On the Desirable Link Between Theory and Practice in Abstract Interpretation (Extended Abstract)

Baudouin Le Charlier[1] and Pierre Flener[2]

[1] Institut d'Informatique
University of Namur, B–5000 Namur, Belgium
Email: ble@info.fundp.ac.be
[2] Department of Computer Engineering and Information Science
Bilkent University, 06533 Bilkent, Ankara, Turkey
Email: pf@cs.bilkent.edu.tr

Abstract. Abstract interpretation was originally introduced by Patrick and Radhia Cousot as a unifying mathematical framework underlying the design of practical program analyses. In this paper,[3] we reconsider some theoretical concepts and assumptions most often used in abstract interpretation. We argue that they impose too strong mathematical conditions on the abstract semantics of programs and so do not allow a practical and provably correct implementation of complex analyses. As a solution, we advocate the use of a weaker and more general methodology, which is essentially based on the concept of (explicit) specification. We illustrate the methodology through a historical review of the abstract interpretation system *GAIA*.

1 Motivations

Although abstract interpretation is a widely used concept, its role and nature are probably not as clear as they should be. Many authors introduce abstract interpretation as a (semantics based) *methodology* for building program analyses. Other authors emphasize that abstract interpretation is a *theory*. The Cousots themselves often insist that abstract interpretation is a tool for comparing semantics in general and for deriving more "abstract" semantics from more "concrete" ones. This variety of presentations is explained by the large number of researchers, of different backgrounds, working on program analysis: there are engineers, mathematicians, denotational semanticists, programming methodologists, to name but a few. Such educational differences explain why the "theory of abstract interpretation" has in fact split into many formulations and variants guided by different philosophies. As a consequence, it seems to us relevant to question the very idea that abstract interpretation is (or can become) a unified framework.

[3] a longer and (slightly) more technical version of which will be available at the conference.

In this paper, we analyze some theoretical and practical aspects of abstract interpretation from a programming methodology standpoint. In other words, we examine their usefulness with respect to the broad objective of building program analyses that are

provably correct: we must be able to provide convincing arguments about the correctness of our analyses up to any requested level of detail (including the code level);

practically implementable: the "theoretical objects" involved in the analyses are required to have a clear counterpart in the implementation, yet this "tracability" requirement must not entail an efficiency penalty; and

scalable: the analyses can be complexified to reach an arbitrary level of accuracy; i.e., the construction of abstract domains that are as complex as desired must be manageable.

2 A Methodology Based on Specifications

A main difference between computer science and mathematics is that formal texts written by computer scientists (i.e., programs in a broad sense) are eventually run on computers, which may reveal errors (and actually often do), while formal texts written by mathematicians (i.e., definitions, theorems, and proofs) are only read by other mathematicians, so that their correctness results from a social acceptance process [11], without being submitted to a factual risk of contradiction.

This difference between mathematics and computer science is the main reason why programming methodology has developed as an original discipline. According to us, the key concept in programming methodology is the notion of a *specification* that allows one to reason explicitly about the program construction process, by establishing a clear link between relevant parts of the program (e.g., procedures and loops) and their purpose [14]. Following this fundamental intuition, we would like abstract interpretation to provide us with the concepts that are the most convenient for specifying the various steps of a program analysis design. Thus, in the next section, we reconsider some well-known aspects of abstract interpretation with respect to this aim.

3 Theory in Abstract Interpretation

3.1 Overview

The original formulation of abstract interpretation [7] can be viewed as a (mathematical) generalization of several ad hoc data flow analyses that were previously published. The aim of that work is to provide convenient and reusable concepts that ensure an economy of thought while building new analyses. A major goal is that an analysis should be derived in a *systematic way* from the (or: a) standard semantics of the programming language.

An apparently straightforward way of doing so is to design an analysis that "mimics" the normal (concrete) computation on a non-standard (abstract) domain. This purely operational idea can be generalized to deal with other standard semantics, such as denotational semantics (for functional languages) and T_P-like semantics (for logic and constraint logic languages): the idea now becomes that an abstract semantics should be *homomorphic* to the standard one [13]. Thus, the abstract and standard semantics are two different instances of a generic (e.g., denotational) semantics, and they are related by means of an *abstraction* function. The benefit of such an approach is that the correctness of the abstract semantics can be established in an almost obvious way (e.g., by a combination of induction on the syntax and fixpoint induction). The "homomorphic" approach has been mainly developed in the functional programming community, since it naturally appears as an application of the denotational semantics theory [23]. More recently, it has also been extensively used in the logic programming community, but based on the mathematical structure of *Galois insertion* [4].

