A Generic Abstract Interpretation Algorithm and its Complexity Analysis (Extended Abstract)

Baudouin Le Charlier and Kaninda Musumbu[1]
Pascal Van Hentenryck[2]

Technical Report No. CS-90-25

October 1990

[1]University of Namur, 21 rue Grandgagnage, B-5000 Namur Belgium
[2]Computer Science Dept., Brown University, Providence, RI 02912

A Generic Abstract Interpretation Algorithm and its Complexity Analysis
(Extended Abstract)

Baudouin Le Charlier and Kaninda Musumbu
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
Email: pvh@cs.brown.edu

Abstract

A generic abstract interpretation algorithm for Prolog programs is presented and its complexity analysed. The algorithm is parametrized on the abstract domain and can be instantiated by choosing an abstraction of substitutions and an implementation of a number of abstract operations whose specifications are given as consistent abstractions of some concrete operations. The algorithm, which is representation-independent, has been developed by successive refinments from the abstract semantics and includes optimizations such as the detection of definitive results, the early recognition of termination, and the detection of dynamic dependencies. The worst-case complexity of the algorithm is analysed for various classes of programs and behavioural assumptions.

1 Introduction

Abstract Interpretation of logic programs has been devoted much attention in recent years. Many semantic frameworks have been developed (e.g. [8, 13, 14, 15, 11, 10, 1, 2]), and a variety of abstract domains have been proposed to cater for various program analysis tools involving modes, types, occur-check, garbage collection, static detection of parallelism, and program specialization to name a few.

However much less attention has been devoted to generic abstract interpretation algorithms that are the cornerstone between the semantic frameworks and the abstract domains, although several authors have proposed (more or less precise) sketches of some algorithms (e.g., [2, 4, 5, 13, 14]). A generic abstract interpretation algorithm is an algorithm, independent of the abstract domain, that can be instantiated to provide an algorithm tailored to a specific application. Generic abstract interpretation algorithms are parametrized on the abstract domains in terms of abstract substitutions and a number of operations involving them. The abstract operations are actually consistent abstractions of the concrete operations in terms of which the concrete semantics is defined. Instantiating the generic abstract interpretation algorithm amounts to designing abstract substitutions capturing the relevant information and to implementing a consistent version of abstract operations on these substitutions.

In this paper, we describe an efficient generic abstract interpretation algorithm and analyse its

time complexity. The algorithm has been built by successive refinments from a semantic framework and includes optimizations such as the detection of definitive results, the detection of early termination, and the recognition of mutual dependencies. Its worst-case time complexity is analysed on a variety of program classes and behavioural assumptions. This is, to our knowledge, the first complexity analysis of a generic abstract interpretation algorithm and the most efficient algorithm ever proposed for a similar accuracy level.

The rest of the paper is organized in the following way. Section 1 describes the concrete semantics which is fundamental in giving a semantics to the abstract operations and to the algorithm. Section 2 describes the abstract semantics as an abstraction of the concrete semantics and establishes the required relationships between abstract and concrete substitutions and operations. Section 3 describes the algorithm and Section 4 analyses its complexity.

The proofs for the results described here can be found in the full version of the paper [9] that also contains the successive refinments of the algorithm, their formal specifications and proofs of correctness, and discussion of related work.

2 Concrete Semantics

The algorithm works on normalized programs and substitutions which impose syntactic restrictions that greatly simplify the presentation without losing generality.

2.1 Normalized Programs

We assume the existence of sets F_i and P_i $(i \geq 0)$ denoting sets of functors and predicate symbols of arity i and of an infinite set PV of program variables. Variables in PV are ordered and denoted by the $x_1, x_2, \ldots, x_i, \ldots$.

Normalized programs contain clauses with heads of the form $p(x_1, \ldots, x_n)$ where $n \geq 0$ and $p \in P_n$. Normalized clauses also contain bodies of the form $B_1, \ldots, B_n$ $(n \geq 0)$ with B_i of the form $p(x_{i_1}, \ldots, x_{i_n})$ where $x_{i_1}, \ldots, x_{i_n}$ are all distinct variables and $p \in P_n$. Unification is achieved through two built-in predicates, $x_i = x_j$ $(i \neq j)$ and $x_i = f(x_{j_1}, \ldots, x_{j_n})$ where $x_i, x_{j_1}, \ldots, x_{j_n}$ are all distinct variables and $f \in F_n$.

The motivation behind these definitions is to allow the result of any predicate p/n to be expressed as a set of substitutions on program variables $x_1, \ldots, x_n$.

2.2 Normalized Substitutions

We assume the existence of another infinite set RV of renaming variables. Terms and substitutions are constructed using program and renaming variables. We distinguish two kinds of substitutions: *program variable substitutions* (*ps* for short) whose domain and codomain are subsets of PV and RV respectively, and *renaming variable substitutions* (*rs* for short) whose domain and codomain are subsets of RV. In the following, PS denotes the set of *ps* and RS the set of *rs*.

Definition 1 Let θ be a substitution and $D \subseteq dom(\theta)$. The *restriction* of θ to D, denoted $\theta_{/D}$, is the substitution σ such that $dom(\sigma) = D$ and $x\theta = x\sigma$ for all $x \in D$.

The definition of substitution composition is slightly modified to take into account the special role held by program variables. The modification occurs when $\theta \in PS$ and $\sigma \in RS$ for which $\theta\sigma \in PS$ is defined by

- $dom(\theta\sigma) = dom(\theta)$;
- $x(\theta\sigma) = (x\theta)\sigma$ for all $x \in dom(\theta)$.

The notion of *free variable* is non-standard to avoid clashes between variables during renaming. A free variable is represented by a binding to a renaming variable that appears nowhere else.

Definition 2 Let θ be a PS and $x \in dom(\theta)$. x is free in θ if and only if $x\theta$ is a variable and $x\theta$ does not appear in $x'\theta$ for all $x' \in dom(\theta)$ such that $x \neq x'$.

Unifiers are defined as usual but only belong to RS hereafter.

Definition 3 Let Θ be a subset of PS. Θ is complete if and only if, for all $\theta \in \Theta$, θ and θ' are variant implies that $\theta' \in \Theta$.

