pisa, Italy; e-mail: roberta.gori@unipi.it.

ACM 1558-4593/2024/10-ART11
https://doi.org/10.1145/3666014

11, 19, 33, 34, 38, 39]. Nowadays, numerous effective methods and tools are available to support the development of correct software [5, 19]. They all rely on the key idea that for proving the absence of bugs we need to consider over-approximations of the program semantics: the absence of unwanted behaviour in the over-approximation guarantees the correctness of the program. Over-approximations consider supersets of all possible behaviours and are necessary to reduce the complexity of the analysis over the set of concrete values manipulated by the program. For example, the set of all possible integer values assumed by a program variable can be over-approximated by the smallest interval that contains them, which can be easily represented by a single pair of values (the extremes of the interval). Early works on static analysis, like Hoare logic [25], focused on over-approximation to prove the absence of errors, and maybe their influence caused the bias of interest towards over-approximations. However, over-approximations are not so effective to highlight bugs, since any alert found in the over-approximation computed by the analyser may be caused by the over-approximation itself rather than by the program. In this case, we call them false alarms. It is left to the programmer to distinguish between false and true alarms among those reported by the analyser. From the point of view of a software developer, false alarms are undesirable because they undermine the credibility and usefulness of the analysis [6, 35]. Tools for bug catching are essential and their quality depends on the timely detection of true alarms as well as the absence of noise created by false alarms.

Recently, O'Hearn proposed to shift the attention of formal methods from correctness to bug catching by introducing Incorrectness Logic [35], a dual version of Hoare logic thought from the ground up for under-approximation rather than over-approximation. Under-approximations can then expose defects in the code while they are unable to show their absence. Consider the simple code

```
for(i = 0; i < 5; ++i) sum += 1000 / (2 * i) + 100 / (2 * i - 5).
```

An abstract analysis based on the domain Int of intervals allows to over-approximate the set of possible values each variable can take as the smallest interval that contains such values. When applied to the above program, the analysis may detect that the value of variable `i` ranges between 0 and 4 within the body of the loop, so that the arithmetic expression `2 * i` is then over-approximated by the interval $[0, 8]$ while `2 * i - 5` by the interval $[-5, 3]$. Consequently, two warnings for possible division by zero are raised, since it seems that both denominators may assume the value 0. It is worth noting that while the warning on the first expression is a true alarm, the warning on the second one is a false alarm. On the contrary, an analysis based on under-approximation will never raise a warning for the second expression since no value of `i` can cause an error in this case. However, note that not all under-approximations will detect the problem with `2 * i`, because any subset of $\{0, 2, 4, 6, 8\}$ is a valid under-approximation, including, e.g., $\{2, 4, 6, 8\}$.

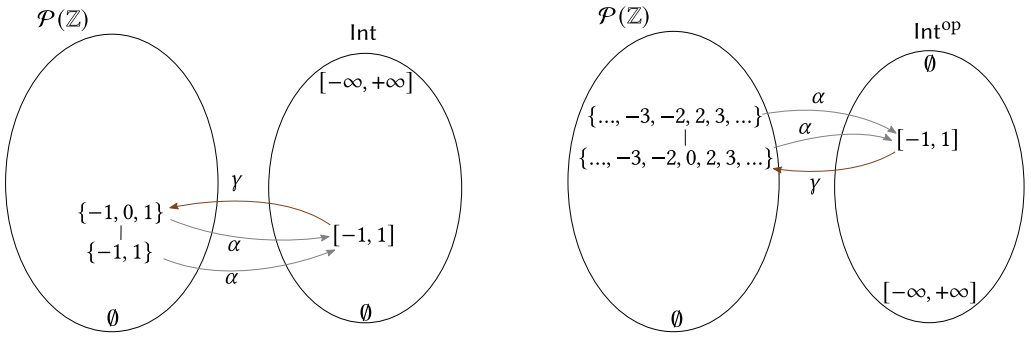
The Problem. Abstract Interpretation [11, 12, 38] is a general framework to define sound analyses based on constructive approximations that found its way through many aspects of modern computer science, such as verification, optimization, security and program transformation (see, e.g., the recent surveys [15, 22]). In his article on Incorrectness Logic, Peter O'Hearn leaves as an open question whether Abstract Interpretation could "*eventually play a guiding and explanatory role for a wide range of static and dynamic under-approximate tools for bug catching, similar to what it already does for over-approximate analyses*" [35, §8]. We took inspiration from his question to investigate the possibility of exploiting under-approximation abstract domains for bug-finding purposes, such as reachability analysis. However, in the end we proved that, under certain hypotheses, the effectiveness of **Galois connection (GC)**-based under-approximation abstract domains is limited, because the analysis is likely to return trivial results.

Related Work. In their first work on Abstract Interpretation [12], Patrick and Radhia Cousot introduced the formal theory that could be used to study both over- and under-approximations. However, while the former has been extensively studied, there are only sparse studies on under-approximation abstract domain. Bourdoncle [8] proposed abstract debugging using over-approximation domains but acknowledged that under-approximation ones could be better suited. Lev-Ami et al. [28] proposed to use complements of over-approximation domains to infer sufficient preconditions for program correctness. However, such an approach is severely limited in proving incorrectness, as we show in Example 3.2. For the same goal, Miné [31] used directly over-approximation domains, giving up the best abstraction and handling the choice of a maximal one with heuristics. To infer necessary conditions for incorrectness, Cousot et al. [16, 17] use Abstract Interpretation techniques but on Boolean formulas, hence bypassing the issue of defining an under-approximation abstract domain. Schmidt [40] uses higher-order domains, defining abstract states with the meaning "there exists a value satisfying this over-approximation property," hence giving rise to an under-approximation of over-approximations. In conclusion, all the above approaches design under-approximation domains starting from over-approximation ones, and, to the extent of our knowledge, there are no abstract domains thought from the ground up for under-approximation. It is worth noting that while the approach based on under-approximation abstract domains has not been studied extensively, there are works that successfully combine over-approximation abstract domains with other under-approximation techniques, not based on **under-approximation GCs (UGC)**, e.g., combining SMT-based and abstract model checking [1, 2, 26] or over-approximation abstract domains and Incorrectness Logic [9].

Contributions. In the context of program analysis over powerset domains of concrete values, we consider the problem of defining meaningful under-approximation abstract domains under the following assumptions: (1) abstract analyses should return non-trivial results for large classes of programs; and (2) to justify the convenience of the abstract analysis, the abstract domain should be significantly "smaller" than the concrete powerset. First, we point out some intuitive reasons that break the symmetry with over-approximation and make this a challenging task. Then, building on these observations, we formally derive some negative results that are applicable to a large class of instances.

From a purely mathematical point of view, the theories of over- and under-approximation are dual. In fact, any over-approximation domain can be turned into an under-approximation by reversing the order of its elements and complementing their interpretation. We call this construction *complement domain*. As an example, consider the (over-approximation) interval domain, where, e.g., the interval $[-1, 1]$ is a correct abstraction for any subset of $\{-1, 0, 1\}$ and is the best abstraction of $\{-1, 1\}$ and $\{-1, 0, 1\}$ (Figure 1(a)). Instead, in the complement domain, the interval $[-1, 1]$ is a correct abstraction of any set containing all values strictly smaller than -1 and all values strictly greater than 1 and is the best abstraction of $\{\dots, -3, -2, 0, 2, 3, \dots\}$ and $\{\dots, -3, -2, 2, 3, \dots\}$ (Figure 1(b)). Note that, being an under-approximation, $[-1, 1]$ represents correctly any set *larger* than its concretization $\{\dots, -3, -2, 2, 3, \dots\}$. However, we argue that complement domains are not useful for analysis: initializations such as `i := 0` or `i := 1000` are abstracted to the interval $[-\infty, \infty]$, which is the best abstraction of any finite set but loses any information about the initial value of `i`. We give more details on complement domains in Example 3.2.

Another important asymmetry we point out is the handling of divergence. In both over- and under-approximation, divergence is represented by the bottom element $\bot$ of the abstract domain. However, $\bot$ as an under-approximation also represents the absence of information; dually, in over-approximation this is described by $\top$. This becomes a problem since many concrete functions are strict, i.e., when applied to a non-terminating expression, they also fail to terminate (they



(a) Over-approximation intervals domain. (b) Under-approximation, complemented interval domain.

Fig. 1. Example of complemented domain, using intervals.

return $\bot$ if one argument is $\bot$), and, to be a correct under-approximation, also the corresponding abstract function needs to be strict. This implies that whenever the analysis cannot determine any meaningful information at some program point, it has to propagate the absence of information along all program paths, at least until a join in the control flow is found. So "recovery" from $\bot$, i.e., producing a result different from $\bot$, once we start with it, is very hard in an under-approximation. On the contrary, "recovery" from $\top$ in an over-approximation is quite easier: e.g., this can happen whenever the code contains a constant assignment.

A last asymmetry is that over-approximation abstract domains are closed under intersection, while under-approximation abstract domains are closed under union. In the case of constant assignments, this asymmetry has serious consequences. While the result of an assignment can be over-approximated by any larger set of values, with different degrees of precision, the only admissible under-approximations are either the singleton or the empty set. If not enough singletons are represented, then the under-approximation analysis is likely to give a trivial result. Vice versa, if too many singletons are represented, then the size of the under-approximation abstract domain will be increased exponentially, because it must be closed under union.

The above intuitive arguments led us to establish some negative results. The general theme is to fix some reasonable hypotheses over the common functions encoded by program fragments and then show that any under-approximation abstract domain with size not exponentially larger than the number of values (i.e., satisfying assumption (1)) will return no useful information for such program fragments. Since the analysis is unable to recover from this lack of information, the result of the analysis of the entire program will be trivial (i.e., violating assumption (2)). Therefore, while any abstract domain for under-approximate reasoning may be effective for some carefully crafted programs, it will return trivial results on the majority of programs.

Formally, we first introduce the new definition of *non-emptying function*, describing functions that do not tamper the analysis (see Definition 4.1). Roughly speaking, a non-emptying function is a function on the concrete domain that admits an under-approximation whose consecutive applications would not waste the analysis result by returning $\bot$. Our first result proves that no abstract domain for integers can be constructed that makes all increments non-emptying. In other words, we prove that an analysis based on under-approximation domains would often report trivial information for programs that involve repeated increments. We do so both for the infinite domain $\mathcal{P}(\mathbb{Z})$ and a finite integer domain $\mathcal{P}([-N, N])$ for large N.

We then study how we can generalize these results to different concrete domains and function families. We first focus on the case where the concrete domain is the powerset of an *infinite*

Result	Concrete domain	Tight hypotheses	Generalizes
Proposition 5.3	$\mathcal{P}(\mathbb{Z})$	–	–
Theorem 6.2	$\mathcal{P}(C)$	High surjectivity	Proposition 5.3
Theorem 6.6	$\mathcal{P}(C)$	High surjectivity	Proposition 5.3

(a) Tabular comparison of our results for infinite domains.

Result	Concrete domain	Tight hypotheses	Generalizes	Inspired by
Proposition 5.6	$\mathcal{P}([-N, N])$	–	–	–
Theorem 7.1	$\mathcal{P}(C)$	None	Proposition 5.6	Theorem 6.6

(b) Tabular comparison of our results for finite domains.

Fig. 2. Summary of our results showing that, under certain hypotheses, there exists no under-approximation abstract domain making all functions in a given family non-emptying. The tables show which concrete domain they are applicable to, which of the hypotheses are (known to be) tight, and the relations between the results, giving a quick reference for comparison.

set of values, and we prove two results, one local and one global, for infinite concrete domains. To do so, we introduce the notion of *highly surjective function family* (see Definition 6.1), of which sums are an instance. The local condition applies to each function in the family, while the global condition is a property of the whole family. Contrary to the definition of non-emptying functions, the notion of highly surjective function family is independent of the abstract domain. Once again the main consequence of our results is that abstract analyses of programs involving the application of functions in the family will often report trivial information. Note that, as in the case of increments, highly surjective function families are commonly coded in programs. Finally, we show that the hypothesis of high surjectivity is tight by presenting mathematical constructions of abstract domains, making all functions in a family non-emptying. Our results for infinite domains are summarized in Figure 2(a): both results for an arbitrary infinite set C generalize the result for integers. The hypothesis of high surjectivity is tight, but we do not know about all the others.

Lastly, we focus on the powerset of a *finite* set of values. We discuss why a straightforward adaptation of the two results for the infinite case to the finite one was not possible, most notably a difference with the very definition of highly surjective function family. We then propose a single general result for this case, inspired by the global condition for the infinite case, but whose details are different. Our results for finite domains are summarized in Figure 2(b): our single result generalizes again the result for integers, but we were not able to prove any of our hypotheses tight.

Structure of the Article. In Section 2, we introduce the notation used in the rest of the article and recall the basics of Abstract Interpretation for over-approximations. In Section 3, we introduce the concept of UGC and use it to present, via examples, some differences between over- and under-approximation. In Section 4, we present the notion of *non-emptying function*, the key concept of our development. In Section 5, we apply our idea to the concrete domain of integers to show that, under some simple conditions, no under-approximation abstract domain can exist. In Section 6, we extend the result obtained for integers to general infinite concrete domains and function families. In Section 7, we focus on the case of finite concrete domains. Section 8 contains some concluding remarks and outlines future research directions.

