Abstract Interpretation Based Formal Methods and Future Challenges

Patrick Cousot

cole normale sup rieure, D partement d'informatique,
45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr http://www.di.ens.fr/~cousot/

Abstract. In order to contribute to the solution of the software reliability problem, tools have been designed to analyze statically the run-time behavior of programs. Because the correctness problem is undecidable, some form of approximation is needed. The purpose of *abstract interpretation* is to formalize this idea of *approximation*. We illustrate informally the application of abstraction to the *semantics* of programming languages as well as to *static program analysis*. The main point is that in order to reason or compute about a complex system, some information must be lost, that is the observation of executions must be either partial or at a high level of abstraction.
A few challenges for static program analysis by abstract interpretation are finally briefly discussed.
The electronic version of this paper includes a comparison with other formal methods: *typing*, *model-checking* and *deductive methods*.

1 Introductory Motivations

The evolution of hardware by a factor of 10^6 over the past 25 years has lead to the explosion of the size of programs in similar proportions. The scope of application of very large programs (from 1 to 40 millions of lines) is likely to widen rapidly in the next decade. Such big programs will have to be designed at a reasonable cost and then modified and maintained during their lifetime (which is often over 20 years). The size and efficiency of the programming and maintenance teams in charge of their design and follow-up cannot grow in similar proportions. At a not so uncommon (and often optimistic) rate of one bug per thousand lines such huge programs might rapidly become hardly manageable in particular for safety critical systems. Therefore in the next 10 years, the *software reliability problem* is likely to become a major concern and challenge to modern highly computer-dependent societies.

In the past decade a lot of progress has been made both on *thinking/methodological tools* (to enhance the human intellectual ability) to cope with complex software systems and *mechanical tools* (using the computer) to help the programmer to reason about programs.

Mechanical tools for computer aided program verification started by executing or simulating the program in as much as possible environments. However

debugging of compiled code or simulation of a model of the source program hardly scale up and often offer a low coverage of dynamic program behavior.

Formal program verification methods attempt to mechanically prove that program execution is correct in all specified environments. This includes *deductive methods*, *model checking*, *program typing* and *static program analysis*.

Since program verification is undecidable, computer aided program verification methods are all partial or incomplete. The undecidability or complexity is always solved by using some form of *approximation*. This means that the mechanical tool will sometimes suffer from practical time and space complexity limitations, rely on finiteness hypotheses or provide only semi-algorithms, require user interaction or be able to consider restricted forms of specifications or programs only. The mechanical program verification tools are all quite similar and essentially differ in their choices regarding the approximations which have to be done in order to cope with undecidability or complexity. The purpose of *abstract interpretation* is to formalize this notion of approximation in a unified framework [10, 17].

2 Abstract Interpretation

Since program verification deals with properties, that is sets (of objects with these properties), abstract interpretation can be formulated in an application independent setting, as a theory for approximating sets and set operations as considered in set (or category) theory, including inductive definitions [24]. A more restricted understanding of abstract interpretation is to view it as a theory of approximation of the behavior of dynamic discrete systems (e.g. the formal semantics of programs or a communication protocol specification). Since such behaviors can be characterized by fixpoints (e.g. corresponding to iteration), an essential part of the theory provides constructive and effective methods for fixpoint approximation and checking by abstraction [19, 23].

2.1 Fixpoint Semantics

The *semantics of a programming language* defines the semantics of any program written in this language. The *semantics of a program* provides a formal mathematical model of all possible behaviors of a computer system executing this program in interaction with any possible environment. In the following we will try to explain informally why the semantics of a program can be defined as the solution of a fixpoint equation. Then, in order to compare semantics, we will show that all the semantics of a program can be organized in a hierarchy by abstraction. By observing computations at different levels of abstraction, one can approximate fixpoints hence organize the semantics of a program in a lattice [15].

2.2 Trace Semantics

Our finer grain of observation of program execution, that is the most pre-

cise of the semantics that we will consider, is that of a *trace semantics* [15, 19]. An execution of a program for a given specific interaction with its environment is a sequence of states, observed at discrete intervals of time, starting from an initial state, then moving from one state to the next state by executing an atomic program step or transition and either ending in a final regular or erroneous state or non terminating, in which case the trace is infinite (see Fig. 1).

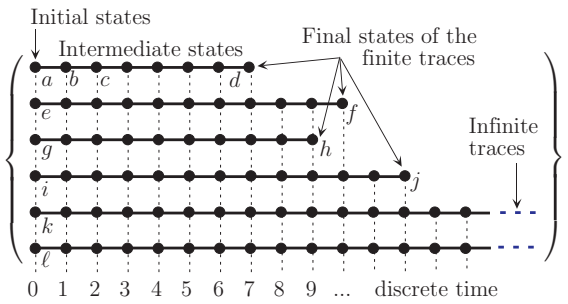


Fig. 1. Examples of Computation Traces

2.3 Least Fixpoint Trace Semantics

Introducing the *computational partial ordering* [15], we define the trace semantics in fixpoint form [15], as the least solution of an equation of the form $X = \mathcal{F}(X)$ where X ranges over sets of finite and infinite traces.

More precisely, let **Behaviors** be the set of execution traces of a program, possibly starting in any state. We denote by $\mathbf{Behaviors}^+$ the subset of finite traces and by $\mathbf{Behaviors}^\infty$ the subset of infinite traces.

A finite trace $\overset{a}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\overset{z}{\bullet}$ in $\mathbf{Behaviors}^+$ is either reduced to a final state (in which case there is no possible transition from state $\overset{a}{\bullet} = \overset{z}{\bullet}$) or the initial state $\overset{a}{\bullet}$ is not final and the trace consists of a first computation step $\overset{a}{\bullet}\!\!-\!\!-\overset{b}{\bullet}$ after which, from the intermediate state $\overset{b}{\bullet}$, the execution goes on with the shorter finite trace $\overset{b}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\overset{z}{\bullet}$ ending in the final state $\overset{z}{\bullet}$. The finite traces are therefore all well defined by induction on their length.