Let D be a subset of PV. $CS_D = \{\Theta : \forall\theta \in \Theta\ dom(\theta) = D$ and Θ is complete$\}$. CS_D is a complete lattice wrt set inclusion $\subseteq$.

2.3 Concrete Operations

We now define the concrete operations on complete sets of substitutions.

Union of Sets of Substitutions Let $\Theta_1, \ldots, \Theta_n \in CS_D$ and $D \subseteq PV$.

$$UNION(\Theta_1, \ldots, \Theta_n) = \Theta_1 \cup \ldots \cup \Theta_n.$$

Unification of two Variables Let $D = \{x_1, x_2\}$ and $\Theta \in CS_D$.

$$AI\_VAR(\Theta) = \{\theta\sigma : \theta \in \Theta, \sigma \in RS, \text{ and } \sigma \in mgu(x_1\theta, x_2\theta)\}$$

Unification of a Variable and a Function This operation unifies x_1 with $f(x_2, \ldots, x_n)$. Let $D = \{x_1, \ldots, x_n\}$, $\Theta \in CS_D$, and $f \in F_{n-1}$.

$$AI\_FUNC(\Theta, f) = \{\theta\sigma : \theta \in \Theta, \sigma \in RS, \text{ and } \sigma \in mgu(x_1\theta, f(x_2, \ldots, x_n)\theta)\}$$

Restriction and Extension of a Set of Substitutions for a Clause The $RESTRC$ operation restricts a set of substitutions on all variables in a clause to the variables in the head. The $EXTC$ operation extends a set of substitutions on variables in the head of a clause to all variables in the clause. Let c a clause, D' be the set of variables in the head and D be the set of variables of c.

$$RESTRC(c, \Theta) = \{\theta_{/_{D'}} : \theta \in \Theta\}.$$

$$EXTC(c, \Theta) = \{\theta : dom(\theta) = D,\ \theta_{/_{D'}} \in \Theta, \text{ and } \forall x \in D \setminus D',\ x \text{ is free in } \theta\}.$$

Restriction and Extension of a Set of Substitutions for a Goal The $RESTRG$ operation expresses a set of substitutions Θ in terms of the formal parameters $x_1, \ldots, x_n$ of a goal g. The $EXTG$ operation extends a set of substitutions Θ with a set of substitutions Θ' representing the result of executing a goal g on Θ. Let D be the domain of Θ, $D'' = \{x_{i_1}, \ldots, x_{i_n}\}$ the set of variables appearing in g exactly in that order, and $D' = \{x_1, \ldots, x_n\}$.

$$RESTRG(g, \Theta) = \{\theta : dom(\theta) = D', \exists \theta' \in \Theta \text{ such that } x_j\theta = x_{i_j}\theta' \ (1 \leq j \leq n)\}.$$

$$\begin{aligned} EXTG(g, \Theta, \Theta') = \{\theta\sigma : \ & \theta \in \Theta,\ \sigma \in RS\ ,\ \theta'\sigma \in \Theta'\ ,\ dom(\sigma) \subseteq codom(\theta'), \\ & (codom(\theta) \setminus codom(\theta')) \cap codom(\sigma) = \emptyset, \\ & dom(\theta') = D'\ ,\ x_j\theta' = x_{i_j}\theta \ (1 \leq j \leq n)\}. \end{aligned}$$

2.4 Sets of Concrete Tuples

We assume in the following an underlying program P.

Definition 4 A concrete tuple is a tuple of the form $(\Theta_{in}, p, \Theta_{out})$ where $\Theta_{in}, \Theta_{out} \in CS_D$, D is $\{x_1, \ldots, x_n\}$, and p, of arity n, appears in the program P.

Θ_{out} is intended to represent the set of output substitutions obtained by executing $p(x_1, \ldots, x_n)$ on the set of input substitutions Θ_{in}. We use *cts* for "set of concrete tuples".

Definition 5 Let *sct* be a *cts*. The *domain* of *sct*, denoted $dom(sct)$, is the set of pairs (Θ, p) for which there exists an Θ' such that $(\Theta, p, \Theta') \in sct$.

We now turn to the properties of *cts*.

Definition 6 Let *sct* be a *cts*. *sct* is *functional* if and only if, for all (Θ, p), there exists at most one set Θ' such that $(\Theta, p, \Theta') \in sct$. This set is denoted $sct(\Theta, p)$.

Definition 7 The *underlying domain* UD is the set of pairs (Θ, p) where p is a predicate symbol of arity n in P, $D = \{x_1, \ldots, x_n\}$ and $\Theta \in CS_D$.

Definition 8 Let *sct* be a functional *cst*. *sct* is *total* if $sct(\Theta, p)$ is defined for all $(\Theta, p) \in UD$. Any functional *cts*, total or not, is *partial*. *sct* is *monotonic* if $\Theta_1 \subseteq \Theta_2 \Rightarrow sct(\Theta_1, p) \subseteq sct(\Theta_2, p)$, each time $sct(\Theta_1, p)$ and $sct(\Theta_2, p)$ are defined.

Definition 9 Let *sct* be a functional, total, and monotonic *cts*. *sct* is *continuous* iff any non-decreasing chain $\Theta_1 \subseteq \Theta_2 \subseteq \ldots \subseteq \Theta_n \subseteq \ldots$ satisfies $sct(\bigcup_{i=1}^{\infty} \Theta_i, p) = \bigcup_{i=1}^{\infty} \{sct(\Theta_i, p)\}$.

We denote by SCT the set of functional and monotonic *cts* and by $SCTT$ the set of total, functional, monotonic, and continuous *cts*. $SCTT$ is endowed with a structure of cpo (i.e. complete partial order) by defining

- $\bot = \{(\Theta, p, \emptyset) : (\Theta, p) \in UD\}$;
- $sct \leq sct' \ \equiv \ \forall (\Theta, p) \in UD \ sct(\Theta, p) \subseteq sct'(\Theta, p)$.

$TSCT(sct) = \{(\Theta, p, \Theta') : (\Theta, p) \in UD \text{ and } \Theta' = T_p(\Theta, p, sct)\}$.

