Multivariant Non-Failure Analysis via Standard Abstract Interpretation

F. Bueno*, P. López-García*, and M. Hermenegildo*,**

(*) School of Computer Science, Technical University of Madrid (UPM)
(**) Depts. of Comp. Science and El. and Comp. Eng., U. of New Mexico (UNM)

{bueno,pedro.lopez,herme}@fi.upm.es
http://www.cliplab.org/

Abstract. Non-failure analysis aims at inferring that calls to the predicates of a program will never fail. This type of information has many applications in functional/logic programming. It is essential for determining lower bounds on the computational cost of calls, useful in the context of program parallelization, instrumental in partial evaluation and other program transformations, and has also been used in query optimization. In this paper, we re-cast the non-failure analysis proposed by Debray et al. as an abstract interpretation, which not only allows to investigate it from a standard and well studied theoretical framework, but has also several practical advantages. It allows us to incorporate non-failure analysis into a standard, generic abstract interpretation engine. The non-failure analysis thus benefits from the fixpoint propagation algorithm, which leads to improved information propagation. Also, the analysis takes advantage of the multi-variance of the generic engine, so that it is now able to infer separate non-failure information for different call patterns for a given predicate in a program. Moreover, the implementation is simpler, and allows to perform non-failure and covering analyses alongside other abstract interpretation based analyses, such as those for modes and types, in the same framework. Finally, besides the precision improvements and the additional simplicity, we also show improvements in efficiency of the analysis from our implementation in the Ciao/CiaoPP multiparadigm programming system.

1 Introduction

Non-failure analysis involves detecting at compile time that, for any call belonging to a particular (possibly infinite) class of calls, a predicate will never fail. As an example, consider a predicate defined by the following two clauses:

$abs(X,Y)$:- $X \geq 0, YisX$.
$abs(X,Y)$:- $X < 0, Yis - X$.

and assume that we know that this predicate will always be called with its first argument bound to an integer, and the second argument a free variable. Obviously, for any particular call, one or the other of the tests '$X \geq 0$' and '$X < 0$' may fail; however, taken together, one of them will always succeed. Thus, we can infer that calls to the predicate will never fail.

Being able to determine statically that a predicate will not fail has many applications. It is essential for determining lower bounds on the computational cost of goals since without such information a lower bound of almost zero (corresponding to an early failure) must often be assumed [9]. Detecting non-failure is also very useful in the context of parallelism because it allows avoiding unnecessary speculative parallelism and ensuring no-slowdown properties for the parallelized programs (in addition to using the lower bounds mentioned previously to perform granularity control) [10]. Non-failure information is also instrumental in partial evaluation and other program transformations, such as reordering of calls, and has also been used in query optimization in deductive databases [7]. Non-failure information is useful in program debugging where it allows verifying user assertions regarding non-failure of predicates [12, 13]. Finally, similar techniques can be used to detect the absence of errors or exceptions when running particular predicates.

A practical non-failure analysis has been proposed by Debray *et al.* [8]. In a similar way to the example above, this approach relies on first inferring mode and type information, and

then testing that the constraints in the clauses of the predicate are entailed by the types of the input arguments, which is called a *covering* test. Covering cannot be inferred by examining the constraints of each clause separately: it is necessary to collect them together and examine the behavior of the predicate as a whole. Furthermore, non-failure of a given predicate depends on non-failure of other predicates being called and also possibly on the constraints in such predicates.
While [8] proposed the basic ideas behind non-failure analysis, only a very simple, monovariant algorithm was proposed for propagating the nonfailure information. In our experience since that proposal, we have found a need to improve it in several ways. First, information propagation needs to be improved, which leads us to a fixpoint propagation algorithm. Furthermore, the analysis really needs to be *multi–variant*, which means that the analysis should be able to infer separate non-failure (and covering) information for different call patterns for a given predicate in a program. This is illustrated by the following example which, although simple, captures the very common case where the same (library) procedure is called from a program (in different points) for different purposes (i.e., with different argument instantiation states):

Example 1 Consider the (exported) predicate `mv`/3 (which uses the library predicate `qsort`/2), defined for the sake of discussion as follows:

```
mv(A,B,C):- qsort(A,B), !, C = B.
mv(A,B,C):- append(A,B,D), qsort(D, C).
```

Assume the following entry assertion for `mv`/3:

```
:- entry mv(A,B,C) : (list(A, num), list(B, num), var(C)).
```

which means that the predicate `mv(A,B,C)` will be called with `A` and `B` bound to lists of numbers, and `C` a free variable.
A multi–variant non-failure analysis would infer two call patterns for predicate `qsort`/2 (i.e. two different ways in which the predicate `qsort`/2 is called):

1. The call pattern `qsort(A,B): (list(A,num), list(B,num))`, for which the analysis infers that *can fail* and is *not covered*, and
2. the call pattern `qsort(A,B): (list(A,num), var(B))`, for which the analysis infers that will *not fail* and *is covered*.