The use of Galois insertions was in fact first introduced by the Cousots in their landmark paper [7], in conjunction with the notion of *collecting semantics*. The collecting semantics basically is a reformulation of the standard semantics where concrete values are replaced by sets of concrete values (which are equivalent to *properties* of concrete values). In this approach, the abstract semantics is homomorphic to the collecting semantics but not to the standard semantics; this clarifies the meaning of the abstract semantics and provides additional flexibility, since the collecting semantics can be designed in various ways. Additionally, the collecting and abstract domains are required to be complete lattices and are related by means of two monotonic functions: the *concretization* function Cc maps an abstract element to its meaning (i.e., the set of concrete values it denotes), while the *abstraction* function Abs maps a set of concrete values to its best approximation. The obtained mathematical structure is quite rich and the motivation for it is clear: it allows one to specify an abstract operation $\mathtt{O}_A$ in an optimal way as $Abs \circ \mathtt{O}_C \circ Cc$, where $\mathtt{O}_C$ is the "collecting" version of a standard operation $\mathtt{O}$.

Finally, note that in both the "homomorphic" and the "collecting" approach, the abstract semantics is (or can be) parameterized on the abstract domain and is defined by a fixpoint construction; assuming that the abstract domains are finite, the fixpoint can be computed in finite time by means of general purpose algorithms.

3.2 Some Problems With Strong Mathematical Frameworks

Strong mathematical frameworks of abstract interpretation based on denotational semantics or Galois insertions have some benefits. Notably, they allow researchers to prove some nice theoretical results (see, e.g., [13, 22]) or to specify some aspects of program analyses in an optimal way. Nevertheless these approaches also exhibit some drawbacks, a few of which we list in the rest of this section.

Restricted form of properties. In the homomorphic approach, it is necessary to require that the abstract and concrete domain share some mathematical structure, which makes it impossible to consider interesting program properties that are incompatible with such structure.

In logic programming, for example, many researchers concentrate on abstract domains that are *closed under instantiation*, since this allows them to implement abstract unification by means of the greatest lower bound operation; moreover, some works assume that the domains are *condensing*, i.e., that the analysis of any goal can be performed by (optimally) specializing the result of the analysis for a most general (i.e., totally uninstantiated) goal. Many useful abstract domains for logic programs do not respect these assumptions.

In functional programming, as a second example, the homomorphic approach requires that properties are *downwards closed* with respect to the ordering of the underlying Scott domain. This rules out many interesting properties.

Limited practicality of Galois insertions. The equation

$$\mathtt{O}_A \quad = \quad Abs \circ \mathtt{O}_C \circ Cc$$

is misleadingly simple because it does not provide an effective way of computing the $\mathtt{O}_A$ operation (as a matter of fact, all three operations Abs, $\mathtt{O}_C$, and Cc are non-computable in general, since they deal with infinite sets). (Part of) the craft of abstract domain design precisely amounts to deriving a practical *implementation* from the *specification* above. Our personal experience has shown that it is almost impossible for all but the simplest abstract domains to discover an implementation of $\mathtt{O}_A$ that exactly satisfies the equation; moreover, such an implementation could be so intricate that the optimality of the operation would be counterbalanced by a significant loss of efficiency. A typical example to which the considerations above apply is abstract unification (see, for instance, [5, 12, 19]. Moreover, to the best of our knowledge, no proof of optimality for an abstract unification algorithm has ever appeared in the literature (not even a proof of monotonicity).[4]

In our opinion, it must be concluded that Galois insertions do not provide an adequate framework for complex domain construction, and that weaker mathematical structures are better. As a final argument, notice that in frameworks based on Galois insertions, the abstract semantics is not an optimal abstraction of the collecting semantics but only a conservative abstraction thereof. (This is basically because the composition of two optimal abstract operations is not an optimal abstraction of the composition of the two corresponding collecting operations.) So why require the optimality of some operations if the global semantics is not optimal anyway? And why pretend that it is reasonable to optimally design operations such as abstract unification if one believes it to be unreasonable to build an optimal abstract semantics. We believe that accuracy of program analyses is (and can only be) a pragmatic issue guided by intuition and practical experiments.

[4] As a matter of fact, optimality implies monotonicity.

