On the Design of Generic Abstract Interpretation Frameworks

Baudouin Le Charlier
University of Namur,
21 rue Grandgagnage, B-5000 Namur (Belgium)
Email: ble@info.fundp.ac.be

Pascal Van Hentenryck
Brown University,
Box 1910, Providence, RI 02912 (USA)
Email: pvh@cs.brown.edu

Abstract

Abstract interpretation is a general methodology for building static analyses of programs. It was introduced by P. and R. Cousot in [5]. The original idea was subsequently adapted, reformulated, and applied to many programming paradigms. As a consequence, this field of research is currently rather fragmented (although very active).

We present, in this paper, the main methodological aspects of a general approach which is widely applicable and was shown very effective for logic programs. We compare it to some previous proposals and argue that this approach is unifying.

1 Introduction

Abstract Interpretation [1, 5] was originally designed by P. and R. Cousot as a *general methodology* for building static analyses of programs. The original ideas have been subsequently developed by many researchers and mainly applied to declarative languages (more amenable to optimization). Although Abstract Interpretation is claimed to provide a unifying approach many different frameworks have been proposed. In the single field of Logic Programming one can cite, among many others, [2, 4, 8, 9, 11, 15, 16, 19, 22, 27].

The objective of our paper is to discuss a number of important issues for the design of abstract interpretation frameworks and to contribute to reunification of the field of abstract interpretation. We mainly discuss abstract interpretation for Prolog because it is clearly the most flourishing and anarchic application field and because we have been working mainly on it. Nevertheless most of the presented issues are of general relevance. The paper is divided in two parts. In section 2, we give a methodological overview of our approach. We insist of its wide applicability and theoretical soundness. We recall also some experimental results to illustrate its effectiveness. Section 3 is devoted to a comparison study with other well-known approaches. We argue that they can be understood and classified with respect to ours which can therefore be seen as a unifying one.

2 A fixpoint approach to the design of abstract interpretation frameworks

In this section, we present an approach to the design of abstract interpretation frameworks that is

1. *theoretically sound*: it encompasses systematic methods for proving correctness of applications;
2. *widely applicable*: it is not restricted to a particular class of languages or analyses;
3. *effective*: it has been used to design pratical systems that have shown efficient and accurate for static analysis of Prolog [6, 11, 12, 13].

A first idea is that packages for static analysis should consists of one or several *generic abstract interpretation algorithms* parametrized on an abstract domain and of one or several implemented *abstract domains* with respect to which the algorithms can be instantiated. This is, for example, the approach of [2]. The approach can be further generalized if generic algorithms are viewed as the instantiation of a *universal fixpoint algorithm* to a particular *fixpoint semantics* (see [14]).

2.1 Abstract fixpoint semantics.

2.1.1 The approach

Static program analyses aim at deriving information about the actual operational behaviour of programs. This information is in fact completely determined by the standard operational semantics of the programming language. It is nevertheless convenient to use another, non standard semantics, as a basis for performing the analyses, because it is in practice impossible to "generate and analyse" all possible execution traces for a given program.

There are several approaches to the choice of the non standard semantics (see a discussion at paragraph 3.1). We basically follow the original approach of P. and R. Cousot. Their so-called *static* semantics provides a fixpoint characterization of the relevant information. (The *collecting* semantics terminology is nowadays more widely accepted).

Let us sketch the approach from a general point of view. The operational semantics of any programming language consists of (or can be rephrased as)

1. a set of states Σ (states are noted σ);
2. an immediate transition relation between states denoted $\longmapsto$ where $\sigma \longmapsto \sigma'$ means that σ' is a possible successor state of σ;
3. a unary predicate on states, denoted final(σ), meaning that execution terminates in state σ.

From this semantics one can derive several fixpoint semantics, that "collect" different kinds of relevant information. For example the following semantics characterizes the set Output(S) of final states reachable from an initial set of states S:

$$\text{Output}(S) \quad = \quad \{\sigma : \sigma \in S \ \& \ \text{final}(\sigma)\} \quad \cup \quad \text{Output}(\{\sigma' : \sigma \in S \ \& \ \sigma \longmapsto \sigma'\})$$

This recursive definition is not computable by usual recursive evaluation for two reasons at least:

1. S can be infinite;

2. sequences of transitions can be infinite.

