Limits and difficulties in the design of under-approximation abstract domains*

Flavio Ascari✉, Roberto Bruni, and Roberta Gori

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

Abstract. Static analyses are mostly designed to show the *absence of bugs*: if the analysis reports no alarms then the program won't exhibit any unwanted behaviours. To this aim they manipulate over-approximations of program semantics and, inevitably, they often report some *false* alarms. Recently, O'Hearn proposed Incorrectness Logic, that is based on under-approximations, as a formal method to *find bugs* that only reports *true* alarms. In this paper we aim to answer one important question raised by O'Hearn, namely which role can Abstract Interpretation play for the development of under-approximate tools for bug catching. In principle, Abstract Interpretation based static analyses can be defined for computing over-approximations as well as under-approximations, but in practice, most techniques exploited the former while few attempts developed the latter. To show why it is difficult to design effective under-approximation abstract domains, we first propose the new definitions of *non emptying functions* and *highly surjective function family* and then we formally prove the limits of under-approximation analysis by showing the non existence of abstract domains able to approximate such functions in a non trivial way. Our results outline the limits of under-approximation Abstract Interpretation and clarify, for the first time, why over- and under-approximation analyzers exhibited such a different development.

Keywords: Abstract Interpretation, Under-approximation, Abstract domains, Impossibility results

1 Introduction

Static program analyses are techniques used to infer properties of programs directly from their source code, without executing them. They have been studied and successfully applied for over 50 years [12,3,13,1,10,17,18,22,23,4] to produce effective methods and tools to support the development of correct software. For all these years, the main focus of static analysis was to prove the absence of bugs by computing over-approximations (supersets of all possible behaviours) of the semantics of programs: the absence of unwanted behaviour

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

© The Author(s) 2022
P. Bouyer and L. Schröder (Eds.): FoSSaCS 2022, LNCS 13242, pp. 21–39, 2022.
https://doi.org/10.1007/978-3-030-99253-8_2

Konstanzer Online-Publikations-System (KOPS)
URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-2-1ix1zxsfcqu311

in the over-approximation guarantees the correctness of the program. However, over-approximations cannot be used to expose bugs, since any alert raised by the analyser may be caused by the over-approximation rather than by the program, i.e. it can be a so called false alarm. From the point of view of a software developer, false alarms are undesirable because they undermine the credibility and usefulness of the analysis. In principle, there is a symmetrical approach to static analysis, that is to compute an under-approximation of the semantics, i.e., a subset of all possible behaviours of a program. Dually to over-approximations, under-approximations can then expose defects in the code, while they are unable to show their absence.

Early works on static analysis, like Hoare logic [13], focused on over-approximation to prove the absence of errors, and maybe their influence directed the focus toward over-approximations. Recently O'Hearn argued for the relevance of bug catching with respect to correctness proofs and proposes the Incorrectness Logic [19], a dual version of Hoare logic thought from the ground up for under-approximation. He also advocates for a similar change of perspective in the static analyses approach.

For instance, 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 the above program, the analysis may detect that the value of variable `i` is 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 $[-5, 3]$. This raises two warnings for possible division by zero, since it seems that both arithmetic expression 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, 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 [6,22,4] 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. Given its broad applicability, in his paper on Incorrectness Logic [19], 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*". The goal of this work is to investigate this topic. The results we have achieved will establish that under-approximation based Abstract Interpretation analyses have serious intrinsic limitations, and therefore our contribution can be read as a negative answer, even if we will then discuss how to overcome some limits.

Related Work: In their first works on Abstract Interpretation [6], Cousot and Cousot introduced the formal theory that could be used to define either over- or under-approximations. However, while the former has been extensively studied, there have been only sparse studies on the latter. Bourdoncle [2] proposed abstract debugging using over-approximation domains, but acknowledged that under-approximation ones could be better suited. Lev-Ami et al. [14] proposed to use complements of over-approximation domains to infer sufficient precondition for program correctness. For the same goal, Miné [15] used directly over-approximation domains, giving up the best abstraction and handling the choice of a maximal one with heuristics. To infer necessary condition for incorrectness, a problem similar to O'Hearn's but studied for a different goal, Cousot et al. [9,8] use Abstract Interpretation techniques but on boolean formulas, hence by-passing the issue of defining an abstract domain. Schmidt [24] uses higher-order domains, defining abstract states with 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. So the question whether it is possible to design an abstract domain for computing under-approximations naturally arises.