This article is a revised and extended version of [3].

2 Background

Notation. We let $\mathcal{P}(S)$ denote the powerset of the set S and $\mathrm{id}_S : S \to S$ be the identity function on a set S. We omit subscripts when obvious from the context. If $f : S \to T$ is a function, then we overload the symbol f to denote also its additive extension $f : \mathcal{P}(S) \to \mathcal{P}(T)$ defined as $f(X) = \{f(x) \mid x \in X\}$ for any $X \subseteq S$. We say a function $f : S \to S$ is *acyclic* if, for any element $x \in S$ and any $n > 0$, we have $f^n(x) \neq x$, where f^n denotes composition of f with itself n times, i.e., $f^0 = \mathrm{id}$ and $f^{n+1} = f \circ f^n$. In ordered structures, such as posets and lattices, we usually denote the ordering with $\preceq$, least upper bounds (lubs) with $\sqcup$, greatest lower bounds (glbs) with $\sqcap$, least element with $\bot$, greatest element with $\top$. If $\preceq$ is an order relation, $\succeq$ is the opposite relation, defined as $s \succeq t$ if and only if $t \preceq s$. We write just S for the poset $(S, \preceq)$ whenever the order relation $\preceq$ is known from the context and we use S^{op} to denote the opposite poset $(S, \succeq)$: hence S^{op} denotes the same set as S, but equipped with the opposite ordering relation $\succeq$. Given a poset T and two functions $f, g : S \to T$, the notation $f \preceq g$ means that, for all $s \in S$, $f(s) \preceq_T g(s)$. Any powerset ordered by inclusion is a complete lattice. In this case, we use standard symbols $\subseteq$, $\cup$, and so on.

Abstract Interpretation. Abstract Interpretation [12, 13, 32] is a general framework to define sound-by-construction static analyses, with the main idea of approximating the program semantics on some abstract domain A instead of working on the concrete domain C. The main tool used to study Abstract Interpretation are Galois connections. Given two complete lattices C and A, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ defines a GC when

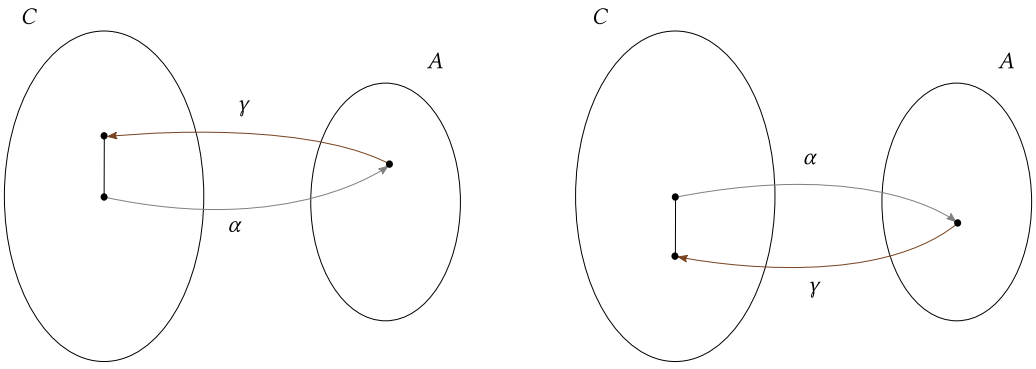
$$\forall c \in C, a \in A. \quad \alpha(c) \preceq a \iff c \preceq \gamma(a),$$

and we denote it with $\langle C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A \rangle$. We call C and A, respectively, the concrete and the abstract domain, α is the abstraction function and γ is the concretization function. In any GC, $\mathrm{id}_C \preceq \gamma \circ \alpha$, $\alpha \circ \gamma \preceq \mathrm{id}_A$, γ preserves glbs and α preserves lubs. In particular, this means that $\gamma(\top_A) = \top_C$ and dually $\alpha(\bot_C) = \bot_A$.

A GC in which $\alpha \circ \gamma = \mathrm{id}_A$ is called a **Galois insertion (GI)**, in which case α is onto and γ is injective. By this last property, there is a bijection between A and $\gamma(A)$, and using this isomorphism, whenever we consider a GI we identify A and its γ-image so that A becomes a subset of C and $\gamma = \mathrm{id}_A$, written as $\langle C \underset{\alpha}{\leftrightarrows} A \rangle$. A GI is said to be trivial if A is the concrete domain or if it only contains $\top_C$. Given a concrete element $c \in C$, we say it is *representable* if it belongs to A, or equivalently if $\alpha(c) = c$.

Given a monotone function $f : C \to C$ and a GC $\langle C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A \rangle$, a function $f^\sharp : A \to A$ is a correct (or sound) approximation of f if $\alpha \circ f \preceq f^\sharp \circ \alpha$. Its best correct approximation (bca) is $f^A = \alpha \circ f \circ \gamma$, and it is the most precise among all the correct approximations of f.

Example 2.1 (Interval Domain). As an example, let us consider $C = \mathcal{P}(\mathbb{Z})$ to be the powerset of integers and $A = \mathsf{Int}$ to be the abstract domain of intervals [12]. Elements of Int are finite intervals $[n, m]$ with $n \leq m$, or infinite intervals of the form $[-\infty, m]$ or $[n, \infty]$, together with the empty interval $\bot$. The top element is $[-\infty, \infty]$. Intervals are ordered by inclusion, the concretisation function γ is defined as usual, while the abstraction function α maps a set of integers to the smallest interval that contains it. If $f(x) = |x|$ is the absolute value function, one of its sound abstractions is $f^\sharp([n, m]) = [0, \max(|n|, |m|)]$ because the interval $[0, \max(|n|, |m|)]$ always contains the entire set $f(S)$ when $n = \min(S)$ and $m = \max(S)$. However, this is not the best possible abstraction: for instance, when $S = \{1\}$ we have $\alpha(S) = [1, 1]$ which yields $f^\sharp(\alpha(S)) = f^\sharp([1, 1]) = [0, 1]$, while



(a) Over-approximation GC (b) Under-approximation GC

Fig. 3. Sketches of GC and UGC.

$f(S) = \{1\}$ whose abstraction is the interval $[1, 1]$. Actually the best correct abstraction f^A is computed as

$$f^A([n, m]) = \alpha \circ f \circ \gamma([n, m]) = \begin{cases} [0, \max(|n|, |m|)] & \text{if } n \leq 0 \leq m \\ [n, m] & \text{if } 0 < n \\ [-m, -n] & \text{if } m < 0 \end{cases}.$$

3 Under-Approximation Galois Connections

As recalled in the Introduction, in theory Abstract Interpretation could be used for under-approximation thanks to the order duality with over-approximation. However, we believe the correspondence is not that straightforward. Therefore, in this section, we introduce the notion of UGC and **under-approximation Galois insertion (UGI)**. Using these, we expand on the idea that over- and under-approximation are not dual to each other: we consider some common over-approximation analyses and techniques and show how they do not fit under-approximation via both intuitive arguments and examples. We start the formal development in the next section.

GC favours over-approximation on the concrete domain: this can be observed in the property $\mathrm{id}_C \preceq \gamma \circ \alpha$, stating that the abstraction of an element is above the element itself (as seen in Figure 3(a)). However, it is also true that $\alpha \circ \gamma \preceq \mathrm{id}_A$, so in a sense the definition favours under-approximation on the abstract domain. This suggest that a GC between A and C, in the "wrong" order, may be a good technical tool. It turns out that such a definition is equivalent to a GC between C^{op} and A^{op}: this matches the intuition that an over-approximation in C^{op} is an under-approximation in C. Following this idea, we introduce the notion of UGC. Given two complete lattices C and A, a pair of monotone functions $\alpha : C \to A$, $\gamma : A \to C$ defines a UGC between C and A when

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

and we denote it with $\langle C \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} A \rangle$. Note the different positions of arrows and their super/subscripts when compared with a GC $\langle C \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} A \rangle$.

The difference between a GC and a UGC is sketched in Figure 3: in the GC on the left γ is above and α below, and their composition is *above* the starting element. Dually, in the UGC on the right γ is below and α above, and their composition is *below* the starting element (as seen in Figure 3(b)).

Using the duality observed above, we get that a UGC enjoys all the properties of a GC with the inequalities reversed: $\gamma \circ \alpha \preceq \mathrm{id}_C$, $\mathrm{id}_A \preceq \alpha \circ \gamma$, γ preserves lubs and α preserves glbs. Just like GCs, we call UGI any UGC in which $\alpha \circ \gamma = \mathrm{id}_A$, and we denote it by $\langle C \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} A \rangle$. Note that when a UGC is a UGI, the corresponding GC between the opposite domains is *not* a GI. Analogously to GIs, in a UGI α is onto and γ is injective, making the same identification of A with $\gamma(A)$ possible. Moreover, in a UGI on a concrete powerset $\langle \mathcal{P}(C) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} A \rangle$, for all $a, a' \in A$, $\gamma(a \cup a') = a \cup a'$, i.e., A is closed under union. In the article, we will use UGIs as much as possible to simplify the notation, but the technical development itself does not rely on this hypothesis.

Dually to standard, over-approximation GCs, given a monotone function $f : C \to C$ and a UGC $\langle C \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} A \rangle$, a function $f^\flat : A \to A$ is a correct (or sound) abstraction of f if $\alpha \circ f \succeq f^\flat \circ \alpha$. Again, $f^A = \alpha \circ f \circ \gamma$ is the best correct under-approximation of f.

Example 3.1 (Intervals Around 0). As an example, let us take again $C = \mathcal{P}(\mathbb{Z})$ and $A = \mathsf{Int}_0$ be the set of integer intervals around 0, i.e., $\mathsf{Int}_0 = \{I \in \mathsf{Int} \mid 0 \in I\} \cup \{\bot\}$. This is an under-approximation abstract domain because it contains $\bot$ and is closed under union: the union of intersecting intervals is an interval too, and all elements of Int_0 intersect at 0. If again $f(x) = |x|$ is the absolute value function, its bca f^A is $f^A([n, m]) = [0, \max(|n|, |m|)]$ since it's always the case that $n \leq 0 \leq m$.

3.1 Comparison with Over-Approximation

We use the concept of UGI to revisit some known over-approximation analyses and techniques, show why they don't work straightforwardly for under-approximation and how this is tied to specific characteristics of under-approximation.

Complement Domain. The first attempt, already briefly discussed in the Introduction, is the use of the complement of an over-approximation abstract domain. This is an application of the order theoretic duality, but it turns out the resulting domains are not useful for analysis.

Example 3.2 (Complement Domain). Whenever the concrete domain is a powerset $\mathcal{P}(C)$, we can exploit the complement $\neg : \mathcal{P}(C) \to \mathcal{P}(C)$ to define a UGC from any given GC by taking complements of the concretization (as done for instance in [28]). Formally, given a GC $\langle \mathcal{P}(C) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A \rangle$, then $\langle \mathcal{P}(C) \underset{\neg\circ\gamma}{\overset{\alpha\circ\neg}{\rightleftarrows}} A^{\mathrm{op}} \rangle$ is a UGC (this stems from $\neg : \mathcal{P}(C) \to \mathcal{P}(C)^{\mathrm{op}}$ being an isomorphism of posets). For instance, given the interval domain, we can define its complement by

$$\gamma_\neg([n, m]) = \neg\gamma([n, m]) = \mathbb{Z} \setminus \{x \in \mathbb{Z} \mid n \leq x \leq m\} = \{x \in \mathbb{Z} \mid x < n \vee m < x\}.$$

The set of abstract elements is the same as Int, but the ordering is reversed, so we call this domain $\mathsf{Int}^{\mathrm{op}}$. The UGC with $\mathcal{P}(\mathbb{Z})$ is given by $\gamma_\neg = \neg \circ \gamma$ and $\alpha_\neg = \alpha \circ \neg$. Note that, thanks to the ordering in $\mathsf{Int}^{\mathrm{op}}$ being the opposite, both $\alpha_\neg$ and $\gamma_\neg$ are monotone. The resulting UGC was already sketched in Figure 1(b).

While $\mathsf{Int}^{\mathrm{op}}$ is a sound under-approximation abstract domain, we argue that it is not useful for incorrectness analysis. Consider a command as simple as the initialization `i := 0` that may happen at the beginning of a loop. This requires the analysis to abstract the concrete element $\{0\}$, the set of values `i` may assume at the beginning of the loop. According to the above definition, we have

$$\alpha_\neg(\{0\}) = \alpha(\neg\{0\}) = \alpha(\mathbb{Z} \setminus \{0\}) = [-\infty, +\infty],$$

i.e., the bottom element of $\mathsf{Int}^{\mathrm{op}}$ since the ordering is reversed. We can better understand the reason for getting bottom by recalling that the meaning of an interval in $\mathsf{Int}^{\mathrm{op}}$ is the set of elements that

are *not* in the interval:

$$\gamma_{\neg}([-\infty, +\infty]) = \neg\gamma([-\infty, +\infty]) = \neg\mathbb{Z} = \emptyset.$$

Other than the intuition that we lost all the information about the initialization, since $[-\infty, +\infty]$ is $\bot$ the analysis incurs in the issue described in the Introduction about "recover" from $\bot$, effectively making the analysis unable to infer anything.