Finite abstract domains. Strong abstract interpretation frameworks often require that the abstract domain is finite, because otherwise the abstract semantics is not finitely computable. Some authors [13] even argue that no additional accuracy can be obtained by considering infinite domains. This is only true because these authors presuppose a "homomorphic" framework. This argumentation has been debunked by the Cousots in [10]. They show that the use of widening operators allows more powerful, yet finite and efficient analyses.

However, there are examples of useful (infinite) abstract domains that are not even complete partial orders [6, 18]. With such domains, widening operations are needed not only to enforce convergence but also as the only practical way of defining the abstract semantics.

Composite abstract domains. As the state of the art of program analysis evolves, it becomes more and more evident that the design of ambitious analyses requires the use of complex abstract domains made of several "communicating" components. Some proposals of combining abstract domains have been made recently, some of which are along the lines of the reduced product originally introduced by the Cousots (see, e.g., [8]). It is worth understanding that operations on abstract domains similar to the reduced product are purely theoretical and provide only specifications of (hypothetical) abstract domains that are still to be designed.

Nevertheless, some practical methods of combining abstract domains have also been proposed [5]. They only require weak mathematical assumptions for the domains, and their accuracy has only been demonstrated experimentally. Once again, this more pragmatic approach seems better to us, since it considers both correctness and practicality issues.

Conclusion. The abovementioned problems suggest to us that a practical approach to the design of complex program analyses can only be based on weak mathematical assumptions about the used abstract domains. In the next section, we elaborate further on this opinion in the light of the experiments conducted with the abstract interpretation system *GAIA*.

4 The Practical System *GAIA*

In this section, we draw some lessons from the history of the abstract interpretation system *GAIA*. We aim at showing that the success of this system, which is demonstrated by the large number of publications related to it,[5] is due to a combination of theory and practice, which essentially relies on the use of explicit specifications. We also show that the evolution of the system was made possible only by weakening the underlying mathematical framework.

Why and how GAIA was successful. The original design of *GAIA* was based on a clear distinction between three aspects: the abstract domains, the abstract semantics, and the fixpoint algorithms. These aspects are well separated in the

[5] See [18] for a bibliography.

implementation, allowing one to modify or replace any component independently of the others. It is worth mentioning that the original version published in [19] was found correct and efficient at once, without any backtracking. Pascal Van Hentenryck, the implementor of the original system, only took the research reports [15, 16, 21] and coded the algorithms and the domains in *C*. The correctness of the system came from the fact that all correctness issues related to the abstract domains, the abstract semantics, and the fixpoint algorithms had been explicitly proven before. The ease of the coding in *C* and the efficiency came from the fact that the chosen abstraction level for the description of abstract substitutions and abstract operations allowed both a straightforward coding and an explicitly written correctness proof.

We think that this approach is different from others, which ignore the explicit use of specifications. In the operational framework [2], for instance, only the two abstract operations `entry` and `exit` are considered. It can be argued that using two operations only is a conceptual simplification. However, the granularity of the operations is so coarse that it is impossible to specify them simply, making it difficult to prove the correctness of instances of the abstract operations. The abstract semantics underlying the original version of *GAIA* uses six abstract operations, each of them having clear and explicit specifications that make explicit proofs possible.

The evolution of GAIA. The theoretical research preliminary to *GAIA* originated from the study of Maurice Bruynooghe's paper [3]. It quickly became apparent to the first author of this paper that a separation between the abstract semantics and the fixpoint algorithm would simplify the correctness proofs. So the framework was structured into three parts as mentioned in the previous paragraph. The first framework was in fact based on a classical collecting semantics approach, but assuming only a concretization function, since some of Bruynooghe's domains were not equipped with an abstraction function. Hence the correctness proofs of abstract operations were only based on the concretization function. Since the existence of an abstract fixpoint requires monotonicity of the abstract operations, it was attempted to prove such properties. This appeared to be of an unexpected difficulty, and the proof attempts were finally given up since it was discovered that, in the absence of monotonicity, the fixpoint algorithm could be modified to compute a postfixpoint that is correct as well.

Based on the first — correct — version of the system, many other experiments and variations were performed in terms of abstract domains, abstract semantics, and fixpoint algorithms. Among other experiments, it is worth mentioning the reexecution semantics [20], which is a collecting semantics that is very different from the standard semantics of Prolog, as it allows one to improve the accuracy of the analysis in a tremendous way for some abstract domains.

