Abstract Interpretation: From 0, 1, To ∞

Patrick Cousot

Abstract This paper starts from zero knowledge about abstract interpretation and provides one rapid introduction for the untaught, goes rapidly over remarkable achievements, and widen to infinitely hard problems to be solved by the savant.

1 Introduction

Abstract interpretation has a long history (Giacobazzi and Ranzato, 2022). It appeared in the seventies to prove the correctness of program static analysis algorithms (Cousot and Cousot, 1976a, 1977a; Cousot, 1978b; Cousot and Cousot, 1979). This justification with respect to a reachability semantics and the use of extrapolators (widenings and their duals) and interpolators (narrowing and their dual) to accelerate fixpoint convergence in infinite abstract domains not satisfying the ascending chain condition was a breakthrough with respect to existing dataflow analysis methods (some of which being even syntactically correct but semantically wrong (Cousot, 2019d)). Then it appeared that the concepts of abstract interpretation also apply to the design of programming languages semantics at various level of abstraction (Cousot, 2002; Cousot and Cousot, 2009). Then further abstractions lead to program verification methods that are sound (as justified with respect to the semantics) and complete (true abstract properties of the program semantics can always be proved) (Alglave and Cousot, 2017; Cousot and Cousot, 2012). Finally, the static analysis algorithms are computable abstractions of proof methods (such as invariance/reachability in (Cousot and Cousot, 1977a)), hence sound but incomplete by undecidability.

This paper is an introduction to abstract interpretation for those with a minimal background in mathematics and no knowledge of abstract interpretation (based on a simplification of (Cousot, 2021b, Ch. 3). At the other extreme, it proposes research challenges for those that don't need this introduction.

Patrick Cousot
New York University, e-mail: pcousot@cs.nyu.edu

2 Abstract Interpretation for the Untaught

The main idea of abstract interpretation is that to prove properties of a program semantics, it is sufficient to restrict oneself to program properties necessary for the proof.

Wassily Kandinsky has a painting dated 1925 entitled "Abstract Interpretation" on view at Yale University Art Gallery (artgallery.yale.edu/collections/objects/43818). The abstraction keeps only shapes and colors maybe reorganized to form a composition.

The very first mathematical example (I know of) of abstract interpretation is by the Indian mathematician Brahmagupta, c. 598 – c. 665 CE who invented the zero and postulated the signs rule for +, −, × and / (Plofker, 2007).

> …
> 18.32. A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.
> …

His is only error was $\frac{0}{0} = 0$, nowadays undefined.

A modern formulation in static program analysis is the following table for the minus operation where $\top_\pm$ denotes the unknown sign (that is either $\leqslant 0$ or $\geqslant 0$) and $\bot_\pm$

$s_1 -_\pm s_2$	s_2 $\bot_\pm$	<0	$=0$	>0	$\leqslant 0$	$\neq 0$	$\geqslant 0$	$\top_\pm$
s_1 $\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
<0	$\bot_\pm$	$\top_\pm$	<0	<0	$\top_\pm$	$\top_\pm$	<0	$\top_\pm$
$=0$	$\bot_\pm$	>0	$=0$	<0	$\geqslant 0$	$\neq 0$	$\leqslant 0$	$\top_\pm$
>0	$\bot_\pm$	>0	>0	$\top_\pm$	>0	$\top_\pm$	$\top_\pm$	$\top_\pm$
$\leqslant 0$	$\bot_\pm$	$\top_\pm$	$\leqslant 0$	<0	$\top_\pm$	$\top_\pm$	$\leqslant 0$	$\top_\pm$
$\neq 0$	$\bot_\pm$	$\top_\pm$	$\neq 0$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$
$\geqslant 0$	$\bot_\pm$	>0	$\geqslant 0$	$\top_\pm$	$\geqslant 0$	$\top_\pm$	$\top_\pm$	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$

Fig. 1 Rule of signs

means no sign e.g. nothing is known before evaluating the sign of the expression.

Given a mathematical definition of integers $\mathbb{Z}$, the modern understanding of signs is a property of integers $\mathbb{Z}$ (or rationals $\mathbb{Q}$ or reals $\mathbb{R}$) represented in extension.

For example, the meaning of "$\geqslant 0 -_\pm \leqslant 0 = \geqslant 0$" is "Take any number n in $\{0, 1, 2, \ldots\}$ and any number m in $\{\ldots, -2, -1, 0\}$ then their difference $n - m$ is in $\{0, 1, 2, \ldots\}$ where $\{0, 1, 2, \ldots\}$ is an extensional definition of the concrete positive numbers (enumerating all of them) and $\{\ldots, -2, -1, 0\}$ is an extensional definition of the concrete negative numbers (enumerating all of them). Therefore,

- >0 is an abstraction of the strictly positives $\{1, 2, 3, \ldots\}$;
- $\geqslant 0$ is an abstraction of the positives $\{0, 1, 2, \ldots\}$;
- $=0$ is an abstraction of $\{0\}$ (to be equal to zero);
- $\neq 0$ is an abstraction of $\{\ldots, -2, -1, 1, 2, \ldots\}$ (to be different from zero);
- $\leqslant 0$ is an abstraction of the negatives $\{\ldots, -2, -1, 0\}$;
- <0 is an abstraction of the strictly negatives $\{\ldots, -3, -2, -1\}$;
- $\top_\pm$ is an abstraction of all numbers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$;
- $\bot_\pm$ is an abstraction of the empty set $\emptyset = \{\}$.

The signs rule "$\geqslant 0 -_\pm \leqslant 0 = \geqslant 0$" means that all possibilities

$$\{n - m \mid n \in \{0, 1, 2, \ldots\} \text{ and } m \in \{\ldots, -2, -1, 0\}$$

are included in

$$\{0, 1, 2, \ldots\}$$

.