This in turn allows the analysis to infer that the predicate `mv`/3 will *not fail* and *is covered* (for the call pattern expressed by the entry assertion).
However, a monovariant analysis only considers one call pattern per predicate. In particular, for predicate `qsort`/2, the call pattern used is `qsort(A,B): (list(A,num), term(B))`[1] (which is the result of "collapsing" all call patterns which can appear in the program, so that precision is lost), for which it infers that `qsort`/2 *can fail* and *is not covered*. This causes the analysis to infer that the predicate `mv`/3 *can fail* (since the calls to `qsort`/2 in both clauses of predicate `mv`/3 are detected as failing) and *is covered*. □

In order to address the different shortcomings of [8] in this paper we start by casting the ideas behind non-failure and covering analysis as an abstract interpretation [4]. This then allows us to incorporate non-failure analysis into a (somewhat modified) standard, generic abstract interpretation engine. This has several advantages. First of all, the analysis is now based on a standard and well studied theoretical framework. But, most importantly, being able to take advantage of standard and well developed analysis engines allows us to obtain a simpler and more efficient implementation, with better propagation of information, performing an efficient fixpoint. The non-failure and covering analyses can be performed alongside other abstract interpretation based analyses, such as those for modes and types, in the same framework. Furthermore, the analysis that we obtain is *multi–variant* (on calls and successes) thus inferring separate non-failure (and covering) information for different

[1] `term(B)` means that argument `B` can be bound to any term.

call patterns for a given predicate in a program. Finally, the abstract domain for non-failure can be easily enhanced to define a domain for determinacy of predicates.
Abstract Interpretation [4] is often proposed as a means for inferring properties of programs at compile–time. The generally intended purpose of this process is to use such knowledge to perform several types of optimizations aimed at improving some characteristics of the program or its execution. It was shown by Bruynooghe [2], Jones and Sondergaard [15], Debray [6], and Mellish [17] that this technique can be extended to flow analysis of programs in logic programming languages, and several frameworks or particular analyses have evolved since (e.g. [16, 20, 22, 23]).
Abstract interpretation has the advantage of allowing the systematic design and verification of data–flow analyses by formalizing the relation between analysis and semantics. Therefore, abstract interpretation is inherently semantics sensitive, different semantic definition styles yielding different approaches to program analysis. For logic programs we distinguish between two main approaches, namely *bottom–up* analysis and *top–down* analysis. While the top–down approach propagates the information in the same direction as SLD-resolution does, the bottom–up approach propagates the information as in the computation of the least fixpoint of the immediate consequences operator T_P. In addition, we distinguish between *goal dependent* and *goal independent* analyses. A goal dependent analysis provides information about the possible behaviors of a specified (set of) initial goal(s) and a given logic program. In contrast, a goal independent analysis considers only the program in itself.
In this paper we use a goal dependent framework, since non-failure analysis is inherently goal dependent. In [3], Bruynooghe describes a framework for the goal-dependent, top–down abstract interpretation of logic programs. In this framework, abstract interpretation is carried out by constructing an *abstract and–or tree* in a top–down fashion for a given (abstract) query and program. Essentially, starting with the abstract call substitution for the query, abstract substitutions at all points of the abstract and–or tree are computed and finally, the success substitution for the query is computed. If the given program has recursive predicates, then fixpoint computation is necessary. We use the PLAI/CiaoPP framework [12, 13], wich follows the ideas above, but incorporating a number of optimizations and efficient fixpoint algorithms, described in [18, 19, 14].
In the rest of the paper, after the necessary preliminaries (Section 2), we present the abstract interpretation framework (Section 3) used in our analyses, the abstract domain (Section 4), and the implementation (Section 5), finalizing with conclusions (Section 6).