However it can be given a mathematical meaning as the least fixpoint of transformation τ such that

$$(\tau f)(S) \quad = \quad \{\sigma : \sigma \in S \ \& \ \text{final}(\sigma)\} \quad \cup \quad f(\{\sigma' : \sigma \in S \ \& \ \sigma \longmapsto \sigma'\})$$

It is straightforward to show from the operational semantics that this definition correctly maps each set of states onto the set of final states reachable from them. The exact and uncomputable fixpoint semantics can then be approximated as follows. First we define a set A of abstract states. An abstract state α is a finite representation of a set S of "concrete" states. We note Cc the function that maps abstract states on their meaning (therefore: $S = \text{Cc}(\alpha)$). The next step is to derive an "abstract" version τ_A of τ that maps monotonic functions from $A \to A$ to $A \to A$. This transformation correctly abstracts τ if the following consistency[1] condition holds:

$$\left.\begin{array}{c} \forall f : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma) \\ \forall f_A : A \to A \\ \forall \alpha \in A \end{array}\right\} : (\forall \alpha' \in A : f(\text{Cc}(\alpha')) \subseteq \text{Cc}(f_A(\alpha'))) \Rightarrow (\tau f)(\text{Cc}(\alpha)) \subseteq \text{Cc}((\tau_A f_A)(\alpha)).$$

Consistency "simply" means that $\tau_A f_A$ safely approximates τf provided that f_A safely approximates f. If τ_A is furthermore monotonic its least fixpoint exists and safely approximates the collecting semantics:

$$\forall \alpha \in A : \mu\tau(\text{Cc}(\alpha)) \subseteq \text{Cc}(\mu\tau_A(\alpha))$$

(where μ stands for the least fixpoint operator.) An *abstract semantics* for the language L is a set of rules to derive transformation τ_A for any program (of L). It is basically defined by mimicking the collecting semantics definition and replacing "concrete" operations by "abstract" ones. In the case of our simplified language, we define two global operations[2]:

$$\begin{array}{lcl} \text{FS}(S) & = & \{\sigma : \sigma \in S \ \& \ final(\sigma)\}, \\ \text{DR}(S) & = & \{\sigma : \exists \sigma' \in S : \sigma' \longmapsto \sigma\}. \end{array}$$

We can then rephrase the definition of τ :

$$(\tau f)(S) = \text{FS}(S) \cup f(\text{DR}(S)).$$

The abstract semantics is identical to the concrete one except that concrete operations are replaced by their abstract counterpart: $\text{FS}_A, \text{DR}_A, \cup_A$:

$$(\tau_A f_A)(\alpha) = \text{FS}_\text{A}(\alpha) \cup_A f_A(\text{DR}_\text{A}(\alpha)).$$

τ_A is consistent provided that the abstract operations are. That is :

$$\begin{array}{lcl} \forall \alpha \in A & : & \text{FS}(\text{Cc}(\alpha)) \subseteq \text{Cc}(\text{FS}_A(\alpha)) \\ \forall \alpha \in A & : & \text{DR}(\text{Cc}(\alpha)) \subseteq \text{Cc}(\text{DR}_A(\alpha)) \\ \forall \alpha_1, \alpha_2 \in A & : & \text{Cc}(\alpha_1) \cup \text{Cc}(\alpha_2) \subseteq \text{Cc}(\alpha_1 \cup_A \alpha_2) \end{array}$$

[1] or safeness or safety ...

[2] FS and DR stand for "Final States" and "Directly reachable", respectively.

$$\beta_{in}\ p(x_1,\ldots,x_n)\ \leftarrow\ \beta_0\ g_1\ \beta_1,\ \ldots\ ,g_m\ \beta_m.\ \beta_{out}$$

Figure 1: Abstract substitutions for a clause

Such an abstract semantics is *generic* with respect to the abstract domain: it does not depend on the particular choice of $A, \mathrm{FS}_A, \mathrm{DR}_A, \cup_A$. This is an important feature because it allows to reuse the same semantics (and therefore all the algorithms derived from it) for many different applications. Note that the collecting semantics is a particular instance of the abstract semantics. Alternatively it is possible to define a generic abstract semantics without relying on an explicit collecting semantics. This only requires to modify the consistency definitions for abstract operations: $\mathrm{FS}_A, \mathrm{DR}_A$ and $\cup_A$ are now consistent if and only if :