The concretization function provides the meaning of signs as sets of integers.

$$\begin{array}{ll} \gamma_\pm(\bot_\pm) = \emptyset \quad \text{(that is } \{\}) & \gamma_\pm(\leqslant 0) = \{\ldots, -2, -1, 0\} \\ \gamma_\pm(<0) = \{\ldots, -3, -2, -1\} & \gamma_\pm(\neq 0) = \{\ldots - 2, -1, 1, 2, \ldots\} \\ \gamma_\pm(=0) = \{0\} & \gamma_\pm(\geqslant 0) = \{0, 1, 2, \ldots\} \\ \gamma_\pm(>0) = \{1, 2, 3, \ldots\} & \gamma_\pm(\top_\pm) = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \end{array} \tag{1}$$

The abstraction function provides the sign abstraction of properties of integers (that is a set S of integers)

$$\begin{array}{l} \alpha_\pm(S) = \text{if } S = \emptyset \text{ then } \bot_\pm \\ \qquad \text{else if } S \subseteq \{\ldots, -2, -1\} \text{ then } <0 \\ \qquad \text{else if } S \subseteq \{0\} \text{ then } =0 \\ \qquad \text{else if } S \subseteq \{1, 2, \ldots\} \text{ then } >0 \\ \qquad \text{else if } S \subseteq \{\ldots, -2, -1, 0\} \text{ then } \leqslant 0 \\ \qquad \text{else if } S \subseteq \{0, 1, 2, \ldots\} \text{ then } \geqslant 0 \\ \qquad \text{else if } S \subseteq \{\ldots, -2, -1, 1, 2, \ldots\} \text{ then } \neq 0 \\ \qquad \text{else } \top_\pm \end{array} \tag{2}$$

$\alpha_\pm(S)$ is the best we can say on the sign, e.g.

$$\text{The absolute value is one} \qquad \alpha_\pm(\{-1, 1\}) = \neq 0 \tag{3}$$

$$\text{The absolute value is less that one} \quad \alpha_\pm(\{-1, 0, 1\}) = \top_\pm \tag{4}$$

Here "best" means "the most precise" that is "included in". For example, $\{1\}$ is the property "to be one", which implies "to be a positive integer", which implies "to be a positive or zero integer", which implies "to be an integer", that is $\{1\} \subseteq \{1, 2, 3, \ldots\} \subseteq \{0, 1, 2, 3, \ldots\} \subseteq \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$.

The idea of the rule of signs is to take the meaning of any two sign, consider all possibilities for the minus, and abstract the result.

$$s_1 -_\pm s_2 = \alpha_\pm(\{n - m \mid n \in \gamma_\pm(s_1) \text{ and } m \in \gamma_\pm(s_2)\}) \tag{5}$$

This formula (5) means that taking any number $n \in \gamma_\pm(s_1)$ with sign s_1 and any number $m \in \gamma_\pm(s_2)$ with sign s_2, the sign of their difference $n - m$ must be $s_1 -_\pm s_2$. The formula (5) uses sets to express that the rule of signs is valid for any number of a given sign by considering the sets $\gamma_\pm(s_1)$ and $\gamma_\pm(s_2)$ of all of them and then considering the best sign abstraction $\alpha_\pm(\{n - m \mid n \in \gamma_\pm(s_1) \text{ and } m \in \gamma_\pm(s_2)\})$ of the set $\{n - m \mid n \in \gamma_\pm(s_1) \text{ and } m \in \gamma_\pm(s_2)\}$ of all their possible differences.

Figure 1 is designed by calculating $s_1 -_\pm s_2$ for all possibilities $s_1, s_2 \in \{\bot_\pm, <0, =0, >0, \leqslant 0, \neq 0, \geqslant 0, \top_\pm\}$. A simple example is $\bot_\pm -_\pm \top_\pm = \bot_\pm$, as follows.

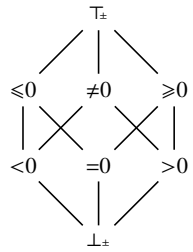
$$\begin{aligned}
& \bot_\pm -_\pm \top_\pm \\
= \ & \alpha_\pm(\{n - m \mid n \in \gamma_\pm(\bot_\pm) \text{ and } m \in \gamma_\pm(\top_\pm)\}) && \text{by (5)} \\
= \ & \alpha_\pm(\{n - m \mid n \in \emptyset \text{ and } m \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}\}) && \text{by (1)} \\
= \ & \alpha_\pm(\{n - m \mid \text{false}\}) && \text{since } n \in \emptyset \text{ is false in the conjunction} \\
= \ & \alpha_\pm(\emptyset) && \text{definition of the empty set } \emptyset \\
= \ & \bot_\pm && \text{definition (2) of } \alpha_\pm.
\end{aligned}$$

Such calculations are foundations of the calculation design of abstract interpreters (Cousot, 1999a, 2021a).

The idea of "best abstraction" is that among the possible abstractions of sets of integers by signs (considered as sets) there is always a smallest one (for inclusion $\subseteq$). This "best" or "more precise" abstraction is given by $\alpha_\pm$. This 'best abstraction" follows from the fact that the pair $(\alpha_\pm, \gamma_\pm)$ of functions is a Galois connection, a generalization by the mathematician Oyster (Ore, 1944) of an algebraic concept introduced by Évariste Galois. This means that

$$\forall S, s \ . \ \alpha_\pm(S) \sqsubseteq_\pm s \iff S \subseteq \gamma_\pm(s).$$

The partial order $\sqsubseteq_\pm$ is defined by the following Hasse diagram



such that, for example, $\bot_\pm \sqsubseteq_\pm >0 \sqsubseteq_\pm \leqslant 0 \sqsubseteq_\pm \top_\pm$ while <0, $=0$, and >0 are not comparable by $\sqsubseteq_\pm$. We have $s \sqsubseteq_\pm s' \iff \gamma_\pm(s) \subseteq \gamma_\pm(s')$ so that $\sqsubseteq_\pm$ in the abstract is set inclusion $\subseteq$ in the concrete, that is logical implication. For example <0 implies $\leqslant 0$. Because of this equivalence, every property true in the abstract will also be true in the concrete.