An infinite trace $\overset{a}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!\cdots$ in $\mathbf{Behaviors}^\infty$ starts with a first computation step $\overset{a}{\bullet}\!\!-\!\!-\overset{b}{\bullet}$ after which, from the intermediate state $\overset{b}{\bullet}$, the execution goes on with an infinite trace $\overset{b}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!\cdots$ starting from the intermediate state $\overset{b}{\bullet}$. These remarks and $\mathbf{Behaviors} = \mathbf{Behaviors}^+ \cup \mathbf{Behaviors}^\infty$ lead to the following fixpoint equation:

$$
\begin{aligned}
\mathbf{Behaviors} = \ & \{\overset{a}{\bullet} \mid \overset{a}{\bullet} \text{ is a final state}\} \\
& \cup \{\overset{a}{\bullet}\!\!-\!\!-\overset{b}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\overset{z}{\bullet} \mid \overset{a}{\bullet}\!\!-\!\!-\overset{b}{\bullet} \text{ is an elementary step } \&\ \overset{b}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\overset{z}{\bullet} \in \mathbf{Behaviors}^+\} \\
& \cup \{\overset{a}{\bullet}\!\!-\!\!-\overset{b}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!\cdots \mid \overset{a}{\bullet}\!\!-\!\!-\overset{b}{\bullet} \text{ is an elementary step } \&\ \overset{b}{\bullet}\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!\cdots \in \mathbf{Behaviors}^\infty\}
\end{aligned}
$$

In general, the equation has multiple solutions. For example if there is only one non-final state $\overset{a}{\bullet}$ and only possible elementary step $\overset{a}{\bullet}\!\!-\!\!-\overset{a}{\bullet}$ then the equation is

Behaviors $= \{\overset{a}{\bullet}\!\!-\!\!\!-\overset{a}{\bullet}\!\!-\!\!\!-\ldots\!-\!\!\!-\ldots \mid \overset{a}{\bullet}\!\!-\!\!\!-\ldots\!-\!\!\!-\ldots \in \textbf{Behaviors}\}$. One solution is $\{\overset{a}{\bullet}\!\!-\!\!\!-\overset{a}{\bullet}\!\!-\!\!\!-\overset{a}{\bullet}\!\!-\!\!\!-\overset{a}{\bullet}\!\!-\!\!\!-\ldots\!-\!\!\!-\ldots\}$ but another one is the empty set $\emptyset$. Therefore, we choose the least solution for the *computational partial ordering* [15]:

« More finite traces & less infinite traces » .

2.4 Abstractions & Abstract Domains

A programming language semantics is more or less precise according to the considered observation level of program execution. This intuitive idea can be formalized by *Abstract interpretation* [15] and applied to different languages, including for proof methods.

The *theory of abstract interpretation* formalizes this notion of approximation and abstraction in a mathematical setting which is independent of particular applications. In particular, abstractions must be provided for all mathematical constructions used in semantic definitions of programming and specification languages [19, 23].

An *abstract domain* is an abstraction of the concrete semantics in the form of *abstract properties* (approximating the concrete properties **Behaviors**) and *abstract operations* (including abstractions of the concrete approximation and computational partial orderings, an approximation of the concrete fixpoint transformer $\mathcal{F}$, etc.). Abstract domains for complex approximations of designed by composing abstract domains for simpler components [19], see Sec. 2.10.

If the approximation is coarse enough, the abstraction of a concrete semantics can lead to an abstract semantics which is less precise, but is *effectively computable* by a computer. By effective computation of the abstract semantics, the computer is able to analyze the behavior of programs and of software before and without executing them [16]. *Abstract interpretation algorithms* provide approximate methods for computing this abstract semantics. The most important algorithms in abstract interpretation are those providing effective methods for the exact or approximate iterative resolution of fixpoint equations [17].

We will first illustrate formal and effective abstractions for sets. Then we will show that such abstractions can be lifted to functions and finally to fixpoints.

The abstraction idea and its formalization are equally applicable in other areas of computer science such as artificial intelligence e.g. for intelligent planning, proof checking, automated deduction, theorem proving, etc.

2.5 Hierarchy of Abstractions

As shown in Fig. 2 (from [15], where **Behaviors**, denoted τ^∞ for short, is the lattice infimum), all abstractions of a semantics can be organized in a lattice (which is part of the lattice of abstract interpretations introduced in [19]). The *approximation partial ordering* of this lattice formally corresponds to logical implication, intuitively to the idea that one semantics is more precise than another one.

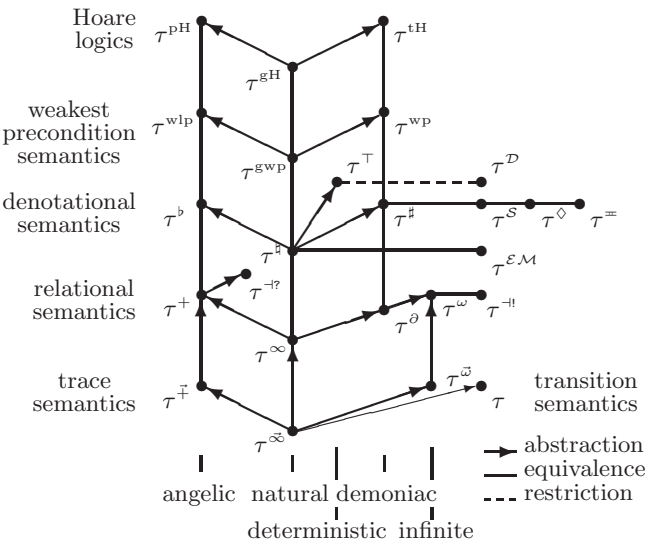


Fig. 2. The Hierarchy of Semantics

Fig. 3 illustrates the derivation of a *relational semantics* (denoted τ^{∞} in Fig. 2) from a trace semantics (denoted $\tau^{\infty\!\!\!\!\rightarrow}$ in Fig. 2). The abstraction α_r from trace to relational semantics consists in replacing the finite traces $\overset{a}{\bullet}\!\!-\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!-\overset{z}{\bullet}$ by the pair $\langle a, z\rangle$ of the initial and final states. The infinite traces $\overset{a}{\bullet}\!\!-\!\!-\!\overset{b}{\bullet}\!\!-\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!-\!\!\cdots$ are replaced by the pair $\langle a, \bot\rangle$ where the symbol $\bot$ denotes non-termination. Therefore the abstraction is:

$$\alpha_r(X) = \{\langle a, z\rangle \mid \overset{a}{\bullet}\!\!-\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!-\overset{z}{\bullet} \in X\} \cup \{\langle a, \bot\rangle \mid \overset{a}{\bullet}\!\!-\!\!-\!\overset{b}{\bullet}\!\!-\!\!-\!\!-\!\!\cdots\!\!-\!\!-\!\!-\!\!\cdots \in X\} .$$

