ACM Workshop on Strategic Directions in Computing Research

MIT Laboratory for Computer Science
Cambridge, Massachusetts, USA
June 14-15, 1996

Draft Program Analysis Position Statement: **The Abstract Interpretation Perspective**

Patrick Cousot*

Program Analysis Position Statement:

- Semantic analysis (of programs, systems, etc.) should become a mature discipline to be included in educational curricula in computer science;
- This requires the discipline to evolve from an ill-assorted collection of methods and algorithms to a large scope formal reasoning and design methodology;
- In particular, program analysis methods should be explained in language, program encoding, semantics, property, property encoding, and approximation independent way, to be of very broad scope and wide usefulness;
- This might be possible by reasoning on the approximation of the structures involved in semantical specifications;
- This effort would culminate in a general theory of semantical approximation leading to a tractable composable parameterized semantical analysis design methodology;
- From a more practical point of view, the methodology would be applied to the uniform design of:
  - hierarchies of semantics,
  - hierarchies of abstract algebras

  at various levels of abstraction/refinement;
- This would lead to generic program analysis tools most of which could be included in reusable and composable libraries;
- An ultimate practical goal would be to automate the generation of such semantic analysis tools primarily for program analysis and more generally for system specification analysis.

An extended argumentation in favor of this position statement is given below.

1. Situation of Program Analysis

1.1 Definition

Program semantic analysis consists of designing and writing program analyzers. A program analyzer is a program which takes as input data a (possibly annotated) program (written in some language) and (fully automatically) produces as output answers to questions about runtime properties which are valid for all (or sometimes for some) possible executions of that program. Because of undecidability or complexity, these answers are necessarily partial, but should be irrefutable.

*DMI, École Normale Supérieure, 45 rue d'Ulm, 75230 Paris Cedex 05, France. Email: cousot@dmi.ens.fr, URL: http://www.ens.fr/~cousot.

1.2 From Programs to Systems

Traditional applications of program analysis concern:

- Program compilation (e.g. guaranteeing the correctness of a compiler optimization [58]);
- Program debugging (e.g. automatic test case generation, abstract debugging [3] (where data abstract properties are used instead of data values in classical debugging) or automatic verification of assertions);
- Program transformation (e.g. ensuring termination of partial evaluations [40]).

for imperative, functional, logic, parallel, object-oriented, etc. languages.

The methods used in program analysis, which do cope with undecidability, have a much broader potential scope of application which is still largely unexplored. For example:

- Analysis of high confidence reactive, real-time, parallel, or distributed infinite state computer systems (e.g. decomposition and simplification into a partial analysis by abstraction);
- Debugging of software specifications (e.g. of air traffic control systems, public transportation systems, banking networks, etc.);
- Analysis of atomicity, concurrency, coherence, ... in data and knowledge bases.

1.3 From Theory to Practice

Program analysis is both a theoretical activity (close to formal specification methods, semantics, etc.) and a practical activity (close to compiler design and development). In a sense this situation is quite creative: looking at a real and vast application problem is a fruitful guide to the process of formalization of the semantics of programming languages, proof methods, etc. This is also a difficult position akin to the old debate between pure and applied mathematics. Often practitioners are not worried about theories while theorists rarely take time to find applications of their theoretical work which is loosely coupled with practical problems.

We would like to illustrate these two aspects by considering methodological and implementation problems related to program analysis and by suggesting some research directions.

2. Methodological Aspects

2.1 Need for Design Methodologies

Numerous program analysis technologies exist which are applicable to specific languages, specific encoding of programs, specific semantics, specific presentations of semantics, specific program properties, specific encoding of these properties, etc. For example, dataflow analysis [45] is strongly bound to imperative languages, graph (flowchart) presentation of programs, finite traces semantics (so-called merge over all paths) or even absence of formal semantics, boolean properties (or at least finitary ones). Adaptation of dataflow analysis to e.g. higher-order functional programming languages is not foreseeable. For a second example, type based analysis [48] is strongly related to functional languages, operational (sometimes denotational) semantics, abstract syntax, prescriptive type properties, term structure and unification, etc. Adaptation of type/inference based analysis to e.g. model checking of transition systems, or to logic languages has not been considered (in this last case, despite the sharing of the term structure). We could multiply such specific (and certainly debatable) examples.