Contributions: We believe the absence of under-approximation abstract domains to be caused by intrinsic difficulties in their design. In this article, we determine and explain the reasons behind these difficulties. In the following we point out some intuitive asymmetries that suggest why under-approximations are not as immediate to use as over-approximations for program analysis.

While over- and under-approximation can be thought as dual theories, they have a deep asymmetry when dealing with the semantics of basic constructs of the language, the so called basic transfer functions. For instance, given an over-approximation abstract domain, we can define an under-approximation domain by taking the opposite interpretation of abstract elements: the idea is that an abstract element represents all concrete elements that may not be present in the set of possible values. As a consequence of being an under-approximation, this means that all the other concrete elements (the complement of the set) *must* be actual values. Considering the abstract domain of (complemented) intervals, it happens, e.g. that an arithmetic expression such as a sum of variables is often under-approximated as the whole $\mathbb{Z}$. It is also worth noticing that, while basic transfer functions are the same, over-approximation abstract domains are closed under intersection, while under-approximation abstract domains are closed under union and can grow large very easily.

Another asymmetry we point out is the handling of divergence. Divergence is represented in over- and under-approximation by the same abstract element $\bot$, but note that $\bot$ as an under-approximations also represents the absence of information (dually to $\top$ in over-approximations). This becomes a problem since many concrete functions are strict, that is, when applied to a non-terminating expression, they also fail to terminate (they 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 can't determine any meaningful information at some program point, it has to propagate this absence of information along all program paths, at least until a join in the control flow is found. So "recovery" from $\bot$, that is, producing a result different from $\bot$, once we start with it, is very hard in an under-approximation. Note that, on the contrary, "recovery" from $\top$ in an over-approximation is quite easier, e.g. by a constant assignment.

The previous arguments are substantiated by formal impossibility results for building meaningful under-approximation abstract domains. First, we introduce the new definition of *non emptying function*, describing functions that don't tamper the analysis and we prove that no abstract domain for integers can be constructed that makes all sums non emptying. Second, we propose two generalizations (one local and one global) of the result for integers domains to arbitrary concrete domains and function families, by introducing the notion of *highly surjective function family*, 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. Finally, we study hypothesis for the existence of abstract domains making all functions in a family non emptying to show first that the hypothesis of high surjectivity is tight, and then that further conditions on the function family must hold.

Structure of the paper: In Section 2 we introduce the notation used in the rest of the paper and recall the basics of Abstract Interpretation for over- and under-approximations. In Section 3 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 4 we extend the result obtained for integers to arbitrary concrete domains and function families. In Section 5 we show that the hypothesis of high surjectivity is needed and explore other requirements for the function family. Section 6 contains some concluding remarks and an outline of future research directions. Due to space limitation, only informal proof sketches are included in this proceedings.

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 $f(X) = \{f(x) \,|\, x \in X\}$ 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. 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 S^{op} comes 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 $s \in S$, $f(s) \preceq g(s)$. Any powerset is a complete lattice with ordering given by the inclusion relation. In this case, we use standard symbols $\subseteq$, $\cup$, etc.

Abstract Interpretation. Abstract Interpretation [6,7,16] 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 Interpretations are Galois connections. Given two complete lattices C and A, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ define a Galois connection (GC) when

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

and we denote it with $\langle C \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} A \rangle$. 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 Galois insertion (GI), and if this is the case also 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 it only contains $\top_C$.

Given a monotone function $f : C \to C$ and a GC $\langle C \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} A \rangle$, 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 of all the correct approximation of f.