This line of reasoning can be generalized to any finite concrete set X: $\mathbb{Z} \setminus X$ is not bounded, as it contains all integers greater than $\max(X)$ and smaller than $\min(X)$ (which are both finite), so its abstraction through α is $[-\infty, +\infty]$. □

Compositionality. An important property of Abstract Interpretation analysis is compositionality. However, citing O'Hearn [35, §8], "for incorrectness reasoning, you must remember information as you go along a path [...]." This means that a compositional under-approximation analysis must be precise enough to have locally all the informations which should be carried over to the next piece of code. Over-approximation instead is way less restricting in this sense, since "for correctness reasoning, you get to forget information as you go along a path" (O'Hearn [35, §8]). As an example, we present dependency analysis, which is compositional for over-approximation but it is not for under-approximation.

Example 3.3 (Dependency Analysis). A dependency analysis (see, e.g., [4], section 6) aims to compute, for each variable at every program point, the set of values it depends on. They are used for instance to check security properties such as information flow constraints.

The result of the analysis is usually expressed with a collection of atomic dependencies $x \rightsquigarrow y$, meaning that the *current* value of y depends on the *initial* value of x. For instance, consider the code

```
if (x == pwd) { y = y + 100 }; x:= 0.
```

The analysis of this fragment returns the dependencies $x \rightsquigarrow y$, $y \rightsquigarrow y$, as the final value of y depends on the initial values of both x and y. Note that the analysis does not compute any dependency with x on the right as the final value of x is always 0, hence it carries no dependency.

Over-approximation dependency analysis heavily exploits *transitivity* (as it enables compositional reasoning): if C_1 exhibit the dependency $x \rightsquigarrow y$ and C_2 exhibits $y \rightsquigarrow w$, then $C_1; C_2$ *may* induce $x \rightsquigarrow w$. As a trivial example, `y := x; w := y` yields the dependency $x \rightsquigarrow w$. However, transitivity is not well behaved for under-approximation, whose goal is to find true dependencies only. The issue is that dependencies may cancel each other when composed. As a simple example, consider the code

$$C_1 = \texttt{y := x; z := -x}$$
$$C_2 = \texttt{w := y + z}.$$

An under-approximation dependency analysis for C_1 may return $x \rightsquigarrow y$ and $x \rightsquigarrow z$ because they are true dependencies. Similarly, for C_2 it could deduce $y \rightsquigarrow w$. However, for the composition $C_1; C_2$ the transitive inference $x \rightsquigarrow y \rightsquigarrow w$ is not sound because w is always 0, so it does not depend on anything. □

Closure Under Union. Lastly, we consider non-relational analyses. Intuitively, a non-relational analysis cannot capture relationships between different variables. However, as recalled above, under-approximation cannot forget information along a path, including relations between variables. This means that non-relational domains cannot be used for under-approximation. Note that the complement of a non-relational abstract domain, constructed as in Example 3.2, is relational.

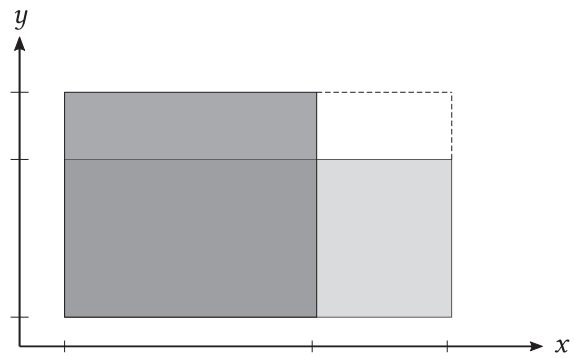


Fig. 4. Non-relational domains are not closed under union.

Example 3.4 (Non-Relational Domain). Informally, a non-relational abstract domain is a tuple of elements, one for each variable x, and describes the set of concrete states where each variable belongs to the values in its abstract coordinate. The abstraction is performed on each variable independently, projecting all states in S on that variable and then abstracting the resulting set. The concretization is performed on each variable independently, and then the results are combined in all possible ways to get concrete values.

As an example, consider the product of one interval domains for each variable. For instance, take the code

```
y := 5 - x; z := x + y,
```

and assume at the beginning the variable x assumes values in the interval $[0, 3]$. An interval analysis on this fragment would find that y takes values in the interval $5 - [0, 3] = [2, 5]$, and then z is in the result of $[0, 3] + [2, 5] = [2, 8]$. However at the end of the program z is always 5, so this is a sound over-approximation but is not as precise as it can be. The issue here is that the values of x and y are not independent, so an operation between these two variables cannot actually receive all possible inputs with x in $[0, 3]$ and y in $[2, 5]$, but just those that also satisfy the *relationship* `y = 5 - x`. However, the interval domain knows nothing about this relationship since it abstracts each variable independently. More precisely, the possible pair of values for x, y are $\{(0, 5), (1, 4), (2, 3), (3, 2)\}$, but the abstraction is computed projecting on one variable (for instance x) and then computing the interval over-approximating that set (so that the abstraction for x is $[0, 3]$). Then, the concretization contains all the pairs in the product $[0, 3] \times [2, 5]$, which are much more.

In general, this projection is not sound for under-approximation, since the concretization is not able to recover which of the original pairs were in the concrete set and which were not. On a more abstract level, such a domain is not closed under union: elements of the abstract domain are "rectangles" in the Cartesian plane with variables on the axes (since they are concretized to the Cartesian product of the abstractions for each variable) and the union of rectangles is not a rectangle. This is shown intuitively in Figure 4: the union of the light and dark rectangles is not a rectangle as it misses the top-left "corner." □

4 Non-Emptying Functions

As anticipated in the Introduction, our focus is on basic transfer functions, the semantics of basic constructs of the language. The key concept in our development is the following definition of

non-emptying function. To understand the rationale behind it, recall that $\bot$ means no information in the under-approximation setting, while any other abstract value means "something" interesting.

Definition 4.1 (Non-Emptying Function). Let $\langle C \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} A\rangle$ be a UGC, $f : C \to C$ a monotone function and $f^A = \alpha \circ f \circ \gamma$ its bca. We say that f is *non-emptying* (in A) if, for any concrete value c, $\alpha(c) \neq \bot$ and $\alpha(f(c)) \neq \bot$ imply $f^A(\alpha(c)) \neq \bot$.

The idea behind this definition is that if the analysis starts from something meaningful ($\alpha(c) \neq \bot$) and it can find something significant ($\alpha(f(c)) \neq \bot$) then it will find at least one of the possible results ($f^A(\alpha(c)) \neq \bot$), thus not degrading the result of the analysis to $\bot$ and avoiding the issues discussed in the Introduction. The meaning of the definition is illustrated by the following toy example.

Example 4.2. Consider the simple imperative fragment

```
if (x ≠ 0) then { while (x < 10) { y := 7 / x; x := x + 1; } }.
```

where a careless programmer used the condition `x ≠ 0` instead of the expected `x > 0`: on any initial state where x is negative the program incurs a division by 0 error.

For the analysis, suppose `x` is an integer value and consider the domain $\mathsf{Int}_{01} = \{I \in \mathsf{Int} \mid 0 \in I \vee 1 \in I\} \cup \{\bot\}$, a variation of Int_0 such that each interval in Int_{01} must contain at least one of 0 and 1. By an argument similar to that for Int_0 it can be shown that Int_{01} is closed under union (since 0 and 1 are consecutive values in the integer domain), and thus is an under-approximation domain, whose abstraction function maps each set of integers to the largest interval that is included in the set and that contains 0 or 1.

In this domain, the semantics f of the increment $x := x + 1$ is not non-emptying: for instance, on the concrete value $c = \{-1, 1, 2, \ldots, 10\}$ we have $\alpha(c) = [1, 10] \neq \bot$ and $\alpha(f(c)) = \alpha(\{0, 2, 3, \ldots, 11\}) = [0] \neq \bot$ but $f^A(\alpha(c)) = f^A([1, 10]) = \alpha(f(\gamma([1, 10]))) = \alpha(\{2, 3, \ldots, 11\}) = \bot$. We show in the following the effect of this on the analysis.

Assume to start the analysis in this domain with the initial condition $[-1, 10]$ for variable x: remember that this being an under-approximation analysis, the abstract state $[-1, 10]$ means that x may assume all the values in that interval at the beginning of the code fragment. In the concrete execution, the filter $x \neq 0$ then produces the concrete set of values $c = \{-1, 1, 2, \ldots, 10\}$, but the abstract interpreter must abstract this to its largest subset that is an interval containing 0 or 1, i.e., $[1, 10]$. The abstract analysis of the cycle then proceeds straightforwardly, finding $\bot$ after one iteration of the loop body (since after the increment the set of values for x is $\{2, 3, \ldots, 11\}$ that is abstracted to $\bot$ because it contains neither 0 nor 1) and so the abstract fixpoint of the loop is the interval $[1, 10]$. This yields no error, even though the concrete execution starting at $x = -1$ does indeed fail after one iteration. □

For the remainder of the article, we assume to be given a set of *values* C and a UGI $\langle \mathcal{P}(C) \overset{\alpha}{\rightleftarrows} A\rangle$ whose concrete domain is $\mathcal{P}(C)$. In the following, we present the basic technical tools we use to prove our theorems (Propositions 5.3 and 5.6 and Theorems 6.2, 6.6, and 7.1), which all follows the same pattern. First we make the assumption that the abstract domain is not *too large* to hinder the analysis, but then we show that if all the functions in a given family were non-emptying, the abstract domain would grow too large. Particularly, we start from a representable element (which is assumed to exist by hypothesis), then use Lemma 4.4 to build other representable elements, up until the size of the abstract domain blows up. To get this, we exploit an exponential increase in the size of A because "closure under union" is demanded for under-approximation domains, i.e., the fact that if two concrete elements are representable in the abstract domain, their union is representable as well.

Definition 4.3. Let $S \subseteq C$ be a subset of C, and $\langle \mathcal{P}(C) \overset{\alpha}{\rightleftarrows} A \rangle$ be an a UGI. We say that $d \in C$ is *representable with* S if $S \cup \{d\}$ is representable in A. We call $R_A(S)$ the set of elements of C representable with S, i.e.,

$$R_A(S) = \{d \in C \mid \alpha(\{d\} \cup S) = \{d\} \cup S\}.$$

For the sake of brevity, we omit the subscript A whenever it is clear from the context, we write R for $R(\emptyset)$, the set of representable values of C, and use the shorthand $R(c)$ for $R(\{c\})$ when $c \in C$ is a concrete value. The following lemma, valid for non-emptying functions, explains the role played by Definition 4.1 in proving all our negative results. Roughly speaking, it states that for f to be non-emptying, some concrete values must be representable in the abstract domain.

Lemma 4.4. *Let $f : C \to C$ be non-emptying, $c \in R$ and $\tilde{c} \notin R(c)$. If $f(\tilde{c}) \in R$, then $f(c) \in R$.*

Proof. By hypothesis c is representable but the pair $\{c, \tilde{c}\}$ is not representable. Since α is monotone and c is representable we have $\alpha(\{c, \tilde{c}\}) \supseteq \alpha(\{c\}) = \{c\}$. Since by correctness $\{c, \tilde{c}\} \supseteq \alpha(\{c, \tilde{c}\})$ and $\alpha(\{c, \tilde{c}\}) \neq \{c, \tilde{c}\}$ because this pair is not representable and hence not in the image of α, it must be the case that $\alpha(\{c, \tilde{c}\}) = \{c\}$.

Now

$$\alpha(f(\{c, \tilde{c}\})) = \alpha(\{f(c), f(\tilde{c})\}) \supseteq \alpha(\{f(\tilde{c})\}) = \{f(\tilde{c})\},$$

where the last equality follows by the hypothesis that $f(\tilde{c}) \in R$. This in particular means that $\alpha(f(\{c, \bar{c}\})) \neq \emptyset$, and together with the fact that $\alpha(\{c, \bar{c}\}) = \{c\} \neq \emptyset$, we get that $f^A(\alpha(\{c, \bar{c}\})) \neq \emptyset$, because f is non-emptying.

From this it follows:

$$\begin{aligned}
\emptyset \subset f^A(\alpha(\{c, \tilde{c}\})) & \qquad [\text{shown above}] \\
= f^A(\{c\}) & \qquad [\alpha(\{c, \tilde{c}\}) = \{c\}] \\
= \alpha(f(\{c\})) & \qquad [\text{definition of } f^A] \\
= \alpha(\{f(c)\}) & \qquad [\text{additivity of } f] \\
\subseteq \{f(c)\} & \qquad [\text{correctness}].
\end{aligned}$$

Since $\alpha(\{f(c)\})$ cannot be empty it must be exactly $\alpha(\{f(c)\}) = \{f(c)\}$, i.e., $f(c) \in R$. □

While the broad outline of all of our theorems is always the same, they differ in how they solve two key issues: first, it must be possible to apply Lemma 4.4; second, all the new representable elements obtained by applying it must be different from one another. In the following, we present various sets of conditions that can guarantee these two points, hence getting hypotheses for non-existence of under-approximation abstract domains.