After almost three decades of research on program analysis, researchers have produced thousands of profoundly (and sometimes superficially) incomparable algorithms (and papers). In order to extract common principles, it seems necessary to provide a program analyzer design methodology.

This point of view is not always shared. After all, a program analyzer is just an algorithm among many other ones which computer scientists can produce easily, mainly by mimicking or modifying existing ones. However, one can think of several arguments in favor of formal methods as opposed to empirical design:

complexity: the task of producing a program analyzer is complex, in some cases much more complex than producing a traditional compiler. An important aspect of a program analysis design methodology is to provide guidance for the decomposition of this complex task into simpler separate ones;

correctness: a program analysis specification can pinpoint early design errors (such as complexity analysis which can be anticipated from the abstraction) and is indispensable to prove correctness. Such a specification can hardly be informal since it must refer to a semantics, most often presented formally;

universality: specific algorithms can be hard to understand, modify and apply by engineers who would like to implement program analyzers. Formal methods enforce underlying principles to be well-understood, widely applicable and easily explained;

taxonomy: universal formal specification methods allow for a comparative investigation of program analysis methods at a very high level of abstraction (see e.g. [27] for pointer analysis)

teachability: there is a difficulty in teaching specific techniques which have a relatively short lifetime as opposed to fundational and reusable theories;

wide spectrum: since the number of formal methods that can be assimilated by a computer scientist is limited, the formal method should have a much broader scope than just program analysis and consequently should be based on simple mathematical concepts of familiar use in computer science.

2.2 Abstract Interpretation as a Theory of Semantic Approximation

Abstract interpretation is such a methodology. It relies on the idea that the specification of an analyzer is an approximation of a semantics, where concrete or exact properties are replaced by abstract or approximate properties. This idea of approximation, and dually of refinement, is central to computer science although it is often left informal. We will discuss it, in the context of program analysis only, although the discussion is also relevant to system analysis, design of semantics, etc.

2.3 Scope of a Design Methodology

When discussing program analysis methods it is important to evaluate their respective scopes. To be of general scope, a program analysis design methodology should emphasize underlying concepts and therefore be presented in a language, semantics and property independent way thus providing modular techniques and tools which can be composed to design actual program analyzers which are correct (and efficient), by construction. It appears however, that the development of such intellectual tools is insufficient without training in their use.

2.4 Programming Language Independence

It is not easy to present program analysis techniques in a way which is independent of a particular program representation. One common technique, originating from compiler design, is therefore to use intermediate languages. Early papers [8, 9] took flowcharts as examples, because flowcharts were considered at the time as a widespread and suitable intermediate language for imperative program analysis. This was abandoned immediately, e.g. to handle recursive procedures [11]. Low-level intermediate languages are clearly not suitable for e.g. higher-order functional languages for which various attempts have been made to propose the lambda-calculus or its multi-level variants [53] as such a universal intermediate language. But the lambda-calculus is not quite adequate e.g. for (concurrent constraint) logic languages. We think that although, for a specific analyzer, it might be a good idea to use an adequate intermediate language (simply because some properties such as storage allocation and side effects can only be explained in terms of this intermediate language), it is not a good idea to base a program analysis design methodology on the choice of a particular language (whether intermediate or high-level). It is quite surprising how long the underlying principles of [9] were confused with their *application* to the example of flowcharts, in particular it was long thought that direct application of these principles to higher-order functional languages was impossible. On the contrary it turned out that the language independent presentation of abstract interpretation when *applied* to denotational semantics based comportment analysis [18], leads to a generalization of previous work on denotational semantics based strictness analysis [52].

2.5 Property Representation Independence

We think that a program analysis design methodology should not be bound to a specific representation or encoding of properties because this would either limit its scope or enforce clumsy property encodings. A counter-example would be the once and for all choice of a specific class of formal languages for representing program properties. Type inference, effect systems, or set-based analysis do make such an arbitrary choice and therefore limit their scope of applicability. In addition, inference based approaches are bound to an algorithm for building the inference tree which unduly influences the specification level. Other examples are program analysis in logical form (e.g. for strictness analysis [38]) or the exclusive use of continuation passing style which are equivalent to the use of (very time and space expensive) disjunctive completions.