The *denotational semantics* (denoted $\tau^{\sharp}$ in Fig. 2) is the isomorphic representation of a relation by its right-image:

$$\alpha_d(R) = \boldsymbol{\lambda}\, a \cdot \{x \mid \langle a, x\rangle \in R\}.$$

The abstraction from relational to *big-step operational or natural semantics* (denoted τ^{+} in Fig. 2) simply consists in forgetting everything about non-termination, so $\alpha_n(R) = \{\langle a, x\rangle \in R \mid x \neq \bot\}$, as illustrated in Fig. 3.

A non comparable abstraction consists in collecting the set of initial and final states as well as all transitions $\langle x, y\rangle$ appearing along some finite or infinite trace $\overset{a}{\bullet}\!\!-\!\!-\!\!-\!\!\cdots\overset{x}{\bullet}\!\!-\!\!-\!\!-\overset{y}{\bullet}\cdots$ of the trace semantics. One gets the *small-step operational or transition semantics* (denoted τ in Fig. 2 and also called Kripke structure in modal logic) as illustrated in Fig. 4.

A further abstraction consists in collecting all states appearing along some finite or infinite trace as illustrated in Fig. 5. This is the *partial correctness semantics* or the *static/collecting semantics* for proving invariance properties of programs.

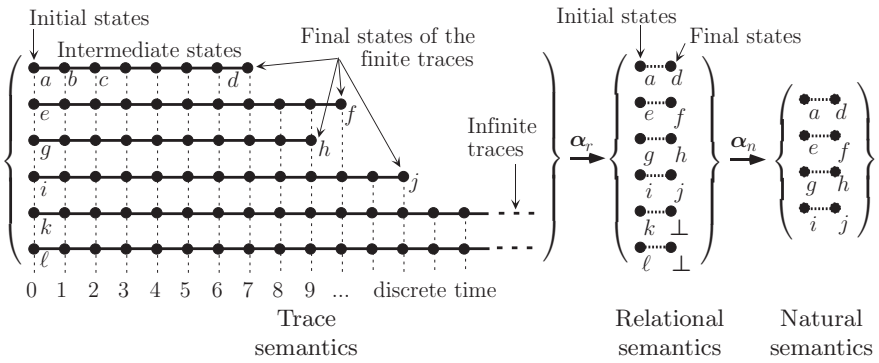


Fig. 3. Abstraction from Trace to Relational and Natural Semantics

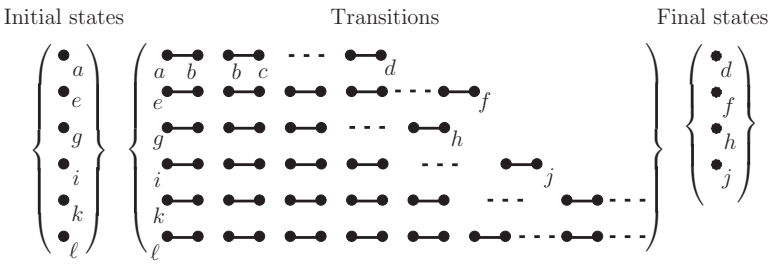


Fig. 4. Transition Semantics

All abstractions considered in this paper are "from above" so that the abstract semantics describes a superset or logical consequence of the concrete semantics. Abstractions "from below" are dual and consider a subset of the concrete semantics. An example of approximation "from below" is provided by debugging techniques which consider a subset of the possible program executions or by existential checking where one wants to prove the existence of an execution trace prefix fulfilling some given specification. In order to avoid repeating two times dual concepts and as we do usually, we only consider approximations "from above", knowing that approximations "from below" can be easily derived by applying the duality principle (as found e.g. in lattice theory).

2.6 Effective Abstractions

Numerical Abstractions Assume that a program has two integer variables `X` and `Y`. The trace semantics of the program (Fig. 1) can be abstracted in the static/collecting semantics (Fig. 5). A further abstraction consists in forgetting in a state all but the values x and y of variables `X` and `Y`. In this way the trace semantics is abstracted to a set of points (pairs of values), as illustrated in the plane by Fig. 6(a).

We now illustrate informally a number of effective abstractions of an [in]finite set of points.