5 Integer Domains

In this section, we apply the idea from the previous section to the concrete domain of integers and prove that any under-approximation abstract domain will likely return trivial analyses for programs that include sums inside arithmetic expressions.

5.1 Infinite Integer Domain

As a first example, we consider the infinite domain $\mathcal{P}(\mathbb{Z})$ over all integers.

Assumption 5.1. *We assume that an abstract domain A, to be feasible for analyses, must be at most countable.*

We make this assumption because we want the analysis to have a complexity comparable to that of a single concrete execution: if the analysis could be as complex as an arbitrary set of concrete executions, we could use those instead of the abstract domain. Therefore, we require the abstract domain to have the same size of the set $\mathbb{Z}$ of values handled by the program, and not the concrete domain $\mathcal{P}(\mathbb{Z})$. Many abstract domains, such as intervals, octagons [30] and polyhedrons [18] with at most n edges, satisfy it; some, such as general polyhedrons, do not, but indeed they also exhibit a worst-case exponential cost.

Based on Assumption 5.1, we prove a simple cardinality estimate that is exploited to prove that there are few representable elements.

Lemma 5.2. *For any fixed subset $S \subseteq \mathbb{Z}$, $R(S)$ is finite.*

Proof. By union closure of the abstract domain, any set $S \cup T$ for $T \subseteq R(S)$ is representable too, since it can be expressed as the union of representable sets

$$S \cup T = \bigcup_{x \in T} (S \cup \{x\})$$

and each $S \cup \{x\}$ is representable because $x \in T \subseteq R(S)$. The number of those sets is given by the cardinality of $\mathcal{P}(R(S))$. If $R(S)$ were infinite, it would be at least countable, so its powerset $\mathcal{P}(R(S))$ would have a greater cardinality. However, this would conflict with the Assumption 5.1 saying that A is at most countable. Therefore, $R(S)$ must be finite. □

The result for integers now shows that no under-approximation abstract domain makes all sums non-emptying. The idea of the proof is to define an infinite sequence of representable elements, which is in contradiction with the previous lemma that says that $R = R(\emptyset)$ is finite. To define such a sequence, we want to use Lemma 4.4: we start from an initial representable n_0 and from a value $\bar{n}$ not representable with it, then find a non-emptying f that maps $\bar{n}$ into n_0, so that $f(\bar{n})$ is representable and we can then apply the lemma to get the new representable element $f(n_0)$. We then iterate this procedure, changing f, to build the infinite sequence. We believe the hypothesis that there exists an initial representable value is not very restrictive since initializations like `x := 0` must be abstracted to $\bot$ if 0 is not representable.

Proposition 5.3. *Let $\langle \mathcal{P}(\mathbb{Z}) \overset{\alpha}{\rightleftarrows} A \rangle$ be a UGI, and assume that there is an integer n_0 that is representable. Then, it cannot be the case that all the functions of the form $f_n(x) = x + n$ are non-emptying in A.*

Proof. Towards a contradiction, let us assume that all f_n are non-emptying in A. By hypothesis, $n_0 \in R$ and $R(n_0)$ is at most finite by Lemma 5.2. Since $\mathbb{Z}$ is infinite, there exists an $\bar{n} \in \mathbb{Z} \setminus (R(n_0) \cup \{n_0\})$, i.e., $\bar{n}$ is an element such that the pair $\{n_0, \bar{n}\}$ is not representable. Let $d = n_0 - \bar{n}$, and let us prove by induction on t that $n_0 + td$ is representable for all t. The base case $t = 0$ is the hypothesis that n_0 is representable. For the inductive case, assume $n_0 + td$ is representable, and consider the non-emptying function $f_{(t+1)d}$. We get

$$f_{(t+1)d}(\bar{n}) = \bar{n} + (t+1)d = \bar{n} + n_0 - \bar{n} + td = n_0 + td,$$

that is representable by inductive hypothesis. By the instance of Lemma 4.4 for the pair $\{n_0, \bar{n}\}$ and the function $f_{(t+1)d}$, we get that $f_{(t+1)d}(n_0) = n_0 + (t+1)d$ is representable too, that is exactly the inductive step.

Since $\bar{n} \neq n_0$ also $d \neq 0$, hence $\{n_0 + td \mid t \in \mathbb{N}\}$ is infinite. Moreover $\{n_0 + td \mid t \in \mathbb{N}\} \subseteq R$ by the induction above, but this is impossible since R must be finite by Lemma 5.2. □

The meaning of this proposition for program analysis is the fact that a domain small enough (by Assumption 5.1) is probably unable to deduce meaningful information on an integer domain: if it does not contain representable singletons, it must abstract to $\bot$ any variable initialization, and otherwise, it cannot be non-emptying for all sums, hence getting $\bot$ when values are manipulated using this operation. In both cases, because of strictness, the abstract $\bot$ is propagated along program paths, yielding it as the final result of the analysis, which means exactly it cannot determine any information. This issue is not bound to manifest for all programs, but for any domain, there exist programs for which it does.

Note that the only hypothesis on the abstract domain is related to its size, by Assumption 5.1. This would include all classical numerical domains such as intervals, octagons or congruences [24]. Of course, these are over-approximation domains, so this result does not apply to them, but it shows that our hypotheses are very general and would include many known abstract domains.

5.2 Finite Integer Domain

An analogous result can be obtained for a finite integer domain $\mathcal{P}([-N, N])$, where N is some big integer. This concrete domain models machine integers that are constrained within an interval, so we assume that operations are performed in machine arithmetic, that is wrapping around in case of overflows. This is modelled working modulo $2N + 1$, the length of the interval, and taking the unique representative of each congruence class in the interval $[-N, N]$ of interest. It is worth noting that we take an interval that is symmetric around 0 to simplify notation, but there is no conceptual difference in using an asymmetric one instead.

Assumption 5.4. *We assume that an abstract domain A, to be feasible for the concrete domain $\mathcal{P}([-N, N])$, must have a cardinality that is polynomial in N.*

This assumption, just like Assumption 5.1, guarantees that for any set of input values, the cost of the analysis is always polynomial in that of a single concrete execution, while the concrete analysis could require an exponential cost.

In the following, we use asymptotic notation for some quantities. Therefore, for each N we consider the concrete set of values $C_N = [-N, N]$ and define an abstract domains A_N for the concrete domain $\mathcal{P}([-N, N])$. Intuitively, each A_N represents the same abstraction, just instantiated for different values of N. With this notation, Assumption 5.4 becomes $|A_N| = O(\text{poly}(N))$.

The next lemma is analogous to Lemma 5.2 in proving that some sets are small under Assumption 5.4 on the cardinality of A_N. Here, the "exponential increase" we mentioned is explicit, as we are dealing with finite quantities. For brevity, we write R_N instead of R_{A_N}.

Lemma 5.5. *Let $S \subseteq [-N_0, N_0]$ for some N_0. Then, $|R_N(S)| = O(\log(N))$.*

Proof. For any $N \geq N_0$, as in the proof of Lemma 5.2, by union closure any set $S \cup T$ for $T \subseteq R_N(S)$ is representable in A_N. Hence, we have

$$\text{poly}(N) = |A_N| \geq |\mathcal{P}(R_N(S))| = 2^{|R_N(S)|}$$

so, taking log at both sides, $|R_N(S)| = O(\log(N))$. □

The following proposition uses the same proof line as Proposition 5.3 above: we define a sequence of representable elements and prove that they are too many since, by the previous lemma, R_N is quite small.

Proposition 5.6. *Let $\langle \mathcal{P}([-N, N]) \overset{\alpha}{\rightleftarrows} A_N \rangle$ be a UGI for all N, and assume that there is an integer n_0 that is representable in all the A_N. Then, there exists an N_0 such that, for all $N > N_0$, it cannot be the case that all the functions of the form $f_n(x) = x + n$ (modulo $2N + 1$) are non-emptying in A_N.*

Proof. Let $r_N = |R_N(n_0)|$. By the previous Lemma 5.5 we know that $r_N = O(\log(N))$. For all N, fix an element $\bar{n}_N \notin R_N(n_0)$ not representable with n_0 in A_N such that $d_N = n_0 - \bar{n}_N \leq r_N + 1$. This element exists because otherwise all elements in the interval $[n_0 - r_N - 1, n_0 - 1]$ (modulo $2N + 1$) would be representable with n_0, that is impossible since that interval contains $r_N + 1 = |R_N(n_0)| + 1$ elements.

Consider an N such that all the functions of the form $f_n(x) = x + n$ (modulo $2N + 1$) are non-emptying in A_N. Following the proof of Proposition 5.3, we can show by induction on $t \geq 1$ that the value $f_{td_N}(\bar{n}) = n_0 + (t - 1)d_N$ is representable in A_N. All these values are different from one another for

$$1 \leq t < \frac{2N + 1}{d_N}$$

so that

$$|R_N| \geq \frac{2N + 1}{d_N}.$$

However, we know that $|R_N| = O(\log(N))$ and $(2N + 1)/d_N = \omega(\log(N))$; therefore, there exists some N_0 such that for any $N > N_0$ this inequality does not hold. This in turn implies that for all $N > N_0$ it is not possible that all the functions of the form $f_n(x) = x + n$ (modulo $2N + 1$) are non-emptying in A_N. □

Again, the only assumption on A_N is about its size, so our result applies to many kinds of domains. For instance, a predicate abstraction [23] with $\Omega(\log(N))$ predicates exceeds this bound, as its size is doubly exponential in the number n of predicates.

6 General Infinite Concrete Domains

In this section, we try to extend the results of Section 5.1 on the infinite concrete domain of integers to other infinite concrete domains. More precisely, we deal with an infinite set C of concrete values, and a UGI $\langle \mathcal{P}(C) \overset{\alpha}{\rightleftarrows} A \rangle$. Again, we take the Assumption 5.1 on the size of A. Under this assumption, we can prove again Lemma 5.2 that does not depend on the specific integer domain considered in the previous section.

All conditions we propose in this section are mainly on the family of functions considered and not on the abstract domain. The reason for this is that first we fix a function family, corresponding to a program, and then we look for a domain well suited to analyse the specific family at hand. In other words, the family is given by the applicative context, while the domain can be adapted to it.

Definition 6.1. (Highly Surjective Function Family). Given a family F of functions from C to itself and an element $c \in C$, let

$$P_F(c) = \{d \in C \mid \exists f \in F.\, f(d) = c\}$$

be the set of *preimages of* c, elements of C that can be mapped to c by a function in F. We say that the family F is *highly surjective* if $P_F(c)$ is infinite for any possible choice of $c \in C$.

This property is needed together with Lemma 5.2 to apply Lemma 4.4 and get a new representable element: since there are infinitely many preimages of c but $R(c)$ is finite, there are elements $\tilde{c} \in P_F(c)$ not in $R(c)$; then by definition of $P_F(c)$, there is an f such that $f(\tilde{c}) = c \in R$, so we can apply the lemma to get $f(c) \in R$. The reason for requiring $f(\tilde{c}) = c$ instead of just its membership in R is that, at the beginning of the proof, we assume R to contain only one element, hence the two conditions are equivalent. Starting from this basic idea, we present two sets of sufficient conditions to prove the non-existence of any under-approximation abstract domain.

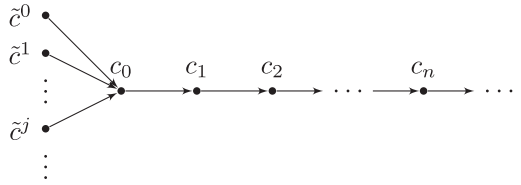


Fig. 5. Graphical representation of the "final" f.

6.1 Local Requirements for Impossibility

The first set of conditions we propose is in a sense more "local," in that it requires conditions on each function in the family F, independently of the others.

The proof of this result proceeds as follows: it starts from a representable $c_0 \in R$ and iteratively creates an infinite sequence $\{c_n\}_{n\in\mathbb{N}}$ of representable elements. This yields a contradiction since R should be finite by Lemma 5.2.

