Abstract Interpretation

Web page maintained by P. Cousot

Last update: August 5, 2008

Contents

1 Introduction to Abstract Interpretation

The *formal verification* of a program (and more generally a computer system) consists in proving that its *semantics* (describing "what the program executions actually do") satisfies its *specification* (describing "what the program executions are supposed to do").

Abstract interpretation [14, 22, 26] formalizes the idea that this formal proof can be done at some level of abstraction where irrelevant details about the semantics and the specification are ignored. This amounts to proving that an *abstract semantics* satisfies an *abstract specification*. An example of abstract semantics is Hoare logic while examples of abstract specifications are invariance, partial, or total correctness. This abstracts away from concrete properties such as execution times.

Abstractions should preferably be *sound* (no conclusion derived from the abstract semantics is wrong relative to the program concrete semantics and specification). Otherwise stated, a proof that the abstract semantics satisfies the abstract specification should imply that the concrete semantics also satisfies the concrete specification. Hoare logic is a sound verification method, debugging is not (since some executions are left out), bounded model checking is not either (since parts of some executions are left out). Unsound abstractions lead to *false negatives* (the program may be claimed to be correct/non erroneous with respect to the specification whereas it is indeed incorrect).

Abstractions should also preferably be *complete* (no aspect of the semantics relevant to the specification is left out). So if the concrete semantics satisfies the concrete specification this should be provable in the abstract. However program proofs are undecidable, and so automatic tools for reasoning about programs are all incomplete (for non trivial program properties such as safety, liveness, or security) and must therefore fail on some programs. This can be achieved by allowing the tool not to terminate, to be unsound (e.g. debugging tools omit possible executions), or to be incomplete (e.g. static analysis tools may produce false alarms). Incomplete abstractions lead to *false positives* or *false alarms* (the specification is claimed to be potentially violated by some program executions while it is not).

Potentially non-terminating and unsound program verification tools are easy to design. Terminating and sound tools are much more difficult to design. Complete tools are impossible to design, by undecidability. However static analysis tools producing very few or no false alarms have been designed and used in industrial contexts for specific families of properties and programs [42]. In all cases, abstract interpretation provides a systematic construction method based on the effective approximation of the concrete semantics, which can be (partly) automated and/or formally verified.

Abstract interpretation aims at

- providing a basic coherent and conceptual theory for understanding in a unified framework the thousands of ideas, concepts, reasonings, methods, and tools on formal program analysis and verification [22, 26];

- guiding the correct formal design of automatic tools for *program analysis* (computing an abstract semantics) and *program verification* (proving that an abstract semantics satisfies an abstract specification) [17].

Abstract interpretation theory studies semantics (formal models of computer systems), abstractions, their soundness, and completeness.

The informal presentation "Abstract Interpretation in a Nutshell" aims at providing a short intuitive introduction to the theory. The "basic concepts of abstract interpretation" and an elementary "course on abstract interpretation" can also be found on the web.

2 What can be formalized by abstract interpretation?

Abstract interpretation is useful in computer science to formalize reasonings involving the [sound [and complete]] approximation of the semantics of formal systems. A non-exhaustive list of typical examples of application is given below.

2.1 Syntax

The analysis of properties of grammars as well as parsing are abstract interpretations of the grammar operational semantics and its abstractions (such as parse trees, protolanguages, or terminal languages) [39, 41].

2.2 Semantics

The semantics of programs describes their possible runtime executions in all possible execution environments at some level of abstraction. The hierarchy of semantics, including trace, small-step, and big-step operational semantics, relational semantics, denotational semantics, predicate transformers, and axiomatic semantics in their angelic, natural, and demoniac versions can be designed by abstract interpretation [32, 19]. An extension to transfinite behaviors is useful to formalize e.g. *semantic slicing* [54].

2.3 Proofs

Formal proofs of program correctness involve an abstraction since specifications always ignore some aspects of program execution which need not be taken into account in the proof [18].

2.4 Static analysis

Static analysis is the automatic determination of abstract program properties [22, 44, 26, 15]. This was the motivating application behind the introduction of the abstract interpretation theory.

2.5 Data-flow Analysis

Abstract-interpretation-based static analysis includes finitary versions such as *Dataflow Analysis* [67], [26, 36], *Set-based Analysis* [35], etc where the lattice height is assumed to be finite.

2.6 Control-flow Analysis

Control-flow analysis (CFA) [70, 82, 105] is an abstract interpretation [81] statically approximating the flow of control of lambda expressions.

2.7 Types

Static typing and type inference [97] can be understood as an abstract interpretation of runtime type checking thus providing a "correct by construction" design method [16], [11].

2.8 Model-checking

Model-checking exhaustively verifies temporal properties on a finite model of hardware or software computer systems [10]. This abstraction of a system into a model is often left implicit. *Abstract model checking*, as formalized by abstract interpretation, makes this abstraction explicit [9], [36], [101].

Model-checking is reputed to be terminating, sound, and complete on the model. From an abstract interpretation point of view, relating the system to its model, it may be sound on the model but unsound on the system (e.g. the model is correct for safety properties but wrong for liveness properties), it is often incomplete (no finite model can cover the specified behaviors of the system [100]) and, in practice, may explode combinatorially. In all cases abstract interpretations of the system into a model have to be considered. All transition models are abstract semantics but the converse is not true.

2.9 Predicate abstraction

Predicate abstraction [61] can be used to reduce any static analysis on a *finite* abstract domain to boolean fixpoint computations as performed by a model-checker using a theorem prover to automatically derive the abstract transformers involved in the fixpoint definition. Parametric predicate abstraction is an extension to infinite abstract domains [20].

2.10 Counter-example-based refinement

Spurious counter-example-based refinement in abstract model-checking is formalized as an abstract domain completion problem in the abstract interpretation theory [56].

2.11 Strong Preservation

The problem of modifying finite abstract model checking by minimal refinements in order to get strong preservation for some specification language, including partition refinement algorithms, is a completion problem in the abstract interpretation theory [101].