$$\forall\sigma\in\Sigma:\forall\alpha\in A:\ \left.\begin{array}{c}\sigma\in\mathrm{Cc}(\alpha)\\ \mathrm{final}(\sigma)\end{array}\right\}\Rightarrow\sigma\in\mathrm{Cc}(\mathrm{FS}_A(\alpha))$$

$$\forall\sigma,\sigma'\in\Sigma:\forall\alpha\in A:\ \left.\begin{array}{c}\sigma\in\mathrm{Cc}(\alpha)\\ \sigma\longmapsto\sigma'\end{array}\right\}\Rightarrow\sigma'\in\mathrm{Cc}(\mathrm{DR}_A(\alpha))$$

$$\forall\sigma\in\Sigma:\forall\alpha_1,\alpha_2\in A:\ \left.\begin{array}{c}\sigma\in\mathrm{Cc}(\alpha_1)\\ \text{or}\\ \sigma\in\mathrm{Cc}(\alpha_2)\end{array}\right\}\Rightarrow\sigma\in\mathrm{Cc}(\alpha_1\cup_A\alpha_2)$$

Basic operations from the standard operational semantics are now used instead of the "collecting" ones: FS, DR. Relying on basic operations allows for simpler definitions in practice. (Compare, for example, the collecting semantics in [11] with the instrumental semantics in [13].) However the later approach can look less systematic because consistency definitions do no longer fit in the same formal scheme. It must be noticed that a collecting semantics still exists implicitely: it can be defined as the more precise (or "concrete") instantiation of the generic semantics. Simplicity stems from the fact that it is no longer *explicitely* defined.

2.1.2 Real size definitions.

The above presentation looks very simple because it was assumed that relation $\longmapsto$ and predicate final are primitive operations of the language. In real size applications those (complex) operations are defined by means of (possibly many) simpler primitives and the abstract semantics uses operations abstracting those simpler primitives. Real size abstract semantics are therefore more complex but the design principle is entirely the same.

2.1.3 Core semantics versus collecting semantics.

In actual applications one is also interested in a richer collecting semantics than the input/output one sketched in the example above.

As a motivating example, let us consider abstract interpretation of Prolog. Most applications require to derive information about concrete substitutions at each program point. Let us note $\beta_0,\ldots,\beta_n$ the corresponding pieces of information for a given clause (see figure 1). They are

called *abstract substitutions*. In this figure, $\beta_{in}(\beta_{out})$ abstractly describes the set of input(output) substitutions for the clause.

The abstract input/output semantics that we used in [6, 11, 12][3] only defines the set of tuples $(\beta_{in}, p, \beta_{out})$. $\beta_0, \ldots, \beta_i, \ldots, \beta_m$ are generated during the computation. They are not collected however. It can be pointed out that the input/semantics can be used as an *oracle* to compute the abstract substitutions at each program point. Therefore it is relevant to distinguish between two generic semantics. We call *core* semantics the "most fundamental" one. It is generally an input/output fixpoint semantics and plays the role of an oracle for the computation (definition) of (possibly several) richer *collecting* semantics.

Distinction between core and collecting semantics can be traced back to [8]. However we introduce an important novelty: the collecting semantics is no longer a fixpoint one. This allows for simpler definitions as well as for faster and simpler algorithms (see paragraph 3.1). In the following we use the core and collecting semantics terminology to speak of generic abstract semantics in the above mentioned sense. We do not restrict the terminology to the "most concrete" version of them as in [8].

2.1.4 Best core semantics and instrumental operational semantics

Core semantics can be defined in several semantically equivalent ways. However different definitions can induce differences in efficiency at the algorithmic level. Let us consider Prolog again. Our core semantics characterizes the input/output behaviour of programs at the *procedure level*. A semantically equivalent alternative consists in defining it at the *goal level* (recall that a goal is a sequence of atoms).

For algorithmic issues however the two approaches require very different kind of loop checks to avoid infinite computations: it is more effective to test equality of two abstract procedure calls than equality (or equivalence) of two abstract goals.

Choosing a more effective core semantics may nevertheless entail more work to prove consistency because the core semantics can be less close to the standard operational semantics. It is typically the case of our core semantics for Prolog: the standard operational semantics (SLD-resolution) is based on the *goal* notion, not on the *procedure* notion. It is then useful to define an *intermediate (instrumental) operational semantics* allowing to break the consistency proof in two independent parts (see [10, 13]).