Observe that reflexivity $\alpha_\pm(S) \sqsubseteq_\pm \alpha_\pm(S)$ implies $S \subseteq \gamma_\pm(\alpha_\pm(S))$. Therefore the meaning of the sign is an over-approximation of the concrete property. For example $\{1, 42\} \subseteq \gamma_\pm(\alpha_\pm(\{1, 42\})) = \gamma_\pm(>0) = \{1, 2, 3, \ldots, 42, \ldots\}$. This is called soundness, when abstracting to sign, we consider over approximations so no concrete case is ever omitted.

Any sign s such that $S \subseteq \gamma_\pm(s)$ is an overapproximation of S. For example $\{1, 42\} \subseteq \gamma_\pm(>0) \subseteq \gamma_\pm(\geqslant 0) \subseteq \gamma_\pm(\top_\pm)$ and $\{1, 42\} \subseteq \gamma_\pm(\neq 0)$. But $\alpha_\pm(S)$, that is >0 in our example, is the most precise. This is because if $S \subseteq \gamma_\pm(s)$ then $\alpha_\pm(S) \sqsubseteq_\pm s$ by definition of a Galois connection. So $\alpha_\pm(S)$ is the best (or most precise) abstraction of S as a sign.

Such connections were invented by Évariste Galois to provide a connection between field theory (originally the field of rationals) and group theory (group of permutations of the polynomial roots), allowing to reduce certain problems in field theory to group theory, which makes them simpler and easier to understand.

Galois characterized in this way the polynomial equations that are solvable by radicals in terms of properties of the permutation group of their roots. An equation is solvable by radicals if its roots are a function of the polynome coefficients using only integers, $+$, $-$, $\times$, $/$, and n-th root operation. This implies the Abel–Ruffini theorem asserting that a general polynomial of degree at least five cannot be solved by radicals.

Galois theory has been used to solve classic problems including showing that two problems of antiquity cannot be solved as they were stated (doubling the cube and trisecting the angle), and characterizing the regular polygons that are constructible (this characterization was previously given by Gauss, but all known proofs that this characterization is complete require Galois theory).

Other elementary examples of Galois connections are the casting out of nine (invented by Carl Friedrich Gauss, to check a multiplication), Paul Bachmann and Edmund Landau's Big O notation (e.g. $log\ n + 10n^2 + n^3 = O(n^3)$ when $n \to \infty$), the mean (expected value), etc.…

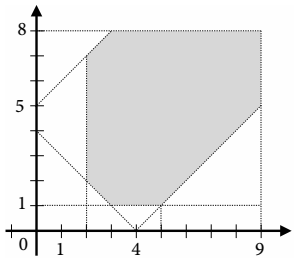
The main idea of abstract interpretation if to use abstraction to simplify hard problems to provide (partial) solutions. The abstraction may be both simple and precise (e.g. the rule of signs for multiplication). The solution may also be *partial*, meaning that the abstraction may be too imprecise to give a precise answer (e.g. $>0 -_\pm >0 = \top_\pm$). But the abstraction can always be refined to get a more precise answer, as understood by Brahmagupta, (Plofker, 2007).

> …
> 18.31. [If] a smaller [positive] is to be subtracted from a larger positive, [the result] is positive, ….

However the refined abstraction might not be computable by a machine, so finding efficiently computable and precise abstractions is a very difficult art!

Besides applications to the design of semantics of programming languages and program proof methods, the main application of abstract interpretation is to program analysis.

An example is octagonal analysis invented by Antoine (Miné, 2006) where a set of points is enclosed within an octagon.



Given the hypothesis $n \geqslant 0$ on the input value n of the program parameter `n`, an octagonal analysis of the following program

```
{n ⩾ 0} (hypothesis on input parameter n)
i = 0;
while (i < n) {i ⩾ 0}
    {0 ⩽ i ⩽ n-1}
    i = (i + 1);
{i = n}
```

will automatically discover the invariants *{ ... }* which holds of the values of the variables each time execution reaches the program where they are given. This abstract domain in combination with dozens of other ones is used in the Astrée static analyzer (Cousot et al., 2005) which scales to over 10 000 000 lines of C/C++ code. Astrée is developed and commercialized by AbsInt (`www.absint.com/astree/index.htm`) and used by thousands of engineers in the industry.

3 Abstract Interpretation for the Savant

3.1 Software engineering

Most computer scientists start by learning a programming language through debugging rather than by relying on typing, static analysis, and verification tools. While advancing in their studies, they will be lucky to have one or two classes on static analysis, mainly in compilation courses, using standard examples from dataflow analysis. No one cares about soundness/correctness in compilation classes and teaching a semantically incorrect dataflow algorithm (see e.g. Cousot (2019d)) is not a problem (with very few exceptions, of course (Leroy, 2016)). When becoming professionals, programmers are encouraged to produce rapidly rather than to deliver products of high quality. No one feels the need to use static analysis tools since this might slow down the delivery of the final product, which anyway will be debugged by the end users, supposedly at no cost for the programmer.

There are several reasons for that situation beyond that of education. First many static analysis analyzers are academic tools designed for competitions on small programs. Most do not to scale up, and are not usable for production-quality software. The second is that, contrary to other industries, programmers are not responsible for their bugs, and so feels no need to use tools. The third is that if, exceptionally, there is an obligation to use a static analysis tool, it is always possible to find a cheap poor tool offering no guarantee. There is no obligation for tools to be qualified. For example, in 2020, the US National Institute of Standards and Technology determined Astrée and Frame C, both abstract interpretation based, to be the only two tools that satisfy their criteria for sound static code analysis (Black and Walia, 2000). The tools not satisfying the criteria are not cited but claim[1] "Don't overestimate the limited value of standard test suites such as Juliet[2]. These suites often exercise language features that are not appropriate for safety-critical code. Historically, the overlap between findings of different tools that were run over the same Juliet test suite has been surprisingly small." (It does not look to be understood that unsound static analysis tools may produce different sets of alarms, which intersection may even by empty.)

Fortunately, the above pessimistic picture of the state of the art of static analysis in research and industrial applications is counter-balanced by outstanding successes.

