Abstract Interpretation and Application to Logic Programs *

Patrick Cousot
LIENS, École Normale Supérieure
45, rue d'Ulm
75230 Paris cedex 05 (France)
cousot@dmi.ens.fr

Radhia Cousot
LIX, École Polytechnique
91128 Palaiseau cedex (France)
radhia@polytechnique.fr

March 1, 1992

Abstract. Abstract interpretation is a theory of semantics approximation which is used for the construction of semantics-based program analysis algorithms (sometimes called "data flow analysis"), the comparison of formal semantics (e.g., construction of a denotational semantics from an operational one), the design of proof methods, etc.

Automatic program analysers are used for determining statically conservative approximations of dynamic properties of programs. Such properties of the run-time behavior of programs are useful for debugging (e.g., type inference), code optimization (e.g., compile-time garbage collection, useless occur-check elimination), program transformation (e.g., partial evaluation, parallelization), and even program correctness proofs (e.g., termination proof).

After a few simple introductory examples, we recall the classical framework for abstract interpretation of programs. Starting from a standard operational semantics formalized as a transition system, classes of program properties are first encapsulated in collecting semantics expressed as fixpoints on partial orders representing concrete program properties. We consider invariance properties characterizing the descendant states of the initial states (corresponding to top/down or forward analyses), the ascendant states of the final states (corresponding to bottom/up or backward analyses) as well as a combination of the two. Then we choose specific approximate abstract properties to be gathered about program behaviors and express them as elements of a poset of abstract properties. The correspondence between concrete and abstract properties is established by a concretization and abstraction function that is a Galois connection formalizing the loss of information. We can then constructively derive the abstract program properties from the collecting semantics by a formal computation leading to a fixpoint expression in terms of abstract operators on the domain of abstract properties. The design of the abstract interpreter then involves the choice of a chaotic iteration strategy to solve this abstract fixpoint equation. We insist on the compositional design of this abstract interpreter, which is formalized by a series of propositions for designing Galois connections (such as Moore families, decomposition by partitioning, reduced product, down-set completion, etc.). Then we recall the convergence acceleration methods using widening and narrowing allowing for the use of very expressive infinite domains of abstract properties.

We show that this classical formal framework can be applied in extenso to logic programs. For simplicity, we use a variant of SLD-resolution as the standard operational semantics. The first example is groundness analysis, which is a variant of Mellish mode analysis. It is extended to a combination of top/down and bottom/up analyses. The second example is the derivation of constraints among argument sizes, which involves an infinite abstract domain requiring the use of convergence acceleration methods. We end up with a short thematic guide to the literature on abstract interpretation of logic programs.

Keywords: Abstract interpretation, fixpoint approximation, abstraction, concretization, Galois connection, compositionality, chaotic iteration, convergence acceleration, widening/narrowing, operational and collecting semantics, top/down, bottom/up and combined analyses, logic programming, groundness analysis, argument sizes analysis.

* To appear in the special issue on Abstract Interpretation of the Journal of Logic Programming. This work was supported in part by Esprit BRA 3124 *Sémantique* and the CNRS GDR C^3.

1. INTRODUCTION

Program manipulators (such as programmers who write, debug, and attempt to understand programs or computer programs which interpret, compile, or execute programs) reason upon or are constructed by relying on the syntax but mainly on the semantics of these programs. The *semantics* of a program describes the set of all possible behaviors of that program when executed for all possible input data. For logic programs, the input data are questions. The behaviors can be non-termination, termination with a run-time error, failure, or correct termination delivering one or more output answers.

For a given type of reasoning about programs, not all aspects and details about their possible behaviors during execution have to be considered. Each program manipulation is facilitated by reasoning upon a well-adapted semantics, abstracting away from irrelevant matters. For example, logical programs debugging often refers to a small-step operational semantics with backtracking. On the contrary, programs explanation often refers to the declarative aspect of a logic program providing the relation between questions and answers. Therefore, there is no universal general-purpose semantics of programs, and, in everyday life, more or less formal, more or less precise, special-purpose semantics are in current use. *Abstract interpretation* is a method for relating these semantics.