2 Preliminaries

We will denote $\mathcal{C}$ the universal set of constraints. A *constraint* is essentially a conjunction (disjunction) of expressions built from predefined predicates (such as term equations or inequalities over the reals) whose arguments are constructed using predefined functions (such as real addition). We let $\overline{\exists}_L\theta$ be the constraint θ restricted to the variables of the syntactic object L. We denote constraint entailment by $\models$, so that $c_1 \models c_2$ denotes that c_1 entails c_2.
An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H :- B where H, the *head*, is an atom and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules. The *definition* of an atom A in program P, $defn_P(A)$, is the set of variable renamings of rules in P such that each renaming has A as a head and has distinct new local variables.
The operational semantics of a program is in terms of its "derivations" which are sequences of reductions between "states". A *state* $\langle G \mid \theta \rangle$ consists of a goal G and a constraint store (or *store* for short) θ. A state $\langle L :: G \mid \theta \rangle$, where L is a literal and :: denotes concatenation of sequences, can be *reduced* as follows:

1. If L is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G \mid \theta \wedge L \rangle$.

2. If L is an atom, it is reduced to $\langle B :: G \mid \theta \rangle$ for some rule $(L$:- $B) \in defn_P(L)$.

assuming for simplicity that the underlying constraint solver is complete. We use $S \rightsquigarrow_P S'$ to indicate that in program P a reduction can be applied to state S to obtain state S'. Also, $S \rightsquigarrow_P^* S'$ indicates that there is a sequence of reduction steps from state S to state S'. A

derivation from state S for program P is a sequence of states $S_0 \leadsto_P S_1 \leadsto_P ... \leadsto_P S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. Given a non-empty derivation D, we denote by *curr_goal(D)* and *curr_store*(D) the first goal and the store in the last state of D, respectively. E.g., if D is the derivation $S_0 \leadsto_P^* S_n$ with $S_n = \langle g :: G \mid \theta \rangle$ then $curr\_goal(D) = g$ and $curr\_store(D) = \theta$. A *query* is a pair (L, θ) where L is a literal and θ a store for which the CLP system starts a computation from state $\langle L \mid \theta \rangle$. The set of all derivations from Q for P is denoted $derivations(P, Q)$. We will denote sets of queries by $\mathcal{Q}$. We extend *derivations* to operate on sets of queries as follows: $derivations(P, \mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} derivations(P, Q)$.
The observational behavior of a program is given by its "answers" to queries. A finite derivation from a query (L, θ) for program P is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a query (L, θ) is *successful* if the last state is of the form $\langle nil \mid \theta' \rangle$, where *nil* denotes the empty sequence. The constraint $\bar{\exists}_L \theta'$ is an *answer* to (L, θ) We denote by $answers(P, Q)$ the set of answers to query Q. A finished derivation is *failed* if the last state is not of the form $\langle nil \mid \theta \rangle$. Note that $derivations(P, \mathcal{Q})$ contains not only finished derivations but also all intermediate derivations from a query. A query Q *finitely fails* in P if $derivations(P, Q)$ is finite and contains no successful derivation.

Abstract Interpretation. Abstract interpretation [4] is a technique for static program analysis in which execution of the program is simulated on an *abstract domain* (D_α) which is simpler than the actual, *concrete domain* (D). An abstract value is a finite representation of a, possibly infinite, set of actual values in the concrete domain. The set of all possible abstract semantic values represents an abstract domain D_α which is usually a complete lattice or cpo which is ascending chain finite. However, for this study, abstract interpretation is restricted to complete lattices over sets both for the concrete $\langle 2^D, \subseteq \rangle$ and abstract $\langle D_\alpha, \sqsubseteq \rangle$ domains.
Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : 2^D \to D_\alpha$, and *concretization* $\gamma : D_\alpha \to 2^D$, such that
$\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad$ and $\quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. In general $\sqsubseteq$ is induced by $\subseteq$ and α. Similarly, the operations of *least upper bound* $(\sqcup)$ and *greatest lower bound* $(\sqcap)$ mimic those of 2^D in a precise sense:

$$\forall \lambda, \lambda' \in D_\alpha : \ \lambda \sqsubseteq \lambda' \Leftrightarrow \gamma(\lambda) \subseteq \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcup \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cup \gamma(\lambda_2) = \gamma(\lambda')$$
$$\forall \lambda_1, \lambda_2, \lambda' \in D_\alpha : \ \lambda_1 \sqcap \lambda_2 = \lambda' \Leftrightarrow \gamma(\lambda_1) \cap \gamma(\lambda_2) = \gamma(\lambda')$$