A more recent version of *GAIA* [1, 17, 18] is based on a denotational semantics of Prolog and allows one to derive precise information about the cut and the number of solutions to a goal. It is interesting to note that, although this version of the system uses a completely different "standard" semantics, most of the

abstract operations are defined as upgraded versions of previous ones. This is possible only because of the systematic use of specifications, since a correct reuse of previous operations can be based on explicit reasoning. Moreover, the mathematical framework used by this version is still weaker than the previous ones, since there is no abstract fixpoint at all, even in theory. Finally, this analysis considers properties of higher-order objects (i.e., substitution sequences) that are not downwards closed, contrary to many frameworks based on denotational semantics.

A limitation of GAIA. To date [24], more than seven (substantially) different versions of *GAIA* have been completed in order to conduct various experiments in logic and constraint logic program analysis. They amount to more than 50,000 lines of *C* code. Thus, in its current form, *GAIA* is not a completely generic system, but rather a series of systems that were rather straightforwardly derived from the first original version. Although the objective of a "real" generic system seems a priori desirable, we believe that it is in fact difficult to foresee all future applications of *GAIA*, since experience has shown that every new experiment may entail unexpected variations of the framework; thus we think that it is probably wiser to continue moving from one version to another one, based on experiments. However, this is wise only if we keep a clear understanding of the system, based on good specifications of all its parts.

5 Conclusion

Is abstract interpretation a unified framework? In this paper, we have mostly answered this question negatively. Much of our argumentation is personal opinion, but it is based on actual and largely successful experiments with a practical system. Nevertheless, notions such as Galois insertions, collecting semantics, and widening operators provide deep insights into how to properly design a program analysis. Once these notions are well understood, one can relax or modify them in order to find the framework that is best adapted to one's goals [9]. Finding the ultimate framework is an endless quest!

Acknowledgements

Baudouin Le Charlier wishes to thank Pascal Van Hentenryck for his invaluable contributions to *GAIA*.

References

1. C. Braem, B. Le Charlier, S. Modard, and P. Van Hentenryck. Cardinality analysis of Prolog. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.
2. M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

3. M. Bruynooghe, G. Janssens, A. Callebaut, and B. Demoen. Abstract interpretation: Towards the global optimization of Prolog programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, California, August 1987. Computer Society Press of the IEEE.
4. P. Codognet and G. Filé. Computations, abstractions and constraints in logic programs. In *Proceedings of the fourth International Conference on Computer Languages (ICCL'92)*, Oakland, U.S.A., April 1992.
5. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combination of abstract domains for logic programming. In *Proceedings of the 21th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.
6. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 23(3):237–278, June 1995.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992.
9. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
10. P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation (invited paper). In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Workshop on Programming Language Implementation and Logic Programming (PLILP'92)*, Lecture Notes in Computer Science, Leuven, August 1992. Springer-Verlag.
11. R.A. De Millo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979. Comments in *Communications of the ACM*, 22(11):621–630, 1979.
12. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.
13. R.B. Kieburtz and M. Napierala. Abstract semantics. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 7, pages 143–180. Ellis Horwood Limited, 1987.
14. B. Le Charlier and P. Flener. Specifications are necessarily informal, or: Some more myths of formal methods. Accepted by *Journal of Systems and Software, Special Issue on Formal Methods Technology Transfer*, forthcoming.
15. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. Efficient and accurate algorithms for the abstract interpretation of Prolog programs. Technical Report 37/90, Institute of Computer Science, University of Namur, Belgium, 1990.
16. B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A generic abstract interpretation algorithm and its complexity analysis. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming (ICLP'91)*, Paris, France, June 1991. MIT Press.
17. B. Le Charlier, S. Rossi, and P. Van Hentenryck. An abstract interpretation framework which accurately handles Prolog search-rule and the cut. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS'94)*, Ithaca NY, USA, November 1994. MIT Press.

18. B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-Based Abstract Interpretation of Prolog. Technical Report RR-97-001, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, January 1997.
19. B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(1):35–101, January 1994.
20. B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.
21. K. Musumbu. *Interprétation Abstraite de Programmes Prolog.* PhD thesis, Institute of Computer Science, University of Namur, Belgium, September 1990. In French.
22. U.S. Reddy and S.N. Kamin. On the power of abstract interpretation. In J. Cordy, editor, *Proceedings of the IEEE fourth International Conference on Computer Languages (ICCL'92)*, Oakland, U.S.A., April 1992. IEEE Press.
23. J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge Mass., 1977.
24. P. Van Hentenryck. Personal communication, June 1997.

This article was processed using the LaTeX macro package with LLNCS style