Abstract Interpretation Based Static Analysis Parameterized by Semantics

(abstract)

Patrick Cousot

École normale supérieure, DMI, 45 rue d'Ulm, 75230 Paris cedex 05, France
cousot@dmi.ens.fr http://www.dmi.ens.fr/~cousot

Abstract. We review how the dependence upon semantics has been taken into account in abstract interpretation based program analysis and next propose to design general purpose abstract interpreters taking semantics as a parameter, either that of the program to be analyzed or that of a programming language.

1 Semantics Used for Static Analysis by Abstract Interpretation

A contribution of abstract interpretation was to understand that program static analyzers can be formally designed by discrete approximation of programming language semantics. An impressive number of semantics has been used right at the beginning of the formalization of abstract interpretation. To cite a few:

- small-step operational semantics of transition systems[1] (called "state transition function" in [6, sec. 3.2, p. 240]);
- prefix-closed finite execution trace semantics (called "computation sequence" in [6, sec. 3.2, p. 240] and "paths" in [9, sec. 2, p. 270]);
- first-order fixpoint[2] symbolic execution tree semantics [7, sec. 6, pp. 6–7];
- first-order fixpoint big-step operational semantics of transition systems (called "initial to final state transition function" in [6, sec. 3.2, p. 240]);
- first-order fixpoint collecting semantics (called "static semantics" in [6, sec. 4, p. 240]);
- first-order fixpoint strongest post-condition predicate transformer semantics of transition systems (called "deductive semantics" in [6, sec. 6, pp. 242–243] and "forward deductive semantics" in [9, sec. 3.1, pp. 270–271]) and second-order fixpoint[3] ones for recursive procedures (called "deductive semantics" in [8, sec. 3, pp. 243–251]);

[1] More precisely of partitioned deterministic transition systems $\langle States, I\text{-}states, \underline{n\text{-}state}\rangle$ where $States$ is a partitioned set of states, $I\text{-}states \subseteq States$ is the set of initial states and $\underline{n\text{-}state} \in States \longrightarrow States$ is the state transition function. For the example application to flowcharts considered in [6], the states are partitioned accordingly, by program control points.

[2] i.e. $x = f(x)$ where $f \in D \longrightarrow D$.

[3] i.e. $f(x) = F(f)(x)$ where $F \in (D \longrightarrow D) \longrightarrow (D \longrightarrow D)$.

– first-order fixpoint weakest pre-condition predicate transformer semantics of transition systems[4] (called "backward deductive semantics" in [9, sec. 3.2, p. 271]);
– axiomatic semantics (called "invariance proof method" in [11, sec. F, p. 258–266]);
– second-order fixpoint denotational semantics [23];
– second-order fixpoint relational semantics [15];
– higher-order denotational semantics [2, 16];

This variety of semantics is almost unavoidable because program static analysis deals with many different run-time properties and one looks for the semantics for which the considered program property is most easily expressible. The spectrum of semantics to be taken into consideration is even much larger when analyzing other families of languages besides imperative and functional ones like logical [19, 12] and parallel [10, 11] languages.

2 Programming Language and Semantics Independent Static Analysis Frameworks

It is very difficult to provide both programming language and semantics independent static analysis frameworks which are general enough to be easily adapted to specific instances. Our approach has been as follows:

– We refrain from presenting the theory of abstract interpretation in the specific context of a particular semantics of a peculiar programming language. This is in contrast with approaches almost exclusively devoted to the denotational semantics of the lambda-calculus such as, for instance, [1, 22–24].
– We first attempted to consider a universal model of programs in the form of transition systems [3]. This is quite effective for imperative, logic, constraint, functional (e.g. through term graph rewriting or interaction nets) and shared or distributed parallel programming languages, but fails for the higher-order domain-theoretic denotational semantics of functional languages.
– We then considered formalization of abstract interpretation theory in terms of the mathematical concepts used to present semantics. The premisses appear in [9] where the basic assumption is that the semantics can be presented in fixpoint form and [13] where it is considered in transfinite iterative form. This is probably clearer in [16] where abstractions (for sets of functions, for binary relations, etc.) and combinations of abstractions (composition, disjunctive completion, etc.) are considered independently of a particular semantics and then applied to comportment analysis of the typed lambda-calculus. No new concept is necessary to handle apparently different program analysis methods such as set-based analysis [18] or type inference [5].