We will explain the abstract interpretation framework that we introduced in [25], [26], [28], [29], [32], [34] and illustrate it for logic programs. Thanks to examples, we will consider two essential utilizations of abstract interpretation: (a) The first utilization is the design of an abstract semantics in order to show off an underlying structure in a concrete, more detailed semantics. Hence, properties of programs are induced, without loss of indispensable information, from a concrete into a more abstract setting. A typical example consists in designing a proof method starting from a collecting semantics [27]. (b) The second utilization of abstract interpretation is the design of an abstract semantics in order to specify an automatic program analyser for the static determination of dynamic properties of programs. Here, properties of programs are approximated, with an inevitable loss of information, from a concrete to a less precise abstract setting. Such semantics-based sound but approximate information is indispensable to identify errors in a program, as performed by program debuggers and type checkers. Another use is in program transformers such as compilers, partial evaluators, and parallelizers, where the analysis determines the applicability of various transformations.

After a presentation of abstract interpretation, we will consider its application to static analysis of logic programs starting from a variant of SLD-resolution as operational semantics. We will illustrate the design of abstract interpretations by the typical example of groundness analysis (which will be extended to a bi-directional combination of top/down and bottom/up analyses) and the atypical example of argument size relation (involving an infinite domain). Finally, we will very briefly review the main applications to logic programs that have been considered in the already abundant literature.

2. SIMPLE EXAMPLES OF ABSTRACT INTERPRETATION

As a first approximation, abstract interpretation can be understood as a nonstandard semantics, i.e., one in which the domain of values is replaced by a domain of descriptions of values, and in which the operators are given a corresponding nonstandard interpretation.

2.1. Rule of Signs

For example, rather than using integers as concrete values, an abstract interpretation may use abstract values -1 and $+1$ to describe negative and positive integers, respectively [138]. Then by reinterpreting operations like addition or multiplication according to the "rule of signs" due to the ancient Greek mathematicians, the abstract interpretation may establish certain properties

of a program such as "whenever this loop body is entered, variable x is assigned a positive value (or perhaps is uninitialized)."

2.1.1. The Rule of Signs Calculus

For example, $(x \times x) + (y \times y)$ yields the value 25 when x is 3 and y is -4 and when $\times$ and $+$ are the usual arithmetical multiplication and addition. But when applying the "rule of signs":

$$\begin{array}{lll} +1 + +1 = +1 & +1 \times +1 = +1 & -1 \times +1 = -1 \\ -1 + -1 = -1 & +1 \times -1 = -1 & -1 \times -1 = +1 \end{array}$$

(where the abstract value $+1$ represents any positive integer, while -1 represents any negative integer) one concludes that the sign of $(3 \times 3) + (-4 \times -4)$ is always $+1$ since $(+1 \times +1) + (-1 \times -1) = (+1) + (+1) = +1$. However, this simple abstract calculus fails to prove that $x^2 + 2 \times x \times y + y^2$ is always positive.

Although very simple, this example shows that abstract interpretations may fail. To avoid errors due to such failures in a partial abstract calculus, we choose to use a total abstract calculus where an abstract value $\top$ is introduced to represent the fact that nothing is known about the result:

$$\begin{array}{llll} +1 + -1 = \top & +1 + \top = \top & \top \times +1 = \top & -1 \times \top = \top \\ -1 + +1 = \top & -1 + \top = \top & \top \times -1 = \top & \top \times \top = \top \\ \top + +1 = \top & \top + \top = \top & +1 \times \top = \top & \\ \top + -1 = \top & & & \end{array}$$

Now, several abstract values can be used to approximate a given concrete value. For example, the concrete value 5 can be approximated by $+1$ or $\top$. A partial order relation $\preceq$ can be introduced to compare the precision of abstract values ([155], [95]). For example, $-1 \preceq \top$ and $+1 \preceq \top$ since -1 or $+1$ are more precise than $\top$, whereas -1 and $+1$ are not comparable since no one can always safely replace the other.