2.1.5 More elaborate core semantics

Although abstract interpretation is generally presented as "simply mimicking the concrete computation on an abstract domain", it can be convenient to define (core) abstract semantics that are very far from simulating the actual operational behaviour of the language.

A very interesting example is the "reexecution semantics" that we presented in [13]. It embodies the idea that recomputing abstractly a goal can improve the accuracy of the analysis. This semantics was used to derive new and more effective algorithms. Many other examples can be imagined: we are presently working on a *forward/backward semantics* which provides information about *terminating* computations.

[3] It was first proposed in [20]; its consistency proof is also presented in [10].

2.2 Generic algorithms

In our approach, the derivation of a generic abstract interpretation algorithm is based on a top-down universal algorithm for fixpoint computation. This algorithm is used to compute the core (input/output) semantics. The collecting semantics is computed afterwards and uses the values of the core semantics (to solve recursive calls). Several optimizations are applicable to the algorithms. Their usefulness depends on the particular semantics and abstract domain.

2.2.1 Top-down universal fixpoint algorithms

A core semantics is (very close to) a functional recursive definition of the form

$$f(\alpha) = expr\langle f, \alpha \rangle$$

where expression *expr* only uses f, α and abstract operations. Therefore techniques for recursive function computation can be adapted to compute this core semantics. We present in [14] a very general algorithm which extends those techniques to the computation of the least fixpoint of an arbitrary functional transformation (provided that it is computable by a (higher-order) procedure `tau(f:A→A; α:A):A`). The main advantage of this algorithm is that it focuses only on the part of the fixpoint relevant for the computation of the output $f(\alpha)$ corresponding to a given initial value (α). (See paragraph 3.2 for a comparison with other fixpoint algorithms).

2.2.2 Computing the abstract semantics

An algorithm to compute the core semantics is derived by instantiating the universal algorithm to it.

We presented in [11] a generic abstract interpretation algorithm designed thanks to this method. It uses the core semantics first proposed in [20]. A more elaborate algorithm, embodying the idea of reexecution, was derived by the same method from a reexecution core semantics [13].

Those algorithms compute and store in a data structure the part of the core semantics which is relevant for a given abstract input. The collecting semantics is computed afterwards in a straightforward way (without any fixpoint computation). This "two passes" approach is simpler and more efficient than the usual one. In [13] we applied the method to collect information about *input* and *output modes* and *built-in specializations*. Each program clause is (abstractly) reexecuted only once to collect the needed information. Therefore the collecting part is not very time consuming (see a comparison with other approaches in paragraph 3.1).

2.2.3 Optimizations

Correctness is a major and difficult issue in abstract interpretation because final implementations are long and very complex (8000 lines of C code for each algorithm described in [6]). It is therefore very useful to organize the correctness proof (or at least the argumentation) in several independent parts. In particular, algorithmic optimizations should not interfere with semantic issues. In [11] we presented a generic algorithm for Prolog which exhibits two new optimization techniques and an older one (due to P. Cousot):

1. a *dependency graph* which allows to prune useless iterations at the procedure level;

2. two operations *Extend* and *Adjust* which exploit monotonicity of the least fixpoint to faster derive better approximations;

3. a *widening* operation to ensure termination on infinite domains and to speed up computation on finite ones.

We showed in [14] that those techniques are applicable to any kind of fixpoint computation (and hence to any abstract interpretation). So they can be reused systematically. Two other optimizations were proposed and used for Prolog in [6]:

1. *Caching* is a general technique applicable when abstract operations are very time consuming and have good chances to be recomputed; it can be handled independently of other algorithmic aspects.

2. *Prefix optimization* is an improvement of the dependency graph at the clause level. It is an instance of a more general paradigm: detecting parts of the transformation code the recomputation of which is useless.

3 Discussion and Comparison with related works

An amazingly large number of authors have proposed frameworks for abstract interpretation (of logic programs, in particular). In this section, we compare our approach with some aspects of those related works. For the sake of simplicity we mainly focus on logic programming.

3.1 Framework style

3.1.1 Fixpoint approach

Our approach to abstract interpretation is close to the original proposal of P. and R. Cousot [5]. However, we insist more on generic aspects and algorithm reusability whereas the Cousots' approach is mainly methodological. The similarity lies in the two following aspects :

1. fixpoint semantics is used to characterize the information to be computed;

2. the algorithms to compute the fixpoint are designed independently of the original operational semantics.