[4] This time nondeterministic transition systems obtained e.g. by inversion of deterministic imperative programs, see [9, example 3.2.0.1, p. 271].

- A further step towards semantic independence is to recognize that a given semantics can be presented compositionally in many different inductive styles such as fixpoints, equational systems, constraints, closure conditions, rule-based formal systems, games, etc., which are all formally equivalent and preserved by abstraction [17].
- Even more important is the fact that semantics can be organized hierarchically and may be considered, after suitable generalization, as abstract interpretations of one another. Starting in [4] from a maximal trace semantics of a transition system, we derive a big-step semantics, termination and non-termination semantics, natural, demoniac and angelic relational semantics and equivalent nondeterministic denotational semantics, D. Scott's deterministic denotational semantics, generalized/conservative/liberal predicate transformer semantics, generalized/total/partial correctness axiomatic semantics and corresponding proof methods. All semantics are presented in uniform fixpoint form and the correspondence between these semantics are established by abstract interpretation.

By presenting the semantics of the programming languages in the same mathematical framework and by considering the comparable semantics as abstractions/refinements of one-another, we guarantee the semantic coherence of the various independent analyzes which can be performed on a program and offer the possibility of composing and combining them consistently in the lattice of abstract interpretations [9, sec. 8, pp. 278–279].

3 Static Analysis Parameterized by Abstract Domains

In the classical approach, a program analyzer is usually designed for a specific programming language and its soundness is based upon a specific standard semantics. The analyzer maps programs P of a given programming language to the abstract semantics (or an approximation) $S^\sharp[\![P]\!]$ specifying the program properties discovered by the analysis:

$$\begin{array}{ccc} P & \longmapsto\!\!\!\begin{array}{c}\text{Abstract}\\ \text{interpreter}\end{array}\!\!\!\longrightarrow & S^\sharp[\![P]\!] \\ \text{program} & & \text{abstract semantics} \\ & & \text{(approximation)} \end{array}$$

The abstract interpreter is usually designed in two phases. In the first phase, a translator maps the program concrete syntax to an abstract semantics specification $\mathcal{S}^\sharp[\![P]\!]$, e.g. a system of equations or unsolved contraints which, in a second phase, is solved to compute the abstract semantics $S^\sharp[\![P]\!]$:

$$\begin{array}{ccccc} P & \longmapsto \text{Translator} \longrightarrow & \mathcal{S}^\sharp[\![P]\!] & \longmapsto \text{Solver} \longrightarrow & S^\sharp[\![P]\!] \\ \text{program} & & \text{abstract} & & \text{abstract} \\ & & \text{semantics} & & \text{semantics} \\ & & \text{specification} & & \text{(approximation)} \end{array}$$

The analyzer may be generic, more or less precise analyzes being obtained by considering various abstract algebras (i.e. abstract domains together with the corresponding abstract operations) which can be integrated in the analyzer as independent modules. In this case the generic analyzer takes modules implementing an abstract algebra and programs as parameters to produce information about run-time execution.

4 Static Analysis Parameterized by the Program Semantics

To achieve independence vis-à-vis the semantics, a proposed alternative approach consists in introducing the concrete program semantics as a parameter to the abstract interpreter.

$$\underset{\substack{\text{concrete}\\\text{semantics}\\\text{specification}}}{\mathcal{S}} \times \underset{\substack{\text{abstract}\\\text{semantics}\\\text{specification}\\\text{function}}}{\mathcal{S} \mapsto \mathcal{S}^\sharp[\![\mathcal{S}]\!]} \quad \longmapsto\!\!\!\!\underset{\text{interpreter}}{\overset{\text{Abstract}}{}}\!\!\!\!\longrightarrow \quad \underset{\substack{\text{abstract}\\\text{semantics}\\\text{(approximation)}}}{S^\sharp[\![\mathcal{S}]\!]}$$

An abstract domain together with a meta-language e.g., a fixpoint calculus on the abstract domain, have to be designed so as to express the input concrete semantics specification $\mathcal{S}$ and the abstract semantics specification function $\mathcal{S} \mapsto \mathcal{S}^\sharp[\![\mathcal{S}]\!]$. The correctness of the abstract semantics specification is proved with respect to the meta-semantics of the meta-language.