2.6 Hierarchies of Abstract Algebras

To a unique encoding of abstract properties (e.g. using logical formulæ or term algebra), we prefer the idea of hierarchies of application independent abstract algebras (the term abstract algebra is better than abstract domain since the design must include abstract properties but also abstract operations, widenings, ...). An example of multi-uses abstract algebra is linear inequalities for the convex-hull approximation of a set of vectors of numbers by a polyhedron [21] which have been used to estimate the size of arguments of logic programs [13, 23] and for infinite state model-checking [36, 22] including hybrid systems [37]. Another example is BDDs originating from model-checking but also useful to cope with size explosion in strictness analysis [47]. Not enough such analysis independent abstract algebras are presently available, in particular for discrete structures (see a.o. [20, 25]) as opposed to numerical structures where the spectrum is wider [8, 43, 42, 21, 33, 46]. Some abstract algebras are used without being clearly identified (e.g. for type inference, see however [50] for references). In this area, mainly of algorithmic nature, not enough constructive methods and algorithms from algebra (see e.g. [30]), functional analysis or discrete mathematics have been investigated. Such algorithmic problems are rather involved and require experts, but their results could be easily incorporated in portable widely distributed libraries as is the case e.g. in the fields of numerical analysis and operational research.

2.7 Semantics Independence

It was recognized early on that a program analysis design methodology bound to a particular (class of) language(s) has great chances to have a confined usability.

2.7.1 Transition Systems/Operational Semantics

Instead, it was proposed to reason on transition systems [12, 5]. They correspond to small-step operational semantics and can be given many different computation models (such as, in [12], traces to cope with syntactic dataflow properties or set of states transformers to cope with invariance/safety properties). Transition systems are well-suited for semantic analysis of programs but also systems [6, 37] so that they have been successfully used in other fields (model-checking, process algebra). For program analysis, transition systems are independent of any programming language and reasonably independent of a particular representation of programs (although a natural notion of states is needed). This may explain why the abstract interpretation theory of transition systems turned out to be directly applicable to languages (e.g. to logic languages [13, 23]) which are hardly comparable to the original domain of application (procedural languages). Transition systems can easily handle non-determinism and have nice generalizations e.g. higher-dimensional automata to handle true concurrency [30] and real-time [31]. However, transition systems are not the ultimate choice for basing a program analysis design methodology. For example, higher-order functions can only be handled using syntactic encodings (e.g. term-graph rewriting [32] and geometry of interaction machine [39]).

2.7.2 Denotational Semantics

The first application of abstract interpretation to strictness analysis was based on a denotational semantics [52]. One clear advantage, at least at the specification level, is compositionality [41]. Surprisingly, these are very few program analyses based on denotational semantics apart from trivial finite domains. We think there are two main reasons for this situation:

Domain theory Denotational semantics is based often on domain theory [35] whose purpose is to provide models of effectively computable objects. Program analysis is concerned with properties, which is slightly different, e.g. deals with non computable objects. Expressing computations and properties in the same model may be misleading. Very often abstract interpretation is understood as a non-standard execution using abstract values instead of concrete ones, that is in terms of computations and not of properties. This point of view is quite limited, since e.g. relational analyses (e.g. [21]) cannot be explained in this naïve way. The confusion between values and properties may also explain that for long the computational ordering and the approximation ordering (implication of properties) have not been clearly distinguished both at the concrete and abstract levels (so that e.g. absence, originating from projection analysis [57] was not expressible [18, 56]). Finally mimicking the structure of computations to obtain the structure of properties is often not adequate (e.g. when modeling properties of functions $D \mapsto D$ by functions $\wp(D) \mapsto \wp(D)$ (since e.g. absence is not expressible in this way [18]) or similarly for data structures where properties are necessarily uniform (compare with e.g. [26])).

Compositionality Denotational semantics is compositional, in that it associates models of computations to programs by induction on their abstract syntax[1]. An immediate consequence is that most denotational based program analysis methods are bound to the particular abstract syntax for

[1] "The main idea of Denotational Semantics is that each phrase of the language described is given a *denotation*: a mathematical object that represents the contribution of the phrase to the meaning of any complete program in which it occurs. Moreover, the denotation of each phrase is determined just by the denotations of its subphrases" in [51].

a particular language (often some variant of the lambda-calculus). A more important consequence is that the compositional presentation of the abstract semantics although directly implementable has inherent costs (e.g. for continuations, for higher-order functions, even on tiny basic domains).