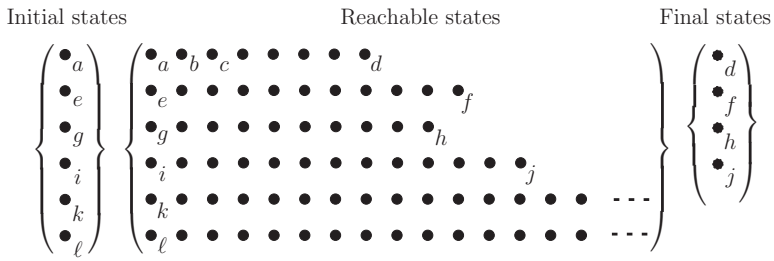
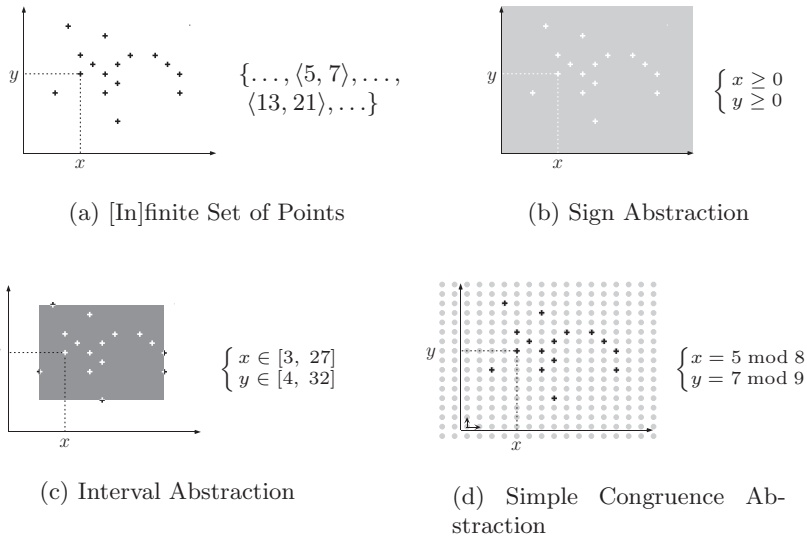


Fig. 5. Static / Collecting / Partial Correctness Semantics



(a) [In]finite Set of Points

(b) Sign Abstraction

(c) Interval Abstraction

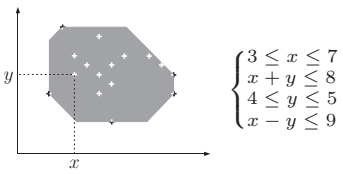
(d) Simple Congruence Abstraction

Fig. 6. Non-relational Abstractions

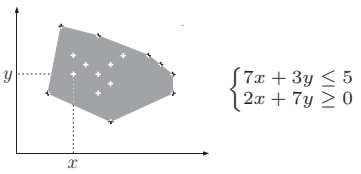
Non-relational Abstractions The non-relational, attribute independent or cartesian abstractions [19, example 6.2.0.2] consists in ignoring the possible relationships between the values of the `X` and `Y` variables. So a set of pairs is approximated through projection by a pair of sets. Each such set may still be infinite and in general not exactly computer representable. Further abstractions are therefore needed.

The *sign abstraction* [19] illustrated in Fig. 6(b) consists in replacing integers by their sign thus ignoring their absolute value. The *interval abstraction* [16] illustrated in Fig. 6(c) is more precise since it approximates a set of integers by it minimal and maximal values (including $-\infty$ and $+\infty$ as well as the empty set if necessary).

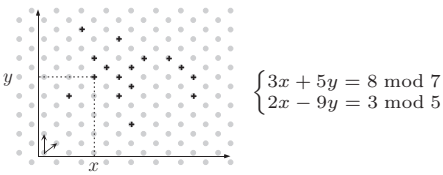
The *congruence abstraction* [37] (generalizing the parity abstraction [19]) is not comparable, as illustrated in Fig. 6(d).



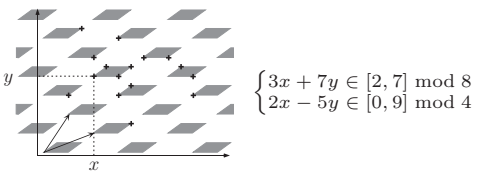
(a) Octagonal Abstraction



(b) Polyhedral Abstraction



(c) Relational Congruence Abstraction



(d) Trapezoidal Congruence Abstraction

Fig. 7. Relational Abstractions

Relational Abstractions Relational abstractions are more precise than non relational ones in that some of the relationships between values of the program states are preserved by the abstraction.

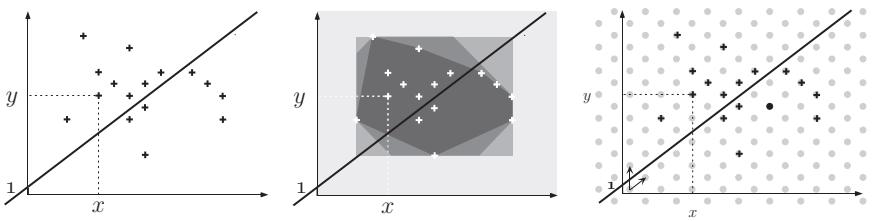
For example the *polyhedral abstraction* [29] illustrated in Fig. 7(b) approximates a set of integers by its convex hull. Only non-linear relationships between the values of the program variables are forgotten.

The use of an *octagonal abstraction* illustrated in Fig. 7(a) is less precise since only some shapes of polyhedra are retained or equivalently only linear relations between any two variables are considered with coefficients +1 or -1 (of the form $\pm x \pm y \leq c$ where c is an integer constant).

A non comparable relational abstraction is the *linear congruence abstraction* [38] illustrated in Fig. 7(c).

A combination of non-relational dense approximations (like intervals) and relational sparse approximations (like congruences) is the *trapezoidal linear congruence abstraction* [47] as illustrated in Fig. 7(d).

Symbolic Abstractions Most structures manipulated by programs are *symbolic structures* such as control structures (call graphs), data structures (search trees, pointers [32, 33, 53, 58]), communication structures (distributed & mobile programs [35, 40, 57]), etc. It is very difficult to find compact and expressive abstractions of such sets of objects (sets of languages, sets of automata, sets of trees or graphs, etc.). For example B chi automata or automata on trees are very expressive but algorithmically expensive.



(a) yes (b) unkown (c) yes

Fig. 8. Is `1/(X+1-Y)` well-defined?

A compromise between semantic expressivity and algorithmic efficiency was recently introduced by [48] using *Binary Decision Graphs* and *Tree Schemata* to abstract infinite sets of infinite trees.

2.7 Information Loss

Any abstraction introduces some loss of information. For example the abstraction of the trace semantics into relational or denotational semantics loses all information on the computation cost since all intermediate steps in the execution are removed.