We call this the *fixpoint approach*. It allows to separate cleanly semantic and algorithmic aspects (what is to be computed versus how to compute it). Correctness proofs are therefore made easier and safer because they are broken in two independent parts. Moreover the semantic part can be easily changed without redesigning all the algorithmic part. Conversely algorithms can be chosen, modified and optimized independently of what is computed. We compare now the fixpoint approach to other important ones.

3.1.2 Operational approach

The *operational approach* basically applies the idea that abstract interpretation should simply *mimick* the concrete computation on the abstract domain. The main virtue of this approach is

its simplicity. It is also easy to understand. However it suffers from a major drawback: semantic and algorithmic aspects of abstract interpretation are intermingled. Therefore it is difficult to modify the semantic part without redesigning the algorithms or to introduce optimizations without reworking the semantic part. Moreover the abstract computation follows the concrete one which is not always the more efficient way to compute the abstract results.

This approach is followed by D.S. Warren [25] and P. Codognet and G. Filé [4], for logic programs. The logic program to be analyzed is first translated into an abstract program which performs the abstract interpretation of the original one. In [25] OLDT-resolution [24] is used to execute the abstract program. A similar mechanism is used by [4].

The two above proposals are very systematic and can be effective and easy to use in practice. We prefer our approach however for the following reasons :

1. fixpoint semantics are more powerful than abstract programs to express what is to be computed.

2. OLDT-resolution is already a particular instance of a more general algorithm applied to logic programs. So it is too low level to be used for other kinds of fixpoint computation.

There are also some problems with the collecting semantics: a simple abstract computation does not collect information about the intermediate steps. This is overcome in [4] by using an AND/OR tree summarizing the computation history. A similar approach was previously proposed by Bruynooghe [2, 3]. This solution is very memory consuming however and can restrict the method applicability.

3.1.3 Denotational Frameworks

The denotational approach to abstract interpretation was first introduced by F. Nielson [21]. It was then further developed by many authors including [8, 18, 26]. In this approach, the elegant notations of denotational semantics and the mathematical framework of the Cousot's are put together to provide high level descriptions of static analyses. Authors relying on this approach are mostly concerned with semantic issues. They assume the existence of well-accepted and efficient algorithms for fixpoint computation. O'Keefe algorithm (see paragraph 3.2.1) is often referred to as well as algorithms designed for deductive data bases and OLDT-resolution. Our universal algorithm provides a versatile alternative to O'Keefe's algorithm to implement the denotational approach given its wide applicability.

3.2 Algorithms for fixpoint computation

In this section we compare our universal algorithm for fixpoint computation to other proposals.

3.2.1 Bottom-up Fixpoint Algorithms

In our approach, we view the computation of the least fixpoint of a monotonic transformation as a generalization of computing the values of a function from its recursive definition. This leads to a top-down approach in contrast to the more usual approach inspired by the Kleene's sequence. We now compare it with a representative bottom-up algorithm proposed by O'Keefe [23].

O'Keefe's algorithm solves finite sets of equations of the form $x_i = expr_i \ \ (1 \leq i \leq n)$ where $x_1, \ldots, x_n$ are distinct variables, ranging on lattices of finite depth $T_1, \ldots, T_n$, and $expr_1, \ldots, expr_n$ are well-typed monotonic expressions possibly involving $x_1, \ldots, x_n$. The algorithm computes the least fixpoint of the transformation

$$\begin{array}{rl} \tau: & T_1 \times \ldots \times T_n \mapsto T_1 \times \ldots \times T_n \\ & \langle x_1, \ldots, x_n \rangle \rightsquigarrow \langle expr_1, \ldots, expr_n \rangle \end{array}$$

It proceeds as follows. Variables x_1 to x_n are initialized to $\bot$ and pushed onto a stack. Then variables are popped from the stack until it becomes empty. Each time a variable x_i is popped its value is recomputed. If the new value is greater than the previous one, all variables that depends on x_i and are not on the stack are pushed on it.