The main idea is to pick a suitable f in F and define the sequence as the iterates $c_{n+1} = f(c_n)$. This function is sketched in Figure 5. The initial set of elements $\tilde{c}^j$ mapped to c_0 is required to apply Lemma 4.4 to all pairs $\{\tilde{c}^j, c_n\}$ and get that $c_{n+1} = f(c_n)$ is representable, since $f(\tilde{c}^j) = c_0 \in R$. The difficulty in realizing this idea is that we do not have enough information on the sequence to pick the right function f at the beginning, so we bring along a list of candidate functions that all coincide on a prefix of the sequence. At step n, we pick a new element of the sequence among the possible images of c_{n-1} through all candidate f we have at that point and discard all those functions that cannot match the choice. Actually, instead of directly considering functions, we represent them with elements $\tilde{c}$ of C. Each one represents a function $f_{\tilde{c}}$ that satisfies $f_{\tilde{c}}(\tilde{c}) = c_0$. Note that this can be done for "enough" (i.e., infinitely many) $\tilde{c}$ because of the high surjectivity hypothesis. We call E_n the set of element $\tilde{c}$ that represent functions that are "valid" for the prefix up to n, i.e., they map c_i to c_{i+1} for $0 \le i \le n-1$. The core of the proof is an induction that proves that E_n always contains infinitely many elements and that the newly chosen c_n is different from all the previous ones.

Theorem 6.2. *Let F be a highly surjective function family from C to itself such that all functions $f \in F$ are either injective or acyclic. Assume also that $R \neq \emptyset$. Then there is at least one function $f \in F$ that is not non-emptying in A.*

Proof. Assume by contradiction that A is non-emptying for all $f \in F$. By hypothesis, R is not empty and thus we can take a $c_0 \in R$. We then define recursively a sequence of representable elements c_n.

Given an element $c \in C$, let

$$NR(c) = C \setminus R(c) = \{\tilde{c} \in C \mid \{c, \tilde{c}\} \text{ is not representable}\}$$

be the set of elements that are *not* representable with c. To ease presentation, we define here E_n, a set of elements of C that depends on the sequence c_n that we construct along the proof. Note that the definition of E_n only depends on elements of the sequence up to c_n

$$\begin{aligned} E_n = \{\tilde{c} \in C \mid & \forall\, 0 \le i \le n\ \tilde{c} \in NR(c_i), \\ & f_{\tilde{c}}(\tilde{c}) = c_0, \\ & \forall\, 0 \le i \le n-1\ f_{\tilde{c}}(c_i) = c_{i+1}\}. \end{aligned}$$

To improve readability, the conditions on $\tilde{c}$ are separated on three different lines. The first line is needed to make Lemma 4.4 applicable to the pair $\{c_i, \tilde{c}\}$ and conclude that $f_{\tilde{c}}(c_i) = c_{i+1}$ is

representable. The second line means just that $\tilde{c}$ represents $f_{\tilde{c}}$, and the last line expresses the requirement that $f_{\tilde{c}}$ coincides on the prefix of the sequence up to c_n.

We observe that the sequence E_n can also be defined inductively by

$$\begin{aligned} E_0 &= \{\tilde{c} \in C \mid \tilde{c} \in NR(c_0), f_{\tilde{c}}(\tilde{c}) = c_0\} \\ &= NR(c_0) \cap P_F(c_0) \\ E_{n+1} &= \{\tilde{c} \in E_n \mid \tilde{c} \in NR(c_{n+1}), f_{\tilde{c}}(c_n) = c_{n+1}\} \\ &= NR(c_{n+1}) \cap \{\tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\}. \end{aligned} \tag{1}$$

E_0 is the intersection of $NR(c_0)$ and the set of $\tilde{c}$ for which there exists $f_{\tilde{c}}$. Using Lemma 5.2 to say $R(c_0)$ is finite and recalling that $P_F(c_0)$ is infinite by high surjectivity, we observe that

$$E_0 = P_F(c_0) \cap NR(c_0) = P_F(c_0) \setminus (C \setminus NR(c_0)) = P_F(c_0) \setminus R(c_0)$$

is infinite too.

We then prove by induction on n the following three statements:

(1) c_n is representable, i.e., $c_n \in R$;
(2) E_n is infinite;
(3) c_n is different from all c_i for $0 \leq i \leq n-1$.

We have already proved the base case for $n = 0$: c_0 is representable by hypothesis, E_0 is infinite as shown above and the third condition is vacuous since there are no $0 \leq i \leq -1$.

For the inductive step, assume the three hypothesis hold for n and let us prove them for $n+1$. Consider the set

$$S = \{f_{\tilde{c}}(c_n) \mid \tilde{c} \in E_n\}$$

of possible candidates for the role of c_{n+1}. Since $c_n \in R$, $\tilde{c} \in E_n \subseteq NR(c_n)$ and $f_{\tilde{c}}(\tilde{c}) = c_0 \in R$ (by inductive hypotheses) we can apply Lemma 4.4 to get $f_{\tilde{c}}(c_n) \in R$ too, hence S is a subset of R. Since R is finite also S must be, and by inductive hypothesis we know E_n is infinite, so there must be an element c_{n+1} in S such that an infinite amount of $\tilde{c} \in E_n$ satisfies $f_{\tilde{c}}(c_n) = c_{n+1}$. We observe that, as shown above, $c_{n+1} \in R$. Moreover we chose c_{n+1} such that

$$\{\tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\}$$

is infinite, so we get

$$\{\tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\} \cap NR(c_{n+1}) = \{\tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\} \setminus R(c_{n+1})$$

is infinite too because $R(c_{n+1})$ is finite. But this, by Equation (1) above, is exactly E_{n+1}.

We only have to show that $c_{n+1} \neq c_i$ for all $0 \leq i \leq n$. Assume by contradiction that this is not the case, so that for some $0 \leq j \leq n$ it holds $c_{n+1} = c_j$. If $f_{\tilde{c}}$ is acyclic this is a contradiction because it would create the cycle $f_{\tilde{c}}^{n+2-j}(c_j) = f_{\tilde{c}}(c_{n+1}) = c_j$. If $f_{\tilde{c}}$ is injective, let us distinguish two cases. If $j = 0$ we get $f_{\tilde{c}}(c_n) = c_{n+1} = c_0 = f_{\tilde{c}}(\tilde{c})$, that would imply $c_n = \tilde{c}$: this is not possible, because the former is representable and the latter is not. If otherwise $j > 0$ we get $f_{\tilde{c}}(c_n) = c_{n+1} = c_j = f_{\tilde{c}}(c_{j-1})$, that would imply $c_n = c_{j-1}$, that is not the case by inductive hypothesis. So $c_{n+1} \neq c_j$, and this concludes the inductive proof.

By the above induction, we conclude that all the infinitely many c_n are elements of R and are all distinct. This yields the desired contradiction because R is finite by Lemma 5.2. □

Remark 6.3. In the previous section, we developed an ad hoc proof for the family of sums over integers in Proposition 5.3, but the same result can also be obtained as an application of Theorem 6.2: if $C = \mathbb{Z}$ and $F = \{\lambda x.x + n \mid n \in \mathbb{Z}\}$, the family is highly surjective (actually $P_F(c) = \mathbb{Z}$

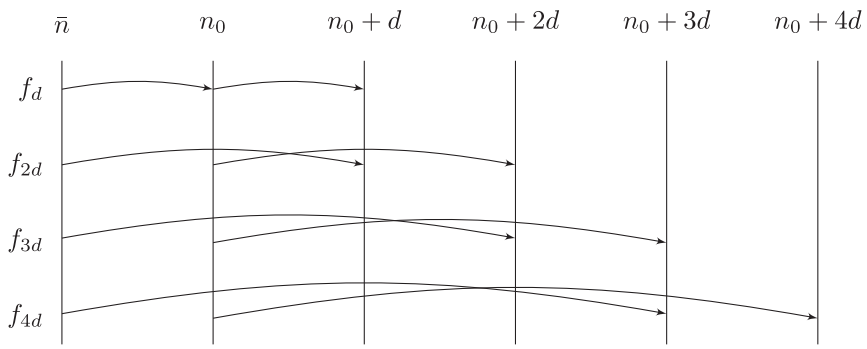


Fig. 6. Graphical representation of the proof of Proposition 5.3.

for all c) and all these functions are injective, so it meets the hypotheses of the theorem. However, it is interesting to note that the proof of Theorem 6.2 is not a generalization of the proof of Proposition 5.3. Here we iterate a single f to build the entire sequence, while in the previous one, we change the function every time, mapping the non representable $\tilde{n}$ to the newly found representable $n_0 + td$ to get that the image of n_0 through that function is representable too, as sketched in Figure 6.

Another example are rational or real numbers, with sums or products.

Example 6.4. Take $C = \mathbb{Q} \setminus \{0\}$ and $F = \{\lambda x.x \cdot q \mid q \in \mathbb{Q} \setminus \{0\}\}$. The family is highly surjective since $P_F(c) = \mathbb{Q} \setminus \{0\}$ for all c, and all these functions are invertible, hence injective. □

A possibly more interesting example of application are floating-point numbers as described by the IEEE Standard.

Example 6.5. Take $C = \mathcal{F} \setminus \{0\}$ the set of non-zero floating-point numbers that can be represented with a fixed number of significant digits, say t bits, but with an arbitrary precision exponent. We choose infinite precision exponents but a finite number of significant digits to have an infinite domain, as required by the theorem, but also to preserve characteristics of floating-point arithmetic.

Let $\cdot$ and $\odot$ denote real product and its floating-point approximation, respectively, and consider the function family $F = \{\lambda x.x \odot y \mid y \in C\}$. The function family is highly surjective, e.g., considering that all numbers with the same significant digits as a floating-point x but different exponent can be mapped into x by multiplying them by 2 to the power of the difference of exponents. For the second condition, if $y = \pm 1$ we have that the function $\lambda x.x \odot y$ is invertible, hence injective. Otherwise, assume without loss of generality that $y > 1$ (other cases are analogous), and by contradiction assume it has a cycle $f^n(x_0) = x_0$. By monotonicity of $\odot$ we have $f(x) = x \odot y \geq x \odot 1 = x$, hence $x_0 \leq f(x_0) \leq f^2(x_0) \leq \cdots \leq f^n(x_0) = x_0$ so all the elements of the cycle are equal, and in particular $f(x_0) = x_0$. However, if $y \neq 1$, the product $x \odot y$ is never equal to x, that is a contradiction. Hence, the function is acyclic. This means F meets the hypotheses of Theorem 6.2, hence no abstract domain on floating-point numbers can be non-emptying for all multiplications. □

6.2 Global Requirements for Impossibility

The second set of conditions we propose is "global," in the sense that it requires the family F to satisfy a property as a whole.

The proof of this theorem starts from the infinite set $P_F(c_0)$ and, using the hypotheses that some sets are finite, propagates its infiniteness down to R, yielding a contradiction with Lemma 5.2, stating that R is finite.

Theorem 6.6. *Let F be a highly surjective family of functions from C to itself such that*

(1) for all pair of elements $c, d \in C$, the set $\{f \in F \mid f(d) = c\}$ is finite;
(2) for all pair of an element $c \in C$ and a function $f \in F$, the set $\{d \in C \mid f(d) = c\}$ is finite.

Assume also that R is not empty. Then, there is at least one function $f \in F$ that is not non-emptying in A.

Proof. Towards a contradiction, let us assume that A is non-emptying for all $f \in F$. By hypothesis, R is not empty, and thus we can take a $c_0 \in R$.

Since F is a highly surjective family, $P_F(c_0)$ is infinite. By Lemma 5.2, we know that $R(c_0)$ is finite. Therefore, we get that the set

$$\begin{aligned} E &= \{\tilde{c} \in C \mid \tilde{c} \notin R(c_0), \exists f_{\tilde{c}} \in F \; f_{\tilde{c}}(\tilde{c}) = c_0\} \\ &= P_F(c_0) \setminus R(c_0) \end{aligned}$$

is infinite. By Lemma 4.4, for all $\tilde{c} \in E$, we have $f_{\tilde{c}}(c_0)$ is representable.

Now fix a function $f \in F$, and let $J(f)$ be the set of $\tilde{c}$ for which f can play the role of $f_{\tilde{c}}$

$$J(f) = \{\tilde{c} \in C \setminus R(c_0) \mid f(\tilde{c}) = c_0\}.$$

By hypothesis (2), the set $J(f)$ is finite.

Now let G be the set of functions in F that can play the role of $f_{\tilde{c}}$ for some $\tilde{c} \in E$

$$G = \{f \in F \mid \exists \tilde{c} \in E.\, f(\tilde{c}) = c_0\}.$$

Clearly

$$E = \bigcup_{f \in G} J(f).$$

Since E is infinite while each $J(f)$ is finite, the set G must be infinite too.

The above observation about $f_{\tilde{c}}(c_0)$ being representable can be equivalently restated by saying that for all $f \in G$, $f(c_0)$ is representable. So consider the set I of all possible images of c_0 through functions in G

$$I = \{f(c_0) \mid f \in G\}.$$

This set is a subset of R because all its elements are representable.

Clearly

$$G = \bigcup_{d \in I} \{f \in G \mid f(c_0) = d\}.$$

Now observe that for any $d \in C$, by hypothesis (1) the set

$$\{f \in F \mid f(c_0) = d\}$$

is finite, and this is a superset of $\{f \in G \mid f(c_0) = d\}$, which must be finite too. Since we know that G is infinite, the set I must be infinite too.