A concrete value may be approximated by several minimal values. For example, 0 can be approximated by minimal abstract values -1 or $+1$. In this case, the best choice may depend upon the expression to be analysed. For example, when analysing $0+x$ it is better to approximate 0 by $+1$ if x is known to be positive and by -1 when x is negative. In order to avoid having to do the choice during the abstract calculus or to explore all alternatives, it is always possible to enrich the abstract domain so that the set of upper approximations of any given concrete value has a best element [34]. For our example, this leads to the introduction of an abstract value 0:

$$\begin{array}{llll} 0 + +1 = +1 & +1 + 0 = +1 & 0 \times +1 = 0 & +1 \times 0 = 0 \\ 0 + -1 = -1 & -1 + 0 = -1 & 0 \times -1 = 0 & -1 \times 0 = 0 \\ 0 + \top = \top & \top + 0 = \top & 0 \times \top = 0 & \top \times 0 = 0 \\ 0 + 0 = 0 & & 0 \times 0 = 0 & \end{array}$$

2.1.2. Generalization to Interval Analysis

In [28], this "rule of signs" idea was generalized to interval analysis, i.e., to properties of the form $l \leq x \leq u$ where $l, u \in \mathbb{Z} \cup \{-\infty, +\infty\}$, $\mathbb{Z}$ is the set of integers, and $l \leq u$. The main innovations were the idea of soundness proof by relating the abstract interpretations to an operational semantics and the use of infinite abstract domains, which led to very powerful analyses, as shown by the following results (where the comments have been generated automatically [7]):

```
function F(X : integer) : integer;
begin
  if X > 100 then begin
    F := X - 10
    { X∈[101, maxint] ∧ F∈[91, maxint - 10] }
  end else begin
    F := F(F(X + 11))
```

```
      { X ∈ [minint, 100] ∧ F = 91 }
    end;
  end;
```

This analysis supersedes the most sophisticated methods based upon data flow analysis. Let us consider the following program given in [76]:

```
program AnOldLookAtOptimizingArrayBoundChecking;
  var
    i, j, k, l, m : integer;
    a : array[1..100] of real;
begin
  read(i, j);
  { i∈⋆[1, 99]⋆ ∧ j∈[-maxint-1, maxint] }
  k := i;
  { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 99] }
  l := 1;
  { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 99] ∧ l∈[1, 1]}
  while l <= i do begin
    { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[1, 99] }
    m := i;
    { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[1, 99] ∧ m∈[1, 99] }
    while m <= j do begin
      { i∈[1, 99] ∧ j∈[1, maxint] ∧ k∈[1, maxint-1] ∧ l∈[1, 99] ∧ m∈[1, maxint-1] }
      k := k + m;
      { i∈[1, 99] ∧ j∈[1, maxint] ∧ k∈[2, maxint]# ∧ l∈[1, 99] ∧ m∈[1, maxint-1] }
      m := m + 1;
      { i∈[1, 99] ∧ j∈[1, maxint] ∧ k∈[2, maxint] ∧ l∈[1, 99] ∧ m∈[2, maxint] }
    end;
    { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, maxint] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
    a[l] := k;
    { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, maxint] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
    if k < 1000 then begin
      { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
      write(k);
      { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
    end else begin
      { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1000, maxint] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
      k := i;
      { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 99] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
    end;
    { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
    a[l + 1] := a[l] / 2;
    { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[1, 99] ∧ m∈[1, maxint] }
    l := l + 1;
    { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[2, 100] ∧ m∈[1, maxint] }
  end;
  { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[2, 100] ∧ m∈[1, maxint] }
  write(a[i]);
  { i∈[1, 99] ∧ j∈[-maxint-1, maxint] ∧ k∈[1, 999] ∧ l∈[2, 100] ∧ m∈[1, maxint] }
end.
```

The invariants given in comments have been discovered automatically. They hold during any execution of the program without run-time error. If any one of these invariants is violated during execution, then a later run-time error is inevitable. To detect these run-time errors before they occur, it is shown automatically that only two bound checks (marked $\star$) are necessary upon initialization as well as an overflow check (marked $\#$) within the loop. This analysis seems well out of reach of the data flow analysis of [76] based upon the syntactical elimination, propagation, and combination of range checks.

2.2. Dimension Calculus

Let us now consider a familiar example from elementary physics.

