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]);

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

\(\text{i.e. } x = f(x) \text{ where } f \in D \rightarrow D.\)

\(\text{i.e. } f(x) = F(f)(x) \text{ where } F \in (D \rightarrow D) \rightarrow (D \rightarrow D).\)
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].