2.12 Program transformation

Program transformations (such as *partial evaluation* [72]) often require a static analysis of the source program, as formalized by abstract interpretation (e.g. [71]). Moreover, the transformation itself, from source to object programs, involves a loss of information on the original program or a limitation on the possible program behaviors. This approximation can be formalized by abstract interpretation. This was exemplified on dead-code elimination, slicing, partial evaluation, or program monitoring [38].

2.13 Watermarking

Looking for program watermarks that are not subject to obfuscation, one can think to an abstract interpretation of the program semantics [40].

2.14 Information hiding

In language-based software security, the information to be hidden to an intruder can be formalized as an abstract interpretation of the program semantics [55].

2.15 Code obfuscation

The aim of code obfuscation is to prevent malicious users from discovering properties of the original source program. This goal can be precisely modeled by abstract interpretation, where the hiding of properties corresponds to abstracting the semantics [99].

2.16 Malware detection

An obfuscated malware is better detected by the effects of its execution as recognized by an abstract interpretation rather than by a syntactic signature [98].

2.17 Termination

A relational abstract-interpretation-based static analysis on a well-founded abstract domain can be systematically extended to a termination analysis [3] (whence liveness analyses). Termination analysis may require probabilistic hypotheses [87] or fairness hypotheses on parallel processes [78].

3 Topics in abstract interpretation

3.1 Abstract domains

An abstract domain is an abstract algebra, often implemented as a library module (see **Sec. 5**), providing a description of abstract program properties and of operations on these abstract properties (such as abstract property transformers describing the operational effect of program instructions and commands in the abstract). Abstract domains are often complete lattices, an abstraction of powersets [26].

3.2 Finite versus infinite abstract domains

The first infinite abstract domain (that of intervals) was introduced in [21]. This abstract domain was later used to prove that, thanks to widenings, infinite abstract domains can lead to effective static analyses for a given programming language that are strictly more precise and equally efficient than any other one using a finite abstract domain or an abstract domain satisfying chain conditions [31].

3.3 Merge over all paths (MOP) in dataflow analysis

The first dataflow analyses where proved correct by comparing the union of the abstraction along all execution paths (so-called MOP[1] solution) with a fixpoint solution using transfer functions (i.e. abstract property transformer) [67]. Instead of comparing one solution of the static analysis problem to another one, the theory of abstract interpretation introduced the idea that the correctness/soundness should be expressed with respect to a formal semantics. A consequence is that static analysis algorithms can be specified and designed by approximation of a program semantics. Moreover, the "fixpoint solution" was shown to be an abstraction of the "MOP solution". These two solutions coincide for distributive abstract interpretations (for which abstract transformers preserve union). Otherwise, the abstract domain used for the fixpoint solution can be minimally enriched into its *disjunctive completion* to get exactly the "MOP solution" in fixpoint form (on the completed abstract domain) [26].

[1] MOP often stands for "Meet Over all Path" where the abstract meet corresponds to a concrete join, the order in dataflow analysis often being the inverse of the one used in abstract interpretation, whence corresponds to a concrete inverse logical implication, which may sometimes be counter-intuitive.

3.4 Standard and collecting semantics

The *collecting semantics* or *non-standard semantics* (after the *static semantics* of [22]) is the semantics of the language formally defining the program execution properties of interest to be analyzed by abstraction. These program executions themselves are formalized by the standard semantics.

For example in [22], the standard semantics is an operational semantics. The concrete properties of interest are invariance properties so the collecting semantics defines the reachable states from initial states. This collecting semantics is expressed as the least fixpoint of equations using strongest postcondition transformers. Thus the considered abstractions provide abstract invariance properties (overapproximating the reachable states).

3.5 Galois connection

Galois connections can be used when the abstract domain always offers a most precise approximation of any concrete property [26]. Indeed the Galois connections introduced in [21, 22] are the semi-dual of the one introduced by Évariste Galois, and so do compose (the abstraction of an abstraction is an abstraction). Equivalent presentations involve closure operators, ideals, congruences, etc [26].

An interesting consequence of the existence of the best abstraction of concrete properties is the existence of best/most precise transformers, which provides guidelines for their [automatic] design [17]. Moreover by abstracting fixpoints by fixpoints, static analysis is specified by an abstract semantics expressed in fixpoint form whereas a static analyzer is an algorithm for computing or overapproximating this fixpoint.

There are several alternative formalizations in absence of best approximations [94], [30].

3.6 Moore family, Closure operators

If the correspondance between the concrete and abstract properties is given by a Galois connection, then the image of the abstract properties in the concrete is a *Moore family*, which is the image of the concrete properties by a *closure operator* [26]. The formalization of the abstraction by closure operators is useful to abstract away from the representation of the abstract properties. Other equivalent formalizations of the existence of the best abstraction of concrete properties include principal ideals, complete join congruences, etc [26].

3.7 Moore completion

If an abstract domains has no best approximations for some concrete properties its *Moore completion* is the most abstract abstract domain more expressive than the original abstract domain which has this property [26]. This introduces a Galois connection.

3.8 Fixpoints

Most program properties can be expressed as fixpoints of monotone or extensive property transformers, a property preserved by abstraction [26]. This reduces program analysis to fixpoint approximation and verification to fixpoint checking [22].

3.9 Inference rules

Most (concrete and abstract) program properties can also be expressed using inference rules, which indeed is equivalent to definitions using fixpoints, equations, constraints, etc [34]. This point of view has the advantage of separating the design of an (abstract) semantics (or a static analysis) from its formal presentation.

3.10 Iteration

A standard method for fixpoints construction is *iteration*. This can be extended to transfinite iterations to prove Tarski's fixpoint theorem [25]. In practice one can accelerate the fixpoint computation using appropriate iteration strategies formalized as chaotic/asynchronous iterations [14]. This extends to higher-order [24], [74].