As an example, let us consider $C = \mathcal{P}(\mathbb{Z})$ be the powerset of integers and $A = \mathsf{Int}$ be the abstract domain of intervals [6]. 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 on $S = \{1\}$ this yields $[0, 1]$ while $f(S) = \{1\}$. Actually the best correct abstraction f^A is computed as

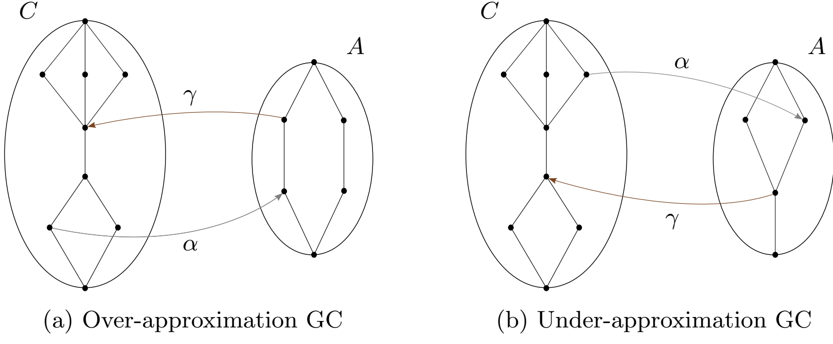
$$f^A([n, m]) = \alpha \circ f \circ \gamma([n, m]) = \begin{cases} [0, \max(|n|, |m|)] & \text{if } n \leq 0 \leq m \\ [n, m] & \text{if } 0 < n \\ [-m, -n] & \text{if } m < 0 \end{cases}$$

2.1 Under-approximation Galois Connections

The definition of GC is not symmetric in γ and α: it favours over-approximation, and is not suited to describe under-approximations. This can be more easily seen from the property $\mathrm{id}_C \preceq \gamma \circ \alpha$, that means the abstraction $\gamma(\alpha(c))$ of a concrete element c is greater than (ie. an over-approximation of) c itself. For this reason we introduce the notion of under-approximation Galois connection (UGC). Formally, an UGC is just a GC between A and C, in the reverse order, or equivalently a GC in which we replaced C and A with C^{op} and A^{op}. However, we believe this definition to allow a better notation, helping the reader's intuition. Given two complete lattices C and A, a pair of monotone functions $\alpha : C \to A$, $\gamma : A \to C$ defines an 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 such UGC 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 $\langle C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A \rangle$. The difference



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

Fig. 1: Sketches of GC and UGC

between a GC and an UGC is sketched in Figure 1: in the GC (on the left) γ is above and α below, while in the UGC (on the right) the two are reversed. Using the duality observed above, from standard properties of GCs we get, reversing inequalities, that $\gamma \circ \alpha \preceq \mathrm{id}_C$, $\mathrm{id}_A \preceq \alpha \circ \gamma$, γ preserves lubs and α preserves glbs. Moreover, an under-approximation Galois insertion (UGI) is an UGC in which $\alpha \circ \gamma = \mathrm{id}_A$, and has the properties of α being onto and γ being injective, making the same identification of A with $\gamma(A)$ possible, written as $\langle C \overset{\alpha}{\rightleftarrows} A \rangle$. In particular, this means that in an UGI on a concrete powerset $\langle \mathcal{P}(C) \overset{\alpha}{\rightleftarrows} A \rangle$, for all $a, a' \in A$, $\gamma(a \cup a') = a \cup a'$, that is A is closed under union.

Dually to standard, over-approximation GCs, given a monotone function $f : C \to C$ and an 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 approximation of f.

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, ie. $\mathsf{Int}_0 = \{I \in \mathsf{Int} \,|\, 0 \in I\} \cup \{\bot\}$. 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 intersects 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 Integer Domains

In this section we focus on under-approximations of integer domains and prove that any under-approximation abstract domain will mostly return trivial analyses for programs that include sums inside arithmetic expressions.