This leads to the desired contradiction: R is finite by Lemma 5.2, but we found that I is an infinite set of representable elements. □

Remark 6.7. Again, Theorem 6.6 can be used to prove Proposition 5.3, but the proofs are different. The former starts from the infinite set $P_F(c_0) \setminus R(c_0)$ of preimages $\tilde{c}$ of c_0 that are not representable with it. This yields an infinite list of pairs $\{\tilde{c}, c_0\}$ to apply Lemma 4.4: for any such pair, since $\tilde{c}$ is in $P_F(c_0)$, we get a function $f_{\tilde{c}}$ such that $f_{\tilde{c}}(\tilde{c}) = c_0 \in R$, so that $f_{\tilde{c}}(c_0) \in R$, too. The proof then exploits the remaining hypotheses to prove that there are infinitely many distinct $f_{\tilde{c}}(c_0)$. The proof of the latter relies on multiple functions to apply Lemma 4.4, but it always uses the same pair $\{\bar{n}, n_0\}$: at every step, it finds a function that maps $\bar{n}$ to the representable element found at the previous step.

Similarly to Theorem 6.2, this result can be used to prove the impossibility of building an abstract domain for floating-point numbers.

Example 6.8. Take $C = \mathcal{F} \setminus \{0\}$ the set of non-zero floating-point numbers with t bits significands and arbitrary precision exponents, and $F = \{\lambda x.x \odot z \mid z \in \mathcal{F} \setminus \{0\}\}$. As observed in Example 6.5, this family is highly surjective. Fixed now two floating-point numbers x, y, and letting $\mathbf{u}$ be the machine precision of floating-point arithmetic, we have that $y = x \odot z$ only if

$$\left|\frac{y - (x \cdot z)}{x \cdot z}\right| < \mathbf{u}$$

i.e.,

$$\left|\frac{y}{x}\right| \frac{1}{1+\mathbf{u}} < |z| < \left|\frac{y}{x}\right| \frac{1}{1-\mathbf{u}}.$$

This is a bounded interval since $x \neq 0$, and hence contains only a finite amount of floating-point numbers. This in turn means that, fixed x and y, there is only a finite amount of functions of the form $\lambda x.x \odot z$ such that $f(x) = y$. Analogously, fixed a floating-point y and a function $f(x) = x \odot z$, we have that $y = f(x)$ only if $|x|$ belong to a bounded interval, that contains a finite amount of floating-point numbers. So, fixed y and a function $f = \lambda x.x \odot z$, only a finite number of x satisfies $f(x) = y$. So, through Theorem 6.6 above, we proved again that no abstract domain on floating-point numbers can be non-emptying for all multiplications. □

We point out once more that the only hypothesis on the abstract domain is about its size for both theorems in this section. For instance, our result applies to any under-approximation predicate abstraction domain [23]. Such a domain is defined by a (finite) list of predicates, each one representing the set of states satisfying the predicate, and an abstract state is a subset of predicates. By duality with respect to over-approximation, the concretization of such a set is the *union* of the states described by each predicate. Analogously, the abstraction of a set of concrete states S is the set of predicates which are *entirely contained* in S. Note that this interpretation is consistent with the use of under-approximations in Incorrectness Logic: the difference is that the logic can use any under-approximation formula, while predicate abstraction is constrained to use just a finite set of predicates, fixed beforehand.

6.3 On the Necessity of High Surjectivity

Both sets of conditions we proposed in this section require the function family to be highly surjective. This turns out to be necessary to prove that no under-approximation abstract domain exists, as we show in this section.

Proposition 6.9. *For any fixed family F of functions from C to itself that is not highly surjective, there exists an abstract domain A_F for $\mathcal{P}(C)$ such that:*

—A_F is finite, and
—all functions $f \in F$ are non-emptying in A_F.

Proof. Since F is not highly surjective, there exists $c_0 \in C$ such that $P_F(c_0)$ is finite. We then define A_F as follows. The only element of C representable on its own is c_0 itself, i.e., $R = \{c_0\}$. A pair of elements of C is representable if and only if one of its elements is c_0 and the other is in $P_F(c_0)$. This also means that $R(c_0) = P_F(c_0)$. Subsets of C with at least three elements are representable if and only if they are unions of representable pairs. The complete definition of A_F is then

$$A_F = \{\emptyset\} \cup \{\{c_0\} \cup T \mid T \subseteq P_F(c_0)\}.$$

A_F is an opposite Moore family with respect to $\mathcal{P}(C)$: it contains the minimal element, i.e., $\emptyset$, and is closed by union, that are lubs. Hence A_F is an under-approximation abstract domain. Moreover, since $R(c_0) = P_F(c_0)$ is finite, we get that A_F is finite too

$$|A_F| = 1 + 2^{|P_F(c_0)|}.$$

Now we want to show that an arbitrary f in F is non-emptying in A_F. We first observe that a subset $S \subseteq C$ is such that $\alpha(S) \neq \emptyset$ if and only if $c_0 \in S$. Suppose that $c_0 \in S$, then

$$\alpha(S) \supseteq \alpha(\{c_0\}) = \{c_0\} \supset \emptyset.$$

Conversely, suppose that $\alpha(S) \neq \emptyset$. Since all elements of A_F but the empty set contains c_0, by correctness we have

$$c_0 \in \alpha(S) \subseteq S.$$

So fix now $S \subseteq C$ an element of the concrete domain, and assume that both $\alpha(S) \neq \emptyset$ and $\alpha(f(S)) \neq \emptyset$. These two assumptions are equivalent to the conditions $c_0 \in S$ and $c_0 \in f(S)$, respectively, and the second can be rewritten as $\exists d \in S\ f(d) = c_0$. By definition of A_F we know this is equivalent to $d \in P_F(c_0) = R(c_0)$. Hence

$$\begin{aligned}
& S \supseteq \{c_0, d\} \\
\implies & \alpha(S) \supseteq \alpha(\{c_0, d\}) = \{c_0, d\} && [d \in R(c_0)] \\
\implies & f(\alpha(S)) \supseteq f(\{c_0, d\}) \supseteq \{f(d)\} = \{c_0\} && [f(d) = c_0] \\
\implies & f^A(\alpha(S)) = \alpha(f(\alpha(S))) \supseteq \alpha(\{c_0\}) = \{c_0\} && [c_0 \in R],
\end{aligned}$$

where in the second line $d \in R(c_0)$ entails $\alpha(\{c_0, d\}) = \{c_0, d\}$. The last line implies $f^A(\alpha(S)) \neq \emptyset$, so f is non-emptying in A_F. □

Notably, the above proof is constructive. We present an example of such domain construction below.

Example 6.10. Fix the pair of functions $f(x) = x - 1$ and $g(x) = x - 2$ on $\mathbb{Z}$. The family $F = \{f, g\}$ is not highly surjective, so we build an under-approximation abstract domain for which these functions are non-emptying. First, take an integer n_0 such that $P_F(n_0)$ (computed with respect to F) is finite. With this F, any integer is a suitable candidate, so let us fix $n_0 = 0$.

The set of preimages of 0 is $P_F(0) = \{1, 2\}$. We define the abstract domain A_F as

$$A_F = \{\emptyset\} \cup \{X \cup \{0\} \mid X \subseteq P_F(0)\} = \{\emptyset, \{0\}, \{0, 1\}, \{0, 2\}, \{0, 1, 2\}\}.$$

In this abstract domain, a set is abstracted to $\emptyset$ if and only if it does not contain 0 since all elements of A_F but $\emptyset$ contains 0 and the abstraction of a set S must be a subset of S.

To check that f is non-emptying in A_F, fix a set $S \subseteq \mathbb{Z}$. If $\alpha(S) = \emptyset$ the non-emptying condition is vacuously true, so assume this is not the case, or, equivalently, that $0 \in S$. Analogously, if

$\alpha(f(S)) = \emptyset$, the condition is true, so assume $0 \in f(S)$ or, equivalently, $1 \in S$. Using these two assumptions we get

$$\begin{aligned} f^A(\alpha(S)) &= \alpha(f(\alpha(S))) && [\text{def. of } f^A] \\ &\supseteq \alpha(f(\alpha(\{0,1\}))) && [\alpha, f \text{ monotone}, S \supseteq \{0,1\}] \\ &= \alpha(f(\{0,1\})) && [\alpha(\{0,1\}) = \{0,1\}] \\ &= \alpha(\{-1,0\}) = \{0\} && [\text{def. of } f \text{ and } \alpha]. \end{aligned}$$

The check for g is analogous. □

Even though this proposition defines an under-approximation abstract domain, it should not be interpreted as a positive result, since the resulting domain is almost a power set and hence too large to be feasible in practice. Instead, the proposition should be regarded as a way to show that one of the hypotheses required in the previous theorems is tight and cannot be weakened. In particular, since these kinds of results require high surjectivity, they are ill-suited when the focus is on a single function.

This proposition can be generalized to consider sets $S \subseteq C$ whose preimages are finite, but a little care is needed when lifting the definition of preimages to sets of values: a preimage is a set for which there exists a function that maps it to S, not the union of the preimages of elements in S. Formally, we let:

$$P_F(S) = \{T \subseteq C \mid \exists f \in F.\, f(T) = S\}.$$

Using this definition, we can now easily generalize Proposition 6.9:

Proposition 6.11. *Let F be a family of functions from C in itself, and assume there is a set $S_0 \subseteq C$ such that $P_F(S_0)$ is finite. Then there exists a finite abstract domain A_F for $\mathcal{P}(C)$ such that all functions $f \in F$ are non-emptying in A_F.*

Proof. Since the proof is very similar to that of Proposition 6.9 above, we gloss over some details.

First, we define a *basis* for the abstract domain as

$$B_F = \{S_0\} \cup \{S_0 \cup S \mid S \in P_F(S_0)\}$$

and then consider its closure under union

$$A_F = \left\{ \bigcup_{T \in \Gamma} T \mid \Gamma \subseteq B_F \right\}.$$

This is an opposite Moore family and hence is an under-approximation abstract domain for $\mathcal{P}(C)$, and is finite because

$$|A_F| \leq |\mathcal{P}(B_F)|$$

and

$$|B_F| \leq 1 + |\mathcal{P}(P_F(S_0))|,$$

with $P_F(S_0)$ finite by hypothesis.

Again we observe that a subset $T \subseteq C$ is such that $\alpha(T) \neq \emptyset$ if and only if $S_0 \subseteq T$ because all elements of the abstract domain but the empty set contains S_0. Then, the proof proceeds as above:

fix $f \in F$ and $T \subseteq C$ such that $\alpha(T) \neq \emptyset$ and $\alpha(f(T)) \neq \emptyset$, that in turn are equivalent to $S_0 \subseteq T$ and $\exists S \subseteq T. f(S) = S_0$. Then, by definition of A_F, this means $S_0 \cup S \in A_F$, so

$$\begin{aligned} & T \supseteq S_0 \cup S \\ \implies & \alpha(T) \supseteq \alpha(S_0 \cup S) = S_0 \cup S && [S_0 \cup S \in A_F] \\ \implies & f(\alpha(T)) \supseteq f(S_0 \cup S) \supseteq f(S) = S_0 && [f(S) = S_0] \\ \implies & f^A(\alpha(T)) = \alpha(f(\alpha(T))) \supseteq \alpha(S_0) = S_0 && [S_0 \in A_F] \end{aligned}$$

i.e., f is non-emptying. □

This proposition may also be applied to the concrete domain of finite lists to show that a natural function family to consider cannot be used to prove non-existence of under-approximation domains using non-emptying functions.

Example 6.12. Fix the concrete domain C as the set of all lists of finite length over a finite, non-empty alphabet Γ, i.e., $C = \Gamma^*$. For $\alpha \in \Gamma^*$ a finite string, let

$$\mathrm{concat}_\alpha(\beta) = \alpha\beta$$

be the function that prefixes its argument by the string α. The family

$$F = \{\mathrm{concat}_\alpha \mid \alpha \in \Gamma^*\}$$

is not highly surjective, because fixed a string γ only its suffixes can be mapped into γ by a function in F, and they are a finite amount. Hence, we can define an under-approximation abstract domain for which all these functions are non-emptying by means of Proposition 6.11. Such domains are defined with a construction similar to that of Example 6.10, and in particular, if ϵ is the empty list, considering the set $S_0 = \{\epsilon\}$ whose preimage is only S_0 itself, the construction yields

$$A_F = \{\emptyset, \{\epsilon\}\}.$$

It is easy to check that all functions concat_α are non-emptying in this abstract domain. □

The previous proposition focuses on preimages, stating that if there is a concrete element that has a finite amount of preimages then it is possible to define an under-approximation domain. A natural dual of this proposition can be formulated in terms of images. For a subset $S \subseteq C$, the set of its images is defined as follows

$$I_F(S) = \{f(S) \mid f \in F\}.$$

This definition is exactly dual to that of preimages and can be used to formulate a similar result.