3.11 Convergence acceleration by widening/narrowing

Convergence acceleration is mandatory in infinite abstract domains in which the iterative computation of abstract fixpoints may diverge. Widenings/narrowings and their duals are universal methods to do so [22, 14]. This extends to higher-order [24, 6].

3.12 Constraint resolution

Thanks to Tarski's theorem, fixpoints can be re-expressed as equality or inequality constraints whence static analysis can be equally viewed as a fixpoint approximation or as a constraint resolution problem. Some constraint resolution methods are mere fixpoint iteration (e.g. set-based analysis [35]).

3.13 Modularity

The static analysis of programs by parts [37]. This includes separate interprocedural analysis [24].

3.14 Abstract domain combination

The design of abstractions can be done by parts, by choosing basic abstractions and then by composing them using abstract domain combinators [26]. This provides a unifying view on abstract domain design [52].

There are numerous examples of such abstract domain combinations including the *reduced sum*, the *reduced product* and the *reduced power* [26].

3.15 Refinement

The enrichment of an abstract domain so has to express in the refined domain properties which cannot be expressed in the original domain. Examples are the *Moore completion* **Sec. 3.7**, *partitioning* [15, 5, 79], [69], the *disjunctive completion* [26], [53], the *Heyting completion* [59], *complementation* [13], fixpoint refinement [43], etc.

3.16 Completion

The refinement of an abstract domain into the most abstract abstract domain which is precise enough to prove a given specification [26], [93, 57, 58, 104].

3.17 Lattice of abstract interpretations

A consequence of the formalization of static analyses by Galois connections is that they can be organized into a complete lattice according to the partial order of their relative precisions [26]. All semantics and analyses are present in this formally defined lattice and so it remains to find the useful ones, which is the real question!

4 Examples of abstract-interpretation-based static analyses

4.1 Numerical Analysis

Contrary to dataflow analyses on lattices satisfying the ascending/descending chain condition in the seventies, the generalization to precise static analyses using numerical domains such as intervals [21], octagons [83] or polyhedra [44] are in infinite abstract domains whence required the introduction of new widening-based iteration convergence acceleration methods to become effective.

4.2 Interval analysis

The analysis of the interval in which the values of variables do range [21, 22].

4.3 Octagonal analysis

The analysis of the octagon in which the values of pairs of variables do range (that is the determination of constraints of the form $+/-x+/-y \leq c$ where x and y are variables and c a rational constant automatically discovered by the analysis) [83, 84, 85].

4.4 Polyhedral analysis

The analysis of the polyhedron in which the values of tuples of variables do range. This abstract interpretation can be used for the automatic discovery of invariant linear inequalities among numerical variables of a program [44].

Delay analysis in synchronous programs [64], safety analysis of reactive systems [65], quantitative time properties of synchronous programs, and linear hybrid systems [66] are among the many applications.

4.5 Congruence analysis

The analysis of the congruence invariants satisfied by the values of one [62] or several [63] numerical variables [84].

4.6 Roundoff error analysis

The analysis of the origin of roundoff errors in floating point computations [60, 77].

4.7 Symbolic analysis

The analysis of symbolic properties of programs, usually as opposed to numerical program properties e.g. heap reachability analysis [23, 24]. Also understood as the symbolic representation of abstract domains, which is almost always the case, since abstract properties cannot generally be simplistically represented as the set of concrete values which have this property (except in enumerative model-checking [10]).

4.8 Strictness and comportment analysis

Strictness analysis determines whether a parameter in a lazy language is always evaluated in a procedure call or this call does not terminate. This information is useful to replace a call-by-need by a more efficient call by value [91]. Introduced by Alan Mycroft [92], this analysis was at the origin of the use of abstract interpretation in the functional language community [1, 73]. Strictness analysis [8, 95] can be embedded in the lattice of comportment analyses [33].

4.9 Control flow analysis (CFA)

Control flow analysis is an abstract interpretation aiming at determining the possible flows of computation in higher-order functional languages [70, 105, 106, 82, 81].

4.10 Escape analysis

Escape analysis is a static analysis that determines whether the lifetime of data exceeds its static scope [96, 47], [4].

4.11 Mode inference/groundness/sharing analysis

The necessity of optimizing Prolog implementations, in particular the occur checks during unification, is at the origin of the use of abstract interpretation for the static analysis of Prolog [80, 45, 107, 75, 76, 90, 7, 12, 2], [29].

4.12 Aliasing and pointer analysis

The static analysis of imperative programs must take accesses and side effects through aliases and pointers into account [23]. The profuse literature on pointer analysis is surveyed in [46, 68].

4.13 Heap analysis

Heap analysis consists in overapproximating the sets of sequences of graphs representing the memory heap at all time instants during execution. Applications range from *memory leak detection* to *shape analysis* that is the determination of the shape of data structures allocated on the heap [103, 102].

4.14 Concurrency analysis

The static analysis of concurrent programs from shared-memory [28] to [a]synchronous communication [27], a vast and difficult subject, with few analyses that do scale up.

4.15 Mobility analysis

The analysis of dynamically allocated mobile processes (such as the π-calculus). This involves the overapproximation of the set of sequences of graphs representing the process allocations and communications at all time instants during execution [109, 51].

4.16 Probabilistic analysis

The static analysis of probabilistic programs [86, 88, 89], [48].

4.17 WCET analysis

The static determination of the worst-case execution time [49] which involves the analysis of the cache behavior [50], the pipelines, etc.

4.18 Hardware analysis

The static analysis of hardware formal descriptions such as asynchronous circuits [108].

5 Libraries of abstract domains

The notion of abstract domain in abstract interpretation (see **Sec. 3.1**) is independent of particular applications and can be implemented in libraries which can be used as parts of many different static analyses and even substituted for one another. Such libraries implement a representation of abstract properties and algorithms encoding operations on these properties such as logical operations, property transformers, widenings, etc.

5.1 The NewPolka polyhedral library

The NewPolka library handles convex polyhedra, whose constraints and generators have rational coefficients (with C and OCaml interfaces).