To this aim, we introduce the concept of *non emptying function.*

Definition 1 (Non emptying function). *Let $\langle C \overset{\alpha}{\underset{\gamma}{\rightleftarrows}} A\rangle$ be an 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 $f^A(\alpha(c)) \neq \bot$.*

Remember that $\bot$ does not give any interesting information in the under-approximation setting, because it can mean divergence as well as complete loss of precision. On the contrary, any abstract element different than $\bot$ means "something" interesting. The rationale behind the definition of non emptying function is that if the analysis starts from something ($\alpha(c) \neq \bot$) and it can find something ($\alpha(f(c)) \neq \bot$) then it will find at least one of the possible results ($f^A(\alpha(c)) \neq \bot$), thus not falling to $\bot$ and avoiding the issues discussed in the Introduction. The meaning of Definition 1 is illustrated by the following toy example.

Example 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 $\mathsf{Int}_{01} = \{I \in \mathsf{Int} \,|\, 0 \in I \vee 1 \in I\} \cup \{\bot\}$, 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.

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, that is $[1;10]$. The abstract analysis of the cycle then proceeds straightforwardly, finding $\perp$ after one iteration of the loop body (since after the increment the set of values for x is $\{2,3,\dots,11\}$ that is abstracted to $\perp$ because it doesn't contain neither 0 nor 1) and so the abstract fixpoint of the loop $[1;10]$. This yields no error, even though the concrete execution starting at $x=-1$ does indeed fail after one iteration. The issue here is that the semantics f of the increment $x := x+1$ is not non emptying in Int_{01}: on the concrete value $c=\{-1,1,2,\dots,10\}$, its input in this program, we have $\alpha(f(c)) = \alpha(\{0,2,3,\dots,11\}) = [0] \neq \perp$ but $f^A(\alpha(c)) = f^A([1;10]) = \alpha(f(\gamma([1;10]))) = \alpha(\{2,3,\dots,11\}) = \perp$.

For the remainder of the paper we assume a set of concrete *values* C, an UGI $\langle \mathcal{P}(C) \overset{\alpha}{\rightleftarrows} A \rangle$ with concrete domain $\mathcal{P}(C)$, and we say an element $S \in \mathcal{P}(C)$ is *representable* if it belongs to A, or equivalently if $\alpha(S) = S$.

Definition 3. *Let $S \subseteq C$ be a subset of C. We say that $d \in C$ is* representable with S *if $S \cup \{d\}$ is representable. We call $R(S)$ the set of elements of C representable with S, ie.*

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

For the sake of brevity, we shall write R for $R(\emptyset)$, the set of representable values of C, and $R(c)$ for $R(\{c\})$ where $c \in C$ is any concrete value. The following is a technical lemma valid for non emptying functions, that explains the role played by Definition 1 in proving all our negative results (Propositions 7, 10 and Theorems 12, 15).

Lemma 4. *Let $f : C \to C$ be non emptying, $c \in R$ and the pair $\{c, \bar{c}\}$ be not representable, ie. $\bar{c} \notin R(c)$. If $f(\bar{c}) \in R$ then also $f(c) \in R$.*

The main proof line of all our impossibility results is the same, and exploit this Lemma. All our results requires the size of the abstract domain to be comparable with that of the set of concrete values C (whose powerset $\mathcal{P}(C)$ is the concrete domain), and this in turn implies that representable elements are few. Then, assuming that all functions in a certain family are non emptying, we use repeatedly Lemma 4 to get many new representable elements, thus finding a contradiction. The key issues in the proofs are two: first, it must be possible to apply Lemma 4; second, all the new representable elements obtained applying it must be different from one another. In the following, we present some sets of conditions that are able to guarantee these two points, hence getting hypothesis for non existence of under-approximation abstract domain.