2.8 Hierarchies of Semantics

Founding a program analysis methodology on a specific semantics (small/big step operational, denotational, weakest-preconditions, etc.) turns out not to be very convenient. This is because a program analysis methodology based on a compositional semantics (e.g. denotational or axiomatic) can hardly be formulated independently of a specific language syntax and a specific class of program properties. Consequently the choice of a specific semantics may lead to elegant or tortuous specifications of properties whether they are directly expressible by the semantics or not: think of invariance properties expressed in terms of a denotational semantics or termination in terms of a small-step operational semantics. Consequently no particular semantics can be considered as general purpose.

We suggest instead to consider hierarchies of semantics which can describe program properties, that is program executions at various levels of abstraction or refinement [7]. Then for a given class of properties there should be a natural choice of semantics in the hierarchy (for example strictness analysis is awkward to explain using a denotational semantics [52] but very simple using a *relational semantics* [17] since there is a simple way to explain how a relation can be understood as a logical approximation of another, whereas this is not so simple for functions. Relations are too abstract to express absence, or more generally comportment which requires a further refinement [18]).

2.9 Semantics Presentation Independence

Since it is not conceivable to have a specific program analyzer design methodology for all possible semantics, we suggest to attain semantics independence by explaining the design of program analyzers in terms of semantics specification methods, for which notions of abstractions can be defined independently of the defined language [16]. In this way, fixpoint based, constraint based, inference based program analysis methods can be presented in a uniform, semantic independent, framework [19].

The ultimate abstraction, to be language, semantics and property independent is to reason in terms of mathematical structures commonly used to describe these objects and their properties e.g. abstract interpretation of compositional inductive definitions [16, 19]. This leads to a modular design of program analyzers, by composing abstractions of the mathematical structures used in the semantics.

2.10 Abstraction Independence

Several methods for relating concrete and abstract semantics have been considered and discussed (soundness relation, abstraction function, concretization function, abstraction/concretization Galois connection [14]). The widening approach is more general [15] since it requires weaker hypotheses on the semantics and the abstraction (because the approximation is done on the fly so that e.g. missing bounds can be approximated dynamically (see e.g. [21])). Since it is not possible to write papers where each alternative is considered, a particular formalization of abstraction has to be chosen e.g. Galois connections for elegance or widenings for generality or a mixture of the two for realism. But some potential users will always be unsatisfied because the exact case corresponding to their problem (e.g. (non-)existence of best approximations, of limits, (in-)finiteness, etc.) is not handled in the paper.

No generic concept encompasses all others which could be properly instantiated by incorporating the hypotheses at hand. An ultimate such concept (encompassing a.o. order-theoretic as well as metric-theoretic approximation) has to be studied. But such meta-theories (e.g. set theory parameterized by axioms) are hard to imagine.

2.11 Mathematical Tool Adequacy

The previous remark on generic concepts encompassing many similar formalisms applies at a more general level to the use of formal methods in computer science. It is difficult to choose an adequate mathematical structure, both at a technical and sociological level. For technical adequacy, lattices, metric spaces, topological spaces, categories, toposes, all have notions of approximation but we still lack a unifying concept widely applicable e.g. to approximation in semantics, image processing and numerical analysis. In a sense present mathematics is not fully adequate for our purpose (physicists often have a similar problem). For sociological adequacy, choosing one of these mathematical structures immediately excludes those computer scientists who are not prepared to read formal papers using this particular mathematical structure.

2.12 Universality versus Simplicity

By looking at a universal program analysis design methodology, we inevitably end up with abstract concepts which are hard to manipulate. Therefore we are inclined to restrict ourselves to simple concepts. But this is not the panacea because of the difficulty to instantiate them. In this respect the number of papers on program analysis which incorrectly claim not to be amenable to abstract interpretation is impressive (see [20] for an example among many others). A user of a design method would also like to have all its possible applications and developments explicitly stated. Otherwise the formal design method is thought not to be applicable. So in complement to theoretical work on program analysis design methodologies, it seems essential to provide numerous examples and to implement program analyzers which can serve as examples of application of the theory.