5.2 The Parma Polyhedra Library (PPL)

The Parma Polyhedra Library (PPL) is a user-friendly, fully dynamic, portable, exception-safe, and efficient C++ library to handle convex polyhedra that can be defined as the intersection of a finite number of (open or closed) hyperspaces, each described by an equality or inequality (strict or non-strict) with rational coefficients.

5.3 The octagon abstract domain library

The Octagon Abstract Domain Library is a library for manipulating special kinds of polyhedra called octagons that correspond to sets of constraints of the form $+/-x+/-y \leq c$ (where x and y are variables and c a rational constant and so, in dimension two, these polyhedra have at most eight faces) [85].

6 Abstract-interpretation-based tools

The following list of static analysis and verification automatic tools provides an idea of the scope of practical applications of abstract interpretation to software verification (but this list is definitely not exhaustive). These tools aim at soundness. Since no universal tool can exist (because of undecidability) such tools are necessarily incomplete and vary in their capacity to produce very few false alarms and to scale up to very large programs.

6.1 Airac5

A static analyzer from the Programming Research Laboratory of the Seoul National University and RopasWork for automatic detection of buffer overrun errors in C programs.

6.2 aiT WCET Analyzers

Commercial static analyzers from AbsInt Angewandte Informatik to statically compute tight bounds for the worst-case execution time (WCET) of tasks in real-time systems.

6.3 ASTRÉE

A static analyzer from the École normale supérieure for proving the absence of runtime errors specialized for synchronous control/command programs written in C.

6.4 C Global Surveyor

A research prototype static analyzer from the Intelligent Systems Division at the NASA Ames Research Center for finding runtime errors in C programs.

6.5 Fluctuat

A static analyzer from CEA-LIST to study the propagation of rounding errors in floating-point computations of C or assembler programs.

6.6 PolySpace Verifier

A commercial general-purpose static analyzer of PolySpace Technologies (1999–2007) for detecting runtime errors in Ada, C, C++, and Java. Acquired by The Mathworks in April 2007.

6.7 TERMINATOR

A static analyzer from Microsoft Research Cambridge for proving program termination and general liveness properties of C programs.

6.8 TVLA

TVLA is a research tool developed at Tel Aviv University for experimenting with abstract interpretation using a logical description of program semantics with a predefined abstraction of predicates.

7 Main conferences in abstract interpretation

- Generalist conferences:
  - The Asian symposium on Programming Languages And Systems APLAS'06,
  - The European Symposium on Programming ESOP'07
  - The Annual ACM SIGPLAN–SIGACT symposium on Principles Of Programming Languages POPL'07;
- Specialized conferences:
  - The International Static Analysis Symposium SAS'06;
  - The International Conference on Verification, Model Checking and Abstract Interpretation VMCAI'07.

8 Short bibliography

Short bibliography

[1] S. Abramsky and C. Hankin, editors. – *Abstract Interpretation of Declarative Languages*. – Ellis Horwood, 1987, *Computers and their Applications*.

[2] R. Barbuti, R. Giacobazzi and G. Levi. – A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *TOPLAS*, Vol. 15, n° 1, Jan. 1993, pp. 133–181.

[3] J. Berdine, A. Chawdhary, B. Cook, D. Distefano and P. O'Hearn. – Variance analyses from invariance analyses. *In : 34^{th} POPL*, edited by M. F. Martin Hofmann, Nice, FR, 2007. pp. 211–224. – ACM Press.

[4] B. Blanchet. – Escape Analysis for Java™. Theory and Practice. *TOPLAS*, Vol. 25, n° 6, Nov. 2003, pp. 713–775.

[5] F. Bourdoncle. – Abstract Interpretation by Dynamic Partitioning. *J. Func. Prog.*, Vol. 2, n° 4, 1992, pp. 407–435.

[6] F. Bourdoncle. – Efficient Chaotic Iteration Strategies with Widenings. *In : Proc. FMPA*, edited by D. Bjørner, M. Broy and I. Pottosin. *Akademgorodok, Novosibirsk, RU, LNCS 735*, pp. 128–141. – Springer, 28 June –2 Jul. 1993.

[7] M. Bruynooghe. – A Practical Framework for the Abstract Interpretation of Logic Programs. *J. Logic Programming*, Vol. 10, n° 1,2,3&4, Jan. 1991, pp. 91–124.

[8] G. Burn, C. Hankin and S. Abramsky. – Strictness Analysis of Higher-Order Functions. *Sci. Comput. Programming*, Vol. 7, Nov. 1986, pp. 249–278.

[9] E. Clarke, O. Grumberg and D. Long. – Model Checking and Abstraction. *TOPLAS*, Vol. 16, n° 5, Sep. 1994, pp. 1512–1542.

[10] E. Clarke, O. Grumberg and D. Peled. – *Model Checking*. – MIT Press, 1999.

[11] M. Comini, F. Damiani and S. Vrech. – On polymorphic recursion, type systems, and abstract interrpetation. *In : Proc. 15^{th} Int. Symp. SAS '08*, edited by M. Alpuente and G. Vidal, pp. 144–158. – Springer, 2008, *Valencia, ES, 16–18 Jul. 2008, LNCS 5079*.

[12] A. Cortesi and G. Filé. – Abstract interpretation of logic programs: an abstract domain for groundness, sharing, freeness and compoundness analysis. *In : Proc. PEPM '91*, edited by P. Hudak and N. Jones, pp. 52–61. – ACM Press, Sep. 1991, *Yale U., New Haven, CT, US, 17–19 June 1991, ACM SIGPLAN Not. 26(9).*

[13] A. Cortesi, G. Filé, R. Giacobazzi, C. Palamidessi and F. Ranzato. – Complementation in Abstract Interpretation. *TOPLAS*, Vol. 19, n° 1, Jan. 1997, pp. 7–47.

[14] P. Cousot. – *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French).* – Grenoble, FR, Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, 21 Mar. 1978.