3.1 Infinite Integer Domain

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

Assumption 5 *We assume that an abstract domain A, to be feasible for analyses, must be at most countable.*

We make this assumption because we want to represent abstract elements with an amount of bits comparable with that of concrete *values*, to have a complexity comparable with a single concrete execution of the program and not exponentially larger. Thus, we require the size of the abstract domain to be that of $\mathbb{Z}$, the set of values handled by the program, and not the concrete domain $\mathcal{P}(\mathbb{Z})$. Many abstract domains satisfy it, for instance intervals, octagons and polyhedrons with at most n edges, for any n; some, such as general polyhedrons, don't, but they also exhibit a worst case exponential cost.

Based on Assumption 5, we prove a simple cardinality estimate that is used, as anticipated before, to prove that there are few representable elements.

Lemma 6. *For any fixed subset $S \subseteq \mathbb{Z}$, $R(S)$ is 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, that is in contradiction with the previous lemma that says that R is finite. In order to define such a sequence, we want to use Lemma 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 7. *Let $\langle \mathcal{P}(\mathbb{Z}) \overset{\alpha}{\rightleftarrows} A \rangle$ be an UGI, and assume that there is an integer n_0 that is representable. Then it can't be the case that all the functions of the form $f_n(x) = x + n$ are non emptying in A.*

The meaning of this proposition for program analysis is the fact that a domain small enough (by Assumption 5) is probably unable to deduce meaningful informations on an integer domain: if it doesn't contain representable singletons it must abstract to $\bot$ any variable initialization, and otherwise it can't 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, that means exactly it can't determine any information. This issue is not bound to manifest for all programs, but for any domain there exists programs for which it does.

3.2 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 the interval is taken symmetric around 0 to simplify notation, but there is no conceptual difficulty in using an asymmetric one.

Assumption 8 *We assume that an abstract domain A, to be feasible, must have a cardinality that is polynomial in N.*

This assumption guarantees that the number of bits required to represent an abstract element is linear in that for concrete elements so that, again, the cost of the analysis is polynomial and not exponential in that of a concrete execution.

In the following we'll use asymptotic notation for some quantities. For this to be completely formal we should define a sequence of abstract domain A_N, each one for the concrete domain $\mathcal{P}([-N, N])$, then define a sequence of values for each quantity we want to estimate, and take the limit of this sequence for N going to infinity. However we do believe all these formal details would clutter notation, making hard to get insight. For this reason, we avoid all this, just (ab)using the intuitive meaning associated with the notation.

The next lemma is analogous to Lemma 6 in proving that some sets are small under Assumption 8 on the cardinality of A.

Lemma 9. *For any fixed subset $S \subseteq \mathbb{Z}$, $|R(S)| = O(\log(N))$.*

The following proposition uses the same proof line as Proposition 7 above: we define a sequence of representable elements, and prove that they are too many since, by the previous lemma, R is quite small.

Proposition 10. *Let $\langle \mathcal{P}([-N, N]) \overset{\alpha}{\rightleftarrows} A \rangle$ be an under-approximation Galois insertion, and assume that there is an integer n_0 that is representable. Then it can't be the case that all the functions of the form $f_n(x) = x + n$ (modulo $2N + 1$) are non emptying in A.*

4 Arbitrary domains

The definition of non emptying function is fully general and not limited to the concrete integer domain, hence we use it to propose conditions that are independent of the concrete domain. In this section, we deal with an infinite set C of concrete values, and an UGI $\langle \mathcal{P}(C) \overset{\alpha}{\rightleftarrows} A \rangle$. Again, we take the Assumption 5 on the size of A. Under this assumption we can prove again Lemma 6, that doesn't 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 11 (Highly surjective function family). *Given a family F of functions from C to itself and an element $c \in C$, let*

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

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* $P(c)$ *is infinite for any possible choice of* $c \in C$.

This property is needed together with Lemma 6 to apply Lemma 4 and get a new representable element: since there are infinite preimages of c but $R(c)$ is finite, there are elements $\bar{c} \in P(c)$ not in $R(c)$; then by definition of $P(c)$ there is an f such that $f(\bar{c}) = c \in R$, so we can apply the lemma to get $f(c) \in R$. The reason for requiring $f(\bar{c}) = c$ instead of just in R is that, at the beginning of the proof, we only assume R to contain one element, hence the two conditions are equivalent. Starting from this basic idea, we present two set of sufficient conditions to prove the non existence of any under-approximation abstract domain.