Goal dependent abstract interpretation takes as input a program P, an abstract domain D_α, and a description $\mathcal{Q}_\alpha$ of the possible initial queries to the program given as a set of abstract queries. An *abstract query* is a pair (L, λ), where L is an atom (for one of the exported predicates) and $\lambda \in D_\alpha$ an abstract constraint which describes the initial stores for L. A set of abstract queries $\mathcal{Q}_\alpha$ represents a set of queries, denoted $\gamma(\mathcal{Q}_\alpha)$, which is defined as $\gamma(\mathcal{Q}_\alpha) = \{(L, \theta) \mid (L, \lambda) \in \mathcal{Q}_\alpha \wedge \theta \in \gamma(\lambda)\}$. Such an abstract interpretation computes a set of triples $Analysis(P, \mathcal{Q}_\alpha, D_\alpha) = \{\langle L_p, \lambda^c, \lambda^s \rangle \mid p \text{ is a predicate of } P\}$, where L_p is a (program) atom for predicate p. Note that, the analysis being multivariant (on calls), it may compute several tuples of the form $\langle L_p, \lambda^c, \lambda^s \rangle$ for different call patterns $\langle L_p, \lambda^c \rangle$ of each predicate p (including different program atoms L_p). If p is detected to be dead code then $\lambda^c = \lambda^s = \bot$. As usual in abstract interpretation, $\bot$ denotes the abstract constraint such that $\gamma(\bot) = \emptyset$, whereas $\top$ denotes the most general abstract constraint, i.e., $\gamma(\top) = D$.

3 The Abstract Interpretation Framework

The PLAI abstract interpretation system is a top–down framework based on the abstract interpretation framework of Bruynooghe [3] with the optimizations described in [18]. The framework works on an abstraction of the (SLD) AND-OR trees of the execution of a program for given entry points. The abstract AND-OR graph makes it possible to provide information at each program point, a feature which is crucial for many applications (such as, for example, reordering, automatic parallelization or garbage collection).

Program points and abstract substitutions are related as follows. Consider a clause h :- $p_1, \ldots, p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal p_i, $1 \leq i \leq n$ in this clause. Then λ_i and λ_{i+1} are, respectively, the *abstract call substitution* and the *abstract success substitution* for the subgoal p_i. For this same clause, λ_1 is the *abstract entry substitution* and λ_{n+1} is the *abstract exit substitution*. Entry and exit substitutions are denoted respectively β_{entry} and β_{exit} when projected on the variables of the clause head.

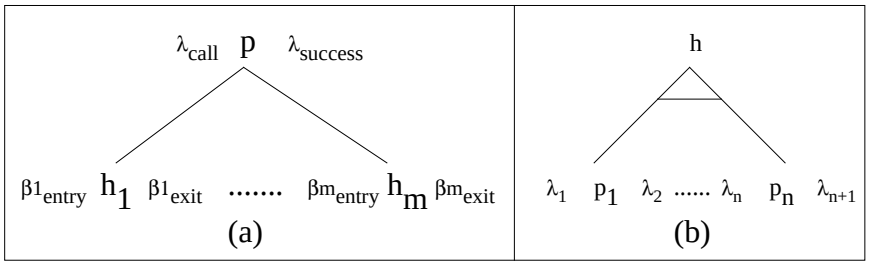

Fig. 1. Illustration of the Top–Down Abstract Interpretation Process

Computing the *success* substitution from the *call* substitution is done as follows (see Figure 1(a)). Given a call substitution λ_{call} for a subgoal p, let $h_1, \ldots, h_m$ be the heads of clauses which unify with p. Compute the entry substitutions $\beta 1_{entry}, \ldots, \beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit}, \ldots, \beta m_{exit}$ as explained below. Compute the success substitutions $\lambda 1_{success}, \ldots, \lambda m_{success}$ from the exit substitutions corresponding to these clauses. At this point, all different success substitutions can be considered for the rest of the analysis, or a single success substitution $\lambda_{success}$ for subgoal p computed by means of an aggregation operation for $\lambda 1_{success}, \ldots, \lambda m_{success}$. This aggregator is usually the LUB (least upper bound) operation of the abstract domain.

The choice of following the analysis with one or more success substitutions has been considered in [21] as the *grain level* of the *abstract result* of the analysis. Computing the LUB and considering a single success substitution is the coarsest level possible. Considering all distinct success substitutions computed for the different clauses is the finest level. In a different terminology, this is related to whether the analysis is *multi–variant* on successes or not.

Computing the *exit* substitution from the *entry* substitution is straightforward (see Figure 1(b)). Given a clause h :- $p_1, \ldots, p_n$ and an entry substitution β_{entry} for the clause head h, λ_1 is the call substitution for p_1. This one is computed simply by adding to β_{entry} an abstraction for the free variables in the clause. The success substitution λ_2 for p_1 is computed as explained above (essentially, by repeating this same process for the clauses which match p_1). Similarly, $\lambda_3, \ldots, \lambda_{n+1}$ are computed. The exit substitution B_{exit} for this clause is precisely the projection onto h of λ_{n+1}.