All answers given by the abstract semantics are always correct with respect to the concrete semantics. For example, if termination is proved using the relational semantics then there is no execution abstracted to $\langle a, \bot \rangle$, so there is no infinite trace $\overset{a}{\bullet}\!\!-\!\!-\overset{b}{\bullet}\!\!-\!\!-\ldots\!\!-\!\!-\ldots$ in the trace semantics, whence non termination is impossible when starting execution in initial state a.

However, because of the information loss, not all questions can be definitely answered with the abstract semantics. For example, the natural semantics cannot answer questions about termination as can be done with the relational or denotational semantics. These semantics cannot answer questions about concrete computation costs.

The more concrete is the semantics, the more questions it can answer. The more abstract semantics are simpler. Non comparable abstract semantics (such as intervals and congruences) answer non comparable sets of questions.

To illustrate the loss of information, let us consider the problem of deciding whether the operation `1/(X+1-Y)` appearing in a program is always well defined at run-time. The answer can certainly be given by the concrete semantics since it has no point on the line $x + 1 - y = 0$, as shown in Fig. 8(a).

In practice the concrete abstraction is not computable so it is hardly usable in a useful effective tool. The dense abstractions that we have considered are too approximate as is illustrated in Fig. 8(b).

However the answer is positive when using the relational congruence abstraction, as shown in Fig. 8(c).

2.8 Function Abstraction

We now show how the abstraction of complex mathematical objects used in the semantics of programming or specification languages can be defined by composing abstractions of simpler mathematical structures.

For example knowing abstractions of the parameter and result of a monotonic function on sets, a function F can be abstracted into an abstract function $F^\sharp$ as illustrated in Fig. 9 [19]. Mathematically, $F^\sharp$ takes its parameter x in the abstract domain. Let $\gamma(x)$ be the corresponding concrete set (γ is the adjoined, intuitively the inverse of the abstraction function α). The function F can be applied to get the concrete result $\circ F \circ \gamma(x)$. The abstraction function α can then be applied to approximate the result $F^\sharp(x) = \alpha \circ F \circ \gamma(x)$.
In general, neither F, α nor γ are computable even though the abstraction α may be effective. So we have got a formal specification of the abstract function $F^\sharp$ and an algorithm has to be found for an effective implementation.

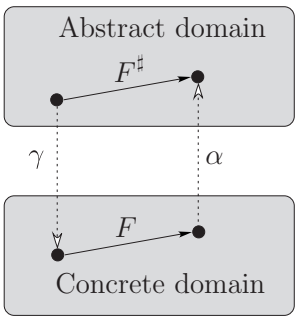


Fig. 9. Function Abstraction

2.9 Fixpoint Abstraction

A fixpoint of a function F can often be obtained as the limit of the iterations of F from a given initial value $\bot$. In this case the abstraction of the fixpoint can often be obtained as the abstract limit of the iteration of the abstraction $F^\sharp$ of F starting from the abstraction $\alpha(\bot)$ of the initial value $\bot$. The basic result is that the concretization of the abstract fixpoint is related to the concrete fixpoint by the approximation relation expressing the soundness of the abstraction [19]. This is illustrated in Fig. 10.

Often states have some finite component (e.g. a program counter) which can be used to partition into fixpoint system of equations by projection along that component. Then *chaotic* [18] and *asynchronous iteration strategies* [10] can be used to solve the equations iteratively. Various efficient iteration strategies have been studied, including ones taking particular properties of abstractions into account and others to speed up the convergence of the iterates [26].

2.10 Composing Abstractions

Abstractions hence abstract interpreters for static program analysis can be designed compositionally by stepwise abstraction, combination or refinement [36, 13].

An example of stepwise abstraction is the functional abstraction of Sec. 2.8. The abstraction of a function is parameterized by abstractions for the function

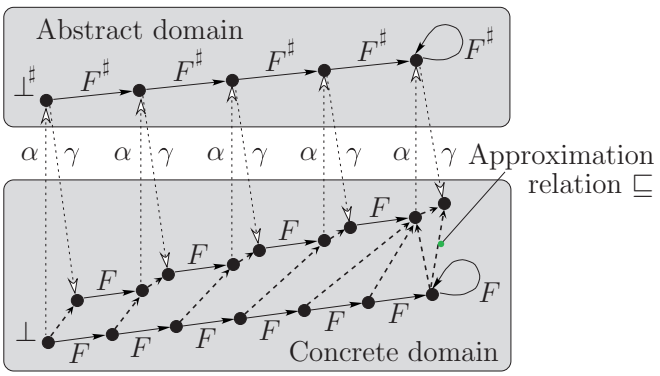


Fig. 10. Fixpoint Abstraction $\mathrm{lfp}\, F \sqsubseteq \gamma(\mathrm{lfp}\, F^\sharp)$

parameters and the function result which can be chosen later in the modular design of the abstract interpreter.

An example of abstraction combination is the *reduced product* of two abstractions [19] which is the most abstract abstraction more precise than these two abstractions or the *reduce cardinal power* [19] generalizing case analysis. Such combination of abstract domains can be implemented as parameterized modules in static analyzer generators (e.g. [45]) so as to partially automate the design of expressive analyses from simpler ones.

An example of refinement is the *disjunctive completion* [19] which completes an abstract domain by adding concrete disjunctions missing in the abstract domain. Another example of abstract domain refinement is the *complementation* [8] adding concrete negations missing in the abstract domain.

2.11 Sound and Complete Abstractions

Abstract interpretation theory has mainly been concerned with the *soundness* of the abstract semantics/interpreter, relative to which questions can be answered correctly despite the loss of information [17]. Soundness is essential in practice and leads to a formal design method [19].

However *completeness*, relative to the formalization of the loss of information in a controlled way so as to answer a given set of questions, has also been intensively studied [19, 36], including in the context of model checking [14].

In practice complete abstractions, including a most abstract one, always exist to check that a given program semantics satisfies a given specification. Moreover any given abstraction can be refined to a complete one. Nevertheless this approach has severe practical limitations since, in general, the design of such complete abstractions or the refinement of a given one is logically equivalent to the design of an inductive argument for the formal proof that the given program satisfies the given specification, while the soundness proof of this abstraction logically amounts to checking the inductive verification conditions or

proof obligations of this formal proof [14]. Such proofs can hardly be fully automated hence human interaction is unavoidable. Moreover the whole process has to be repeated each time the program or specification is modified.

