Abstract Interpretation

PATRICK COUSOT

DMI, École Normale Supérieure, Paris ⟨cousot@dmi.ens.fr⟩ ⟨http://www.ens.fr/~cousot⟩

Abstract interpretation [Cousot and Cousot 1977, 1979] is a general theory for approximating the semantics of discrete dynamic systems, e.g. computations of programs. In particular program analysis algorithms can be constructively derived from these abstract semantics.

PRINCIPLES OF ABSTRACT INTERPRETATION

A semantics S of a programming language L associates a semantic value $S[\![p]\!] \in \mathcal{D}$ in the semantic domain $\mathcal{D}$ to each program p of L. The semantic domain $\mathcal{D}$ can be transition systems (for small-step operational semantics), pomsets, traces, relations (for big-step operational semantics), higher-order functions (for denotational semantics), and so on. $\mathcal{D}$ is usually defined compositionally by induction on the structure of run-time objects (computations, data, etc.). S is defined compositionally by induction on the syntactical structure of programs, using e.g. fixpoint definitions to handle iteration, recursion, and the like.

An empirical approach to abstract interpretation consists in a priori choosing a problem-specific abstract semantics domain $\mathcal{D}^\sharp$ and an abstract semantics $S^\sharp \in \mathsf{L} \mapsto \mathcal{D}^\sharp$ which is designed intuitively for a specific language L. Then safety, correctness or soundness is established by proving that a soundness relation σ satisfies

$$\forall p \in \mathsf{L} : \sigma(S[\![p]\!], S^\sharp[\![p]\!]).$$

If the abstract semantics is computable ($\mathcal{D}^\sharp$ is usually assumed to be finite), we can infer that the abstract interpretation is sound in the sense that:

$$S[\![p]\!] \quad \in \quad \{S \mid \sigma(S, S^\sharp[\![p]\!])\},$$

which may be sufficient to prove program properties e.g. that some "programs cannot go wrong". However, the choice of $\mathcal{D}^\sharp$ and σ offers no guideline for the design of the abstract semantics $S^\sharp$ with respect to the concrete semantics S.

The approach propounded in [Cousot and Cousot 1977, 1979] is constructive in the sense that once the standard semantics S and an approximation α are chosen, one can derive the best choice for the corresponding abstract semantics $S^\sharp$. More precisely let us call elements of the powerset

$$\mathcal{D}_{\text{Coll}} \quad \stackrel{\text{def}}{=} \quad \wp(\mathcal{D})$$

program (concrete) properties. Define $S_{\text{Coll}} \in \mathsf{L} \mapsto \mathcal{D}_{\text{Coll}}$ to be the collecting semantics:

$$S_{\text{Coll}}[\![p]\!] \quad \stackrel{\text{def}}{=} \quad \{S[\![p]\!]\}.$$

(This is a conceptual step, since no other detailed specification of S_{Coll} is needed, but for the design of formal proof methods). $S_{\text{Coll}}[\![p]\!]$ is the strongest program property. We have seen that an abstract property P, such as $\{S \mid \sigma(S, S^\sharp[\![p]\!])\}$ above, is weaker in that $S_{\text{Coll}}[\![p]\!] \subseteq P$. We call $\subseteq$ the approximation ordering. Now the abstraction function is a map $\alpha \in \mathcal{D}_{\text{Coll}} \mapsto \mathcal{D}_{\text{Coll}}$. We call $\alpha[\mathcal{D}] \stackrel{\text{def}}{=}$

$\{\alpha(S) \mid S \in \mathcal{D}\}$ the abstract domain. We often use an isomorphic representation $\mathcal{D}^\sharp$ for $\alpha[\mathcal{D}]$ and directly define $\alpha \in \mathcal{D}_{\text{Coll}} \mapsto \mathcal{D}^\sharp$. An example would be the approximation of a trace-based semantics by a relational/denotational semantics:

$$\alpha(P) \stackrel{\text{def}}{=} \{\langle s_0, s_n\rangle \mid s_0 s_1 \ldots s_n \in P\} \cup \{\langle s_0, \bot\rangle \mid s_0 s_1 \ldots \in P \text{ is an infinite trace}\}.$$

Another approximation, ignoring distinction between finite and infinite computations, would be the approximation of traces by sets of states:

$$\alpha(P) \stackrel{\text{def}}{=} \{s_i \mid \exists s_0 s_1 \ldots s_i \ldots \in P\},$$

which is adequate for safety/invariance properties.