3. On the Design of a Program Analyzer

3.1 Design Methodology Versus A Posteriori Verification

An important aspect of a program analysis design methodology is its capacity to serve as a guideline for the effective design of a specific program analyzer. In particular the abstraction of a semantics can be designed to inherit its structure, which can be a quite helpful guideline (in addition to the advantage of correctness by construction) if the semantics is adequately chosen. This is in contrast to a posteriori soundness/correctness verification, which offers no help in the design of the analyzer (except maybe that it has to be redone when proved erroneous). Besides design guidance, the abstraction of a semantics offers a synthetic understanding of the analyzer (see e.g. [27] as opposed to an impenetrable set of rules, see e.g. [28]).

3.2 Separation of Concerns

The separation between:
- The specification of the program properties to be determined (e.g. by a semantics and an abstraction function);
- The presentation of the specification (using fixpoints, constraints, inference system, etc.);
- The form of questions for interfacing the analyzer (complete or partial (demand-driven) abstract properties, etc.)
- The abstract semantic domains/algebra which are used in the analysis;
- The resolution algorithms (iterative fixpoint computation, direct resolution by elimination, etc.)

leads to clearly separated problems which can studied independently.

3.3 Hierarchies of Properties and Layered Program Analyzer Specification

The specification of a program analyzer can be decomposed at various levels of successive abstractions. For example the first level of abstraction can be chosen as a property class such as invariance/safety, parameterized by an abstraction of invariants. A second level can be an abstraction from relational to attribute independent properties, parameterized by an abstraction of a set of values. Such layered designs lead to reusable specifications. This also leads to parameterized abstract interpreters (see a.o. [44]) which can be instantiated with specific properties presented as parameterized abstract algebras and implemented as composable and reusable libraries corresponding to many particular domains of abstract properties.

3.4 Modular Implementation of Abstract Algebras

Nevertheless, any particular program analyzer must use a specific domain of properties. There is presently no best way to determine an optimal abstract algebra offering the best cost-precision compromise. Consequently it is very useful to design and implement application independent abstract algebras for reusability and experimentation. An important aspect of such implementations is to offer means for controlling cost versus precision, through e.g. hierarchies of widenings (offering e.g. a dynamic compromise between monovariant and polyvariant procedural analysis [11, 1, 2, 4]).

3.5 Composition of Abstract Domains

The next step is to be able to compose simple domains into complex ones and reciprocally decompose complex domains into the composition of simpler ones. Semantic [12, 13] and more syntactic [34] domain compositions have been studied. Semantic decomposition is starting to be studied, mainly with simplification objectives [29]. It remains to extend to the implementation level e.g. by a (non-trivial) composition of library modules.

3.6 Linguistic Interface

As an alternative to the use of a programming language dependent intermediate language as a basis for a generic analyzer, we might explore a domain dependent intermediate language. For example with an equation based presentation of the analysis, one can design a domain-dependent intermediate language for writing these equations (together with a separate solver and query mechanism). The design of an analyzer for a specific language is then essentially a translator from the program to equations [20]. The design of linguistic interfaces to library modules and the combination of analyses in this framework are not obvious.

3.7 Resolution Algorithms

By resolution algorithms we understand fixpoint computation, constraint solving or theorem proving depending upon the equational, constraint or inference-based presentation of the analysis [19]. Few general algorithms exist (such as asynchronous/chaotic iterations with convergence acceleration by widening [10, 11]) and more work is needed e.g. to take specific forms of equations into account (e.g. [4, 54]), to investigate efficient implementations such as parallel resolution methods or direct elimination, non-iterative resolution methods (e.g. [55]) and to abstract wide-scope methods from particular algorithms (such as e.g. recognizing the use of widenings in type inference [49]).

3.8 Predictability, User Interface

It is often argued that inference based analysis has the advantage of predictability. This is true, in principle, since the user can always apply the rules e.g. to check the inferred type. Our teaching experience of languages such as ML shows however that the rules offer a few surprises. For example the most general type of a polymorphic function changes when its definition is textually moved from one part of the program to another or even worse when the same program is compiled with different versions of the compiler. So the so-called predictability is not true in practice since the inference algorithm effectively used in the compiler is not available to the user in simple form.