Among these stands Astrée used in aeronautic, automotive, and nuclear industry to check for runtime errors and standards (such as DO-178B/C, ISO 26262, IEC 61508, EN-50128, Misra, etc.) in safety critical C or C++ software. Zoncolan is used at Facebook to detect and prevent security issues in Hack code (Logozzo et al., 2019). It finds 70% of the privacy bugs found every day in the ever changing 100 million lines code base.

For the field to make significant progress, there are many research and education subjects on which to contribute. We discuss a few possible research objectives, other ones are considered in (Cousot, 2005b; Cousot and Cousot, 2014). Some are hard unresolved issues pending for years (Cousot, 1999b) for which progress has been much needed, continuous, but slow. There are also emerging subjects and applications. It seems that there is no limit since abstract interpretation has accompanied the evolution of computer science over several decades since, as originally shown by Galois, the idea of understanding a complex structure using a more abstract and simple one looks to be a universally applicable idea.

3.2 Education

Successful projects in static analysis have mostly been developed by researchers knowing well abstract interpretation, some with a very deep knowledge and experience. But it is not easy to acquire the necessary knowledge and experience starting from zero. There are numerous introductions (Cousot (1996a,b, 1999a); Cousot and

[1] `https://www.synopsys.com/content/dam/synopsys/sig-assets/whitepapers/coverity-risk-mitigation-for-do178c.pdf`

[2] Juliet Test Suites are available at `https://samate.nist.gov/SRD/testsuite.php`.

Cousot (2010a, 2004); Miné (2017); Cousot and Cousot (2010b); Cousot (2019b) among others) or videos[3], but classes on abstract interpretation are necessary to go more in depth. Classes going deep into the intricacies of abstract interpretation are proposed in a few universities, sometimes online[4]. Recently, books on abstract interpretation (Rival and Yi, 2020; Cousot, 2021b) have been published which can also be very helpful.

3.3 Scope of Abstract Interpretation

Abstract interpretation started in the mid-seventies by interval analysis of flowchart programs (Cousot and Cousot, 1976b) and, although some imagine that it is still there, it evolved as a general theory to approximate the possible semantics (i.e. formal specifications of the runtime computations) of computer programs, with a sound construction methodology. Abstract interpretation can be used to design static analyzers, type systems, proof methods, and semantics. It can be both operational (by reasoning on program steps), structural (by reasoning on the denotation of program components), and compositional (by composing diverse abstractions).

Many other methods of reasoning on program semantics such as type theory, symbolic execution, bisimulation, security analysis, etc. have their own way of thinking about programs and expressing soundness. These are also abstract interpretations, although this is mostly not understood. Although generality by abstraction is the main aim in pure mathematics where merging several theories into a single formalization is considered a progress, this is not the case in computer science where the multiplication of concepts, specialized by communities, does not encourage cross-fertilization. Such unification would make formal methods more easily teachable, applicable, and composable.

3.4 More Complex Properties

Most static analyzers aim at discovering semantic properties quantifying on one execution trace at a time, mainly safety properties such as reachability. Liveness properties, such as termination, have also been considered Cousot and Cousot (2012), which require sophisticated abstract domains and widenings (Urban and Miné, 2017; Urban et al., 2018).

More subtle properties than safety and liveness involve quantification on two execution traces at a time like ,dependency analysis (Cousot, 2019a). When abstracted to a property on one execution trace (like taint analysis for dependency in Tripp et al. (2013)), there is a loss of precision. And this is even more difficult with properties such as responsibility involving the comparison of any execution of a program with

[3] `https://www.youtube.com/watch?v=IBlfJerAcRw`

[4] such as the slightly outdated `https://web.mit.edu/16.399/www/` d.

all its other possible executions (Deng and Cousot, 2022). Much more foundational work is needed to abstract higher-order program properties (including so-called hyper properties) precisely and efficiently.

3.5 Properties of More Complex Data Structures

Most static analyzers have been designed for languages where taking into account simple properties of the data structures manipulated by programs such as integers, floats, pointers, arrays, etc. is sufficient to report few and significant alarms. There are even libraries of abstract domains that can be reused for analyzing these properties, mainly numerical ones (such as Apron (Jeannet and Miné, 2009) and Elina (Singh et al., 2015, 2017)). Beyond linearity (i.e. intervals in one dimension and affine subspaces in Euclidian spaces), abstractions by non-linear (closed) convex sets and functions (Hiriart-Urruty and Lemaréchal, 2004) is very challenging but necessary in the analysis on control/command software (beyond specialized domains such as filters (Feret, 2004)), especially to take the interaction of the controller and controlled system into account (Cousot, 2005a).

Despite recent progress in the analysis of complex data structures (see e.g. Nicole et al. (2022); Illous et al. (2021); Ko et al. (2019)), general-purpose reusable libraries do not exist for more complex symbolic data structures like sets of graphs, or sets of sets of graphs, etc. commonly found in complex computer applications like social networks data bases, where capturing succinctly the evolution of unbounded graphs over time is essential.

Another example is sets of functions. Of course there are known methods to analyze procedures (by copy as in Astrée (Blanchet et al., 2003), by functional abstraction (Cousot and Cousot, 1977b), and by compilation (Sharir and Pnueli, 1981)), but the are not really applicable to higher-order first-class functional languages where functions take as parameters and return functions.

3.6 Properties of More Complex Control Structures

Very few static analyzers go beyond sequential programs. Counter examples are Astrée and Zoncolan. Parallelism is difficult because in can be formalized in many different ways, from data parallelism, shared memory models (Cousot and Cousot, 1984; Miné, 2014), including with weak memory models (Alglave and Cousot, 2017; Suzanne and Miné, 2016), to distributed systems with synchronous (Cousot and Cousot, 1980) or asynchronous communications, with various hypotheses on the communications achievability, static or dynamic process creation, etc. We lack good semantic models of general applicability and corresponding expressive abstractions.

3.7 Computation Spaces

Originally, abstract interpretation dealt with reachability properties, that is sets of states, but had to move to more complex ones such as sets of traces (e.g. for the soundness of dataflow analysis ((Cousot and Cousot, 1979, example 7.2.0.6); Cousot (2019d)), to sets of sets of traces. This fits well for discrete computations. For continuous or hybrid ones, more complex models are needed (Cousot, 2022; Farjudian and Moggi, 2022) and mainly, abstractions that go beyond time-limited/bounded ones would be most helpful.