Instead of considering such strong specifications for a given specific program, the objective of static program analysis is to consider (often predefined) specifications and all possible programs. The practical problem in static program analysis is therefore to design useful abstractions which are computable for all programs and expressive enough to yield interesting information for most programs.

3 Static Program Analysis

Static program analysis is the automatic static determination of dynamic runtime properties of programs.

3.1 Foundational Ideas of Static Program Analysis

Given a program and a specification, a program analyzer will check if the program semantics satisfies the specification (Fig. 11). In case of failure, the analyzer will provide hints to understand the origin of errors (e.g. by a backward analysis providing necessary conditions to be satisfied by counter-examples).

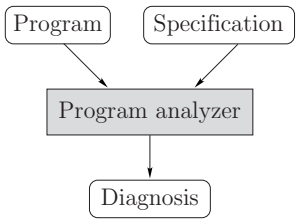


Fig. 11. Program Analysis

The principle of the analysis is to compute an approximate semantics of the program in order to check a given specification. Abstract interpretation is used to derive, from a standard semantics, the approximate and computable abstract semantics. The derivation can often be done by composing standard abstractions to fit a particular kind of information which has to be discovered about program execution. This derivation is itself not (fully) mechanizable but *static analyzer generators* such as PAG [46] and others can provide generic abstractions to be composed with problem specific ones.

In practice, the program analyzer contains a *generator* reading the program text and producing equations or constraints whose solution is a computer representation of the program abstract semantics. A *solver* is then used to solve these abstract equations/constraints. A popular resolution method is to use iteration. Of the numerical abstractions considered in Sec. 2.6, only the sign and simple congruence abstractions ensure the finite convergence of the iterates. If the limit of the iterates is inexistent (which may be the case e.g. for the polyhedral abstraction) or it is reached after infinitely many iteration steps (e.g. interval and octagonal abstractions), the convergence may have to be ensured and/or accelerated using a *widening* to over estimate the solution in finitely many steps followed by a *narrowing* to improve it [10, 17, 26].

In *abstract compilation*, the generator and solver are directly compiled into a program which directly yields the approximate solution.

This solution is an approximation of the abstract semantics which is then used by a *diagnoser* to check the specification. Because of the loss of information, the diagnosis is always of the form "yes", "no", "unknown" or "irrelevant" (e.g. a safety specification for unreachable code). The general structure of program analyzers is illustrated in Fig. 12. Besides diagnosis, static program analysis is also used for other applications in which case the diagnoser is replaced by an *optimiser* (for compile-time optimization), a *program transformer* (for partial evaluation [43]), etc.

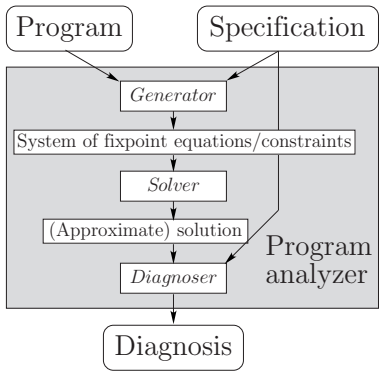


Fig. 12. Principle of Program Analysis

3.2 Shortcomings of Static Program Analysis

Static program analysis can be used for large programs (e.g. 220,000 lines of C) without user interaction. The abstractions are chosen to be of wide scope without specialization to a particular program. Abstract algebras can be designed and implemented into libraries which are reusable for different programming languages. The objective is to discover invariants that are likely to appear in many programs so that the abstraction must be widely reusable for the program analyzer to be of economic interest.

The drawback of this general scope is that the considered abstract specifications and properties are often simple, mainly concerning elementary safety properties such as absence of run-time errors. For example non-linear abstractions of sets of points are very difficult and very few mathematical results are of practical interest and directly applicable to program analysis. Checking termination and similar liveness properties is trivial with finite state systems, at least from a theoretical if not algorithmic point of view (e.g. finding loops in finite graphs). The same problem is much more difficult for infinite state systems because of fairness [48] or of potentially infinite data structures (as considered e.g. in partial evaluation) which do not amount to finite cycles so that termination or inevitability proofs require the discovery of *variant functions on well-founded sets* which is very difficult in full generality.

Even when considering restricted simple abstract properties, the semantics of real-life programming languages is very complex (recursion, concurrency, modularity, etc.) whence so is the corresponding abstract interpreter. The abstraction of this semantics, hence the design of the analyzer is mostly manual (and beyond the ability of casual programmers or theorem provers) whence costly. The considered abstractions must have a large scope of application and must be easily reusable to be of economic interest.

From a user point of view, the results of the analysis have to be presented in a simple way (for example by pointing at errors only or by providing abstract counter-examples, or less frequently concrete ones). Experience shows that the cases of uncertainty represent 5 to 10 % of the possible cases. They must be handled with other empirical or formal methods (including more refined abstract interpretations).

3.3 Applications of Static Program Analysis

Among the numerous applications of static program analysis, let us cite *data flow analysis* [52, 28]; *program optimization and transformation* (including partial evaluation and program specialization [43] and data dependence analysis for the parallelisation of sequential languages); *set-based analysis* [27]; *type inference* [12] (including undecidable systems and soft typing); verification of *reactive* [39, 42], *real-time* and *(linear) hybrid systems* including state space reduction; *cryptographic protocol* analysis; *abstract model-checking* of infinite systems [28]; *abstract debugging*, testing and verification ; *cache and pipeline behavior prediction* [34]; *probabilistic analysis* [49]; *communication topology analysis* for mobile/distributed code [35, 40, 57]; *automatic differentiation* of numerical programs; *abstract simulation* of temporal specifications; Semantic tattooing/*watermarking* of software [54]; etc.