[15] P. Cousot. – Semantic Foundations of Program Analysis, invited chapter. *In : Program Flow Analysis: Theory and Applications*, edited by S. Muchnick and N. Jones, Chapter 10, pp. 303–342. – Prentice-Hall, 1981.

[16] P. Cousot. – Types as Abstract Interpretations, invited paper. *In : 24^{th} POPL*, Paris, FR, Jan. 1997. pp. 316–331. – ACM Press.

[17] P. Cousot. – The Calculational Design of a Generic Abstract Interpreter, invited chapter. *In : Calculational System Design*, edited by M. Broy and R. Steinbrüggen, pp. 421–505. – NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, 1999, Volume 173.

[18] P. Cousot. – Partial Completeness of Abstract Fixpoint Checking, invited paper. *In : Proc. 4^{th} Int. Symp. SARA '2000*, edited by B. Choueiry and T. Walsh, pp. 1–25. – Springer, 26–29 Jul. 2000, *Horseshoe Bay, TX, US, LNAI 1864.*

[19] P. Cousot. – Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoret. Comput. Sci.*, Vol. 277, n° 1—2, 2002, pp. 47–103.

[20] P. Cousot. – Verification by Abstract Interpretation, invited chapter. *In : Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, edited by N. Dershowitz, pp. 243–268. – Taormina, IT, LNCS 2772, Springer, 29 June – 4 Jul. 2003.

[21] P. Cousot and R. Cousot. – Static determination of dynamic properties of programs. *In : Proc. 2^{nd} Int. Symp. on Programming*, Paris, FR, 1976. pp. 106–130. – Dunod.

[22] P. Cousot and R. Cousot. – Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In : 4^{th} POPL*, Los Angeles, CA, 1977. pp. 238–252. – ACM Press.

[23] P. Cousot and R. Cousot. – Static determination of dynamic properties of generalized type unions. *In : ACM Symposium on Language Design for Reliable Software*, Raleigh, NC, ACM SIGPLAN Not. 12(3):77–94, 1977.

[24] P. Cousot and R. Cousot. – Static determination of dynamic properties of recursive procedures. *In : IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, edited by E. Neuhold. pp. 237–277. – North-Holland, 1977.

[25] P. Cousot and R. Cousot. – Constructive versions of Tarski's fixed point theorems. *Pacific J. Math.*, Vol. 82, n° 1, 1979, pp. 43–57.

[26] P. Cousot and R. Cousot. – Systematic design of program analysis frameworks. *In : 6^{th} POPL*, San Antonio, TX, 1979. pp. 269–282. – ACM Press.

[27] P. Cousot and R. Cousot. – Semantic analysis of communicating sequential processes. *In : 7^{th} ICALP*, edited by J. de Bakker and J. van Leeuwen. *LNCS 85*, pp. 119–133. – Springer, Jul. 1980.

[28] P. Cousot and R. Cousot. – Invariance Proof Methods and Analysis Techniques For Parallel Programs, invited chapter. *In : Automatic Program Construction Techniques*, edited by A. Biermann, G. Guiho and Y. Kodratoff, Chapter 12, pp. 243–271. – Macmillan, 1984.

[29] P. Cousot and R. Cousot. – Abstract Interpretation and Application to Logic Programs. *J. Logic Programming*, Vol. 13, n° 2–3, 1992, pp. 103–179. – (The editor of J. Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.).

[30] P. Cousot and R. Cousot. – Abstract Interpretation Frameworks. *J. Logic and Comp.*, Vol. 2, n° 4, Aug. 1992, pp. 511–547.

[31] P. Cousot and R. Cousot. – Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. *In : Proc. 4^{th} Int. Symp. on PLILP '92*, edited by M. Bruynooghe and M. Wirsing. *Leuven, BE, 26–28 Aug. 1992, LNCS 631*, pp. 269–295. – Springer, 1992.

[32] P. Cousot and R. Cousot. – Inductive Definitions, Semantics and Abstract Interpretation. *In : 19^{th} POPL*, Albuquerque, NM, US, 1992. pp. 83–94. – ACM Press.

[33] P. Cousot and R. Cousot. – Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages), invited paper. *In : Proc. 1994 ICCL*, Toulouse, FR, 16–19 May 1994. pp. 95–112. – IEEE Comp. Soc. Press.

[34] P. Cousot and R. Cousot. – Compositional and Inductive Semantic Definitions in Fixpoint, Equational, Constraint, Closure-condition, Rule-based and Game-Theoretic Form, invited paper. *In: Proc. 7th Int. Conf. CAV '95*, edited by P. Wolper. *Liège, BE, LNCS 939*, pp. 293–308. – Springer, 3–5 Jul. 1995.

[35] P. Cousot and R. Cousot. – Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. *In: Proc. 7th FPCA*, La Jolla, CA, US, 25–28 June 1995. pp. 170–181. – ACM Press.

[36] P. Cousot and R. Cousot. – Temporal Abstract Interpretation. *In: 27th POPL*, Boston, MA, US, Jan. 2000. pp. 12–25. – ACM Press.

[37] P. Cousot and R. Cousot. – Modular Static Program Analysis, invited paper. *In: Proc. 11th Int. Conf. CC '2002*, edited by R. Horspool, Grenoble, FR, 6–14 Apr. 2002. pp. 159–178. – LNCS 2304, Springer.

[38] P. Cousot and R. Cousot. – Systematic Design of Program Transformation Frameworks by Abstract Interpretation. *In: 29th POPL*, Portland, OR, US, Jan. 2002. pp. 178–190. – ACM Press.

[39] P. Cousot and R. Cousot. – Parsing as Abstract Interpretation of Grammar Semantics. *Theoret. Comput. Sci.*, Vol. 290, n^{o} 1, Jan. 2003, pp. 531–544.

[40] P. Cousot and R. Cousot. – An Abstract Interpretation-Based Framework for Software Watermarking. *In: 31st POPL*, Venice, IT, 14–16 Jan. 2004. pp. 173–185. – ACM Press.