3.8 Choosing Precise and Efficient Abstractions

The choice of abstractions is completely empirical and experimental. At least, we have a lattice of abstractions allowing for combinations of abstractions like the reduced product (Cousot and Cousot, 1979, Section 10). This allows for the refinement of static analyzers by adding and combining various abstractions. The automation of this process during the analysis (e.g. Cousot et al. (2007) and many others) has not be very successful. This is because the refinement with respect to the exact collecting semantics is too costly while the refinement with respect to an abstract domain (e.g. when using SMT solvers necessarily restricted to a given combination of abstract domains Cousot et al. (2011)) cannot help when the needed properties are not exactly expressible in this abstract domain.

Moreover, because the terminating extrapolation operators cannot be increasing (monotone), a more precise domain does not necessarily result in a more precise analysis. Finally, the lattice of abstract domain provides no information to compare the precision of incomparable domains in this lattice, meaning that properties expressible by one are not a subset of the properties expressible by the other. So a rigorous formalization of the relative precision of abstract domains and abstract interpretations would be welcomed. (Casso et al., 2019) is an innovative and interesting step in this direction.

3.9 Induction Abstraction

Most reasonings on program executions require some form of mathematical induction such as recurrence on the number of loop iterations or recursive program calls (Cousot, 2019c). In abstract interpretation this is approximated by an abstract domain and by extrapolation (widening, dual widening) and interpolation (narrowing and dual narrowing) operators (Cousot and Cousot, 1992; Cousot, 2015), which are weak forms of induction that guarantee the soundness of the approximation and the termination of the induction (the same way that a proof by recurrence allows for the finite presentation of an otherwise infinite proof for each of the naturals).

This is certainly the most difficult part of program verification which is evaded in deductive methods (by asking the user to provide the inductive argument) and model checking (by requiring finiteness or else bounded model checking, a trivial widening) and make analysis harder than combinatorial enumeration and verification (Cousot et al., 2018). More sophisticated extrapolation and interpolation methods are needed to boost the precision of static analysis.

3.10 Calculational Design of Abstract Interpreters

Given a semantics, and an abstraction, an abstract interpreter can be designed by calculus, mostly by folding and unfolding definitions, simplifications, and, at crucial points in the proof, by well-chosen approximations. If such proofs can be checked a posteriori (Jourdan et al., 2015; Barthe et al., 2017; Franceschino et al., 2021), we lack tools that would automatize the symbolic computations, except may be the few approximation points, although progress in program synthesis would certainly be applicable to such formal computations.

3.11 Language Independent Abstract Interpretation

If the theory of abstract interpretation has aimed at being independent of a specific programming language, the practice has been very different. Type inference, verification, static analysis, etc. is almost always tightly tight to a specific model of programs if not to a specific programming language. Designing at multi-language abstract interpreter is a challenge. If it is nowadays possible to design static analyzers that can handle both C and Python (Journault et al., 2019), it is much more difficult to consider say C++, Java, Lisa (Alglave and Cousot, 2016), Lustre (Halbwachs, 1998; Halbwachs et al., 1997), Parallel Prolog (Dovier et al., 2021), and Ocaml. Even something looking simpler, like extending Astrée to analyze inlined assembly code, is a very complex task (Chevalier and Feret, 2020).

The most common practical approach is to compile to an intermediate language and then analyze this “universal” intermediate language. The approach is manual and the difficulty is to connect the source and object languages so that the analysis at the object level can take into account the specificities of the source and report its findings with respect to the source. A counter-example is Verasco (Jourdan, 2016) which makes the analysis of C code using one of the intermediate languages of CompCert, and reports with respect to this intermediate language, so that messages are not related to the source and therefore hardly understandable.

One of the big difficulties to achieve language-independence is that the semantics of these languages, whether operational, denotational, or axiomatic are completely language dependent, and so are their abstractions.

Transitions systems, as used in Cousot (1978a) and later in model checking, are certainly language independent, can be given any of the know semantics by abstraction (Cousot, 2002), but are of very limited expressivity. An example is (Cousot and Cousot, 1989) attempting to describe the abstract syntax and transitional semantics, including parallelism, in a language independent way. It is applied to the soundness proof of a variant of Hoare logic that can be further abstracted using abstract domains.

Another attempt is (Cousot, 1997) consists in considering a meta-language to define the language (collecting) semantics accompanied by abstractions of this meta-language constructs and data which yield abstractions of the defined language. Instances of this approach are (Mirliaz and Pichardie, 2022) and the "skeletal semantics" of (Bodin et al., 2019). Again, it is very difficult to be flexible in the representation of computations used in the meta-semantics (an not make an arbitrary choice such as as execution traces or sets of reachability states for the collecting semantics) and to incorporate language dependent abstractions in the meta language (e.g. for pointer analysis).

But this venue has not been largely explored and progress is certainly possible beyond the first steps taken by (Journault et al., 2019).

3.12 New Computation Models and Challenges

Being of general scope, abstract interpretation has shown to be largely applicable in computer science and is certainly the only verification methods that has scaled up to huge industrial software over the past decades (think e.g. to Astrée and Zoncolan). Innovations in computer science evolve rapidly, which opens new needs for sound verification, hence abstract interpretation. Let us just mention a few examples.

Probabilistic Programming and Analyses:

The static analysis of probabilistic properties or probabilistic programs has a long history (Monniaux, 2000, 2001; Cousot and Monerau, 2012; Adjé et al., 2013; Bouissou et al., 2016) but precise abstraction and inference of probabilistic properties (like sets of distributions) are not well-developed enough.

Smart Contracts:

Smart contracts are commonly associated with cryptocurrencies and decentralized finance applications that takes place on a blockchain or distributed ledger. A smart contract can be any kind of computer program writen in a Turing-complete language and so subject to penalizing errors. The need for verification and analysis is obvious and abstract interpretation directly applicable (Bau et al. (2022); Perez-Carrasco et al. (2020); Henglein et al. (2020)).