Proposition 6.13. *Let F be a family of total functions (i.e., if $S \neq \emptyset$ then $f(S) \neq \emptyset$) from $\mathcal{P}(C)$ to itself, and assume there is a non-empty set $S_0 \subseteq C$ such that $I_F(S_0)$ is finite. Then, there exists a finite abstract domain A_F such that all functions $f \in F$ are non-emptying in A_F.*

Proof. Define a *basis* for the abstract domain

$$B_F = \{S_0\} \cup \{S_0 \cup S \mid S \in I_F(S_0)\}$$

and consider its closure under union

$$A_F = \left\{ \bigcup_{T \in \Gamma} T \mid \Gamma \subseteq B_F \right\}.$$

Again this is a correct finite under-approximation abstract domain for $\mathcal{P}(C)$ satisfying that $\alpha(T) \neq \emptyset$ if and only if $S_0 \subseteq T$ (this can be shown in the very same way as in the proof of Proposition 6.11).

Taken an arbitrary $f \in F$, let us show that it is non-emptying. Fix a set $T \subseteq C$ such that $\alpha(T) \neq \emptyset$. This equivalently means that $S_0 \subseteq T$, so

$$\begin{aligned} & \alpha(T) \supseteq \alpha(S_0) = S_0 \\ \implies & f(\alpha(T)) \supseteq f(S_0) \\ \implies & f^A(\alpha(T)) = \alpha(f(\alpha(T))) \supseteq \alpha(f(S_0)). \end{aligned}$$

But $f(S_0) \in B_F$, so $\alpha(f(S_0)) = f(S_0) \neq \emptyset$ by the hypothesis that f is total, and so $f^A(\alpha(T)) \neq \emptyset$, from which we conclude that f is non-emptying. □

Even though Proposition 6.13 introduces the technical hypothesis that all $f \in F$ are total, this condition is not very restrictive in practice, because our results are applicable when F is a family of basic transfer functions, that seldom introduce divergence: in programming languages, non-termination is often due to control-flow constructs and not to single assignments or guards. As an immediate application of the proposition that exploits images instead of pre-images, we consider lists again and rule out another natural function family.

Example 6.14. Fix again $C = \Gamma^*$, and consider all functions of the form $\mathrm{drop}_n : \Gamma^* \to \Gamma^*$ that, taken a list, drop its first n elements and return the resulting list. If the input list is shorter than n, the output of drop_n is the empty list ϵ. The function family

$$F = \{\mathrm{drop}_n \mid n \in \mathbb{N}\}$$

is highly surjective since, for any fixed list $\alpha \in \Gamma^*$ and any n, we can prefix α by n arbitrary characters, and map this extended list back to α via drop_n. However, images through this function family are finite:

$$I_F(\alpha) = \{\mathrm{drop}_n(\alpha) \mid n \in \mathbb{N}\}$$

is finite because $I_F(\alpha)$ coincides with the set of all tails of α. By Proposition 6.13, we can define an under-approximation abstract domain such that all functions drop_n are non-emptying. Again, these domains are constructed from sets S_0 with a finite amount of images, and considering $S_0 = \{\epsilon\}$, that satisfies $I_F(S_0) = \{\epsilon\}$, we get

$$A_F = \{\emptyset, \{\epsilon\}\}.$$

It can be easily checked that all functions drop_n are non-emptying in A_F. □

These last two propositions consider opposite situations in which it is possible to define an under-approximation domain: the former requires to be able to go backward using F in infinitely many ways, while the latter to go forward. This often is not the case in the presence of "boundaries" in the concrete domain, which are points with respect to which functions tend to walk either up or away: for instance, ϵ is such a point with finite strings because concat functions go away from it while drop functions move towards it. Another example of such a boundary is 0 in the domain of integers $\mathbb{Z}$ for multiplications and (rounded) divisions: the former increase absolute value, moving away from 0 (even though 0 itself is never a preimage), while the latter decrease it. Also considering a function family made of both kinds of functions does not work: a slight adaptation of the constructions for the two propositions above shows that, if F can be partitioned into two subfamilies, each satisfying the hypothesis of one of the two propositions, then there exists an under-approximation abstract domain. An example of this is in the set of finite lists, taking as F both concat and drop functions. The construction then yields exactly $A_F = \{\emptyset, \{\epsilon\}\}$, for which all these functions are non-emptying, as shown in Examples 6.12 and 6.14. In light of these observations, to apply the definition of non-emptying function in an effective way for proving the

non-existence of abstract domains, for all possible boundaries there is the need for a function that can both enter and exit it. This happens for integers since there is no boundary, but does not for finite lists, with $\{\epsilon\}$ being often either a sink or a source for many functions on lists.

7 General Finite Concrete Domains

The discussion of Section 6 requires C to be infinite. While this is a common simplification, since concrete domains have usually a very large size, an interesting and important question is whether our findings can be extended to the case of finite concrete domains. However, we believe that the two Theorems 6.2 and 6.6 cannot be straightforwardly adapted to the finite setting.

For both these results, the proof of non-existence of the abstract domain was carried out by showing the existence of infinitely many representable elements (defining an infinite sequence in the local result Theorem 6.2, extracting a fraction of the infinite $P_F(c_0)$ in the global Theorem 6.6), against Lemma 5.2. For a finite C, if $N = |C|$, one possibility would be to resort to Lemma 5.5 to show that $|R| = O(\log(N))$ and get the contradiction by proving the existence of $\omega(\log(N))$ representable elements. Unfortunately, the same and similar constructions do not yield the desired bound: when C is infinite we exploited the fact that finite combinations of finite numbers are also finite, while for finite C we should require that arbitrary combinations of logarithmic factors are $O(N)$, which is not the case.

The definition of highly surjective family itself is not so easy to translate. The construction of Proposition 6.9 on a finite C yields an abstract domain with the needed features already when $|P_F(c)| = O(\log(N))$. However, to carry out proofs along the lines of those applicable to infinite domains, we would need stronger constraints on the cardinality of $P_F(c)$ than just $|P_F(c)| = \omega(\log(N))$, the bare minimum to avoid that construction, possibly up to $|P_F(c)| = \Theta(N)$.

Even with these theoretical considerations against stating finite counterparts of Theorems 6.2 and 6.6, we have been able to carry out the proofs for the special case of the finite domain $C = [-N, N]$ of integers (Proposition 5.6). Our result exploits the precise structure of both the given concrete domain and function family, but the key points for the proof are two. First, functions produce elements that are not "too far away" with respect to the size of the domain (i.e., $n_{i+1} - n_i = O(\log(N))$), and this allows to prove that there are enough distinct representable elements. Second, the domain is circular and hence has no boundaries. If the domain had them, we could have applied results, such as Propositions 6.12 or 6.13. This was not the case for integers because additions overflow, so the domain has no boundaries near $-N$ and N.

Building on the above discussion, we present a version of the global condition for finite concrete domains. To overcome the limitation of boundaries, we explicitly constrain the initial representable element c_0, writing hypotheses around it. Such hypotheses imply that c_0 is "far enough" from the boundaries of the domain (if any). Conditions (1–2) of the next theorem correspond to those of the infinite version (Theorem 6.6) rewritten with c_0 in mind. The relation between the two bound functions k_1 and k_2 and the number of preimages of c_0 corresponds to the high surjectivity hypothesis. It only constrains the value c_0 because this is the representable value from which the proof begins.

As we did in the specific case of the integer domain $[-N, N]$, we assume the size of A to be polynomial in N (Assumption 5.4). To say this formally, as we did for finite integers (cfr. Section 5.2), we consider a sequence C_N of sets of concrete values with size $O(N)$. In the specific instance of integers, such sets were the intervals $[-N, N]$; in general, we require that $C_{N-1} \subseteq C_N$ for all N. This inclusion formalizes the intuition that all the C_N are the "same" concrete domain instantiated for different sizes. We also assume to have an abstract domain A_N for each concrete domain $\mathcal{P}(C_N)$ such that $|A_N| = O(\text{poly}(N))$. As before, we write R_N for R_{A_N}, and we remark that, under Assumption 5.4, Lemma 5.5 holds.

Theorem 7.1. *Assume $|C_N| = O(N)$, and let $\langle \mathcal{P}(C_N) \overset{\alpha_N}{\rightleftarrows} A_N \rangle$ be a UGI for all N. Assume there exists a number N_1 and a value $c_0 \in C_{N_1}$ such that, for all $N > N_1$, c_0 is representable in A_N (i.e., $c_0 \in R_N$). Given two functions $k_1, k_2 : \mathbb{N} \to \mathbb{N}_{>0}$, for any N let F_N be a family of functions from C_N to itself such that:*

(1) for all elements $d \in C_N$, $|\{f \in F_N \mid f(c_0) = d\}| \leq k_1(N)$;
(2) for all functions $f \in F_N$, $|\{d \in C_N \mid f(d) = c_0\}| \leq k_2(N)$.

Lastly, assume that $|P_{F_N}(c_0)| = \omega(\log(N) \cdot k_1(N) \cdot k_2(N))$.

Then, there exists N_0 such that, for all $N > N_0$, it is not possible that all $f \in F_N$ are non-emptying in A_N.

Proof. Fix an N such that all $f \in F_N$ are non-emptying in A_N.

Define

$$\begin{aligned} E &= \{\tilde{c} \in C_N \mid \tilde{c} \notin R_N(c_0), \exists f_{\tilde{c}} \in F_N \; f_{\tilde{c}}(\tilde{c}) = c_0\} \\ &= P_{F_N}(c_0) \setminus R_N(c_0). \end{aligned}$$

Now fix a function $f \in F_N$, and let $J(f)$ be the set of $\tilde{c}$ for which f can play the role of $f_{\tilde{c}}$, namely

$$J(f) = \{\tilde{c} \in C_N \setminus R_N(c_0) \mid f(\tilde{c}) = c_0\}.$$

By hypothesis (2), $|J(f)| \leq k_2(N)$.

Now let G be the set of functions in F_N that can play the role of $f_{\tilde{c}}$ for some $\tilde{c} \in E$

$$G = \{f \in F_N \mid \exists \tilde{c} \in E \; f(\tilde{c}) = c_0\}.$$

Clearly

$$E = \bigcup_{f \in G} J(f).$$

But we know that $|J(f)| \leq k_2(N)$ for all f, so

$$|E| \leq \sum_{f \in G} |J(f)| \leq |G| \cdot k_2(N).$$

By Lemma 4.4, for all $\tilde{c} \in E$ we have $f_{\tilde{c}}(c_0)$ is representable. This can be equivalently restated saying that for all $f \in G$, $f(c_0)$ is representable. So consider the set I of all possible images of c_0 through functions in G

$$I = \{f(c_0) \mid f \in G\}.$$

This set is a subset of R_N because all its elements are representable.

Clearly,

$$G = \bigcup_{d \in I} \{f \in G \mid f(c_0) = d\}.$$

Now observe that, for any $d \in C$, by hypothesis (1) we have

$$|\{f \in G \mid f(c_0) = d\}| \leq k_1(N)$$

so

$$|G| \leq \sum_{d \in I} |\{f \in G \mid f(c_0) = d\}| \leq |I| \cdot k_1(N)$$

that in turn implies

$$|E| \leq |G| \cdot k_2(N) \leq |I| \cdot k_1(N) \cdot k_2(N).$$

Since I is a subset of R_N, we get

$$|E| \leq |I| \cdot k_1(N) \cdot k_2(N) \leq |R_N| \cdot k_1(N) \cdot k_2(N).$$

Lastly, we recall that $E = P_{F_N}(c_0) \setminus R_N(c_0)$. Therefore, if all $f \in F_N$ are non-emptying in A_N, then

$$|P_{F_N}(c_0)| \leq |E| \leq |R_N| \cdot k_1(N) \cdot k_2(N).$$

The last hypothesis of the theorem states that $|P_{F_N}(c_0)| = \omega(\log(N) \cdot k_1(N) \cdot k_2(N))$. Lemma 5.5 states that $|R_N| = O(\log(N))$. Therefore, there exists an N_0 such that the inequality

$$\omega(\log(N) \cdot k_1(N) \cdot k_2(N)) = |P_{F_N}(c_0)| \leq |R_N| \cdot k_1(N) \cdot k_2(N) = O(\log(N) \cdot k_1(N) \cdot k_2(N))$$

does not hold for any $N > N_0$. This in turn implies that for all $N > N_0$ it is not possible that all the functions $f \in F_N$ are non-emptying in A_N. □

A straightforward corollary is that, whenever we can verify the hypotheses for all values in C, there is no under-approximation domain with a representable element. This is for instance the case for integers and sums, so we recover Proposition 5.6:

Example 7.2. Let $C_N = [-N, N]$ and

$$F_N = \{\lambda x.x + n(\text{modulo } 2N + 1) \mid n \in [-N, N]\}.$$

Fixed any $n_0 \in [-N, N]$, it is easy to check that $P_{F_N}(n_0) = [-N, N]$ and $k_1(N) = k_2(N) = 1$. The condition

$$|P_{F_N}(n_0)| = \omega(\log(2N + 1) \cdot k_1(2N + 1) \cdot k_2(2N + 1))$$

thus reduces to

$$2N + 1 = \omega(\log(N))$$

that is true. Hence, there is no under-approximation abstract domain with at least one integer n_0 representable. □

The example of integers has the nice property of not having boundaries, but this is seldom the case. For instance, consider floating point numbers: they overflow and underflow to special values, hence they do have boundaries. Nevertheless, if we pick a suitable initial element c_0 we can still apply the theorem above.