[41] P. Cousot and R. Cousot. – Grammar Analysis and Parsing by Abstract Interpretation, invited chapter. *In: Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, edited by T. Reps, M. Sagiv and J. Bauer, pp. 178–203. – Springer, 2006, *LNCS 4444*.

[42] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. – The ASTRÉE analyser. *In: Proc. 14th ESOP '2005, Edinburg, UK*, edited by M. Sagiv, pp. 21–30. – Springer, 2–10 Apr. 2005, *LNCS*, Vol. 3444.

[43] P. Cousot, P. Ganty and J.-F. Raskin. – Fixpoint-Guided Abstraction Refinements. *In: Proc. 14th Int. Symp. SAS '07*, edited by G. Filé and H. Riis-Nielson, pp. 333–348. – Springer, 22–24 Aug. 2007, *Kongens Lyngby, DK, LNCS 4634*.

[44] P. Cousot and N. Halbwachs. – Automatic discovery of linear restraints among variables of a program. *In: 5th POPL*, Tucson, AZ, 1978. pp. 84–97. – ACM Press.

[45] S. Debray and D. Warren. – Automatic mode inferencing for Prolog programs. *In : Proc. 1986 Int. Symp. on Logic Programming*, pp. 78–88. – IEEE Comp. Soc. Press, Sep. 1986, *Salt Lake City, UT.*

[46] A. Deutsch. – Semantic models and abstract interpretation techniques for inductive data structures and pointers, , invited paper. *In : Proc. PEPM '95*, La Jolla, CA, 21–23 June 1995. pp. 226–229. – ACM Press.

[47] A. Deutsch. – On the complexity of escape analysis. *In : 24^{th} POPL*, Paris, FR, Jan. 1997. pp. 358–371. – ACM Press.

[48] A. Di Pierro, C. Hankin and H. Wiklicky. – Probabilistic lambda calculus and quantitative program analysis. *J. Logic and Comp.*, Vol. 15, n° 2, 2005, pp. 159–179.

[49] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing and R. Wilhelm. – Reliable and Precise WCET Determination for a Real-Life Processor. *In : Proc. 1^{st} Int. Work. EMSOFT '2001*, edited by T. Henzinger and C. Kirsch, pp. 469–485. – Springer, 2001, *LNCS*, Vol. 2211.

[50] C. Ferdinand, F. Martin, R. Wilhelm and M. Alt. – Cache behavior prediction by abstract interpretation. *Sci. Comput. Programming*, Vol. 35, n° 1, 1999, pp. 163–189.

[51] J. Feret. – Abstract Interpretation-Based Static Analysis of Mobile Ambients. *In : Proc. 8^{th} Int. Symp. SAS '01*, edited by P. Cousot. *Paris, FR, LNCS 2126*, pp. 413–431. – Springer, 16–18 Jul. 2001.

[52] G. Filè, R. Giacobazzi and F. Ranzato. – A Unifying View on Abstract Domain Design. *ACM Computing Surveys*, Vol. 28, n° 2, 1996, pp. 333–336.

[53] G. Filé and F. Ranzato. – The Powerset Operator on Abstract Interpretations. *Theoret. Comput. Sci.*, Vol. 222, n° 1-2, Jul. 1999, pp. 77–111.

[54] R. Giacobazzi and I. Mastroeni. – Non-Standard Semantics for Program Slicing. *Higher-Order and Symbolic Computation*, Vol. 16, n° 4, 2003, pp. 297–339.

[55] R. Giacobazzi and I. Mastroeni. – Timed Abstract Non-Interference. *In : Proc. 3^{rd} Int. Conf. FORMATS '05*, edited by B. Le Charlier, pp. 289–303. – Springer, 2005, *Uppsala, SE, 26-28 Sep. 2005, LNCS 3829.*

[56] R. Giacobazzi and E. Quintarelli. – Incompleteness, Counterexamples and Refinements in Abstract Model-Checking. *In : Proc. 8^{th} Int. Symp. SAS '01*, edited by P. Cousot. *Paris, FR, LNCS 2126*, pp. 356–373. – Springer, 16–18 Jul. 2001.

[57] R. Giacobazzi and F. Ranzato. – Completeness in Abstract Interpretation: A Domain Perspective. *In : Proc. 6th Int. Conf. AMAST '97, Sydney, AU*, edited by M. Johnson. *LNCS*, Vol. 1349, pp. 231–245. – Springer, 13–18 Dec. 1997.

[58] R. Giacobazzi, F. Ranzato and F. Scozzari. – Making Abstract Interpretations Complete. *J. ACM*, Vol. 47, n° 2, 2000, pp. 361–416.

[59] R. Giacobazzi and F. Scozzari. – A logical model for relational abstract domains. *TOPLAS*, Vol. 20, n° 5, 1998, pp. 1067–1109.

[60] É. Goubault, M. Martel and S. Putot. – Asserting the precision of floating-point computations: a simple abstract interpreter. *In : Proc. 11th ESOP '2002*, edited by D. Le Métayer, pp. 209–212. – Springer, 8–12 Apr. 2002, *Grenoble, FR, LNCS 2305*.

[61] S. Graf and H. Saïdi. – Verifying Invariants Using Theorem Proving. *In : Proc. 8th Int. Conf. CAV '97*, edited by R. Alur and T. Henzinger. *New Brunswick, NJ, US, LNCS 1102*, pp. 196–207. – Springer, Jul. 31 — Aug. 3 1996.

[62] P. Granger. – Static Analysis of Arithmetical Congruences. *Int. J. Comput. Math.*, Vol. 30, 1989, pp. 165–190.

[63] P. Granger. – Static Analysis of Linear Congruence Equalities among Variables of a Program. *In : Proc. Int. J. Conf. TAPSOFT '91, Volume 1 (CAAP '91)*, edited by S. Abramsky and T. Maibaum. *Brighton, GB, LNCS 493*, pp. 169–192. – Springer, 1991.