$T_p(\Theta, p, sct) = UNION(\Theta_1, \ldots, \Theta_n)$
where $\Theta_i = T_c(\Theta, c_i, sct)$,
$c_1, \ldots, c_n$ are the clauses of p.

$T_c(\Theta, c, sct) = RESTRC(c, \Theta')$
where $\Theta' = T_b(EXTC(c, \Theta), b, sct)$,
b is the body of c.

$T_b(\Theta, <>, sct) = \Theta$.
$T_b(\Theta, g.gs, sct) = T_b(\Theta_3, gs, sct)$
where $\Theta_3 = EXTG(g, \Theta, \Theta_2)$,

$$\Theta_2 = \begin{cases} sct(\Theta_1, p) & \text{if } g \text{ is } p(\ldots) \\ AI\_VAR(\Theta_1) & \text{if } g \text{ is } x_i = x_j \\ AI\_FUNC(\Theta_1, f) & \text{if } g \text{ is } x_i = f(\ldots), \end{cases}$$

$\Theta_1 = RESTRG(g, \Theta)$.

Figure 1: A Fixpoint Semantics

2.5 A Fixpoint Concrete Semantics

The concrete semantics is defined in terms of three functions and one transformation given in Figure 1. We assume an underlying program P and let b and c be a sequence of atoms and a clause using only predicate symbols from P.

The transformation and functions are monotonic and continuous wrt $SCCT$ and the canonical ordering on the cartesian product $CS_D \times SCTT$ respectively. Since $SCTT$ is a cpo, the semantics of logic programs is the least fixpoint of the transformation $TSCT$, denoted $\mu(TSCT)$. This fixpoint can be shown to be equivalent (in a restricted sense) to SLD-resolution.

Theorem 10 Let P be a program, $g = p(x_1, \ldots, x_n)$ be a predicate, θ_{in} be a program substitution with $dom(\theta_{in}) = \{x_1, \ldots, x_n\}$, sct be $\mu(TSCT)$ and $\Theta_{in} = \{\theta \in PS : \theta \text{ and } \theta_{in} \text{ are variant}\}$. The following two statements are true (we assume that SLD-Resolution uses renaming variables belonging to RS).

1. If σ is an answer-substitution of SLD-resolution applied to $P \cup \{\leftarrow g\theta_{in}\}$, then there exists a substitution $\theta_{out} \in sct(\Theta_{in}, p)$ such that $\theta_{out} = \theta_{in}\sigma$.
2. If $\theta_{out} \in sct(\Theta_{in}, p)$, then there exists an answer-substitution σ of SLD-resolution applied to $P \cup \{\leftarrow g\theta_{in}\}$ such that $\theta_{in}\sigma$ is more general than θ_{out}, i.e. $\theta_{out} = \theta_{in}\sigma\sigma'$ for some $\sigma' \in RS$.

3 Abstract Semantics

We abstract the concrete semantics and establish the relationships between the abstract and concrete semantics.

3.1 The Abstract Domain

Definition 11 An *abstract domain* is the association of a cpo $(AS_D, \leq)$ with each finite set of variables D . Elements of AS_D are called *abstract substitutions.*

The correspondence between the abstract and concrete domains is established through a concretization function.

Definition 12 A *concretization function* for an abstract domain AS_D is a monotonic and continuous function $Cc : AS_D \rightarrow CS_D$.

The monotonicity of the concretization function is a sufficient condition to ensure that the least fixpoint of the abstract transformation is the most accurate. The continuity requirement, although natural, is not essential for the theory.

3.2 Abstract Operations

To each of the concrete operations, we associate an abstract operation with the same name and signature except that abstract substitutions replace complete sets of substitutions. In addition, all abstract versions should be consistent to ensure the correctness of the abstract semantics. If all abstract operations are consistent, then any fixpoint of the abstract transformation is consistent.

Definition 13 A primitive abstract operation $ao : AS_{D_1} \times \ldots \times AS_{D_n} \rightarrow AS_D$ is *consistent* wrt to a primitive operation $o : CS_{D_1} \times \ldots \times CS_{D_n} \rightarrow CS_D$ if and only if

$$\forall \beta_1 \in AS_{D_1}, \ldots, \beta_n \in AS_{D_n} : \quad o(Cc(\beta_1), \ldots, Cc(\beta_n)) \subseteq Cc(ao(\beta_1, \ldots, \beta_n)).$$

The abstract operations are also required to be monotonic and continuous. The monotonicity of the abstract operations ensures the existence of a least fixpoint. The continuity ensures that the least fixpoint is the limit of a sequence converging by step ω.

3.3 Sets of Abstract Tuples

Abstract tuples are the counterpart of concrete tuples with abstract substitutions replacing complete sets of substitutions. We use *ats* for "set of abstract tuples". We denote SAT the set of all functional and monotonic *ats*, $SATT$ the subset of SAT whose elements are total and continuous. As the generic algorithm works on partial *ats*, we define an ordering on SAT.

Definition 14 Let $sat, sat' \in SAT$, $sat \leq sat'$ if and only if, for all $(\beta, p) \in dom(sat), (\beta, p) \in dom(sat')$ and $sat(\beta, p) \leq sat'(\beta, p)$.

3.4 A Fixpoint Abstract Semantics

The abstract fixpoint semantics is defined in exactly the same way as the concrete semantics with abstract tuples and operations replacing concrete tuples and operations. The semantics is defined on $SATT$.

It is possible to show that the abstract transformation $TSAT$ is monotonic and continuous and thus to define the abstract semantics as the least fixpoint of $TSAT$, say $\mu(TSAT)$. Moreover the semantics can be proven consistent wrt the concrete semantics.

Theorem 15 Let P be a program, $TSAT$ and $TSCT$ the associated transformations, $sat = \mu(TSAT)$ and $sct = \mu(TSCT)$. Let p be a predicate and $D = \{x_1, \ldots, x_n\}$ where n is the arity of p. For each $\beta \in AS_D$: $sct(Cc(\beta), p) \subseteq Cc(sat(\beta, p))$.

4 A Generic Abstract Interpretation Algorithm

We are now in position to present our generic abstract interpretation algorithm. We start by some preliminary definitions and introduce the approach followed by the algorithm. We then present two new abstract operations, an important data-structure, and the algorithm itself.

