Our objective in this talk is to give an intuitive account of abstract interpretation theory [1,2,3,4,5] and to present and discuss its main applications [6].

Abstract interpretation theory formalizes the conservative approximation of the semantics of hardware and software computer systems. The semantics provides a formal model describing all possible behaviors of a computer system in interaction with any possible environment. By approximation we mean the observation of the semantics at some level of abstraction, ignoring irrelevant details. Conservative means that the approximation can never lead to an erroneous conclusion.

Abstract interpretation theory provides thinking tools since the idea of abstraction by conservative approximation is central to reasoning (in particular on computer systems) and mechanical tools since the idea of an effective computable approximation leads to a systematic and constructive formal design methodology of automatic semantics-based program manipulation algorithms and tools (e.g. [7]).

Semantics have been studied in the framework of abstract interpretation [8,9] and compared according to their relative precision. A number of semantics including among others small-step, big-step, termination and nontermination semantics, Plotkin’s natural, Smyth’s demonic, Hoare’s angelic relational and corresponding denotational semantics, Dijkstra’s weakest precondition and weakest liberal precondition predicate transformers and Hoare’s partial and total axiomatic semantics have all been derived by successive abstractions starting from an operational maximal trace semantics of a transition system. This results in a hierarchy of semantics providing a complete account of the structure and relative precision of most well-known semantics of programming languages [10].

Program transformation (such as online and offline partial evaluation, program monitoring (e.g. for security policy enforcement or scheduling), etc.) is an abstract interpretation [11] where the program syntactic transformation is an effective approximation of a corresponding undecidable transformation of the program semantics. The correctness of this program transformation is expressed as an observational equivalence of the subject and transformed semantics at some level of abstraction.
Typing that is formal type systems and type inference algorithms, is an approximation of the denotational semantics of higher-order functional programs [12]. The abstraction is powerful enough to show statically that “typable cannot go wrong” in that the denotational semantics of these programs cannot raise at run-time those errors excluded by typing. This point of view leads to a hierarchy of type systems, which is part of the lattice of abstract interpretation of the untyped lambda-calculus. The hierarchy includes classical Milner/Mycroft and Damas/Milner polymorphic type schemes, Church/Curry monotypes and Hindley principal typing algorithm as well as new à la Church/Curry polytype systems.

Model-checking classical linear-time and branching-time state based algorithms are sound and complete abstract interpretations of the trace-based semantics of transition systems [13]. Surprisingly, for the $\mu$-calculus, a novel general temporal specification language featuring a natural and rich time-symmetric trace-based semantics, model-checking turned out to be incomplete, even for finite systems [13]. Moreover, any model-checking for the $\mu$-calculus abstracting away from sets of traces will be necessarily incomplete [14].

Static program analysis is the first and most prevalent application of abstract interpretation [1,3,4,5]. By effective approximation of the fixpoint semantics of programs through abstraction [4,5] and convergence acceleration [4,15], a program analyzer will produce maybe incomplete but always sound information about the run-time behavior of programs. Abstract interpretation provides a general theory behind all programs analyzers, which only differ in their choice of considered programming languages (e.g. imperative [16,17], parallel [18,19], functional [20], logic [21], etc), program properties (among many others, run-time errors [16,22], precision [23], security [24,25], fair liveness [26], probabilistic termination [27], etc) and their abstractions. Finally, we will discuss the various possible designs of program analyzers, from general-purpose to application-specific ones.

References

21. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Logic Programming 13 (1992) 103–179 (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/).