Data Sciences:

Machine learning has made huge progress this last decade, in particular thanks to breakthroughs in neural networks, with a large variety of highly publicized applications. However, it has also a number of weaknesses not so much advertised. This pave the way for formal methods and static analysis (Urban (2019); Urban and Miné (2021)) of required properties such as robustness (Gehr et al. (2018); Singh et al. (2019); Munakata et al. (2022)), fairness (Mazzucato and Urban, 2021), absence of data leakages (Subotic et al., 2022), sound behavior of neural network controlled physical systems (Goubault and Putot, 2022).

Quantum Computing:

Efforts towards building a physical quantum computer may succeed in the near future. Since quantum computers obey the Church–Turing thesis, verification of quantum programs will be undecidable, and so abstract interpretation comes in (Perdrix, 2008; Yu and Palsberg, 2021).

Static Analysis of Biological Networks:

A biological network are used to represent complex sets of binary interactions or relations between various biological entities and their evaluation over time. Inevitably, the behavior of such dynamic systems must be approximated to cope with enormous complexity, which pave the way for the use of abstract interpretation (Danos et al., 2008; Fages and Soliman, 2008; Boutillier et al., 2018, 2019; Beica et al., 2020).

Economy:

(Discrete) dynamical systems, including games, have applications to a wide variety of fields, including economy, and their evolution over time requires sound approximation, the domain of predilection of abstract interpretation (Ranzato, 2016).

4 Conclusion

The application of abstract interpretation to semantics, verification, and static analysis needs more powerful semantic models, reusable abstractions via libraries (of complex data and control structures), a significant deepening of abstract inference, and achieve the computer-assisted calculations design of sound extensible abstract interpreters. New application domains appear over time (we have only cited a few one to follow the hype) and require more abstractions of formal semantic domains of computation. If the mentioned opened problems do not look difficult enough, you can combine them to get an impossible challenge like using machine learning to au-

tomatically design a sound by design static analyzer for your favorite language (or better any one) running on a quantum computer.

References

Adjé A, Bouissou O, Goubault-Larrecq J, Goubault E, Putot S (2013) Static analysis of programs with imprecise probabilistic inputs. In: VSTTE, Springer, Lecture Notes in Computer Science, vol 8164, pp 22–47

Alglave J, Cousot P (2016) Syntax and analytic semantics of LISA. CoRR abs/1608.06583

Alglave J, Cousot P (2017) Ogre and pythia: an invariance proof method for weak consistency models. In: POPL, ACM, pp 3–18

Barthe G, Blazy S, Laporte V, Pichardie D, Trieu A (2017) Verified translation validation of static analyses. In: CSF, IEEE Computer Society, pp 405–419

Bau G, Miné A, Botbol V, Bouaziz M (2022) Abstract interpretation of michelson smart-contracts. In: SOAP@PLDI, ACM, pp 36–43

Beica A, Feret J, Petrov T (2020) Tropical abstraction of biochemical reaction networks with guarantees. In: SASB, Elsevier, Electronic Notes in Theoretical Computer Science, vol 350, pp 3–32

Black PE, Walia KS (2000) SATE VI Ockham Sound Analysis Criteria. NIST IR 8304, URL `https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8304.pdf`

Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2003) A static analyzer for large safety-critical software. In: PLDI, ACM, pp 196–207

Bodin M, Gardner P, Jensen TP, Schmitt A (2019) Skeletal semantics and their interpretations. Proc ACM Program Lang 3(POPL):44:1–44:31

Bouissou O, Goubault E, Putot S, Chakarov A, Sankaranarayanan S (2016) Uncertainty propagation using probabilistic affine forms and concentration of measure inequalities. In: TACAS, Springer, Lecture Notes in Computer Science, vol 9636, pp 225–243

Boutillier P, Camporesi F, Coquet J, Feret J, Lý KQ, Théret N, Vignet P (2018) Kasa: A static analyzer for kappa. In: CMSB, Springer, Lecture Notes in Computer Science, vol 11095, pp 285–291

Boutillier P, Cristescu I, Feret J (2019) Counters in kappa: Semantics, simulation, and static analysis. In: ESOP, Springer, Lecture Notes in Computer Science, vol 11423, pp 176–204

Casso I, Morales JF, López-García P, Giacobazzi R, Hermenegildo MV (2019) Computing abstract distances in logic programs. In: LOPSTR, Springer, Lecture Notes in Computer Science, vol 12042, pp 57–72

Chevalier M, Feret J (2020) Sharing ghost variables in a collection of abstract domains. In: VMCAI, Springer, Lecture Notes in Computer Science, vol 11990, pp 158–179

Cousot P (1978a) Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France

Cousot P (1978b) Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes. In: University of Grenoble

Cousot P (1996a) Abstract interpretation. ACM Comput Surv 28(2):324–328

Cousot P (1996b) Program analysis: The abstract interpretation perspective. ACM Comput Surv 28(4es):165

Cousot P (1997) Abstract interpretation based static analysis parameterized by semantics. In: SAS, Springer, Lecture Notes in Computer Science, vol 1302, pp 388–394

Cousot P (1999a) The calculational design of a generic abstract interpreter. In: Broy M, Steinbrüggen R (eds) Calculational System Design, NATO ASI Series F. IOS Press, Amsterdam

Cousot P (1999b) Directions for research in approximate system analysis. ACM Comput Surv 31(3es):6

Cousot P (2002) Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor Comput Sci 277(1-2):47–103

Cousot P (2005a) Integrating physical systems in the static analysis of embedded control software. In: APLAS, Springer, Lecture Notes in Computer Science, vol 3780, pp 135–138

Cousot P (2005b) The verification grand challenge and abstract interpretation. In: VSTTE, Springer, Lecture Notes in Computer Science, vol 4171, pp 189–201

Cousot P (2015) Abstracting induction by extrapolation and interpolation. In: VMCAI, Springer, Lecture Notes in Computer Science, vol 8931, pp 19–42

Cousot P (2019a) Abstract semantic dependency. In: SAS, Springer, Lecture Notes in Computer Science, vol 11822, pp 389–410