If, from a different subgoal in the program, a different entry substitution is computed for an already analyzed clause, different call substitutions will appear (for p_1 and possibly the other subgoals). These substitutions can be collapsed by means of the LUB operation of the underlying abstract domain, or a different node in the graph can be computed. In the latter solution, different nodes exist in the graph for each call substitution and subgoal, thus yielding an analysis which is *multi–variant* on calls.

Note that the framework itself is domain independent. Given this basic framework, it is clear that a particular analysis needs to:

- Define an abstract domain and abstract unification, and the $\sqsubseteq$ relation, which in turn defines the LUB operation $\sqcup$. Abstract unification is divided into two parts in this framework, as follows:
- Describe how to compute the entry substitution for a clause C given a subgoal p (which unifies with the head of C) and its call substitution.

– Describe how to compute the success substitution for a subgoal p given its call substitution and the exit substitution for a clause C whose head unifies with p.

We formalize the framework with the functions *entry_to_exit* and *call_to_success* defined in Figure 2. The domain dependent functions used in these definitions are:

– $call\_to\_entry(p(\bar{u}), C, \lambda)$ which gives an abstract substitution describing the effects on $vars(C)$ of unifying $p(\bar{u})$ with $head(C)$ given an abstract substitution λ describing $\bar{u}$,

– $exit\_to\_success(\lambda, p(\bar{u}), C, \beta)$ which gives an abstract substitution describing $\bar{u}$ accordingly to β (which describes $vars(head(C))$) and the effects of unifying $p(\bar{u})$ with $head(C)$ under the abstract substitution λ describing $\bar{u}$,

– $extend(\lambda, \lambda')$ which extends abstract substitution λ to incorporate the information in λ' in a way that it is still consistent,

– $project\_in(\bar{u}, \lambda)$ which extends abstract substitution λ so that it refers to all of the variables $\bar{u}$,

– $project\_out(\bar{u}, \lambda)$ which restricts the abstract substitution λ to only the variables $\bar{u}$.

$entry\_to\_exit(C, \beta_{entry}) \equiv$
  $A_1 := project\_in(vars(C), \beta_{entry})$;
  For $i := 1$ to $length(C)$ do
    $A_{i+1} := call\_to\_success(q_i(\bar{u}_i), A_i)$);
  return $project\_out(vars(head(C)), A_{n+1})$;

$call\_to\_success(p(\bar{u}), \lambda_{call}) \equiv$
  $\lambda := project\_out(\bar{u}, \lambda_{call})$;
  $\lambda' := \epsilon$;
  For each clause C which matches $p(\bar{u})$ do
    $\beta_{exit} := entry\_to\_exit(C, call\_to\_entry(p(\bar{u}), C, \lambda))$;
    $\lambda' := \lambda' \sqcup exit\_to\_success(\lambda, p(\bar{u}), C, \beta_{exit})$;
  od;
  return $extend(\lambda_{call}, \lambda')$;

Fig. 2. The Top–Down Framework

In the presence of recursive predicates, analysis requires of a fixpoint computation. In [18] a fixpoint algorithm was proposed for the framework that localizes fixpoint computations to only the strongly connected components of (mutually) recursive predicates. Additionally, fixpoint computation is initiated with an initial approximation to the fixpoint which is computed from the non-recursive clauses of the recursive predicate. Fixpoint convergence is accelerated by updating this value with the information from every clause analyzed in turn. The algorithm is (schematically) shown in Figure 3. For a complete description see [18].

4 Abstract Framework, Domain, and Operations for Non-Failure Analysis

In the non-failure analysis, the covering test is instrumental. In fact, covering can be seen as a notion that characterizes the fact that execution of a query will not finitely fail, i.e., if it has finished derivations then at least one is successful. Note that, as in [8] non-failure does not imply success: a predicate that is non-failing may nevertheless not produce an answer because it does not terminate.

```
call_to_success_recursive(p(ū), λ_call) ≡
    λ := project_out(ū, λ_call);
    λ' := ϵ;
    For each non-recursive clause C which matches p(ū) do
        β_exit := entry_to_exit(C, call_to_entry(p(ū), C, λ));
        λ' := λ' ⊔ exit_to_success(λ, p(ū), C, β_exit);
    od;
    λ'' := fixpoint(p(ū), λ, λ');
    return extend(λ_call, λ'');

fixpoint(p(ū), λ, λ') ≡
    λ'' := λ';
    For each recursive clause C which matches p(ū) do
        β_exit := entry_to_exit(C, call_to_entry(p(ū), C, λ));
        λ'' := λ'' ⊔ exit_to_success(λ, p(ū), C, β_exit);
    od;
    If λ'' = λ'
      then return λ''
      else return fixpoint(p(ū), λ, λ'');
```