The concrete semantics $\mathcal{S}$ of a program which is fed to the analyzer can be designed "by hand". It is then considered as a specification to be analyzed, which is often the case in model-checking, where the abstraction process is sometimes taken for granted without formal justification. Most often in automatic program static analysis, the abstract semantics will be automatically generated by a front-end pre-processor which has to be proved correct and rewritten for different programming languages:

$$\underset{\text{program}}{P} \longmapsto \text{Translator} \longrightarrow \underset{\substack{\text{concrete semantics}\\\text{specification}}}{\mathcal{S}[\![P]\!]}$$

In both cases the abstract semantics $S^\sharp[\![\mathcal{S}[\![P]\!]]\!]$ of the program P corresponds to a given static approximation which is necessary for inexpressive abstract domains or to master the analysis cost [14, sec. 3, pp. 271–274].

If the abstract domain and the meta-language are rich and expressive enough to specify the standard semantics defining exact program executions, then the concrete semantics specification can be chosen to be that of the standard semantics. One can then consider applications to interactive debugging or model-checking of finite systems. Applications to automatic program analysis and model-checking of infinite systems require powerful dynamic extrapolation operators (widening, narrowing) to allow for effective computation of sound approximations of the abstract semantics [14, sec. 4, pp. 275–278]). The considered abstract domain can also be parameterized to achieve various degrees of approximation. Examples of such abstract domains are A. Deutsch's exponential unitary-prefix monomial relations [20], P. Cousot and R. Cousot's constrained tree grammars [18] and A. Venet's cofibred domains [25] which can all be parameterized by numerical domains.

This approach is much more flexible than specialized abstract interpreters since the abstract interpreter which computes or approximates the abstract semantics is general purpose. Moreover the specialized abstract interpreter can be derived by partial evaluation [21]. However, this approach requires the hand-coding and manual correctness proof of a front-end pre-processor/translator for mapping programs to a specification of their concrete semantics.

5 Static Analysis Parameterized by the Programming Language Semantics

A further idea to achieve both programming language and semantics independence is to have the abstract interpreter take the program abstract syntax A, the concrete semantics specification function mapping the program abstract syntax A to a concrete semantics specification $\mathcal{S}[\![A]\!]$ and the abstract semantics specification function $\mathcal{S} \mapsto \mathcal{S}^\sharp[\![\mathcal{S}]\!]$ as parameters. The concrete semantics specification $\mathcal{S}[\![A]\!]$ can then be provided as parameter of the abstract semantics specification $\mathcal{S}^\sharp[\![\mathcal{S}[\![A]\!]]\!]$ in order to get the abstract semantics (or an approximation) $S^\sharp[\![A]\!]$ specifying the program properties discovered by the analysis:

$$\begin{array}{rcccc}
A \;\times & A \mapsto \mathcal{S}[\![A]\!] & \times\; \mathcal{S} \mapsto \mathcal{S}^\sharp[\![\mathcal{S}]\!] & \longmapsto\!\!\begin{array}{c}\text{Abstract}\\\text{interpreter}\end{array}\!\!\longrightarrow & S^\sharp[\![A]\!] \\
\text{abstract} & \text{concrete} & \text{abstract} & & \text{abstract} \\
\text{syntax} & \text{semantics} & \text{semantics} & & \text{semantics} \\
 & \text{specification} & \text{specification} & & \text{(approximation)} \\
 & \text{function} & & &
\end{array}$$

Again the abstract domain and abstract semantics specification language have to be expressive enough to be able to specify A, $A \mapsto \mathcal{S}[\![A]\!]$ and $\mathcal{S} \mapsto \mathcal{S}^\sharp[\![\mathcal{S}]\!]$. Suitable choices of widenings/narrowing lead to a simplification of the expression $\mathcal{S}^\sharp[\![\mathcal{S}[\![A]\!]]\!]$ by simple expansion so as to obtain an abstract semantics specification on which the solvers considered in the previous sections can directly operate. The only remaining part to be coded is a translator from concrete to abstract syntax:

$$\begin{array}{rcl}
P & \longmapsto \text{Translator} \longrightarrow & A[\![P]\!] \\
\text{program} & & \text{program} \\
\text{(concrete syntax)} & & \text{(abstract syntax)}
\end{array}$$