4.1 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 on the other.

Theorem 12. *Let* F *be an highly surjective function family from* C *to itself such that all functions* $f \in F$ *are either injective or acyclic. Assume also that* R *isn't empty. Then* A *can't be non emptying for all* $f \in F$.

In the previous section we developed an ad hoc proof for the family of sums over integers, but the same result can also be obtained as an application of this theorem: if $C = \mathbb{Z}$ and $F = \{\lambda x.x + n \mid n \in \mathbb{Z}\}$, the family is highly surjective (actually $P(c) = \mathbb{Z}$ for all c) and all these functions are injective, so it meets the hypothesis of the theorem. Another example are rational or real numbers, with sums or products

Example 13. 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 $P(c) = \mathbb{Q} \setminus \{0\}$ for all c, and all these functions are invertible, hence injective.

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

Example 14. 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 make the choice of infinite precision exponents and finite number of significant digits in order to have an infinite domain, as required by the theorem, but also preserve characteristics of floating-point arithmetic.

Let $\cdot$ and $\odot$ denote respectively real product and its floating-point approximation, and consider the function family $F = \{\lambda x.x \odot y \mid y \in C\}$. The function family is highly surjective, eg. considering that all numbers with the same significant digits as a floating-point x but different exponent can be mapped into x multiplying them by 1 times 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, 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 hypothesis of Theorem 12, hence no abstract domain on floating-point numbers can be non emptying for all multiplications.

4.2 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.

Theorem 15. *Let F be an highly surjective function family from C in itself such that*

- *for all pair of elements $c, d \in C$ there exists at most a finite amount of $f \in F$ such that $f(d) = c$*
- *for all pair of an element $c \in C$ and a function $f \in F$, there exists at most a finite amount of elements $d \in C$ such that $f(d) = c$*

Assume also that R isn't empty. Then A can't be non emptying for all $f \in F$.

Again this result can be used to prove the impossibility of building an abstract domain for integers that is non emptying for all sums, or for floating-point numbers.

Example 16. Take $C = \mathcal{F} \setminus \{0\}$ the set of non-zero floating-point numbers with t bits significands and arbitrary precision exponents, and $F = \{\lambda x.x \odot y \,|\, y \in \mathcal{F} \setminus \{0\}\}$. As observed in Example 14 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 $y = f(x) = x \odot z$ only if

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

that is

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

This is a bounded interval since $x \neq 0$, and hence contains only a finite amount of floating-point numbers. Analogously, fixed a floating-point y and a function $f(x) = x \odot z$, we have that $y = x \odot z$ only if $|x|$ belong to a bounded interval, that contains a finite amount of floating-point numbers. So, by means of Theorem 15 above, we proved again that no abstract domain on floating-point numbers can be non emptying for all multiplications.

5 On the necessity of high surjectivity hypothesis

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

Proposition 17. *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*
- *all functions $f \in F$ are non emptying in A_F*

Moreover, the proof of this proposition is constructive, and we present an example of such construction in the following.

Example 18. Fix the pair of functions $f(x) = x - 1$ and $g(x) = x - 2$ on $\mathbb{Z}$. The family $F = \{f, g\}$ is clearly 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 $P(n_0)$ (computed with respect to F) is finite. With this F, any integer is fine, so let us fix $n_0 = 0$.