Cousot P (2019b) A formal introduction to abstract interpretation. In: Pretschner A, Müller P, Stöckle P (eds) Calculational System Design, NATO SPS, Series D, Vol. 53. IOS Press, Amsterdam

Cousot P (2019c) On fixpoint/iteration/variant induction principles for proving total correctness of programs with denotational semantics. In: LOPSTR, Springer, Lecture Notes in Computer Science, vol 12042, pp 3–18

Cousot P (2019d) Syntactic and semantic soundness of structural dataflow analysis. In: SAS, Springer, Lecture Notes in Computer Science, vol 11822, pp 96–117

Cousot P (2021a) Calculational design of a regular model checker by abstract interpretation. Theor Comput Sci 869:62–84

Cousot P (2021b) Principles of Abstract Interpretation, 1st edn. MIT Press

Cousot P (2022) Asynchronous correspondences between hybrid trajectory semantics. In: Tom Henzinger Festschrift, Springer, Lecture Notes in Computer Science, vol 13660, to appear

Cousot P, Cousot R (1976a) Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Dunod, pp 106–130

Cousot P, Cousot R (1976b) Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Dunod, Paris, France, pp 106–130

Cousot P, Cousot R (1977a) Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM, pp 238–252

Cousot P, Cousot R (1977b) Static determination of dynamic properties of recursive procedures. In: Formal Description of Programming Concepts, North-Holland, pp 237–278

Cousot P, Cousot R (1979) Systematic design of program analysis frameworks. In: POPL, ACM Press, pp 269–282

Cousot P, Cousot R (1980) Semantic analysis of communicating sequential processes (shortened version). In: ICALP, Springer, Lecture Notes in Computer Science, vol 85, pp 119–133

Cousot P, Cousot R (1984) Invariance proof methods and analysis techniques for parallel programs. In: Biermann A, Guiho G, Kodratoff Y (eds) Automatic Program Construction Techniques, Macmillan, New York, New York, United States, chap 12, pp 243–271

Cousot P, Cousot R (1989) A language independent proof of the soundness and completeness of generalized hoare logic. Inf Comput 80(2):165–191

Cousot P, Cousot R (1992) Inductive definitions, semantics and abstract interpretation. In: POPL, ACM Press, pp 83–94

Cousot P, Cousot R (2004) Basic concepts of abstract interpretation. In: IFIP Congress Topical Sessions, Kluwer/Springer, IFIP, vol 156, pp 359–366

Cousot P, Cousot R (2009) Bi-inductive structural semantics. Inf Comput 207(2):258–283

Cousot P, Cousot R (2010a) A gentle introduction to formal verification of computer systems by abstract interpretation. In: Esparza J, Grumberg O, Broy M (eds) Logics and Languages for Reliability and Security, NATO Science Series III: Computer and Systems Sciences, IOS Press, pp 1–29

Cousot P, Cousot R (2010b) A gentle introduction to formal verification of computer systems by abstract interpretation. In: Logics and Languages for Reliability and Security, NATO Science for Peace and Security Series - D: Information and Communication Security, vol 25, IOS Press, pp 1–29

Cousot P, Cousot R (2012) An abstract interpretation framework for termination. In: POPL, ACM, pp 245–258

Cousot P, Cousot R (2014) Abstract interpretation: past, present and future. In: CSL-LICS, ACM, pp 2:1–2:10

Cousot P, Monerau M (2012) Probabilistic abstract interpretation. In: ESOP, Springer, Lecture Notes in Computer Science, vol 7211, pp 169–193

Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2005) The astreé analyzer. In: ESOP, Springer, Lecture Notes in Computer Science, vol 3444, pp 21–30

Cousot P, Ganty P, Raskin J (2007) Fixpoint-guided abstraction refinements. In: SAS, Springer, Lecture Notes in Computer Science, vol 4634, pp 333–348

Cousot P, Cousot R, Mauborgne L (2011) The reduced product of abstract domains and the combination of decision procedures. In: FoSSaCS, Springer, Lecture Notes in Computer Science, vol 6604, pp 456–472

Cousot P, Giacobazzi R, Ranzato F (2018) Program analysis is harder than verification: A computability perspective. In: CAV (2), Springer, Lecture Notes in Computer Science, vol 10982, pp 75–95

Danos V, Feret J, Fontana W, Krivine J (2008) Abstract interpretation of cellular signalling networks. In: VMCAI, Springer, Lecture Notes in Computer Science, vol 4905, pp 83–97

Deng C, Cousot P (2022) The systematic design of responsibility analysis by abstract interpretation. ACM Trans Program Lang Syst 44(1):3:1–3:90

Dovier A, Formisano A, Gupta G, Hermenegildo MV, Pontelli E, Rocha R (2021) Parallel logic programming: A sequel. CoRR abs/2111.11218

Fages F, Soliman S (2008) Abstract interpretation and types for systems biology. Theor Comput Sci 403(1):52–70

Farjudian A, Moggi E (2022) Robustness, scott continuity, and computability. DOI 10.48550/ARXIV.2208.12347, URL `https://arxiv.org/abs/2208.12347`

Feret J (2004) Static analysis of digital filters. In: ESOP, Springer, Lecture Notes in Computer Science, vol 2986, pp 33–48

Franceschino L, Pichardie D, Talpin J (2021) Verified functional programming of an abstract interpreter. In: SAS, Springer, Lecture Notes in Computer Science, vol 12913, pp 124–143

Gehr T, Mirman M, Drachsler-Cohen D, Tsankov P, Chaudhuri S, Vechev MT (2018) AI2: safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy, IEEE Computer Society, pp 3–18

Giacobazzi R, Ranzato F (2022) History of abstract interpretation. IEEE Ann Hist Comput 44(2):33–43

Goubault E, Putot S (2022) RINO: robust inner and outer approximated reachability of neural networks controlled systems. In: CAV (1), Springer, Lecture Notes in Computer Science, vol 13371, pp 511–523

Halbwachs N (1998) About synchronous programming and abstract interpretation. Sci Comput Program 31(1):75–89