4.1 Preliminary Definitions

The following definitions aim at characterizing the set of abstract tuples needed to compute $T_p(\beta, p, sat)$. A more formal definition can be found in [9] and is omitted here for brevity.

Definition 16 Let $D \subseteq UD$ and sat be an ats. The restriction of sat to D, denoted $sat_{/D}$, is the set $\{(\beta, p, sat(\beta, p)) : (\beta, p) \in D \cap dom(sat)\}$.

Definition 17 Let β be an abstract substitution and p be a predicate symbol. (β, p) is based in sat iff, informally speaking, for any total ats sat' such that $sat \subseteq sat'$, the computation of $T_p(\beta, p, sat')$ does not require values of sat' not belonging to sat. Hence the notation $T_p(\beta, p, sat)$ can be extended to such partial ats.

We denote by $base(\beta, p, sat)$ the smallest set D such that (β, p) is based in $sat_{/D}$. This set is defined iff (β, p) is based in sat.

Definition 18 Let β be an abstract substitution, p be a predicate symbol, and sat a partial ats. (β, p) is founded in sat if and only if $\exists D : (\beta, p) \in D$ and $\forall (\alpha, q) \in D : (\alpha, q)$ is based in $sat_{/D}$.

We denote by $foundation(\beta, p, sat)$ the smallest set D such that (β, p) is founded in $sat_{/D}$, when it exists.

4.2 The Approach

A straightforward approach to compute the output substitution β_{out} for a pair (β_{in}, p) consists in computing $\mu(TSAT)$ and picking up the value $sat(\beta_{in}, p)$ in the fixpoint. This is possible using a bottom-up approach provided that some restrictions be imposed on the abstract domain to guarantee termination. The main drawback of this approach is that many elements in $\mu(TSAT)$ are not relevant to the computation of β_{out} and hence most of the computation is unnecessary.

The approach followed in our algorithm amounts to computing a partial set of abstract tuples sat containing only elements relevant to the computation of $sat(\beta_{in}, p)$ and to ignoring the others. In other words, the purpose of the algorithm is to converge towards a partial set of abstract tuples sat including $(\beta_{in}, p, \beta_{out})$ but with as few other elements as possible. This subset can be computed through a series of approximations, say $sat_0 < sat_1 < \ldots < sat_{n-1} < sat_n = sat$ such that

- $sat_0 = \{(\beta_{in}, p, \bot)\}$;
- sat_n is such that for all $(\alpha, q) \in sat_n$, $T_p(\alpha, q, sat_n) \leq sat_n(\alpha, q)$ and (α, q) is based in sat_n;

- sat_{i+1} is obtained from sat_i by computing a new tuple $(\alpha_{in}, q, \alpha_{out})$ such that either (α_{in}, q) is not based in sat_i or $T_p(\alpha_{in}, q, sat_i) \leq sat_i(\alpha_{in}, q)$ does not hold.

It is important to notice at this point that, within this approach, we never fully apply $TSAT$ to a complete *ats* but rather we are constantly improving our current *ats* and working with the most accurate one. This idea is closely related to the concept of minimal function graph introduced in [7] and used subsequently by several authors (e.g., [15, 4, 6]) as well as to the idea of *query-directed* semantics [12].

Given this basic idea, the algorithm to compute β_{out} given (β_{in}, p) has to specify a variety of decisions, including the detection of termination and the choice of a new tuple to work on. In our algorithm, the computation starts with the couple (β_{in}, p) and mainly amounts to executing the definition of $T_p(\beta_{in}, p, sat)$. At some point, the computation might need the value of another tuple (α_{in}, q) which is not defined or only approximated in our current *ats*. If this happens, the computation of (β_{in}, p) is suspended and a new subcomputation is started with the couple (α_{in}, q). This subcomputation is carried out in the same way as the computation of (β_{in}, p) except in the case where the value of the tuple (β_{in}, p) is required. In this case, we do not start a new subcomputation but rather we pick up the best approximation of β_{out} available in our current *ats*. The execution of the initial couple (β_{in}, p) is only resumed once the computation of (α_{in}, q) is completed. Note that if the computation of (α_{in}, q) has required the value of (β_{in}, p) then its resulting substitution α_{out} might be an approximation of its definitive value (i.e., its value in $\mu(TSAT)$) and we need to reconsider (α_{in}, q) once more information is available on (β_{in}, p).

We now introduce two new abstract operations on partial sets of abstract tuples.

4.3 Two New Abstract Operations

If the abstract domain is endowed with a *lub* operation, these two operations can be considered derived operations. We make this hypothesis here. For a more general handling, see [9].

4.3.1 Extension of a Partial Set of Abstract Tuples

The purpose of $EXTEND(\beta, p, sat)$ is to extend $sat \in SAT$ not defined on (β, p) so that its extension is defined on (β, p), remains monotonic, and does not go beyond the least fixpoint.

$$EXTEND(\beta, p, sat) = sat \cup \{(\beta, p, \beta')\}$$

where $\beta' = lub\{sat(\beta'', p) : \beta'' \leq \beta \text{ and } (\beta'', p) \in dom(sat)\}$

4.3.2 Adjustment of a Set of Abstract Tuples

The purpose of $ADJUST(\beta, p, \beta', sat)$ is to extend $sat \in SAT$ to include the information that $sat(\beta, p)$ contains at least the substitutions represented by β'. The operation should preserve the monotonicity and not go beyond the least fixpoint.

$$ADJUST(\beta, p, \beta', sat) = sat' \cup \{(\beta'', p, lub\{\beta', sat(\beta'', p)\}) : \beta'' \geq \beta \text{ and } (\beta'', p) \in dom(sat)\}$$

where $sat' = sat \setminus \{(\beta'', p, sat(\beta'', p)) : \beta'' \geq \beta \text{ and } (\beta'', p) \in dom(sat)\}$

In the following, we use a slightly more general version which returns, besides the new *sat*, the set of tuples that have been modified by the adjustment.