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

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

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

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, that is equivalent to $0 \in S$. Analogously, if $\alpha(f(S)) = \emptyset$ the condition is true, so assume $0 \in f(S)$ or, equivalently, $1 \in S$. Using these two 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 shouldn't 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 hypothesis required in the previous theorems is tight and can't be weakened. In particular, since these kind of results need 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:

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

Using this definition, the proposition generalizes straightforwardly:

Proposition 19. *Let F be a family of functions from C in itself, and assume there is a set $S_0 \subseteq C$ such that $P(S_0)$ 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.*

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

Example 20. 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

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

the function that prefix α to its argument. The family

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

is not highly surjective, because fixed a string γ only its prefixes can be mapped into it 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 19. Such domains are defined with a construction similar to that of Example 18, 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's 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 them 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

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

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

Proposition 21. *Let F be a family of total functions (ie. if $S \neq \emptyset$ then $f(S) \neq \emptyset$) from $\mathcal{P}(C)$ in itself, and assume there is a non empty set $S_0 \subseteq C$ such that $I(S_0)$ is finite. Then there exists a finite abstract domain A_F such that all functions $f \in F$ are non emptying in A_F.*

Even though this proposition introduces the technical hypothesis that all $f \in F$ are total, we don't believe this to be very restrictive because these theorems are intended to be applied when F is a family of basic transfer functions, that seldom introduce divergence: in programming languages this is often caused by control-flow constructs. An application of this proposition is again on lists, to rule out another natural function family.

Example 22. Fix again $C = \Gamma^*$, and consider functions $\text{drop}_n : \Gamma^* \to \Gamma^*$ 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

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

is highly surjective since, for any fixed list $\alpha \in \Gamma^*$ and any n, we can extend α with any n character, and map this list to α with drop_n. However, images through this function family are finite:

$$I(\alpha) = \{\text{drop}_n(\alpha) \mid n \in \mathbb{N}\}$$

that is finite since it's the set of all tails of α. Hence by Proposition 21 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 $I(S_0) = \{\epsilon\}$, it yields

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

Again it can be easily checked that all functions drop_n are non emptying in A_F.

These 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 isn't the case in the presence of "boundaries" in the concrete domain, that 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 go towards. Another example of such boundary is 0 in the domain of integers $\mathbb{Z}$ with respect to 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 kind of functions doesn't work: a slight adaptation of the constructions for the two propositions above shows that, if F can be partitioned in 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 20 and 22. In light of these observations, in order to apply effectively the definition of non emptying function to prove non existence of abstract domains, for all possible boundaries there is the need for a function that is able to both enter and exit it. This happens for integers, since there is no boundary, but doesn't for finite lists, with $\{\epsilon\}$ being often either a sink or a source for many functions on lists.

6 Conclusions and Future Works

Until recently, the focus of formal static analyses has been on over-approximation to prove program correctness, but many tools based on this theory are instead deployed to catch bugs [23,10]. Incorrectness Logic promoted the study of a theory for under-approximation to give a formal basis to a new class of tools. This has seldom been done in the last few decades, especially in the framework of Abstract Interpretation. In our work, we point out some asymmetries between over- and under-approximation in Abstract Interpretation, and why those are an obstacle to the design of abstract domains. We have identified functions as the main difference, because they remain the same in both over- and under-approximation thus preventing one theory to be obtained simply as a dual of the other. Handling of divergence is another critical issue. Building on those ideas, we have proposed the new (to the extent of our knowledge) definition of *non emptying function* and studied how it can be used to prove non existence of under-approximation abstract domains. We have presented some general results, and applied them to integer and floating point domains to conclude that, under some assumptions, there are no useful under-approximation domains. Then, we have found conditions under which there do exist under-approximation abstract domains, showing that some of the hypothesis required in our theorems are very tight. However, because of the scarcity of works in this direction, we believe there are many possible subjects for future research.

Under-approximation abstract domains must be closed under union, but known abstract domains are rarely such. However disjunctive completion [11], a known domain transformer, refines any abstract domain in a union-closed one. This has been studied for over-approximation in order to improve precision at the expense of increased complexity. A solution to keep the analysis feasible is to use heuristics to prune disjunctions, trading back complexity for precision, but making the analysis possible for under-approximations. Moreover, practical tools based on the theory of Incorrectness Logic already use heuristic to drop logical disjunctions [19], so taking inspiration from them may be effective also for Abstract Interpretation.