Fig. 3. The Fixpoint Computation

Definition 1 (Covering). *Given computation state* $\langle g :: G \mid \theta \rangle$ *in the execution of program* P*, define the* global answer constraint *of goal* g *in store* θ *as:*

$$c = \vee\{\ curr\_store(D'_i) \mid D'_i \in derivations(P, \langle g, \theta \rangle)\ and\ is\ maximal\ \}$$

Let $\bar{u}$ *denote the variables of* g *already constrained in* θ*, call them the* input *variables. We say that* g *is* covered *in* θ *iff* $\overline{\exists}_{\bar{u}}\theta \models \overline{\exists}_{\bar{u}}c$.

It is not difficult to show that, in a pure language, i.e., in a language where failure can only be caused by constraint store inconsistency, covering is a sufficient condition for non-failure. Indeed, if g is covered in θ, i.e., $\overline{\exists}_{\bar{u}}\theta \models \overline{\exists}_{\bar{u}}c$, then one of the disjunctions in (the projection of) c is entailed. This corresponds to a (maximal) derivation of $\langle g, \theta \rangle$, and this derivation cannot be failed, since, if it was, it would be inconsistent, and no inconsistent constraint can be entailed by a consistent one. Therefore, either such derivation is infinite, or, if finite, it is successful. Thus, we have:

If g is *covered* in θ then $\langle g, \theta \rangle$ does not finitely fail.

A key issue in non-failure analysis will thus be how to approximate the current store and the global answer constraint so that covering can be effectively and accurately approximated. In [8] such approximation is defined in the following terms: A goal is non-failing if there is a subset of clauses of the predicate which do not fail and which match the input types of the goal. This "matching" is the so-called *covering test*, and basically amounts to the analysis being able to gather enough constraints on the input variables of the goal to be able to prove that for each of the variables any element in the corresponding type satisfies at least the constraint imposed in one of the clauses of the predicate. An analysis for non-failure thus needs to traverse the clauses of a predicate to check non-failure of the clause body goals, collect constraints that approximate the global answer constraint, and finally check that they cover the input types of the original goal. In the rest of this section, we show how to accommodate the abstract interpretation based framework of the previous section to perform these tasks, and define an abstract domain suitable for them.

4.1 Abstract Domain

The abstractions for non-failure analysis are made of four components. The first two are (abstractions of) constraints that represent the current store and the global answer constraint for the current goal. This is the core part of the domain. The other two components carry the results of the covering test, specifying if the current constraint store covers the global answer constraint, and if this implies that the computation may fail or not.
The covering and non-failure information is represented by values of the set $\mathcal{B} = \{\top, \bar{0}, \bar{1}, \bot\}$, which is a two-point domain with topmost and bottommost elements with the lattice structure depicted in Figure 4. For covering, $\bar{0}$ is interpreted as "not covered" and $\bar{1}$ as covered. For non-failure, $\bar{0}$ is interpreted as "not failing" and $\bar{1}$ as failing.

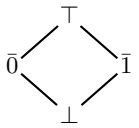

Fig. 4. The Two-point basic lattice

Definition 2 (Abstract Domain). *Let $\mathcal{C}^{\alpha_1}$ and $\mathcal{C}^{\alpha_2}$ be abstract domains for $\mathcal{C}$. The abstract domain for non-failure is the set*

$$\mathcal{F} = \{(s, c, o, f) \mid s \in \mathcal{C}^{\alpha_1}, c \in \mathcal{C}^{\alpha_2}, o \in \mathcal{B}, f \in \mathcal{B}\}$$

The ordering in domain $\mathcal{F}$ is induced from the ordering in $\mathcal{B}$, so that (overloading $\sqsubseteq$):

$$(s_1, c_1, o_1, f_1) \sqsubseteq (s_2, c_2, o_2, f_2) \text{ iff } f_1 \sqsubseteq f_2$$

The covering component encodes in fact the mode and type information that is required for the analysis (in particular, for the covering test). The usual approximation used (e.g., in [8]) for the first two components of an abstract value are types (and modes) for the first one, and a finite set of (concrete) constraints for the second one. This way, the covering test is defined in terms of whether the (input) types imply the (approximated) global answer constraint. We include the covering component (the third one) to make the presentation of the domain self-contained. It can be dropped, under the assumption that mode and type information is available for the same concrete substitutions that correspond to the abstract value $f \in \mathcal{F}$ at hand. Therefore, we can assume given suitable abstractions α_1 and α_2 for the constraint domain $\mathcal{C}$, from which the first two components of elements of $\mathcal{F}$ can be obtained.