Great flexibility is achieved through that design ranging from specialized abstract interpreters obtained by partial evaluation, abstract interpreters working with different kinds of semantics to general purpose language independent abstract interpreters using e.g. an intermediate language. Correctness proof are by construction. The same abstract interpreter can then be used for many different purposes from compile-time program analysis, to abstract debugging, abstract model-checking, etc. Abstract interpretation and partial evaluation of the abstract interpreters is necessary to achieve full efficiency. Other stimulating and interesting challenges are discussed.

References

1. S. Abramsky and C. Hankin. *An introduction to abstract interrpetation*, ch. 1. Ellis Horwood, 1987.
2. G.L. Burn, C.L. Hankin, and S. Abramsky. Strictness analysis of higher-order functions. *Sci. Comput. Prog.*, 7:249–278, 1986.
3. P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, ch. 10, pp. 303–342. Prentice-Hall, 1981.
4. P. Cousot. Design of semantics by abstract interpretation, invited address. In *Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference (MFPS XIII)*, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1997.
5. P. Cousot. Types as abstract interpretations, invited paper. In 24^{th} *POPL*, pp. 316–331, Paris, 1997. ACM Press.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *POPL*, pp. 238–252, Los Angeles, Calif., 1977. ACM Press.
7. P. Cousot and 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.
8. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conference on Formal Description of Programming Concepts*, St-Andrews, N.B., Canada, pp. 237–277. North-Holland, 1977.
9. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In 6^{th} *POPL*, pp. 269–282, San Antonio, Texas, 1979. ACM Press.
10. P. Cousot and R. Cousot. Semantic analysis of communicating sequential processes. In J.W. de Bakker and J. van Leeuwen, editors, 7^{th} *ICALP*, LNCS 85, pp. 119–133. Springer-Verlag, 1980.
11. 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.
12. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 13(2–3):103–179, 1992. (The editor of JLP has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.dmi.ens.fr/~cousot`.).
13. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic and Comp.*, 2(4):511–547, 1992.
14. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. Int. Work. PLILP '92, Leuven, Belgium*, LNCS 631, pages 269–295. Springer-Verlag, 1992.

15. P. Cousot and R. Cousot. Galois connection based abstract interpretations for strictness analysis, invited paper. In D. Bjørner, M. Broy, and I.V. Pottosin, editors, *Proc. FMPA*, Academgorodok, Novosibirsk, Russia, LNCS 735, pages 98–127. Springer-Verlag, 1993.
16. P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proc. 1994 ICCL, Toulouse, France*, pp. 95–112. IEEE Comp. Soc. Press, 1994.
17. P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. In P. Wolper, editor, *Proc. 7th Int. Conf. CAV '95*, Liège, Belgium, LNCS 939, pp. 293–308. Springer-Verlag, 1995.
18. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7th FPCA*, pp. 170–181, La Jolla, Calif.,1995. ACM Press.
19. S.K. Debray. Formal bases for dataflow analysis of logic programs. In G. Levi, editor, *Advances in Logic Programming Theory*, International Schools for Computer Scientists, section 3, pp. 115–182. Clarendon Press, 1994.
20. 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, 1992.
21. N. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
22. N.D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, ch. 5, pp. 527–636. Clarendon Press, 1995.
23. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. Ph.D. Dissertation, CST-15-81, Department of Computer Science, University of Edinburgh, Edinburg, Scotland, 1981.
24. F. Nielson. Two-level semantics and abstract interpretation. *TCS — Fund. St.*, 69:117–242, 1989.
25. A. Venet. Abstract cofibred domains: Application to the alias analysis of untyped programs. In R. Cousot and D.A. Schmidt, editors, *Proc. SAS '96*, Aachen, Germany, LNCS 1145, pp. 368–382. Springer-Verlag, 1996.

Published in the *Proceedings of the 4th International Symposium on Static Analysis, SAS'97*, Paris, France, 8-10 September 1997, P. van Hentenryck (Ed.), Lecture Notes in Computer Science 1302, Springer-Verlag, pp. 388–394.