Program Analysis: The Abstract Interpretation Perspective

Patrick Cousot
École Normale Supérieure, Paris

Program analysis should evolve from a disparate collection of methods and algorithms to a mature discipline founded on a well-established methodology. Abstract interpretation can be used as a basis for such a methodology.

1. PROGRAM ANALYSIS
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.

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.

2. DESIGN METHODOLOGY
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.

3. ABSTRACT INTERPRETATION
Abstract interpretation understood as a theory of semantic approximation is a basis for 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.
To be of general scope, its presentation 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.

4. 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 sys-
tem, etc. [5]);
—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 be studied independently. We dis-
cuss briefly below the specification of the program properties through a hierarchy
of semantics and then the abstract semantic algebra which are used for program
analysis.

5. HIERARCHIES OF SEMANTICS

Founding a program analysis methodology on a specific semantics (small/big step
operational, denotational [14], weakest-preconditions, etc.) turns out not to be very
convenient. This is because a program analysis methodology based on a compo-
sitional semantics (e.g. denotational or axiomatic) can hardly be formulated inde-
pendently 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 seman-
tics 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 pro-
gram properties, that is program executions at various levels of abstraction or
refinement [2]. 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 [18] but very simple using a relational semantics [3]
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 [4]).
6. HIERARCHIES OF ABSTRACT ALGEBRAS

To a unique encoding of abstract properties (e.g. using logical formulae 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 [7] which have been used to estimate the size of arguments of logic programs [1; 9] and for infinite state model-checking [12; 8] including hybrid systems [13]. Another example is BDDs originating from model-checking but also useful to cope with size explosion in strictness analysis [16]. Not enough such analysis independent abstract algebras are presently available, in particular for discrete structures (see a.o. [6; 10; 19]) as opposed to numerical structures where the spectrum is wider [7; 11; 15]. Some abstract algebras are used without being clearly identified (e.g. for type inference, see however [17] for references). In this area, mainly of algorithmic nature, not enough constructive methods and algorithms from algebra, 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.

7. DESIGN OF PROGRAM ANALYZERS

From a more practical point of view, we think that the uniform design of:
— hierarchies of semantics,
— hierarchies of abstract algebras

at various levels of abstraction/refinement 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.

REFERENCES