4.4 The Dependency Graph

The main concern in the design of the algorithm was the elimination of redundant computations. Redundant computations occur in various situations. For instance, at some point of the computation, an abstract tuple t might have reached its definitive value (i.e., $t \in \mu(TSAT)$). Recognizing definitive tuples and avoiding starting a subcomputation for them enable to remove much redundant work but is not sufficient. In the case of mutually recursive programs, it may happen that a tuple be reconsidered although none of the tuples it is depending upon has been updated. To cope with the problem (as well as to handle definitive tuples which is just a particular case), we introduce a data-structure: a dependency graph.

Definition 19 A dependency graph is a set of tuples of the form $\langle(\beta,p),lt\rangle$ where lt is a set $\{(\alpha_1,q_1),\ldots,(\alpha_n,q_n)\}$ $(n \geq 0)$ such that, for each (β,p), there exists at most one lt such that $\langle(\beta,p),lt\rangle \in dp$.

We denote by $dp(\beta,p)$ the set lt such that $\langle(\beta,p),lt\rangle \in dp$ if it exists. We also denote by $dom(dp)$ the set of all (β,p) such that $\langle(\beta,p),lt\rangle \in dp$ and by $codom(dp)$ the set of all (α,q) such that there exists a tuple $\langle(\beta,p),lt\rangle \in dp$ satisfying $(\alpha,q) \in lt$.

The set $dp(\beta,p)$, when defined, represents the dynamic dependencies of (β,p) at some step in the computation. Basically two cases must be distinguished.

If (β,p) is not suspended, $(\beta,p) \in dom(dp)$ means that the current value $sat(\beta,p)$ cannot be improved upon at this stage of the computation. In this case, $dp(\beta,p) = base(\beta,p,sat)$. Note that, if $dp(\beta,p)$ does not contain any suspended tuple, then (β,p) is definitive.

If (β,p) is suspended, $(\beta,p) \in dom(dp)$ means that presently the computation of (β,p) needs no further iteration. Also, $dp(\beta,p)$ contains the set of pairs (α,q) which have been called (directly) by the computation of (β,p).

We also define a number of operations on dependency graphs. The first two definitions allow us to remove elements from a dependency graph.

Intuitively $trans\_dp(\beta,p,dp)$ is the set of all pairs (α,q) at some point of the computation which, if modified, might imply an updating of $sat(\beta,p)$.

Definition 20 Let dp be a dependency graph and assume that $(\beta,p) \in dom(dp)$. The set $trans\_dp(\beta,p,dp)$ is the smallest subset of $codom(dp)$ closed by the following two rules:

1. if $(\alpha,q) \in dp(\beta,p)$ then $(\alpha,q) \in trans\_dp(\beta,p,dp)$;
2. if $(\alpha,q) \in dp(\beta,p)$, $(\alpha,q) \in dom(dp)$, and $(\alpha',q') \in trans\_dp(\alpha,q,dp)$ then $(\alpha',q') \in trans\_dp(\beta,p,dp)$.

Definition 21 Let dp be a dependency graph and let $modified$ be a set of tuples $\{(\alpha_1,q_1),\ldots,(\alpha_n,q_n)\}$ with $n \geq 0$. Then

$$REMOVE\_DP(modified,dp) = \{\langle(\alpha,q),lt\rangle \in dp : modified \cap trans\_dp(\alpha,q,dp) \neq \emptyset\}$$

The next two operations enable to extend a dependency graph. Note that, in the postconditions, we denote by par_0 the value of parameter par at call time.

Specification 22 Operation EXT_DP(β,p,dp) can be specified as follows:

Pre: $(\beta, p) \notin dom(dp)$;
Post: $dp = dp_0 \cup \{\langle(\beta, p), \emptyset\rangle\}$.

Specification 23 Operation `ADD_DP(`β`,p,`α`,q,dp)` can be specified as follows:

Pre: $(\beta, p) \in dom(dp)$;
Post: $dp = dp_0 \setminus \{\langle(\beta, p), dp_0(\beta, p)\rangle\} \cup \{\langle(\beta, p), dp_0(\beta, p) \cup \{(\alpha, q)\}\rangle\}$.

4.5 The Generic Abstract Interpretation Algorithm

We are now in position to present our generic abstract interpretation algorithm. The algorithm is composed of three procedures shown in Figures 2, 3, and 4 in Appendix 1.

The top-level procedure is Procedure `solve` which, given an input substitution β_{in} and a predicate symbol p, returns the set of abstract tuples *sat* containing $(\beta_{in}, p, \beta_{out}) \in \mu(TSAT)$ and the final dependency graph. It is straightforward to compute the base and the foundation of (β_{in}, p) from these results.

Procedure `solve_goal` receives as inputs an abstract substitution β_{in} and its associated predicate symbol, the suspended set mentioned previously, the current set of abstract tuples *sat* and dependency graph *dp*. If (β_{in}, p) is suspended or if $(\beta_{in}, p) \in dom(dp)$, `solve_goal` simply returns. In the first case, we have already started a computation with (β_{in}, p) and we should pick up the best value available while, in the second case, a further computation will not bring any new information.

Otherwise Procedure `solve_goal` computes a new approximation of $sat(\beta_{in}, p)$, possibly after extending *sat*. It enters a `repeat` loop that iterates on all clauses defining p and basically follows the semantic function T_p. Note that the call to Procedure `solve_clause` has an extended suspended set and that the modification of $sat(\beta_{in}, p)$ updates *dp* accordingly to exclude all the tuples that might be improved upon. The `repeat` loop exits when $(\beta_{in}, p) \in dom(dp)$. This condition means that $dp(\beta_{in}, p)$ is the base of (β_{in}, p) and that a further iteration will not produce any improvement.

Finally, Procedure `solve_clause` fulfills, loosely speaking, the functionalities of the semantic functions T_c and T_b. The set *dp* is updated to include the new dependency only if (β_{in}, p) belongs to $dom(dp)$.

5 Complexity Analysis

5.1 Basic Assumptions

5.1.1 Complexity Measures

The complexity measure we are most interested in here is the number of iterations in the **repeat** loop of Procedure `solve_goal`. This measure allows us to deduce information such as the number of iterations in the other loops and the number of times the abstract operations are executed.

5.1.2 Parameters on the Abstract Domain

Note that there are as many abstract domains as different arities among the predicate symbols in the program. For each abstract domain, we need two parameters:

1. the length of its longest increasing chain;

2. its number of elements.