Static program analysis has been intensively studied for a variety of programming languages including procedural languages (e.g. for alias and pointer analysis [32, 33, 53, 58]), functional languages (e.g. for binding time [56], strictness [4, 50] and comportment analysis [25], exception analysis [59]), parallel functional languages, data parallel languages, logic languages including Prolog [1, 22, 31] (e.g. for groundness [9], sharing [7], freeness [5] and their combinations [6], parallelizatiion [3], etc.), database programming languages, concurrent logic languages, functional logic languages, constraint logic languages, concurrent constraint logic languages, specification languages, synchronous languages, procedural/functional concurrent/parallel languages [21], communicating and distributed languages [20] and more recently object-oriented languages [2, 55].

Abstract interpretation based static program analyses have been used for the static analysis of the embedded ADA software of the Ariane 5 launcher[1] and the ARD[2] [44]. The static program analyser aims at the automatic detection of the *definiteness*, *potentiality*, *impossibility* or *inaccessibility* of run-time errors such as scalar and floating-point overflows, array index errors, divisions by zero and related arithmetic exceptions, uninitialized variables, data races on shared data structures, etc. The analyzer was able to automatically discover the Ariane 501 flight error. The static analysis of embedded safety critical software (such as avionic software [51]) is very promising [30].

[1] Flight software (60,000 lines of Ada code) and Inertial Measurement Unit (30,000 lines of Ada code).

[2] Atmospheric Reentry Demonstrator.

3.4 Industrialization of Static Analysis by Abstract Interpretation

The impressive results obtained by the static analysis of real-life embedded critical software [44, 51] is quite promising for the industrialization of abstract interpretation.

This is the explicit objective of AbsInt Angewandte Informatik GmbH (`www.absint.com`) created in Germany by R. Wilhelm and C. Ferdinand in 1998 commercializing the program analyzer generator PAG and an application to determine the worst-case execution time for modern computer architectures with memory caches, pipelines, etc [34].

Polyspace Technologies (`www.polyspace.com`) was created in France by A. Deutsch and D. Pilaud in 1999 to develop and commercialize ADA and C program analyzers.

Other companies like Connected Components Corporation (`www.concmp.com`) created in the U.S.A. by W.L. Harrison in 1993 use abstract interpretation internally e.g. for compiler design [41].

4 Grand Challenge for the Next Decade

We believe that in the next decade the software industry will certainly have to face its responsibility imposed by a computer-dependent society, in particular for safety critical systems. Consequently, *Software reliability*[3] will be a grand challenge for computer science and practice.

The grand challenge for formal methods, in particular abstract interpretation based formal tools, is both the large scale industrialization and the intensification of the fundamental research effort.

General-purpose, expressive and cost-effective abstractions have to be developed e.g. to handle floating point numbers, data dependences (e.g. for parallelization), liveness properties with fairness (to extend finite-state model-checking to software), timing properties for embedded software, probabilistic properties, etc. Present-day tools will have to be enhanced to handle higher-order compositional modular analyses and to cope with new programming paradigms involving complex data and control concepts (such as objects, concurrent threads, distributed/mobile programming, etc.), to automatically combine and locally refine abstractions in particular to cope with "unknow" answers, to interact nicely with users and other formal or informal methods.

The most challenging objective might be to integrate formal analysis by abstract interpretation in the full software development process, from the initial specifications to the ultimate program development.

Acknowledgements I thank Radhia Cousot and Reinhard Wilhelm for their comments on a preliminary version of this paper. This work was supported by the DAEDALUS [30] and TUAMOTU [54] projects.

[3] other suggestions were "trustworthiness" (C. Jones) and "robustness" (R. Leino).

References

1. R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. *TOPLAS*, 15(1):133–181, 1993.
2. B. Blanchet. Escape analysis for object-oriented languages: Application to Java. *OOPSLA '99. SIGPLAN Not. 34(10):20–34*, 1999.
3. F. Bueno, M.J. Garc a de la Banda, and M.V. Hermenegildo. Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming. *TOPLAS*, 21(2):189–239, 1999.
4. G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis of higher-order functions. *Sci. Comput. Programming*, 7:249–278, 1986.
5. M. Codish, D. Dams, G. Fil , and M. Bruynooghe. Freeness analysis for logic programs – and correctness? *Proc. ICLP '93*, pp. 116–131. MIT Press, 1993.
6. M. Codish, H. Søndergaard, and P.J. Stuckey. Sharing and groundness dependencies in logic programs. *TOPLAS*, 21(5):948–976, 1999.
7. A. Cortesi and G. Fil . Sharing is optimal. *J. Logic Programming*, 38(3):371–386, 1999.
8. A. Cortesi, G. Fil , R. Giacobazzi, C. Palamidessi, and F. Ranzato. Complementation in abstract interpretation. *TOPLAS*, 19(1):7–47, 1997.
9. A. Cortesi, G. Fil , and W.H. Winsborough. Optimal groundness analysis using propositional logic. *J. Logic Programming*, 27(2):137–167, 1996.
10. P. Cousot. *M thodes it ratives de construction et d'approximation de points fixes d'op rateurs monotones sur un treillis, analyse s mantique de programmes*. Th se d' tat s sciences math matiques, Univ. of Grenoble, 1978.
11. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *ENTCS*, 6, 1997. `http://www.elsevier.nl/locate/entcs/volume6.html`, 25 pages.
12. P. Cousot. Types as abstract interpretations. *24^{th} POPL*, pp. 316–331. ACM Press, 1997.
13. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbr ggen, editors, *Calculational System Design*, volume 173, pp. 421–505. NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, 1999.
14. P. Cousot. Partial completeness of abstract fixpoint checking. *SARA '2000*, LNAI 1864, pp. 1–25. Springer-Verlag, 2000.
15. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, To appear (Preliminary version in [11]).
16. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. *2^{nd} Int. Symp. on Programming*, pp. 106–130. Dunod, 1976.
17. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4^{th} POPL*, pp. 238–252. ACM Press, 1977.
18. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. *Symp. on Artificial Intelligence & Programming Languages*, SIGPLAN Not. 12(8):1–12, 1977.
19. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *6^{th} POPL*, pp. 269–282. ACM Press, 1979.