Halbwachs N, Proy Y, Roumanoff P (1997) Verification of real-time systems using linear relation analysis. Formal Methods Syst Des 11(2):157–185

Henglein F, Larsen CK, Murawska A (2020) A formally verified static analysis framework for compositional contracts. In: Financial Cryptography Workshops, Springer, Lecture Notes in Computer Science, vol 12063, pp 599–619

Hiriart-Urruty JB, Lemaréchal C (2004) Fundamentals of convex analysis, 2nd edn. Springer

Illous H, Lemerre M, Rival X (2021) A relational shape abstract domain. Formal Methods Syst Des 57(3):343–400

Jeannet B, Miné A (2009) Apron: A library of numerical abstract domains for static analysis. In: CAV, Springer, Lecture Notes in Computer Science, vol 5643, pp 661–667

Jourdan J (2016) Verasco: a formally verified C static analyzer. (verasco: un analyseur statique pour C formellement vérifié). PhD thesis, Paris Diderot University, France

Jourdan J, Laporte V, Blazy S, Leroy X, Pichardie D (2015) A formally-verified C static analyzer. In: POPL, ACM, pp 247–259

Journault M, Miné A, Monat R, Ouadjaout A (2019) Combinations of reusable abstract domains for a multilingual static analyzer. In: VSTTE, Springer, Lecture Notes in Computer Science, vol 12031, pp 1–18

Ko Y, Rival X, Ryu S (2019) Weakly sensitive analysis for javascript object-manipulating programs. Softw Pract Exp 49(5):840–884

Leroy X (2016) Formally verifying a compiler: What does it mean, exactly? In: ICALP, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, LIPIcs, vol 55, pp 2:1–2:1

Logozzo F, Fahndrich M, Mosaad I, Hooimeijer P (2019) Zoncolan: How Facebook uses static analysis to detect and prevent security issues. Engineering at Meta URL `https://engineering.fb.com/2019/08/15/security/zoncolan/`

Mazzucato D, Urban C (2021) Reduced products of abstract domains for fairness certification of neural networks. In: SAS, Springer, Lecture Notes in Computer Science, vol 12913, pp 308–322

Miné A (2014) Relational thread-modular static value analysis by abstract interpretation. In: VMCAI, Springer, Lecture Notes in Computer Science, vol 8318, pp 39–58

Miné A (2017) Tutorial on static inference of numeric invariants by abstract interpretation. Found Trends Program Lang 4(3-4):120–372

Miné A (2006) The octagon abstract domain. Higher–Order and Symbolic Computation 19(1):31–100

Mirliaz S, Pichardie D (2022) A flow-insensitive-complete program representation. In: VMCAI, Springer, Lecture Notes in Computer Science, vol 13182, pp 197–218

Monniaux D (2000) Abstract interpretation of probabilistic semantics. In: SAS, Springer, Lecture Notes in Computer Science, vol 1824, pp 322–339

Monniaux D (2001) Backwards abstract interpretation of probabilistic programs. In: ESOP, Springer, Lecture Notes in Computer Science, vol 2028, pp 367–382

Munakata S, Urban C, Yokoyama H, Yamamoto K, Munakata K (2022) Verifying attention robustness of deep neural networks against semantic perturbations. CoRR abs/2207.05902

Nicole O, Lemerre M, Rival X (2022) Lightweight shape analysis based on physical types. In: VMCAI, Springer, Lecture Notes in Computer Science, vol 13182, pp 219–241

Ore O (1944) Galois connexions. Trans Amer Math Soc 55(3):493–513

Perdrix S (2008) Quantum entanglement analysis based on abstract interpretation. In: SAS, Springer, Lecture Notes in Computer Science, vol 5079, pp 270–282

Perez-Carrasco V, Klemen M, López-García P, Morales JF, Hermenegildo MV (2020) Cost analysis of smart contracts via parametric resource analysis. In: SAS, Springer, Lecture Notes in Computer Science, vol 12389, pp 7–31

Plofker K (2007) Mathematics in India. Princeton University Press

Ranzato F (2016) Abstract interpretation of supermodular games. In: SAS, Springer, Lecture Notes in Computer Science, vol 9837, pp 403–423

Rival X, Yi K (2020) Introduction to Static Analysis. MIT Press

Sharir M, Pnueli A (1981) Two approaches to interprocedural data flow analysis. In: Muchnick S, Jones N (eds) Program Flow Analysis: Theory and Applications, Prentice–Hall, chap 7, pp 189–342

Singh G, Püschel M, Vechev MT (2015) Making numerical program analysis fast. In: PLDI, ACM, pp 303–313

Singh G, Püschel M, Vechev MT (2017) Fast polyhedra abstract domain. In: POPL, ACM, pp 46–59

Singh G, Gehr T, Püschel M, Vechev MT (2019) An abstract domain for certifying neural networks. Proc ACM Program Lang 3(POPL):41:1–41:30

Subotic P, Bojanic U, Stojic M (2022) Statically detecting data leakages in data science code. In: SOAP@PLDI, ACM, pp 16–22

Suzanne T, Miné A (2016) From array domains to abstract interpretation under store-buffer-based memory models. In: SAS, Springer, Lecture Notes in Computer Science, vol 9837, pp 469–488

Tripp O, Pistoia M, Cousot P, Cousot R, Guarnieri S (2013) Andromeda: Accurate and scalable security analysis of web applications. In: FASE, Springer, Lecture Notes in Computer Science, vol 7793, pp 210–225

Urban C (2019) Static analysis of data science software. In: SAS, Springer, Lecture Notes in Computer Science, vol 11822, pp 17–23

Urban C, Miné A (2017) Inference of ranking functions for proving temporal properties by abstract interpretation. Comput Lang Syst Struct 47:77–103

Urban C, Miné A (2021) A review of formal methods applied to machine learning. CoRR abs/2104.02466

Urban C, Ueltschi S, Müller P (2018) Abstract interpretation of CTL properties. In: SAS, Springer, Lecture Notes in Computer Science, vol 11002, pp 402–422

Yu N, Palsberg J (2021) Quantum abstract interpretation. In: PLDI, ACM, pp 542–558