More generally user interfaces (e.g. as needed in abstract debugging [3]) should be better worked out to provide simple annotations and user interaction. There are several difficulties however. The first, well-known to compiler designers, is that the analyzer manipulates objects which are not available at the source language level (e.g. elements profoundly buried in the stack cannot be described by a Hoare logic in Algol-like languages). The second is that, for efficiency reasons, resolution algorithms may introduce further approximations which may confuse users, as is the case with type inference.

3.9 Separate Program Analysis

An important problem in program analysis is separate analysis of program modules. Classical solutions range from ignorance of the interaction with the environment (and may be signaling that the analysis might be incorrect), worst case assumptions, user provided assertions on module interfaces, and the use of relational abstract algebras (allowing for a relation to be established between unknown objects from the environment and known internal objects of the program module). The problem will not be solved until further progress is made in separate semantic specification methods, adequate abstract algebras and user interfaces.

3.10 Complexity and Quality Compromise

Complexity issues are central to program analysis. We do not think here to an analyzer performing a program cost analysis (an interesting application of program analysis [24]) but more specifically to the analysis of the complexity of a program analyzer. This is much more difficult than classical program concrete complexity analysis (order of the mean/maximum execution time as a function of the distribution/size of input data) as one must also consider the *quality* of the output. This notion of quality has still to be worked out, and this is difficult! Up to now only experimental methods are available.

3.11 Automatic Design

Pushing the idea of semantics based design a little further, the abstraction of a semantics is a guide which is precise enough to lead to (semi-)automation. For having computed by hand numerous abstract interpretations, we have observed that for a given language, the formal construction of an abstract semantics/analyzer specification has always the same structure. Therefore we think that by providing a proof strategy and a formal specification of an abstraction it should be possible to program a prover to generate automatically an abstract specification from a concrete one. This approach would also be a way for theoreticians to incorporate their know-how into generic software.

4. Conclusion

In conclusion, we think that the idea of semantic approximation is general enough to be investigated in other contexts than the strict application to program analysis and should attract the attention of researchers in collateral fields such as semantics, type systems, proof methods, program transformation, implementation. Cross-fertilization with fields such as artificial intelligence, concurrency, constraint programming, databases, formal methods and model checking, parallel and distributed computation, object-oriented programming, real time and software quality, which have problems and solutions to offer, would be very fruitful.

References

[1] F. Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In P. Deransart & J. Małuszyński, eds., *Proc. Int. Work. PLILP '90*, Linköping SWE, 20–22 Aug. 1990, LNCS 456, pp. 307–323. Springer-Verlag, 1990.

[2] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Func. Prog.*, 2(4), 1992.

[3] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. PLDI*, pp. 46–55. ACM Press, 1993.

[4] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjø rner, M. Broy, & I. Pottosin, eds., *Proc. FMPA*, Academgorodok, Novosibirsk, RUS, LNCS 735, pp. 128–141. Springer-Verlag, June 28–July 2, 1993.

[5] P. Cousot. Semantic foundations of program analysis. In S. Muchnick & N. Jones, eds., *Program Flow Analysis: Theory and Applications*, ch. 10, pp. 303–342. Prentice-Hall, 1981.

[6] P. Cousot. Abstract model checking, invited lecture. In *7th Int. Conf. CAV '95*, Liège, BEL, 5 July 1995.

[7] P. Cousot. Constructing a hierarchy of semantics by abstract interpretation. In *Workshop on Logic, Domains and Programming Languages*, Darmstadt, GER, May 24–27 1995.

[8] P. Cousot & R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pp. 106–130. Dunod, 1976.

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

[10] P. Cousot & R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, N.Y., SIGPLAN Notices 12(8):1–12, 1977.

[11] P. Cousot & R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, ed., *IFIP Conference on Formal Description of Programming Concepts*, St-Andrews, N.B., CAN, pp. 237–277. North-Holland, 1977.

[12] P. Cousot & R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pp. 269–282, San Antonio, Texas, 1979. ACM Press.

[13] P. Cousot & R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 13(2–3):103–179, 1992.

[14] P. Cousot & R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, Aug. 1992.

[15] P. Cousot & R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe & M. Wirsing, eds., *Proc. Int. Work. PLILP '92*, Leuven, BEL, 13–17 Aug. 1992, LNCS 631, pp. 269–295. Springer-Verlag, 1992.