20. P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes. *7th ICALP*, LNCS 85, pp. 119–133. Springer-Verlag, 1980.
21. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In A.W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, ch. 12, pp. 243–271. Macmillan, 1984.
22. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs[4]. *J. Logic Programming*, 13(2–3):103–179, 1992.
23. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, Aug. 1992.
24. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. *19th POPL*, pp. 83–94. ACM Press, 1992.
25. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). *Proc. 1994 ICCL*, pp. 95–112. IEEE Comp. Soc. Press, 1994.
26. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. *Proc. 4th PLILP '92*, LNCS 631, pp. 269–295. Springer-Verlag, 1992.
27. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. *7th FPCA*, pp. 170–181. ACM Press, 1995.
28. P. Cousot and R. Cousot. Temporal abstract interpretation. *27th POPL*, pp. 12–25. ACM Press, 2000.
29. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *5th POPL*, pp. 84–97. ACM Press, 1978.
30. DAEDALUS: *Validation of critical software by static analysis and abstract testing.* P. Cousot, R. Cousot, A. Deutsch, C. Ferdinand, . Goubault, N. Jones, D. Pilaud, F. Randimbivololona, M. Sagiv, H. Seidel, and R. Wilhelm. Project IST-1999-20527 of the european 5th Framework Programme, Oct. 2000 – Oct. 2002.
31. S.K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in Logic Programming Theory*, Int. Sec. 3, pp. 115–182. Clarendon Press, 1994.
32. A. Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. *Proc. PEPM '95*, pp. 226–229. ACM Press, 1995.
33. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. *Proc. SAS '2000*, LNCS 1824, pp. 115–134. Springer-Verlag, 2000.
34. C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Programming*, 35(1):163–189, 1999.
35. J. Feret. Confidentiality analysis of mobile systems. *Proc. SAS '2000*, LNCS 1824, pp. 135–154. Springer-Verlag, 2000.
36. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.
37. P. Granger. Static analysis of arithmetical congruences. *Int. J. Comput. Math.*, 30:165–190, 1989.
38. P. Granger. Static analysis of linear congruence equalities among variables of a program. 493, pp. 169–192. Springer-Verlag, 1991.
39. N. Halbwachs. About synchronous programming and abstract interpretation. *Sci. Comput. Programming*, 31(1):75–89, 1998.

[4] 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`.

40. R.R. Hansen, J.G. Jensen, F. Nielson, and H. Riis Nielson. Abstract interpretation of mobile ambients. *Proc. SAS '99*, LNCS 1694, pp. 134–138. Springer-Verlag, 1999.
41. W.L. Harrison. Can abstract interpretation become a main stream compiler technology? (abstract). *Proc. SAS '97*, LNCS 1302, p. 395. Springer-Verlag, 1997.
42. T.A. Henzinger, R. Majumbar, F. Mang, and J.-F. Raskin. Abstract interpretation of game properties. *Proc. SAS '2000*, LNCS 1824, pp. 220–239. Springer-Verlag, 2000.
43. N.D. Jones. Combining abstract interpretation and partial evaluation (brief overview). *Proc. SAS '97*, LNCS 1302, pp. 396–405. Springer-Verlag, 1997.
44. P. Lacan, J.N. Monfort, L.V.Q. Ribal, A. Deutsch, and G. Gonthier. The software reliability verification process: The Ariane 5 example. *DASIA '98 – DAta Systems In Aerospace*, ESA Publications, 1998.
45. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *Proc. ICCL 92*, pp. 137–146. IEEE Comp. Soc. Press, 1992.
46. F. Martin. *Generating Program Analyzers*. Pirrot Verlag, Saarbr cken, 1999.
47. F. Masdupuy. Semantic analysis of interval congruences. *FMPA*, LNCS 735, pp. 142–155. Springer-Verlag, 1993.
48. L. Mauborgne. Tree schemata and fair termination. *Proc. SAS '2000*, LNCS 1824, pp. 302–321. Springer-Verlag, 2000.
49. D. Monniaux. Abstract interpretation of probabilistic semantics. *Proc. SAS '2000*, LNCS 1824, pp. 322–339. Springer-Verlag, 2000.
50. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. Ph.D. Dissertation, CST-15-81, Univ. of Edinburgh, 1981.
51. F. Randimbivololona, J. Souyris, and A. Deutsch. Improving avionics software verification cost-effectiveness: Abstract interpretation based technology contribution. *DASIA '2000 – DAta Systems In Aerospace*, ESA Publications, 2000.
52. D.A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. *Proc. SAS '98*, LNCS 1503, pp. 351–380. Springer-Verlag, 1998.
53. J. Stransky. A lattice for abstract interpretation of dynamic (lisp-like) structures. *Inform. and Comput.*, 101(1):70–102, 1992.
54. TUAMOTU: *Tatouage lectronique s mantique de code mobile Java*. P. Cousot, R. Cousot, and M. Riguidel. Project RNRT 1999 n° 95, Oct. 1999 – Oct. 2001.
55. R. Vall e-Rai, H. Hendren, P. Lam, Gagnon, and P. Co. Soot - a Java^{tm} optimization framework. *Proc. CASCON '99*, 1999.
56. F. V drine. Binding-time analysis and strictness analysis by abstract interpretation. *Proc. SAS '95*, LNCS 983, pp. 400–417. Springer-Verlag, 1995.
57. A. Venet. Automatic determination of communication topologies in mobile systems. *Proc. SAS '98*, LNCS 1503, pp. 152–167. Springer-Verlag, 1998.
58. A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Programming*, 35(1):223–248, 1999.
59. Kwangkeun Yi. An abstract interpretation for estimating uncaught exceptions in standard ML programs. *Sci. Comput. Programming*, 31(1):147–173, 1998.

The electronic version of this paper includes additional material on static program analysis applications as well as a comparison with other formal methods (typing, model-checking and deductive methods) which, for lack of space, could not be included in this published version. A broader bibliography is available in its extended version.