Obviously the parameters are only significant when the abstract domain is finite. In case of an infinite abstract domain, the above definitions are replaced by their dynamic counterparts.

In the following, we denote by l_i and s_i respectively the length of the longest increasing chain and the number of elements in the abstract domain of the predicate symbol p_i. When p is not subscripted, we use respectively l_p and s_p.

Note that the parameters l_i can be replaced in practice by the average length of the chain explored before reaching the fixpoint. Hence the results also give an idea of the average complexity. Moreover, in realistic domains such as modes or types, experience has shown that the average length is generally small (around 3 or 4).

5.1.3 Parameters on the Programs

The programs are characterized in terms of the size of their static call tree.

Definition 24 A *static call tree* for a program P and a procedure name p is defined in the following way.

1. The root node is p.
2. If q is a node and $q(x_1, \ldots, x_n) \leftarrow b_1, \ldots, b_m$ is a clause of P, then q_i is a son of q if b_i is a procedure call with predicate symbol q_i and q_i is not an ancestor of q. Note that there can be several sons of q which are labelled with the same predicate symbol.

The *size* of a static call tree T is the number of nodes in T.

5.1.4 Additional Assumptions

We are only concerned with worst-case complexity. It is difficult to achieve a better result without taking into account the properties of the abstract domains and/or of the analysed programs. However, we have identified several classes of programs and several behavioural assumptions typical of logic programs that should give a precise idea of an average behaviour for "realistic" programs on "realistic" abstract domains.

Classes of programs include hierarchical programs, tail-recursive programs, locally recursive programs, and some classes of mutually recursive procedures.

Behavioural assumptions make hypothesis on the kind of input substitutions encountered for procedure calls given an initial call. In the following, we mainly make two behavioural assumptions which are thought to be representative of programs built using standard techniques like structural induction. The intuition is that most recursive calls are in fact of the same form as the initial call, property that can be defined as follows.

Definition 25 A procedure defining p is *substitution preserving* wrt (β, p) iff

$$\forall (\alpha, q) \in foundation(\beta, p, \mu(TSAT)) : q = p \Rightarrow \alpha = \beta.$$

A program P is *substitution preserving* wrt (β, p) iff the procedure defining q in P is substitution preserving wrt (α, q) for all $(\alpha, q) \in foundation(\beta, p, \mu(TSAT))$.

When information is lost during abstract interpretation, the property becomes as follows.

Definition 26 A procedure defining p is *substitution increasing* wrt (β, p) iff

$$\forall(\alpha, q) \in foundation(\beta, p, \mu(TSAT)) : q = p \Rightarrow \alpha \geq \beta.$$

A program P is *substitution increasing* wrt (β, p) iff the procedure defining q in P is substitution increasing wrt (α, q) for all $(\alpha, q) \in foundation(\beta, p, \mu(TSAT))$.

These two properties have important consequences when combined with the various classes of programs due to monotonicity of the operations.

5.2 Hierarchical Programs

The first class of programs we consider is the class of hierarchical programs (no recursion) which is present in some form in many programs. The algorithm is obviously optimal for hierarchical programs.

Theorem 27 Procedure `solve` defined in Appendix 1, given a hierarchical program P and a pair (β, p), spends exactly n iterations in the `repeat` loop of `solve_goal`, where n is the size of the static call tree of P and p.

5.3 Tail-Recursive Programs

Definition 28 A program P is tail-recursive if and only if its clauses are of the form[1]

`p :- q1,...,qn,p` or `p :- q1,...,qn`

where `q1,...,qn` do not call (directly or indirectly) p.

5.3.1 Substitution Preserving Programs

We first study tail-recursive programs satisfying the *substitution preserving* property. Many programs built using techniques like structural induction are substitution preserving for "useful" calls (β, p) if the abstract domains are accurate enough.

Tail-recursive programs with the substitution preserving have the nice property that a pair (β, p) only gives rise to recursive calls of the form (β, p). Indeed consider for instance a tail-recursive clause `p :- q1,...,qn,p`. Because the program is tail-recursive, the results of `q1,...,qn` are the same as in the least fixpoint and the input-substitution for the recursive call is the same as in the least fixpoint as well. Hence the algorithm behaves very efficiently.

Theorem 29 Procedure `solve` defined in Appendix 1, given a pair (β, p_1) and a substitution preserving tail-recursive program P, spends at most

$$\sum_{i=1}^{n}(l_i + 1) \leq O(nl)$$

iterations in the `repeat` loop of `solve_goal`, where n is the size of the static call tree of P and p_1, the predicate symbols in the tree are $p_1, \ldots, p_n$, and $l = max\{l_i\}$.

[1] We omit the arguments in the following to increase legibility.

5.3.2 Substitution Increasing Programs

When the abstract domain is not accurate enough, tail-recursive programs, even built with techniques like structural induction, might not be substitution preserving. The loss of information might induce the presence of recursive calls with substitutions greater than the initial substitution. We therefore consider the substitution increasing property for tail-recursive programs.

The combination of tail-recursive programs with the substitution increasing property implies that a node in the static call tree receives a sequence of input substitutions $\beta_1, \ldots, \beta_n$ such that the total number of executions of the `repeat` loop in Procedure `solve_goal` is at most $l_p + 1$ for each input pair. However nothing prevents the algorithm from exploring various chains such that the complexity depends on s.

Theorem 30 Procedure `solve` defined in Appendix 1, given a pair (β, p_1) and a substitution increasing tail-recursive program P, spends at most

$$\sum_{i=1}^{n}((l_i + 1)s_i) \leq O(nls)$$

iterations in the `repeat` loop of `solve_goal`, where n is the size of the static call tree of P and p_1, the predicate symbols in the tree are $p_1, \ldots, p_n$, $l = max\{l_i\}$, and $s = max\{s_i\}$.

5.3.3 General Case

We now consider tail-recursive programs without behavioural assumptions. Complex recursions are possible giving the following result.