[64] N. Halbwachs. – Delay Analysis in Synchronous Programs. *In : Proc. 5th Int. Conf. CAV '93*, edited by C. Courcoubatis. *Elounda, GR, LNCS 697*, pp. 333–346. – Springer, 28 June –1 Jul. 1993.

[65] N. Halbwachs. – About Synchronous Programming and Abstract Interpretation. *In : Proc. 1st Int. Symp. SAS '94*, edited by B. Le Charlier, pp. 179–192. – Springer, 1994, *Namur, BE, 20–22 Sep. 1994, LNCS 864*.

[66] N. Halbwachs, Y. Proy and P. Roumanoff. – Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, Vol. 11, n° 2, Aug. 1997, pp. 157–185.

[67] M. Hecht. – *Flow Analysis of Computer Programs.* – North-Holland/Elsevier, 1977.

[68] M. Hind. – Pointer Analysis: Haven't We Solved This Problem Yet? *In : 2001 ACM SIGPLAN-SIGSOFT Workshop PASTE '01*, Snowbird, UT, US, 2001.

[69] B. Jeannet, N. Halbwachs and P. Raymond. – Dynamic Partitioning in Analyses of Numerical Properties. *In : Proc. 6th Int. Symp. SAS '99*, edited by A. Cortesi and G. Filé, pp. 18–38. – Springer, 1999, *Venice, IT, 22–24 Sep. 1999, LNCS 1694*.

[70] N. Jones. – Flow Analysis of Lambda Expressions (Preliminary Version). *In : 8th ICALP*, edited by S. Even and O. Kariv. *LNCS 115*, pp. 114–128. – Springer, Jul. 1981.

[71] N. Jones. – Combining Abstract Interpretation and Partial Evaluation (Brief Overview). *In : Proc. 4th Int. Symp. SAS '97*, edited by P. Van Hentenryck, pp. 396–405. – Springer, 1997, *Paris, FR, 8–10 Sep. 1997, LNCS 1302*.

[72] N. Jones, C. Gomard and P. Sestoft. – *Partial Evaluation and Automatic Program Generation*. – Prentice-Hall, June 1993, *Int. Series in Computer Science*.

[73] N. Jones and F. Nielson. – Abstract interpretation: a semantics-based tool for program analysis. *In : Semantic Modelling*, edited by S. Abramsky, D. Gabbay and T. Maibaum, Chapter 5, pp. 527–636. – Clarendon Press, 1995, *Handbook of Logic in Computer Science*, Vol. 4.

[74] N. Jones and M. Rosendahl. – Higher-Order Minimal Function Graphs. *J. Func. and Logic Prog.*, Vol. 1997, n° 2, Feb. 1997.

[75] H. Mannila and E. Ukkonen. – Flow analysis of Prolog programs. *In : Proc. 1987 Int. Symp. on Logic Programming. San Francisco, CA*, pp. 205–214. – IEEE Comp. Soc. Press, 31 Aug. – 4Sep. 1987.

[76] K. Marriott and H. Søndergaard. – Bottom-Up Abstract Interpretation of Logic Programs. *In : Proc. 5th Int. Conf. & Symp. on Logic Programming, Volume 1*, edited by R. Kowalski and K. Bowen. *Seattle, WA, US*, pp. 733–748. – MIT Press, 15–19 Aug. 1988.

[77] M. Martel. – An Overview of Semantics for the Validation of Numerical Programs. *In : Proc. 6th Int. Conf. VMCAI 2005*, edited by R. Cousot, Paris, FR, 17–19 Jan. 2005. pp. 59–77. – LNCS 3385, Springer.

[78] L. Mauborgne. – Tree Schemata and Fair Termination. *In : Proc. 7th Int. Symp. SAS '2000*, edited by J. Palsberg, pp. 302–321. – Springer, 29 June – 1 Jul. 2000, *Santa Barbara, CA, US, LNCS 1824*.

[79] L. Mauborgne and X. Rival. – Trace Partitioning in Abstract Interpretation Based Static Analyzer. *In : Proc. 14th ESOP '2005, Edinburg, UK*, edited by M. Sagiv, pp. 5–20. – Springer, Apr. 2ÃŚ-10, 2005, *LNCS*, Vol. 3444.

[80] C. Mellish. – Abstract Interpretation of Prolog Programs. *In : 3rd ICLP '86*, edited by E. Shapiro, pp. 463–474. – Springer, 14–18 Jul. 1986, *London, GB, LNCS 225*.

[81] J. Midtgaard and T. Jensen. – A calculational approach to control-flow analysis by abstract interrpetation. *In : Proc. 15th Int. Symp. SAS '08*, edited by M. Alpuente and G. Vidal, pp. 347–362. – Springer, 2008, *Valencia, ES, 16–18 Jul. 2008, LNCS 5079*.

[82] M. Might and . Shivers. – Analyzing the environment structure of higher-order languages using frame strings. *Theoret. Comput. Sci.*, Vol. 375, nº 1–3, 2007, pp. 137–168.

[83] A. Miné. – A New Numerical Abstract Domain Based on Difference-Bound Matrices. *In : Proc. 2nd Symp. PADO '2001*, edited by . Danvy and A. Filinski. *Århus, DK, 21–23 May 2001, LNCS 2053*, pp. 155–172. – Springer, 2001.

[84] A. Miné. – A Few Graph-Based Relational Numerical Abstract Domains. *In : Proc. 9th Int. Symp. SAS '02*, edited by M. Hermenegildo and G. Puebla. *LNCS*, Vol. 2477, pp. 117–132. – Springer, 2002.

[85] A. Miné. – The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, Vol. 19, 2006, pp. 31–100.

[86] D. Monniaux. – Abstract interpretation of probabilistic semantics. *In : Proc. 7th Int. Symp. SAS '2000*, edited by J. Palsberg, pp. 322–339. – Springer, 29 June – 1 Jul. 2000, *Santa Barbara, CA, US, LNCS 1824*.

[87] D. Monniaux. – An Abstract Analysis of the Probabilistic Termination of Programs. *In : Proc. 8th Int. Symp. SAS '01*, edited by P. Cousot. *Paris, FR, LNCS 2126*, pp. 111–127. – Springer, 16–18 Jul. 2001.