Once such abstract domain $\mathcal{D}^\sharp$ and abstraction function α have been chosen, it remains to derive the abstract semantics $S^\sharp[\![p]\!]$, $p \in \mathsf{L}$. We proceed compositionally, by induction on the syntactic structure of program p. The price to pay is that in general the ideal $\alpha(S_{\text{Coll}}[\![p]\!])$ has to be approximated by $S^\sharp[\![p]\!] \supseteq \alpha(S_{\text{Coll}}[\![p]\!])$. For example, consider a fixpoint definition (for some syntactic construct $p(p_1, \ldots, p_n)$ with components p_1, $\ldots$, p_n):

$$S[\![p(p_1, \ldots, p_n)]\!] \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq} F_p \cdot [S[\![p_1]\!], \ldots, S[\![p_n]\!]].$$

For simplicity, we consider the simple case when the computational ordering $\sqsubseteq$ coincide with the approximation ordering $\subseteq$. Assuming, by induction hypothesis, that we know a sound abstract semantics for the program components p_1, $\ldots$, p_n:

$$\alpha(S_{\text{Coll}}[\![p_i]\!]) \subseteq S^\sharp[\![p_i]\!],\ i = 1, \ldots, n\ ,$$

then we look, by algebraic formula manipulation, for $F_p^\sharp$ satisfying for all C_1, $\ldots$, $C_n \in \mathcal{D}_{\text{Coll}}$:

$$\alpha(\{F_p[S_1, \ldots, S_n] | \forall i = 1, \ldots, n{:}S_i \in C_i\}) \subseteq F_p^\sharp[\alpha(C_1), \ldots, \alpha(C_n)]$$

in order to conclude (under suitable hypotheses, see Cousot and Cousot [1979]) that

$$\alpha(\text{lfp}^{\sqsubseteq} F_p[C_1, \ldots, C_n]) \subseteq \text{lfp}^{\sqsubseteq^\sharp} F_p^\sharp \cdot [\alpha(C_1), \ldots, \alpha(C_n)].$$

This leads to the definition of the abstract semantics:

$$S^\sharp[\![p(p_1, \ldots, p_n)]\!] \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq^\sharp} F_p^\sharp \cdot [S^\sharp[\![p_1]\!], \ldots, S^\sharp[\![p_n]\!]],$$

which is sound, by construction, so that no a posteriori verification is necessary. When equality holds, we have a completeness property which is useful to design hierarchies of semantics. By identifying useful abstract algebras consisting of an abstract domain $\mathcal{D}^\sharp$ and abstract operations $F^\sharp$ corresponding to common primitive operations F used in the semantic definition of programming languages, one can design abstraction libraries of general scope. Finally, the abstraction can be parameterized with other abstractions in order to obtain generic semantic definitions and abstract interpreters. Contrary to e.g. type inference, soundness (and relative completeness) can be established once for all (may be parameterized by basic abstractions for generic implementations) and not for each particular instance of the program analysis problem.

The abstract interpretation framework sketched above is based on an abstraction function α. The existence of a best (most precise) approximation among all the possible sound ones is ensured by choosing α as the upper-adjoint of a Galois connection. Other equivalent formulations using Moore families, closure operators, ideals, …where introduced in Cousot and Cousot [1979]. The case when the computational ordering $\sqsubseteq$ does not coincide with the approximation ordering $\subseteq$ in their concrete or abstract versions is considered in Cousot and Cousot [1994]. Other alternatives are studied in Cousot and Cousot [1992] (using a soundness relation σ, a concretization γ, an abstraction/concretization pair $\langle\alpha, \gamma\rangle$ or widenings, which allows for the degree of approximation to evolve during program analysis, so that the abstract domain $\mathcal{D}^\sharp$ is not fixed once and for all but evolves during the analysis) .

Under the influence of Mycroft's early application of abstract interpretation to strictness analysis of lazy functional languages, the standard semantics S is often chosen to be a denotational semantics (see Jones and Nielson [1995] for a survey taking this point of view) which is mainly adequate for functional languages. The original operational-based abstract interpretation [Cousot 1981] turned out to be much more adequate for imperative, logic [Debray 1994] and more recently concurrent, distributed and object-oriented languages. A general framework unifies both point of views [Cousot and Cousot 1994], by lifting operational semantics to handle infinite behaviors, considering the equivalence between rule-based and fixpoint presentations of semantics specifications and viewing denotational semantics as part of a hierarchy of abstractions of operational semantics.

APPLICATION OF ABSTRACT INTERPRETATION

The most widespread use of abstract semantics $S^\sharp$ is for the specification of program analyzers as used for highly-performant compilers, program transformation (e.g. program vectorization and parallelization), partial evaluation, test generation for program debugging, abstract debugging (involving abstract values/properties instead of concrete ones), polymorphic type inference, effect systems, model checking, verification of hybrid systems, and the like.