In their recent work, Raad et al. [20] study incorrectness separation logic, the join of separation logic [21] and Incorrectness Logic. They notice that the original separation logic doesn't 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, but a new point different from $\bot$ describing 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 (eg. as proposed for higher-order functional languages [5]), so it may be worth to investigate the possibility to change the concrete domain to account for this new point.

All our results depend on the existence of a 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 because functions defined as additive extensions don't increase cardinality, so they might take the place of singletons. The technical issue is if and how Lemma 4 can be generalized, but we believe it may be possible to relax that hypothesis about singletons.

We have discussed the finite domain of integers at the end of Section 3, but all our general results deal with infinite concrete domains. Both theorems rely on cardinality estimates essentially based on the fact that arbitrary combinations of finite numbers is still finite, hence less than the cardinality of the concrete domain. However, with a finite concrete domain those would be replaced by combinations of logarithmic factors, which may become equal to the size of the concrete domain. For finite domains we can prove a result reminiscent of Theorem 15, but this topic requires thorough investigation to understand the new issues and possibilities they open up.

Acknowledgements. We thank the anonymous reviewers for their helpful comments.

References

1. Boulanger, J.L. (ed.): Static Analysis of Software: The Abstract Interpretation. Wiley (2011)
2. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. SIGPLAN Not. **28**(6), 46–55 (Jun 1993). https://doi.org/10.1145/173262.155095
3. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM'15. LNCS, vol. 9058, pp. 3–11. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
4. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
5. Cousot, P., Cousot, R.: 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). pp. 95–112 (1994). https://doi.org/10.1109/ICCL.1994.288389
6. Cousot, P., Cousot, R.: 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. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). https://doi.org/10.1145/512950.512973
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. p. 269–282. POPL '79, Association for Computing Machinery, New York, NY, USA (1979). https://doi.org/10.1145/567752.567778

8. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7737, pp. 128–148. Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_10
9. Cousot, P., Cousot, R., Logozzo, F.: Precondition inference from intermittent assertions and application to contracts on collections. In: Jhala, R., Schmidt, D.A. (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6538, pp. 150–168. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_12
10. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019). https://doi.org/10.1145/3338112
11. Filé, G., Ranzato, F.: Improving abstract interpretations by systematic lifting to the powerset. In: Proceedings of the 1994 International Symposium on Logic Programming. p. 655–669. ILPS '94, MIT Press, Cambridge, MA, USA (1994)
12. Floyd, R.W.: Assigning meanings to programs. Proceedings of Symposium on Applied Mathematics **19**, 19–32 (1967)
13. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259
14. Lev-Ami, T., Sagiv, M., Reps, T., Gulwani, S.: Backward analysis for inferring quantified preconditions. Tr-2007-12-01, Tel Aviv University (2007)
15. Miné, A.: Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Sci. Comput. Program. **93**, 154–182 (Nov 2014). https://doi.org/10.1016/j.scico.2013.09.014
16. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Found. Trends Program. Lang. **4**(3–4), 120–372 (Dec 2017). https://doi.org/10.1561/2500000034
17. Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis. Springer (2010). https://doi.org/10.1007/978-3-662-03811-6
18. O'Hearn, P.W.: Continuous reasoning: Scaling the impact of formal methods. In: Proc. LICS'18. p. 13–25. ACM (2018). https://doi.org/10.1145/3209108.3209109
19. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371078
20. Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O'Hearn, P.W., Villard, J.: Local reasoning about the presence of bugs: Incorrectness separation logic. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 225–252. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_14
21. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
22. Rival, X., Yi, K.: Introduction to Static Analysis – An Abstract Interpretation Perspective. MIT Press (2020)
23. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at Google. Commun. ACM **61**(4), 58–66 (Mar 2018). https://doi.org/10.1145/3188720

24. Schmidt, D.A.: A calculus of logical relations for over- and underapproximating static analyses. Sci. Comput. Program. **64**(1), 29–53 (2007). https://doi.org/10.1016/j.scico.2006.03.008

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.