Definition 3 (Abstraction Function). *The abstraction of a derivation D in the execution of program P, such that $curr\_store(D) = \theta$ and $curr\_goal(D) = g$, and the input variables and global answer constraint of g in θ are respectively $\bar{u}$ and c, is:*

$$\alpha(D) = (\theta^{\alpha_1}, c^{\alpha_2}, o, f) \text{ where } f = \begin{cases} \bar{1} & \text{if } D \text{ is failed} \\ \bar{0} & \text{otherwise} \end{cases} \quad \text{and } o = \begin{cases} \bar{1} & \text{if } \overline{\exists}_{\bar{u}}\theta^{\alpha_1} \models^{\alpha} \overline{\exists}_{\bar{u}}c^{\alpha_2} \\ \bar{0} & \text{otherwise} \end{cases}$$

It is easy to show that such an abstraction is correct, provided that α_1 and α_2 are also correct abstractions, and that the corresponding abstract covering test ($\models^{\alpha}$) correctly approximates Definition 1. For α_1 we have already mentioned the use of type and mode information. One possibility for α_2 is to use only those constraints appearing explicitly in the clause bodies of the predicate whose covering test is to be performed (the current goal g in the derivation).

Example 2 Consider the following (contrived) predicates:

```
p(X,Y,Z):- X =< Y, q(X,Z).
q(X,Y):- X =< Y.
```

The global answer constraint for `p(X,Y)` is `X =< Y` $\wedge$ `X =< Z`, but it can be approximated simply by `X =< Y`, which is the only constraint appearing in the clauses defining `p`/2. □

One rationale for the above choice might be that collecting all constraints in derivations may not be possible during a compile-time analysis (since such constraints are only known during execution), or may lead to non-termination of the analysis. However, the first problem can be alleviated by proper abstractions of the tests (such as a depth-k abstraction, in a way similar to [5]), and the second problem only occurs for recursive predicates. Thus, the most simple solution to the termination problem is to avoid collecting constraints in recursive calls for the collection of (an approximation to) the global answer constraint.[2]

Example 3 Consider the predicate `sorted`/1 defined below.

```
sorted([]).
sorted([_]).
sorted([X,Y|L]):- X =< Y, sorted([Y|L]).
```

Its global answer constraint includes a constraint for each two elements in the input list, the length of which is not in general known at compile-time. □

Our solution to this problem[3] is to collect only constraints that refer literally to the predicate arguments in the program clause head, which also excludes in general (but not always) the constraints arising from recursive calls.

Example 4 Consider again the predicate `sorted`/1 defined in the previous example. We collect constraints only for the clause head argument `[X,Y|L]`, which amounts to only one constraint: `X =< Y` (since the recursive call does not provide constraints for the head arguments that appear literally in the program).
Consider, on the other hand, predicate `p`/2 of Example 2. In this case the complete global answer constraint for `p(X,Y)` will be collected: `X =< Y` $\wedge$ `X =< Z`, since the two single constraints can be "projected" onto the clause head. □

Note that such a solution yields an under-approximation of the global answer constraints. Given the use of type and mode information, which are in general over-approximations, we have that, for any element $(s, c, o, f) \in \mathcal{F}$, given current constraint store θ and global answer constraint ω, $s = \theta^{\alpha_1}$ is an over-approximation of θ, and $c = \gamma^{\alpha_2}$ is an under-approximation of γ. In this situation, it is not difficult to prove that $\overline{\exists}_{\bar{u}}\theta^{\alpha_1} \models^{\alpha} \overline{\exists}_{\bar{u}}\gamma^{\alpha_2}$ correctly approximates covering: $\overline{\exists}_{\bar{u}}\theta \models \overline{\exists}_{\bar{u}}\gamma$.

4.2 Abstract Operations

Abstract values $(s, c, o, f) \in \mathcal{F}$ are built during analysis in the following way: f is carried along during the abstract computation by the abstract operations below, o is computed from the covering test, c is collected as explained above, and for s, type and mode analysis is performed. Thus, our analysis is in fact three-fold: it carries on mode, type, and non-failure analyses simultaneously. We focus now on the abstract operations for non-failure, given that the operations for types and modes are standard. The abstract operations required by the framework are as follows:

[2] Note that this does not imply that recursive calls are simply ignored. They need be considered to check that it is indeed non-failing, even though its global answer constraint is not computed.