As far as applicability is concerned, it is clear that O'Keefe's algorithm addresses a considerably more restrictive class than ours. It is nevertheless possible to adapt it to deal with more general problems as follows. Consider the problem of solving recursive equations of the form

$$f(x) = expr\langle f, x \rangle$$

on a finite lattice T, where *expr* is an expression built from f, x and constant symbols denoting monotonic functions. To compute $f(v)$ where $v \in T$, we can partially evaluate $expr\langle f, v \rangle$ and identify other subexpressions, say $f(w)$ for which we iterate the process. We obtain a finite set of equations and O'Keefe's algorithm is applicable. The problem is more difficult when f appears in subexpressions of expressions of the form $f(expr_1, \ldots, expr_n)$. In order to evaluate the subexpressions we can initialize $f(v)$ to the bottom element of T for each $v \in T$. So a first set of equations can be derived and solved. The solution is used to derive a new set of equations which is then solved, and so on until no greater values are obtained for f. This complicated process should be contrasted with the application of our algorithm to a procedure `tau` defined as follows:

function `tau(f:T→T; α:T):T;`
  begin `tau:=expr⟨f,α⟩` **end**.

The previous discussion can also be related to the *minimal function graph semantics* discussed below.

Minimal Function Graph and Query Directed Semantics To overcome the applicability problem of O'Keefe's algorithm (and more generally of any bottom-up evaluation), many authors design instrumental semantics introducing supplementary results needed only by the algorithm for fixpoint computation. They are called *minimal function graph semantics* in [7, 26] and *query directed semantics* in [17]. The basic idea is that each iteration provides the set of input values for the next iteration in addition to the output values corresponding to the previous inputs. Such an instrumental semantics is no longer needed with our approach. However, the additional results provided by those derived semantics are sometimes useful for applications. For example, the minimal function graph semantics is useful for program specialization [26]. The same information is obtained for free by our algorithm.

3.2.2 Chaotic Iteration Algorithms

Chaotic iteration was introduced by P. Cousot in [5] as a very general class of methods for finding the least solution of a set of equations of the form $x_i = expr_i \quad (1 \leq i \leq n)$. A chaotic iteration strategy for such a set of equations defines a finite sequence $i_1, \ldots, i_m$ such that the "loop":

$(x_1, \ldots, x_n)$`:=`$(\bot, \ldots, \bot)$`;`
for `j:= 1 to m do` x_{i_j} `:=` $\texttt{expr}_{i_j}$`<`$x_1, \ldots, x_n$`>`

compute the least solution of the system. The best strategy minimizes the amount of computation (roughly speaking, the number m of iterations). Clearly, this is very problem dependent. O'Keefe's algorithm provides a standard (bottom-up) chaotic iteration strategy while ours provides a top-down one. However, chaotic iteration also needs a minimal function graph or a query directed semantics.

3.3 Granularity

Granularity is a useful criterium to compare Abstract Interpretation algorithms and frameworks. It is specified by giving the "program points" where information about all possible executions is considered. It is *finer* if more program points are considered, *coarser* otherwise. It is *static* if the pieces of information corresponding to all execution paths leading to the same point are lumped together. Static granularity has the advantage that a finite set of equations can be associated with the analyzed program. *Dynamic* granularity is achieved when different pieces of information can be associated with the same program point, possibly corresponding to different execution paths. For a fixed abstract domain, a finer granularity is likely to give more precise results at the price of a higher computation cost. Similarly dynamic granularity is a priori more accurate and costly than static granularity. Usefulness of granularity to compare abstract interpretation frameworks can be illustrated by the works of C. Mellish [19], U. Nilsson [22] and M. Bruynooghe [2]. All are frameworks for Abstract Interpretation of (pure) Prolog.

The earlier work is due to C. Mellish. It has static granularity and only two program points are considered for each procedure: procedure entry and procedure exit. The abstract semantics can be expressed as system of $2n$ equations with $2n$ variables where n is the number of procedures. As this system is small the naive bottom-up method based on the Kleene's sequence can be used. Mellish's algorithm is based on it. O' Keefe's paper [23] was motivated by Mellish's work and provides a faster method.

The framework of U. Nilsson has static but finer granularity: all program points are considered (clause entry and exit, and any point between two calls). This results in a system of m equations with m variables where m is the number of program points. The algorithm of U. Nilsson is essentially O'Keefe's one. It is easy to exhibit programs where it gives more precise results than Mellish's algorithm.

Bruynooghe's framework uses the same program points as Nilsson's one but has dynamic granularity. It defines the same abstract information as the algorithm that we presented in [11].

Finest granularity is in fact obtained when the universal algorithm is applied to a core semantics without introducing any widening operation. It is however possible to derive abstract interpretation algorithms with the same granularity as Mellish's and Nilsson's ones by instantiating the universal algorithm to appropriate (less simple) instrumental semantics.