An important aspect of research of abstract interpretation is concerned with the composable design of abstractions α by induction on the mathematical structure of the semantic domain $\mathcal{D}$ (which, for typed languages, often coincide with the type structure of the language) for all possible data and control structures encountered in programming languages. Let us consider a small range of samples of data abstractions:

- For attribute-independent abstractions of sets of vectors of numbers, one can consider signs, intervals, parity, simple congruences. For relational abstractions, one can consider linear equalities, linear inequalities, congruences, congruencial trapezoids, and so on.
- For the context-free abstraction of formal languages, one can consider regular expressions, grammars (thus making the link with set-based analysis). Unitary-prefix monomial decompositions of rational subsets of a free monoid (such as $\{\mathtt{X.tl}^m\mathtt{.hd.tl}^n\mathtt{.hd.tl}^p \mid m = n = p\}$) provide an example of context-sensitive abstraction [Deutsch 1995]. These abstractions can be used for pointer analysis based on location-free/storeless models of computation.
- For the abstraction of sets of graphs, as used e.g. in store-based pointer analysis, one can refer to Deutsch [1995].

To remain succint on control structures, let us consider the example of program loops which involve solving an equation $X = F^\sharp_\ell(X)$ originating from the fixpoint definition $\mathrm{lfp}^{\sqsubseteq^\sharp} F^\sharp_\ell$ of the abstract semantics $S^\sharp[\![\ell]\!]$ of a program ℓ. This is one of the typical algorithmic problems involved in abstract interpretation. One classical solution, when $F^\sharp_\ell$ is a monotonic operator on a poset $\langle\mathcal{D}^\sharp, \sqsubseteq^\sharp\rangle$ with infimum $\perp$, is iteration: $X^0 \stackrel{\text{def}}{=} \perp$, $X^{n+1} \stackrel{\text{def}}{=} F^\sharp_\ell(X^n)$ until convergence. Widenings [Cousot and Cousot 1977] can be used to accelerate convergence (in large or infinite domains) and cope with the lack of least upper bounds during fixpoint iteration. Widenings can be understood as local dynamic change of abstract semantic domain during the analysis [Cousot and Cousot 1992].

The complexity issues in abstract interpretation have only been touched upon. A common error is that abstract interpretation is thought to be inherently exponential (as opposed to polynomial dataflow analysis). This can be exactly the contrary! It can be polynomial or polynomial on the average but exponential in pathological cases which are rare enough to be cut off by widening (e.g. polymorphic type inference à la Hindley-Milner). In general, exponential costs can be avoided by using widenings introducing further approximation as analysis time elapses. The common idea that program analyzers should be as fast as compilers does not necessarily take the cost/benefit trade-off into account. Is it better to spend 8 (night) hours of CPU time or 8 (day) hours of man-power for finding a crucial programming error?

CONCLUSION AND RESEARCH PERSPECTIVES

Although the designer of program analyzers may prefer empirical approaches, abstract interpretation is often an indispensable guideline to avoid conceptual errors since it provides a methodology to design a formal specification. It provides a synthetic understanding of the abundant literature on semantics, type systems, logics of computations, program verification, program analysis, partial evaluation, and the like, all involving more or less refined or abstract semantics of computations.

A number of problems remains to be considered or more thoroughly studied, e.g.:

Semantics

- general frameworks formalizing the notion of semantic approximation;
- models of computations (e.g. true concurrency) to be used in concrete semantics;
- design of hierarchies of parameterized semantics for programming languages (such as concurrent and object-oriented languages);

Domains

- abstract domains for non-numerical objects,
- design of abstraction functions specifying different program analysis methods in order to compare their relative power,
- decomposition/combination of existing program analyses;

Algorithms

- equation resolution and convergence acceleration methods,
- compositional design of widenings;

Abstract interpreters

- design and implementation of general-purpose libraries of abstract domains and their associated operations;
- design of language-specific generic abstract interpreters; and
- formal or experimental study of the complexity/benefit tradeoff of program analyses.

REFERENCES

Cousot, P. 1981. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*. S.S. Muchnick and N. Jones Eds., Prentice-Hall, Englewood Cliffs, NJ, Ch. 10, 303–342.

Cousot, P. and Cousot, R. 1994. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of 1994 ICCL*, Toulouse, France (May 16–19), IEEE, Los Alamitos, CA, 95–112.

Cousot, P. and Cousot, R. 1992. Abstract interpretation frameworks. *J. of Logic and Comput. 2*, 4 (Aug.), 511–547.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th POPL* (San Antonio, TX), ACM Press, New York, 269–282.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th POPL* (Los Angeles, CA), ACM Press, New York, 238–252.

Debray, S. K. 1994. Formal bases for dataflow analysis of logic programs. In *Advances in Logic Programming Theory*, International Schools for Computer Scientists, G. Levi, Ed., Clarendon Press, New York, Sect. 3, 115–182.

Deutsch, A. 1995. Semantic models and abstract interpretation techniques for inductive data structures and pointers. Invited paper. In *Proceedings of PEPM '95* (La Jolla, CA, June 21-23), ACM Press, New York, 226–229.

Jones, N. and Nielson, F. 1995. Abstract interpretation: a semantic-based tool for program analysis. In *Semantic modelling*, S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, Eds., number 4 in *Handbook of Logic in Computer Science*. Clarendon Press, New York.