[88] D. Monniaux. – An Abstract Monte-Carlo Method for the Analysis of Probabilistic Programs (extended abstract). *In : 28th POPL*, London, GB, Jan. 2001. pp. 93–101. – ACM Press.

[89] D. Monniaux. – Backwards abstract interpretation of probabilistic programs. *In : Proc. 10th ESOP '2001*, edited by D. Sands. *Genova, IT, 2–6 Apr. 2001, LNCS 2028*, pp. 367–382. – Springer, 2001.

[90] K. Muthukumar and M. Hermenegildo. – Determination of Variable Dependence Information through Abstract Interpretation. *In : NACLP 1989, Volume 1*, edited by E. Lusk and R. Overbeek. *Cleaveland, OH, US*, pp. 166–185. – MIT Press, 16–20 Oct. 1989.

[91] A. Mycroft. – The theory and practice of transforming call-by-need into call-by-value. *In : Proc. 4th Int. Symp. on Programming*, edited by B. Robinet, pp. 270–281. – Springer, 1980, *Paris, FR, 22–24 Apr. 1980, LNCS 83*.

[92] A. Mycroft. – *Abstract Interpretation and Optimising Transformations for Applicative Programs.* – Edinburg, UK, Ph.D. Dissertation, CST-15-81, Department of Computer Science, University of Edinburgh, Dec. 1981.

[93] A. Mycroft. – Completeness and predicate-based abstract interpretation. *In: Proc. PEPM '93. Copenhagen, DK, 14–16 June 1993*, pp. 80–87. – ACM Press, 1993.

[94] A. Mycroft and N. Jones. – A Relational Framework for Abstract Interpretation. *In: Programs as Data Objects, Proceedings of a Workshop*, edited by N. Jones and H. Ganzinger, pp. 156–171. – Springer, 1986, *Copenhagen, DK, 17-19 Oct. 1985, LNCS 215.*

[95] F. Nielson. – Strictness Analysis and Denotational Abstract Interpretation. *In: 14^{th} POPL*, Munchen, DE, 1987. pp. 120–131. – ACM Press.

[96] Y. G. Park and B. Goldberg. – Escape analysis on lists. *In: Proc. ACM SIGPLAN '92 Conf. PLDI. ACM SIGPLAN Not. 31(5)*, San Francisco, CA, US, 21–24, May 1992. pp. 116–127. – ACM Press.

[97] B. Pierce. – *Types and Programming Languages.* – MIT Press, 2002.

[98] M. D. Preda, M. Christodorescu, S. Jha and S. Debray. – Semantics-Based Approach to Malware Detection. *In: 34^{th} POPL*, Nice, France, 17–19 Jan. 2007. pp. 238–252. – ACM Press.

[99] M. D. Preda and R. Giacobazzi. – Control Code Obfuscation by Abstract Interpretation. *In: Proc. 3^{rd} IEEEInt. Conf. SEFM '05*, Koblenz, DE, 2005. – IEEE Comp. Soc. Press.

[100] F. Ranzato. – On the Completeness of Model Checking. *In: Proc. 10^{th} ESOP '2001*, edited by D. Sands. *Genova, IT, 2–6 Apr. 2001, LNCS 2028*, pp. 137–154. – Springer, 2001.

[101] F. Ranzato and F. Tapparo. – Strong Preservation of Temporal Fixpoint-Based Operators by Abstract Interpretation. *In: Proc. 7^{th} Int. Conf. VMCAI 2006*, edited by A. Emerson and K. Namjoshi, Charleston, SC, US, 8–10 Jan. 2006. pp. 332–347. – LNCS 3855 , Springer.

[102] N. Rinetzky, J. Bauer, T. Reps, S. Sagiv and R. Wilhelm. – A semantics for procedure local heaps and its abstractions. *In: 32^{nd} POPL*, Long Beach, CA, US, 2005. pp. 296–309. – ACM Press.

[103] M. Sagiv, T. Reps and R. Wilhelm. – Shape Analysis. *In: Proc. Int. Conf. CC '2000, LNCS 1781*, edited by D. A. Watt, Berlin, DE, 25 Mar. – 2 Apr. 2000. pp. 1–17. – Springer.

[104] D. Schmidt. – Comparing completeness properties of static analyses and their logics. *In: Proc. 4^{th} APLAS '2006*, edited by N. Kobayashi, Sydney, AU, 8–10 Nov. 2006. pp. 183–199. – LNCS 4279, Springer.

[105] O. Shivers. – Control-Flow Analysis in Scheme. *In : Proc. ACM SIGPLAN '1988 Conf. PLDI. ACM SIGPLAN Not. 23(7)*, Atlanta, GE, US, 22–24 June 1988. pp. 164–174. – ACM Press.

[106] O. Shivers. – Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. *ACM SIGPLAN Not.*, Vol. 39, n° 4, 2004, pp. 257–269.

[107] H. Søndergaard. – An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. *In : Proc. ESOP '86*, edited by B. Robinet and R. Wilhelm, pp. 327–338. – Springer, 1986, *Saarbrücken, DE, 17-19 Mar. 1986, LNCS 213*.

[108] S. Thompson and A. Mycroft. – Abstract Interpretation of Combinational Asynchronous Circuits. *In : Proc. 11th Int. Symp. SAS '04*, edited by R. Giacobazzi. *Verona, IT, LNCS 3148*, pp. 181–196. – Springer, 26–28 Aug. 2004.

[109] A. Venet. – Automatic Determination of Communication Topologies in Mobile Systems. *In : Proc. 5th Int. Symp. SAS '98*, edited by G. Levi, pp. 152–167. – Springer, 1998, *Pisa, IT, 14–16 Sep. 1998, LNCS 1503*.

This document is hopefully sound, may be too abstract and necessarily incomplete. Amendments are welcomed at `Patrick.Cousot@ens.fr`.