An other approach to be investigated in the future is the definition of universal algorithms that should automatically provide a coarser granularity when instantiated to a core semantics. Those algorithms should use widening techniques and/or introduce "hooks" at some "semantic points" to control the granularity level.

It can therefore be concluded that semantic and algorithmic issues to abstract interpretation are in some sense dual. Tuning the granularity can be done at both levels just as a matter of convenience.

3.4 Mathematical assumptions

Theoretical presentation of abstract interpretation often refers to the *Galois insertion* notion. A Galois insertion is a 4-uple $(C, A, \mathrm{Cc}, \mathrm{Abs})$ where C and A are complete lattices and $\mathrm{Cc} : A \to C$ and $\mathrm{Abs} : C \to A$ are monotonic functions such that :

$$\begin{array}{lll} \forall \alpha \in A : & \mathrm{Abs}(\mathrm{Cc}(\alpha)) & = \alpha, \\ \forall \gamma \in C : & \mathrm{Cc}(\mathrm{Abs}(\gamma)) & \geq \gamma. \end{array}$$

Using Galois connections is convenient to build an elegant theory for abstract interpretation. The drawback for practical applications is however that a lot of properties must be proven about abstract domains and abstract operations. Fortunately it can be shown that some mathematical assumptions can be dropped in practice, in particular *monotonicity assumptions*. In fact only *consistency* assumptions are essential. We sketch a simplified proof of this. The reader is referred to [14, 20] for further details.

Let C and A be the concrete and abstract domains. In the simplest case, the concrete and abstract version of the core semantics are given via two transformation:

$$\begin{array}{l} \tau : C \to C \\ \tau_A : A \to A \end{array}$$

The concrete semantics is the least fixpoint of τ, noted $\mu\tau$. The goal of abstract interpretation is to find $\alpha \in A$ which safely approximates $\mu\tau$. That is:

$$\mu\tau \leq \mathrm{Cc}(\alpha).$$

(Note that the relation holds in the *concrete* domain.) We show that existence of such an element depends on the *consistency* of τ_A with respect to τ. That is:

$$\forall \alpha \in A : \tau(\mathrm{Cc}(\alpha)) \leq \mathrm{Cc}(\tau_A(\alpha)).$$

We can assume that τ is monotonic and even continuous because C is a concrete domain where those properties follow from the algorithmic nature of the language to be analyzed. It is also assumed in general that τ_A and Cc are monotonic. In fact one of the two assumptions can be dropped (but not both).

Theorem 1 If τ_A is monotonic but Cc is not, *any* fixpoint of τ_A safely approximates $\mu\tau$.

Proof Consider the increasing sequence $f_0, f_1, \ldots, f_i, \ldots$ of elements of C, defined as follows :

$$\begin{array}{lcl} f_0 & = & \bot \\ f_{i+1} & = & \tau f_i \quad (i \geq 0) \end{array}$$

Let α, a fixpoint of τ_A. The following holds for all $i \geq 0$:

$$f_i \leq \mathrm{Cc}(\alpha).$$

The result is obvious if $i = 0$. Suppose $i > 0$. Then

$$\begin{array}{lcll} f_i & = & \tau f_{i-1} & \\ & \leq & \tau(\mathrm{Cc}(\alpha)) & \text{(Induction hypothesis)} \\ & \leq & \mathrm{Cc}(\tau_A(\alpha)) & \text{(Consistency of } \tau_A\text{)} \\ & = & \mathrm{Cc}(\alpha) & (\alpha \text{ is a fixpoint of } \tau_A.) \end{array}$$

Therefore

$$\mu\tau \quad = \quad \bigsqcup_{i=0}^{\infty} f_i \quad \leq \quad \mathrm{Cc}(\alpha).$$

□ Theorem 1 shows that the ordering in A can be unrelated to the ordering in C provided that an *exact* fixpoint can be computed.

Theorem 2 Let us suppose that Cc is monotonic but τ_A is not. Let α be any post fixpoint of τ_A (that is: $\alpha \geq \tau_A(\alpha)$). Then α safely approximates $\mu\tau$.

Proof The proof is similar to the previous one but uses the facts that Cc is monotonic and α is a post fixpoint instead of the hypothesis that α is a fixpoint. □
Theorem 2 is very useful in practice because monotonicity of τ_A is often hard to prove (and is often "forgotten" by designers of practical applications). Moreover postfixpoints can be easily computed under very general assumptions on A.