Theorem 31 Procedure `solve` defined in Appendix 1, given a pair (β, p_1) and a tail-recursive program P, spends at most

$$\sum_{i=1}^{n}[l_i(\frac{{s_i}^2}{2} + \frac{s_i}{2} + 1)] \leq O(nls^2)$$

iterations in the `repeat` loop of `solve_goal`, where n is the size of the static call tree of P and p_1, the predicate symbols in the tree are $p_1, \ldots, p_n$, $l = max\{l_i\}$, and $s = max\{s_i\}$.

5.4 Locally Recursive Programs

Definition 32 A program is locally recursive if its clauses are of the form

`p :- q1,...,qn`

where `qi` is either `p` or do not call `p` recursively.

Intuitively it means that there can be several recursive calls but no mutually recursive procedures. This is a rather large class of programs including, for instance, divide and conquer algorithms.

Without behavioural assumptions, this class inherits the results from the general case of tail-recursive programs so that the algorithm is $O(nls^2)$. The intuitive justification is that allowing several recursive calls does not increase neither the longest chain of dependencies nor the number of elements to explore in the abstract domains in the worst case.

It is useful to consider locally recursive programs satisfying the substitution preserving assumption as this is likely to occur frequently in practice.

Theorem 33 Procedure solve defined in Appendix 1, given a pair (β, p_1) and a substitution preserving locally recursive program P, spends at most

$$\sum_{i=1}^{n}(l_i + 1) \leq O(nl)$$

iterations in the repeat loop of solve_goal, where n is the size of the static call tree of P and p_1, the predicate symbols in the tree are $p_1, \ldots, p_n$, and $l = max\{l_i\}$.

The result obtained for locally recursive programs with the substitution preserving property illustrates the importance of the property. The algorithm is very efficient in that case, thanks in part to the $ADJUST$ operation. Indeed, $ADJUST$, when updating $sat(\beta, p)$ to β_{out}, makes sure that all $sat(\alpha, p)$ with $\alpha \geq \beta$ are at least β_{out}. This behaviour, in conjunction with the fact that the substitution preserving property entails that the successive input substitutions make up a decreasing chain, implies that the smallest substitution is computed first, propagating its result to all other substitutions in the chain. Needless to say, the monotonicity of the abstract operations plays a major role in the efficiency of the algorithm for programs with the substitution preserving and substitution increasing properties.

5.5 Mutually Recursive Programs

In this subsection, we consider two classes of mutually recursive procedures, MRC-0 and MRC-m. In both classes, programs are constructed from built-ins and a set of mutually recursive predicates. We take the convention of representing by $P[p_1, \ldots, p_n]$ a program built from the predicates $p_1, \ldots, p_n$. The restriction of considering only builtins besides the mutually recursive predicates is justified by the increased simplicity of the presentation. There is no difficulty in generalizing the results. The interest of these classes will be made clear in the final discussion.

5.5.1 The Class MRC-0

In the class MRC-0, a program $P[p_1, \ldots, p_n]$ has to satisfy two constraints: the procedure calls must be terminal and the procedure defining p_i can only call p_{i+1} with the assumption that $p_{n+1} = p_1$.

Definition 34 A program $P[p_1, \ldots, p_n]$ is in MRC-0 iff all the clauses defining predicate p_i with $1 \leq i \leq n$ are of the form $\mathtt{p}_i$:- $\mathtt{b}_1, \ldots, \mathtt{b}_p$ except one of the form

$$\mathtt{p}_i \text{ :- } \mathtt{b}_1, \ldots, \mathtt{b}_p, \mathtt{p}_{i+1}$$

assuming that $\mathtt{b}_1, \ldots, \mathtt{b}_p$ are builtins and $\mathtt{p}_{n+1} = \mathtt{p}_1$.

Theorem 35 Procedure solve defined in Appendix 1, given a program $P[p_1, \ldots, p_n]$ in MRC-0 that is substitution preserving wrt (β_1, p_1) and a pair (β_1, p_1) as inputs, spends at most $n(l+1) = O(nl)$ iterations in the repeat loop of solve_goal, where $l = min\{l_1, \ldots, l_n\}$.

5.5.2 The class MRC-m

Definition 36 A program $P[p_1,\ldots,p_n]$ is in MRC-m iff all the clauses defining predicate p_i with $1 \leq i < n$ are of the form

$$\mathtt{p}_i \texttt{ :- } \mathtt{b}_1,\ldots,\mathtt{b}_p \quad \text{or} \quad \mathtt{p}_i \texttt{ :- } \mathtt{b}_1,\ldots,\mathtt{b}_p,\mathtt{p}_i \quad \text{or} \quad \mathtt{p}_i \texttt{ :- } \mathtt{b}_1,\ldots,\mathtt{b}_p,\mathtt{p}_{i+1}$$

and the clauses of p_n are of the form

$$\mathtt{p}_n \texttt{ :- } \mathtt{b}_1,\ldots,\mathtt{b}_p \quad \text{or} \quad \mathtt{p}_n \texttt{ :- } \mathtt{b}_1,\ldots,\mathtt{b}_p,\mathtt{p}_n \quad \text{or} \quad \mathtt{p}_n \texttt{ :- } \mathtt{b}_1,\ldots,\mathtt{b}_p,\mathtt{p}_j \ (j \leq m)$$

assuming that $\mathtt{b}_1,\ldots,\mathtt{b}_p$ are builtins.

Theorem 37 Procedure `solve` defined in Appendix 1, given a program $P[p_1,\ldots,p_n]$ in MRC-m that is substitution preserving wrt (β_1,p_1) and a pair (β_1,p_1) as inputs, spends at most

$$\sum_{i=m+1}^{n}(l_i+1)+(n-m)\sum_{i=1}^{m}l_i+\sum_{i=1}^{m}[(m-i+1)l_i]+m \ \leq \ O(nml)$$

iterations in the `repeat` loop of `solve_goal`, where $l = max\{l_1,\ldots,l_n\}$.

5.6 Arbitrary Programs

Without assumptions on the form and behaviour of the program, we have the following result.

Theorem 38 Procedure `solve` defined in Appendix 1 , given a pair (β,p_1) and a program P, spends at most

$$\sum_{i=1}^{n}[(n-i+1)l_i(\frac{{s_i}^2}{2}+\frac{s_i}{2})+s_i]-\sum_{i=1}^{n-1}i \quad \leq \quad O(n^2ls^2)$$

iterations in the `repeat` loop of `solve_goal`, where n is the size of the static call tree of P and p_1, the predicate symbols in the tree are $p_1,\ldots,p_n$, and $l = max\{l_i\}$, and $s = max\{s_i\}$.