[16] P. Cousot & R. Cousot. Inductive definitions, semantics and abstract interpretation. In *19th POPL*, pp. 83–94, Albuquerque, N.M., 1992. ACM Press.

[17] P. Cousot & R. Cousot. Galois connection based abstract interpretations for strictness analysis, invited paper. In D. Bjø rner, M. Broy, & I. Pottosin, eds., *Proc. FMPA*, Academgorodok, Novosibirsk, RUS, LNCS 735, pp. 98–127. Springer-Verlag, June 28–July 2, 1993.

[18] P. Cousot & 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, FRA, pp. 95–112. IEEE Comp. Soc. Press, 16–19 May 1994.

[19] P. Cousot & R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. In P. Wolper, ed., *Proc. 7^{th} Int. Conf. CAV '95*, Liège, BEL, LNCS 939, pp. 293–308. Springer-Verlag, 3–5 July 1995.

[20] P. Cousot & R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7^{th} FPCA*, pp. 170–181, La Jolla, Calif., 25–28 June 1995. ACM Press.

[21] P. Cousot & N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5^{th} POPL*, pp. 84–97, Tucson, Ariz., 1978. ACM Press.

[22] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proc. PEPM '95*, La Jolla, Calif., 21–23 June 1995. ACM Press, June 1995.

[23] S. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, ed., *Advances in Logic Programming Theory*, International Schools for Computer Scientists, section 3, pp. 115–182. Clarendon Press, 1994.

[24] S. Debray & N.-W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5):826–875, 1993.

[25] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proc. 1992 ICCL*, Oakland, Calif., pp. 2–13. IEEE Comp. Soc. Press, 20–23 Apr. 1992.

[26] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. PLDI*, pp. 230–241. ACM Press, Orlando, Fla, June 1994.

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

[28] P. Fradet, R. Gaugne, & D. Le Métayer. Static detection of pointer errors: An axiomatization and a checking algorithm. In H. Riis Nielson, ed., *Proc. 6^{th} ESOP '96*, Linköping, SWE, LNCS 1058, pp. 125–140. Springer-Verlag, Apr. 22–26 1996.

[29] R. Giacobazzi & F. Ranzato. Compositional optimization of disjunctive abstract interpretations. In H. Riis Nielson, ed., *Proc. 6^{th} ESOP '96*, Linköping, SWE, LNCS 1058, pp. 141–155. SPRINGER, Apr. 22–26 1996.

[30] E. Goubault. Schedulers as abstract interpretations of higher-dimensional automata. In *Proc. PEPM '95*, La Jolla, Calif., 21–23 June 1995, pp. 134–145. ACM Press, June 1995.

[31] E. Goubault. Duration for truly-concurrent transitions. In H. Riis Nielson, ed., *Proc. 6^{th} ESOP '96*, Linköping, SWE, LNCS 1058, pp. 173–187. Springer-Verlag, Apr. 22–26 1996.

[32] E. Goubault, C. Hankin, M. van Eekelen, & E. Nocker. Abstract reduction: towards a theory via abstract interpretation. In R. Sleep, R. Plasmeijer, & M. van Eekelen, eds., *Term Graph Rewriting: Theory and Practice*, ch. 9, pp. 117–129. John Wiley & Sons, Ltd., 1993.

[33] P. Granger. Static analysis of linear congruence equalities among variables of a program. In S. Abramsky & T. Maibaum, eds., *Proc. Int. J. Conf. TAPSOFT '91*, Brighton, U.K., Volume 1 (CAAP '91), LNCS 493, pp. 169–192. Springer-Verlag, 1991.

[34] P. Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Proc. 12^{th} FST & TCS*, pp. 68–79, New Delhi, IND, 18–20 Dec. 1992, LNCS 652, 1992. Springer-Verlag.

[35] C. Gunter & D. Scott. Semantic domains. In J. van Leeuwen, ed., *Formal Models and Semantics*, vol. B of *Handbook of Theoretical Computer Science*, ch. 12, pp. 633–674. Elsevier, 1990.

[36] N. Halbwachs. About synchronous programming and abstract interpretation. In B. Le Charlier, ed., *Proc. SAS '94*, Namur BEL, 20–22 Sept. 1994, LNCS 864, pp. 179–192. Springer-Verlag, 1994.