4 Conclusion

In this paper, we have presented a general approach to the design of abstract interpretation frameworks and algorithms. This approach is systematic, theoretically sound and widely applicable. Our work has also been related to some other proposals in order to emphasize some of its benefits.

Although the proposed approach may look complex, we believe it is worth using it because it provides more systematic tools and methods for generic abstract interpretation. Most aspects could be automated leading to the implementation of a truly universal system for abstract interpretation.

References

[1] S. Abramsky and C. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, West Sussex, England, 1987.

[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 Programming languages (ICCL'92)*, Oakland, U.S.A., April 1992.

[5] 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.

[6] V. Englebert, B. Le Charlier, D. Roland, and P. Van Hentenryck. Generic abstract interpretation algorithms for prolog: Two optimization techniques and their experimental evaluation (extended abstract). 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.

[7] N.D. Jones and A. Mycroft. Dataflow analysis of applicative programs using minimal function graphs. In *Proceedings of Thirteenth ACM Symposium on Principles of Programming Languages (POPL'86)*, pages 123–142, St. Petersburg, Florida, 1986.

[8] N.D. Jones and H. Søndergaard. A semantic-based framework for the abstract interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 123–142. Ellis Horwood Limited, 1987.

[9] T. Kanamori and T. Kawamura. Analysing success patterns of logic programs by abstract hybrid interpretation. Technical report, ICOT, 1987.

[10] B. Le Charlier and K. Musumbu. Une sémantique opérationnelle instrumentale pour prolog et son application à la preuve de consistance d'un modèle d'interprétation abstraite. In J.-P. Delahaye, editor, *Actes des Journées Francophones de Programmation Logique (JFPL'92)*, Lille, May 1992.

[11] 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.

[12] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. In J. Cordy, editor, *Proceedings of the IEEE fourth International Conference on Programming Languages (ICCL'92)*, Oakland, U.S.A., April 1992. IEEE Press.

[13] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of prolog. In K. Apt, editor, *Proceedings of the Join International Conference and Symposium on Logic Programming (JICSLP'92)*, Washington, U.S.A., November 1992. MIT Press.

[14] B. Le Charlier and P. Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report 92-22, Institute of Computer Science, University of Namur, Belgium, (also Brown University), April 1992.

[15] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In R.A. Kowalski and K.A. Bowen, editors, *Proceeding of Fifth International Conference on Logic Programming (ICLP'88)*, pages 733–748, Seattle, Washington, August 1988. MIT Press.

[16] K. Marriott and H. Søndergaard. Notes for a tutorial on abstract interpretation of logic programs. In *North American Conference on Logic Programming (NACLP'89)*, Cleveland, Ohio, 1989.

[17] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. Ritter, editor, *Information Processing'89*, pages 601–606, San Fransisco, California, 1989.

[18] K. Marriott and H. Søndergaard. Abstract interpretation of logic programs: the denotational approach. Padovà, Italy, June 1990.

[19] C. S. Mellish. Abstract Interpretation of Prolog Programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 8, pages 181–198. Ellis Horwood Limited, 1987.

[20] K. Musumbu. *Interprétation Abstraite de Programmes Prolog*. PhD thesis, Institute of Computer Science, University of Namur, Belgium, September 1990. In French.

[21] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[22] U. Nilsson. Systematic semantic approximations of logic programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming (PLILP'90)*, volume 456 of *Lecture Notes in Computer Science*, pages 293–306, Linköping, Sweden, August 1990. Springer-Verlag.

[23] R.A. O'Keefe. Finite fixed-point problems. In J-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming (ICLP'87)*, pages 729–743, Melbourne, Australia, May 1987. MIT Press.

[24] H. Tamaki and T. Sato. OLD-resolution with tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming (ICLP'86)*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98, London, England, July 1986. Springer-Verlag.

[25] D.S. Warren. Memoization for logic programs. *Communications of the ACM*, 35(3), March 1992.

[26] W. Winsborough. A minimal function graph semantics for logic programs. Technical Report TR-711, Computer Science Department, University of Wisconsin at Madison, August 1987.

[27] W. Winsborough. Multiple specialization using minimal-function graph semantics. Technical report, Department of Computer Science, The Pennsylvania State University, August 1990. To appear in the Journal of Logic Programming.