6 Discussion and Conclusion

This paper has presented for the first time, to the best of our knowledge, a fixpoint framework for abstract interpretation, an efficient & accurate algorithm based on the semantics, and a complexity analysis of the algorithm. The separation of the framework and the algorithm allows for an independent study of correctness, efficiency, and precision. Moreover it enables to analyse the influence of some properties on the efficiency of the algorithm. For instance, monotonicity is an important property to achieve efficiency.

The complexity analysis indicates the practicability of the algorithm (provided that the abstract domain be practical as well which is a major but independent issue). When the abstract domain is fixed, the algorithm is at worst quadratic in the size of the program and probably linear in practice. Of course efficient implementation techniques are required to achieve the bounds.

The algorithm presented in the paper is also significantly more efficient than those based on techniques like extension-tables and OLD-resolution. Another version of the algorithm (see [9])

using these techniques is $O(n^2l)$ for the class MRC-0 instead of $O(nl)$ for our algorithm. This is significant as mutual recursion appears in various ways in abstract interpretation of logic programs.

Further work will include adaptations of the algorithm to take into account other semantics since the improvements we proposed are independent of the semantics. Hence techniques to derive automatically an efficient algorithm from a declarative semantics are an important topic for further research.

Acknowledgments

Maurice Bruynooghe provided the initial motivation for this research by writing [3]. We gratefully thank him as well as Bruno Desart, Yves Deville, and Henri Leroy for helpful discussions on this research.

References

[1] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. (To Appear).

[2] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 1990. (To Appear).

[3] M. Bruynooghe and al. Abstract Interpretation: Towards the Global Optimization of Prolog Programs. In *Proc. 1987 Symposium on Logic Programming*, pages 192–204, San Francisco, CA, August 1987.

[4] M. Codish, J. Gallagher, and E. Shapiro. Using Safe Approximations of Fixed Points for Analysis of Logic Programs. In *Meta-programming in Logic Programming*, Bristol, UK, 1989.

[5] C. Codognet, P. Codognet, and J.M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *Proceedings of the North-American Conference on Logic Programming (NACLP-90)*, Austin, Tx, October 1990.

[6] J. Gallagher. The Derivation of an Algorithm for Program Specialisation. In *Seventh International Conference on Logic Programming*, Jerusalem, Israel, June 1990.

[7] N.D. Jones and A. Mycroft. Dataflow Analysis of Applicative Programs using Minimal Function Graphs. In *Proceedings of 13th ACM symposium on Principles of Programming Languages*, pages 123–142, St. Petersburg, Florida, 1986.

[8] N.D. Jones and H. Sondergaard. *A Semantics-Based Framework for the Abstract Interpretation of Prolog*, volume Abstract Interpretation of Declarative Languages, pages 123–142. Ellis Horwood, 1987.

[9] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. Efficient and Accurate Algorithms for the Abstract Interpretation of Prolog Programs. Research Paper RP-90/9, University of Namur, August 1990.

[10] K. Marriot and H. Sondergaard. Notes for a Tutorial on Abstract Interpretation of Logic Programs. North American Conference on Logic Programming, Cleveland, Ohio, 1989.

[11] K. Marriott and H. Sondergaard. Bottom-up Abstract Interpretation of Logic Programs. In *Proc. of Fifth International Conference on Logic Programming*, pages 733–748, Seattle, WA, August 1988.

[12] K. Marriott and H. Sondergaard. Semantics-based Dataflow Analysis of Logic Programs. In *Information Processing-89*, pages 601–606, San Fransisco, CA, 1989.

[13] C. Mellish. *Abstract Interpretation of Prolog Programs*, volume Abstract Interpretation of Declarative Languages, pages 181–198. Ellis Horwood, 1987.

[14] U. Nilsson. *A Systematic Approach to Abstract Interpretation of Logic Programs.* PhD thesis, Department of Computer and Information Science, Linkoeping University, Linkoeping (Sweden), December 1989.

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

Appendix 1: The Generic Abstract Interpretation Algorithm

```
procedure solve(in β_in,p; out sat,dp)
begin
   sat := ∅;
   dp := ∅;
   solve_goal(β_in,p,∅,sat,dp)
end
```

Figure 2: Procedure solve

```
procedure solve_goal(in β_in,p,suspended; inout sat,dp)
begin
   if (β_in,p) ∉ (dom(dp) ∪ suspended) then
   begin
      if (β_in,p) ∉ dom(sat) then
         sat := EXTEND(β_in,p,sat);
      repeat
         β_out := ⊥;
         EXT_DP(β_in,p,dp);
         for i := 1 to m with c_1,...,c_m clauses-of p do
         begin
            solve_clause(β_in,p,c_i,suspended ∪ {(β_in,p)},β_aux,sat,dp);
            β_out := UNION(β_out,β_aux)
         end;
         if ¬ (β_out ≤ sat(β_in,p)) then
         begin
            (sat,modified) := ADJUST(β_in,p,β_out,sat);
            dp := dp \ REMOVE_DP(modified,dp)
         end
      until (β_in,p) ∈ dom(dp)
   end
end
```

Figure 3: Procedure solve_goal

```
procedure solve_clause(in β_in,p,c,suspended; out β_out; inout sat,dp)
begin
   β_ext := EXTC(c,β_in);
   for i := 1 to m with b_1,...,b_m body-of c do
   begin
      β_aux := RESTRG(b_i,β_ext);
      switch (b_i) of
      case x_j = x_k:
         β_int := AI_VAR(β_aux)
      case x_j = f(...):
         β_int := AI_FUNC(β_aux,f)
      case q(...):
         solve_goal(β_aux,q,suspended,sat,dp);
         β_int := sat(β_aux,q);
         if (β_in,p) ∈ dom(dp) then
            ADD_DP(β_in,p,β_aux,q,dp)
      end;
      β_ext := EXTG(b_i,β_ext,β_int)
   end;
   β_out := RESTRC(c,β_ext)
end
```

Figure 4: The Procedure solve_clause