[37] N. Halbwachs, J.-É. Proy, & P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. Le Charlier, ed., *Proc. SAS '94*, Namur BEL, 20–22 Sept. 1994, LNCS 864, pp. 223–237. Springer-Verlag, 1994.

[38] T. Jensen. Strictness analysis in logical form. In J. Hughes, ed., *Proc. 5^{th} FPCA*, LNCS 523, pp. 352–366. Springer-Verlag, Aug. 1991.

[39] T. Jensen & I. Mackie. Flow analysis in the geometry of interaction. In H. Riis Nielson, ed., *Proc. 6^{th} ESOP '96*, Linköping, SWE, LNCS 1058, pp. 188–203. Springer-Verlag, Apr. 22–26 1996.

[40] N. Jones, C. Gomard, & P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[41] N. Jones & F. Nielson. Abstract interpretation: a semantic-based tool for program analysis. In S. Abramsky, G. D.M., & M. T.S.E., eds., *Semantic modelling*, no. 4 in Handbook of Logic in Computer Science. Clarendon Press, 1995.

[42] M. Karr. Affine relationships among variables of a program. *ACTA I2*, 6:133–151, 1976.

[43] G. Kildall. A unified approach to global program optimization. In *1^{st} POPL*, pp. 194–206, Boston, Mass., Oct. 1973. ACM Press.

[44] B. Le Charlier & P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. In *Proc. 1992 ICCL*, Oakland, Calif., pp. 137–146. IEEEpress, Apr. 20–23, 1992.

[45] T. Marlowe & B. Ryder. Properties of data flow frameworks: A unified model. *Acta Inf.*, 28:121–163, 1990.

[46] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *Proc. ACM Int. Conf. on Supercomputing, ICS '92*, pp. 226–235, Washington D.C., July 1992.

[47] L. Mauborgne. Abstract interpretation using TDGs. In B. Le Charlier, ed., *Proc. SAS '94*, Namur BEL, 20–22 Sept. 1994, LNCS 864, pp. 363–379. Springer-Verlag, 1994.

[48] R. Milner. A theory of polymorphism in programming. *J. Comput. Sys. Sci.*, 17(3):348–375, Dec. 1978.

[49] B. Monsuez. Polymorphic types and widening operators. In P. Cousot, M. Falaschi, G. Filé, & A. Rauzy, eds., *Proc. 3^{rd} Int. Work. WSA '93 on Static Analysis*, Padova, ITA, LNCS 724, pp. 267–281. Springer-Verlag, 22–24 Sept. 1993.

[50] B. Monsuez. System F and abstract interpretation. In A. Mycroft, ed., *Proc. SAS '95*, Glasgow, UK, 25–27 Sept. 1995, LNCS 983, pp. 279–295. Springer-Verlag, 1995.

[51] P. Mosses. Denotational semantics. In J. van Leeuwen,

ed., *Formal Models and Semantics*, vol. B of *Handbook of Theoretical Computer Science*, ch. 11, pp. 575–631. Elsevier, 1990.

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

[53] F. Nielson. Two-level semantics and abstract interpretation. *TCS — Fund. St.*, 69:117–242, 1989.

[54] M. Rosendahl. Higher order chaotic iteration sequences. In *Proc. Int. Work. PLILP '93*, Leuven, BEL, 13–17 Aug. 1993, LNCS 714, pp. 332–345. Springer-Verlag, 1993.

[55] H. Seidel. Least solutions of equations over $\mathcal{N}$. In *Proc. ICALP '96*, LNCS 820, pp. 400–411. Springer-Verlag, 1995.

[56] F. Védrine. Binding-time analysis and strictness analysis by abstract interpretation. In A. Mycroft, ed., *Proc. SAS '95*, Glasgow UK, 25–27 Sept. 1995, LNCS 983, pp. 400–417. Springer-Verlag, 1995.

[57] P. Wadler & R. Hughes. Projections for strictness analysis. In G. Kahn, ed., *Proc.* 2^{nd} *FPCA*, LNCS 274, pp. 385–407. Springer-Verlag, Portland, Ore., Sept. 1987.

[58] R. Wilhelm & D. Maurer. *Compiler Design.* Addison-Wesley, 1995.

PS: The references are obviously non exhaustive.