Example 7.3. Consider the finite set of non-zero floating-point numbers $C = \mathcal{F} \setminus \{0\}$ with t bits significand, one bit sign and e bit exponents. Consider the function family $F = \{\lambda x.x \odot z \mid z \in \mathcal{F} \setminus \{0\}\}$ of floating-point multiplications, where $\odot$ denotes the floating-point approximation of real product.

As shown in Example 6.8, fixed two floating-point numbers x, y, we have that $y = f(x) = x \odot z$ only if

$$|z| \in \left[\left|\frac{y}{x}\right| \frac{1}{1 + \mathbf{u}}, \left|\frac{y}{x}\right| \frac{1}{1 - \mathbf{u}}\right].$$

If $\mathbf{u} \leq 1/2$ this is entirely contained in the interval

$$\left[\left|\frac{y}{x}\right| (1 - 2\mathbf{u}), \left|\frac{y}{x}\right| (1 + 2\mathbf{u})\right].$$

It can be shown that, if $\mathbf{u} < 1/2$, the quantity of floating point numbers in that interval is bounded by a constant c that does not depend on x and y.[1] Analogously, fixed y and z there are at most c floating point numbers x such that $y = f(x) = x \odot z$.

[1]For instance, this can be proved for $c = 17$.

With these two bounds, if x_0 has enough preimages we can verify the last hypothesis of the theorem: letting $N = |C|$

$$|P_F(x_0)| = \omega(\log(N) \cdot k_1(N) \cdot k_2(N)) = \omega(\log(N) \cdot c \cdot c) = \omega(\log(N)).$$

For instance, $x_0 = 1$ satisfies this condition (more than half of all floating-point numbers have a floating-point inverse, hence $|P_F(x_0)| \geq N/2 = \Theta(N)$) but there are many other points satisfying it. By mean of Theorem 7.1, no under-approximation abstract domain is non-emptying for all multiplications whenever one of these points is representable. □

8 Conclusions and Future Works

Until recently, the focus of formal static analyses has been on over-approximation to prove program correctness, but tools based on this theory are often deployed to catch bugs [19, 39], possibly raising many false alarms. Incorrectness Logic promoted the study of a dual theory for bug finding based on under-approximation to give a formal basis to a new class of tools [27, 36]. In this work, we have pointed out some asymmetries between over- and under-approximation in Abstract Interpretation, and why those are an obstacle to the design of abstract domains for program analysis via UGC. The key observation is that the duality between under- and over-approximation is broken by the fact that in program analysis over- and under-approximations have to be applied to the same transfer functions. The handling of divergence in the abstract domain poses another critical issue. Building on those ideas, we proposed the novel definition of *non-emptying function* and studied how it plays a crucial role in proving the non-existence of general, useful under-approximation based abstract domains. Indeed, our results prove that an analysis based on such domains will very often answer $\bot$ (representing either absence of information or divergence) for programs that require repeated applications of non-emptying functions. This is a big limitation since recovery from $\bot$ in an under-approximation is not as easy as recovering from $\top$ in an over-approximation: we can say that it is quite impossible. We applied our general results to several concrete domains to conclude that, under some mild assumptions, there are no useful under-approximation abstract domains. Then, to show that one of the hypotheses in our result is tight, we proposed a construction to craft an under-approximation abstract domain whenever such a hypothesis is not met.

Our investigations open many possible subjects for future research, as summarized below.

Under-approximation abstract domains must be closed under union, but known abstract domains are rarely such. However, disjunctive completion [20], a known domain transformer, refines any abstract domain in a union-closed one. For over-approximation, disjunctive completions have been studied to improve precision at the expense of increased complexity. For under-approximation, this makes the analysis possible, and is, e.g., the approach taken by the recent work [29]. Moreover, for under-approximation, it is sound to drop disjunct, so employing heuristics to decide which disjuncts to drop is a solution to keep the complexity of the analysis low. Practical tools based on the theory of Incorrectness Logic already use heuristics to drop disjuncts, so taking inspiration from them may be effective also for Abstract Interpretation.

In their recent work, Raad et al. [36] study Incorrectness Separation Logic, the join of Separation Logic [37] and Incorrectness Logic. They notice that the original Separation Logic does not distinguish a pointer known to be dangling from one about which it has no information, and they introduce a new kind of heap assertion for dangling pointers. This issue is reminiscent of the difference between divergence and no information we incur into in Abstract Interpretation. This may suggest the introduction of a similar distinction also in under-approximation domains. The problem is that the use of a point different from $\bot$ to represent divergence needs a concretization, and no such element exists in a power set other than $\emptyset$. However, in Abstract Interpretation, it happens at times that more general concrete domains allow more flexibility in the abstraction

(e.g., as proposed for higher-order functional languages [14]), so it may be worth investigating the possibility of changing the concrete domain to account for this new point.

All our results depend on the existence of at least one single representable value. This assumption is motivated by the analysis performed but is not a requirement of Abstract Interpretation itself. A way to remove this hypothesis may be to consider representable sets of minimal cardinality instead because functions defined as additive extensions do not increase cardinality, so such minimal sets can replace the role of singletons. The technical issue is if and how Lemma 4.4 can be generalized to this setting, which we believe to be possible.

We briefly discussed finite domains, but we left many open questions. We presented two results for infinite domains, but we only translated one of them to the finite case, and with additional hypotheses too. We presented a construction to show that high surjectivity is a tight condition for infinite domains, but the same construction for finite ones yields a bound that may not be tight. Finiteness introduces limitations because arbitrary composition of small sets may grow larger, but also opens up new possibilities as this may also allow proving that the number of representable elements is too large. To sum up, finite domains may deserve further investigation.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments.

References

[1] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012. From Under-Approximations to Over-Approximations and Back. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems—18th International Conference, TACAS'12, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS'12*, Lecture Notes in Computer Science, Vol. 7214. Cormac Flanagan and Barbara König (Eds.). Springer, 157–172. DOI: https://doi.org/10.1007/978-3-642-28756-5_12

[2] Aws Albarghouthi, Arie Gurfinkel, Ou Wei, and Marsha Chechik. 2010. Abstract Analysis of Symbolic Executions. In *Proceedings of the Computer Aided Verification, 22nd International Conference, CAV'10,* Lecture Notes in Computer Science, Vol. 6174. Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 495–510. DOI: https://doi.org/10.1007/978-3-642-14295-6_43

[3] Flavio Ascari, Roberto Bruni, and Roberta Gori. 2022. Limits and Difficulties in the Design of Under-Approximation Abstract Domains. In *Proceedings of the Foundations of Software Science and Computation Structures—25th International Conference, FOSSACS'22, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS'22,* Lecture Notes in Computer Science, Vol. 13242. Patricia Bouyer and Lutz Schröder (Eds.). Springer, 21–39. DOI: https://doi.org/10.1007/978-3-030-99253-8_2

[4] Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel, and Frédéric Tronel. 2017. Hypercollecting Semantics and Its Application to Static Analysis of Information Flow. *SIGPLAN Not.* 52, 1 (Jan. 2017), 874–887. DOI: https://doi.org/10.1145/3093333.3009889

[5] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. 2015. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. *Found. Trends Program. Lang.* 2, 2–3 (2015), 71–190. DOI: https://doi.org/10.1561/2500000002

[6] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. 2022. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 57:1–57:69. DOI: https://doi.org/10.1145/3477579

[7] J. -L. Boulanger (Ed.). 2011. *Static Analysis of Software: The Abstract Interpretation*. Wiley.

[8] François Bourdoncle. 1993. Abstract Debugging of Higher-Order Imperative Languages. *SIGPLAN Not.* 28, 6 (Jun. 1993), 46–55. DOI: https://doi.org/10.1145/173262.155095

[9] 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. DOI: https://doi.org/10.1145/3582267

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

[11] Patrick Cousot. 2021. *Principles of Abstract Interpretation*. MIT Press.

[12] 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 SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*. ACM, New York, NY, 238–252. DOI: https://doi.org/10.1145/512950.512973

[13] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. ACM, New York, NY, 269–282. DOI: https://doi.org/10.1145/567752.567778

[14] Patrick Cousot and Radhia Cousot. 1994. Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. 95–112. DOI: https://doi.org/10.1109/ICCL.1994.288389

[15] Patrick Cousot and Radhia Cousot. 2014. Abstract Interpretation: Past, Present and Future. In *Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 20th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14*. Thomas A. Henzinger and Dale Miller (Eds.). ACM, New York, NY, 2:1–2:10. DOI: https://doi.org/10.1145/2603088.2603165

[16] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI'13,* Lecture Notes in Computer Science, Vol. 7737. Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 128–148. DOI: https://doi.org/10.1007/978-3-642-35873-9_10

[17] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation—12th International Conference, VMCAI'11*, Lecture Notes in Computer Science, Vol. 6538. Ranjit Jhala and David A. Schmidt (Eds.). Springer, 150–168. DOI: https://doi.org/10.1007/978-3-642-18275-4_12

[18] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '78)*. ACM, New York, NY, 84–96. DOI: https://doi.org/10.1145/512760.512770

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

[20] Gilberto Filé and Francesco Ranzato. 1994. Improving Abstract Interpretations by Systematic Lifting to the Powerset. In *Proceedings of the 1994 International Symposium on Logic Programming (ILPS '94)*. MIT Press, Cambridge, MA, USA, 655–669.

[21] Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proc. Symp. Appl. Math.* 19, 19–32.

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

[23] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97*, Lecture Notes in Computer Science, Vol. 1254. Orna Grumberg (Ed.). Springer, 72–83. DOI: https://doi.org/10.1007/3-540-63166-6_10

[24] Philippe Granger. 1991. Static Analysis of Linear Congruence Equalities among Variables of a Program. In *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, Volume 1 Colloquium on Trees in Algebra and Programming (CAAP'91), Lecture Notes in Computer Science, Vol. 493. Samson Abramsky and T. S. E. Maibaum (Eds.). Springer, 169–192. DOI: https://doi.org/10.1007/3-540-53982-4_10

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

[26] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. 2013. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *Proceedings of tje Computer Aided Verification—25th International Conference, CAV 2013*, Lecture Notes in Computer Science, Vol. 8044. Natasha Sharygina and Helmut Veith (Eds.). Springer, 846–862. DOI: https://doi.org/10.1007/978-3-642-39799-8_59

[27] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding Real Bugs in Big Programs with Incorrectness Logic. *Proc. ACM Program. Lang.* 6, OOPSLA (2022), 1–27. DOI: https://doi.org/10.1145/3527325

[28] Tal Lev-Ami, Mooly Sagiv, Thomas Reps, and Sumit Gulwani. 2007. Backward Analysis for Inferring Quantified Preconditions. *Tr-2007-12-01*, Tel Aviv University.

[29] Marco Milanese and Antoine Miné. 2024. Generation of Violation Witnesses by Under-Approximating Abstract Interpretation. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI'24*, , Part I, Lecture Notes in Computer Science, Vol. 14499. Rayna Dimitrova, Ori Lahav, and Sebastian Wolff (Eds.). Springer, 50–73. DOI: https://doi.org/10.1007/978-3-031-50524-9_3

[30] Antoine Miné. 2006. The Octagon Abstract Domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100. DOI: https://doi.org/10.1007/s10990-006-8609-1

[31] Antoine Miné. 2014. Backward Under-Approximations in Numeric Abstract Domains to Automatically Infer Sufficient Program Conditions. *Sci. Comput. Program.* 93 (Nov. 2014), 154–182. DOI: https://doi.org/10.1016/j.scico.2013.09.014
[32] Antoine Miné. 2017. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Found. Trends Program. Lang.* 4, 3–4 (Dec. 2017), 120–372. DOI: https://doi.org/10.1561/2500000034
[33] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer. DOI: https://doi.org/10.1007/978-3-662-03811-6
[34] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the Impact of Formal Methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'18)*. ACM, New York, NY, 13–25. DOI: https://doi.org/10.1145/3209108.3209109
[35] Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (Dec. 2019), 32 pages. DOI: https://doi.org/10.1145/3371078
[36] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Proceedings of the Computer Aided Verification—32nd International Conference, CAV'20*, Part II, Lecture Notes in Computer Science, Vol. 12225. Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 225–252. DOI: https://doi.org/10.1007/978-3-030-53291-8_14
[37] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS'22)*. IEEE Computer Society, 55–74. DOI: https://doi.org/10.1109/LICS.2002.1029817
[38] X. Rival and K. Yi. 2020. *Introduction to Static Analysis – An Abstract Interpretation Perspective*. MIT Press.
[39] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (Mar. 2018), 58–66. DOI: https://doi.org/10.1145/3188720
[40] David A. Schmidt. 2007. A Calculus of Logical Relations for Over- and Underapproximating Static Analyses. *Sci. Comput. Program.* 64, 1 (2007), 29–53. DOI: https://doi.org/10.1016/j.scico.2006.03.008

Received 22 December 2022; revised 7 February 2024; accepted 3 May 2024