2.2.1. The Dimension Calculus

The dimension calculus uses the abstract values *length*, *surface*, *volume*, *time*, *speed*, *acceleration*, *mass*, *force*, ..., *nodimension*. The abstract version $\bar{op}$ of an operator *op* is defined as follows:

$$\begin{array}{rcl} \mathit{length} \mathbin{\bar{+}} \mathit{length} & = & \mathit{length} \\ \mathit{length} \mathbin{\bar{\times}} \mathit{length} & = & \mathit{surface} \\ \mathit{length} \mathbin{\bar{/}} \mathit{length} & = & \mathit{nodimension} \\ \mathit{length} \mathbin{\bar{/}} \mathit{time} & = & \mathit{speed} \\ \mathit{speed} \mathbin{\bar{/}} \mathit{time} & = & \mathit{acceleration} \end{array}$$

$$\begin{array}{rcl} \mathit{mass} \mathbin{\bar{\times}} \mathit{acceleration} & = & \mathit{force} \\ & \cdots & \\ x \mathbin{\bar{/}} y^{\overline{n+1}} & = & \bar{(}x \mathbin{\bar{/}} y^{\bar{n}}\bar{)} \mathbin{\bar{/}} y \\ y^{\bar{1}} & = & y \\ \bar{(}x\bar{)} & = & x \\ & \cdots & \end{array}$$

The correspondence between concrete values and abstract values can be formalized by an abstraction function α mapping units to dimensions:

$$\begin{array}{rcl} \alpha(\mathit{meter}) & = & \mathit{length} \\ \alpha(\mathit{mile}) & = & \mathit{length} \\ \alpha(\mathit{acre}) & = & \mathit{surface} \\ \alpha(\mathit{second}) & = & \mathit{time} \\ \alpha(\mathit{minute}) & = & \mathit{time} \\ \alpha(\mathit{hour}) & = & \mathit{time} \\ \alpha(\mathit{kilogram}) & = & \mathit{mass} \end{array}$$

$$\begin{array}{rcl} \alpha(\mathit{pound}) & = & \mathit{mass} \\ \alpha(\mathit{ton}) & = & \mathit{mass} \\ \alpha(\mathit{Newton}) & = & \mathit{force} \\ \alpha(\mathit{nounit}) & = & \mathit{nodimension} \\ & \cdots & \\ \alpha(E_1\ \mathit{op}\ E_2) & = & \alpha(E_1)\ \bar{op}\ \alpha(E_2) \\ \alpha((E)) & = & \bar{(}\alpha(E)\bar{)} \end{array}$$

The abstract interpretation of an expression can be done in two distinct steps: it begins with the derivation of an abstract expression from the concrete expression and goes on with the evaluation of the abstract expression using the definition of the abstract operators. In our example, the abstract expression is first obtained using the abstraction operator α:

$$\begin{array}{rcl} \alpha(kg \times (m/s^2)) & = & \alpha(kg) \mathbin{\bar{\times}} \alpha((m/s^2)) \\ & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\alpha(m \,/\, s^2)\bar{)} \\ & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\alpha(m) \mathbin{\bar{/}} \alpha(s^2)\bar{)} \\ & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\mathit{length} \mathbin{\bar{/}} \alpha(s)^{\bar{2}}\bar{)} \\ & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\mathit{length} \mathbin{\bar{/}} \mathit{time}^{\bar{2}}\bar{)} \end{array}$$

Since, in general, the abstraction function α is not computable, this first phase, which is usually done by hand, can be understood as the design of an abstract compiler. Then, the abstract expression can be evaluated using an abstract interpreter:

$$\begin{array}{rcl} \mathit{mass} \mathbin{\bar{\times}} \bar{(}\mathit{length} \mathbin{\bar{/}} \mathit{time}^{\bar{2}}\bar{)} & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\bar{(}\mathit{length} \mathbin{\bar{/}} \mathit{time}\bar{)} \mathbin{\bar{/}} \mathit{time}\bar{)} \\ & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\bar{(}\mathit{speed}\bar{)} \mathbin{\bar{/}} \mathit{time}\bar{)} \\ & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\mathit{speed} \mathbin{\bar{/}} \mathit{time}\bar{)} \\ & = & \mathit{mass} \mathbin{\bar{\times}} \bar{(}\mathit{acceleration}\bar{)} \\ & = & \mathit{mass} \mathbin{\bar{\times}} \mathit{acceleration} \\ & = & \mathit{force} \end{array}$$

This second phase of abstract execution must always be finitely computable, hence must only involve the finite iterated application of computable abstract operations on finitely representable abstract values.

The main interest of this example is to illustrate the idea of proving the correctness of the abstract interpretation relatively to a semantics via an abstraction operator as introduced in [28] and [29]. The importance of this idea was that by relating abstract interpretations not to