[3] However, we plan to investigate other solutions. In particular, the use of a depth-k abstraction seems to be a very promising one.

– *call_to_entry*$(p(\bar{u}), C, \lambda)$ solves head unification $p(\bar{u}) = head(C)$, and checks that it is consistent with the c component of λ. If it is not, it returns $\bot$, otherwise, the resulting abstraction.
If $p(\bar{u}) \in \mathcal{C}$, i.e., if it happens to be a constraint itself, then no clause C exists, and $p(\bar{u})$ itself is added to the c component. In this case the following *exit_to_success* function is not called.

– *exit_to_success*$(\lambda, p(\bar{u}), C, \beta)$ adds the equations resulting from unification $p(\bar{u}) = head(C)$ to the c component of β and projects it onto the variables $\bar{u}$.
It is the projection performed here that gets rid of useless constraints, like in the case of Example 4. Constraints that cannot be projected onto the (goal) variables $\bar{u}$ are simply dropped in the analysis.

– $\lambda \uplus \lambda'$ adds abstraction λ to the set λ' if λ is non-failing.

– *extend*(λ, λ') performs the covering test for λ' (a set of abstractions); if it is successful, the c component of λ' is merged with that of λ.
This operation uses the covering algorithm described in [8], which takes the global answer constraint c and a *type assignment* for the input variables appearing in c. Given a finite set of variables V, a type assignment over V is a mapping from V to a set of types. This is computed from the type information in the first component of λ. Input variables are determined from the mode information in that same component. The global answer constraint is obtained as the disjunction of the c components of each abstraction in λ'.

4.3 Adapting the Analysis Framework

The framework described in the previous section is not adequate for non-failure analysis. The main reason for this is that the aggregation function for the successive exit abstractions of the different clauses is not the LUB anymore. In non-failure analysis, the constraints for each clause need be gathered together, and a covering test on the set of constraints need be performed. Another difference is that the covering test should only consider constraints from clauses that are not guaranteed to fail altogether;[4] therefore the aggregator must be able to discriminate abstract substitutions on this criterion.
We have adapted the definition of the *call_to_success* function to reflect the aggregation operator. The adapted definition is shown in Figure 5. Note that, as a result of this, λ' in the algorithm is not anymore an abstract substitution, but a set of them. This is input to function *extend*, which is in charge of the covering test.

---

call_to_success$(p(\bar{u}), \lambda_{call}) \equiv$
  $\lambda :=$ *project_out*$(\bar{u}, \lambda_{call})$;
  $\lambda' := \emptyset$;
  For each clause C which matches $p(\bar{u})$ do
    $\beta_{exit} :=$ *entry_to_exit*$(C,$ *call_to_entry*$(p(\bar{u}), C, \lambda))$;
    $\lambda' := \lambda' \uplus$ *exit_to_success*$(\lambda, p(\bar{u}), C, \beta_{exit})$;
  od;
  return *extend*$(\lambda_{call}, \lambda')$;

Fig. 5. The Top–Down Framework for Non-Failure Analysis

When fixpoint computation is required, adapting the framework is a bit more involved. Basically, since the aggregation operator is not LUB, fixpoint detection can not be performed right after the success substitution has been computed. Normally, it is the LUB that is used for updating the successive approximations to the fixpoint value, and fixpoint

[4] Note how this information could be used to improve the results of other analyses.

detection works by simply comparing the initial and the final values for the success substitution. In non-failure analysis, the covering test must be performed first, and only after this one has been performed, the test for the fixpoint can be done. The resulting algorithm is shown in Figure 6. It is basically a simpler fixpoint iterator over the function *call_to_success* abandoning the sophisticated fixpoint computation of Figure 3.

$call\_to\_success\_recursive(p(\bar{u}), \lambda_{call}) \equiv$
  $\lambda := project\_out(\bar{u}, \lambda_{call})$;
  return $fixpoint(p(\bar{u}), \lambda, \bot)$;

$fixpoint(p(\bar{u}), \lambda, \lambda') \equiv$
  $\lambda'' := call\_to\_success(p(\bar{u}), \lambda)$;
  If $\lambda'' = \lambda'$
    then return λ''
    else return $fixpoint(p(\bar{u}), \lambda, \lambda'')$;

Fig. 6. The Fixpoint Computation for Non-Failure Analysis

4.4 A Running Example

We now illustrate our analysis by means of a "running example". Consider the program (fragment) below:

```
qsort(As,Bs):- qsort(As,Bs,[]).

qsort([X|L],R,R2) :-
    partition(L,X,L1,L2),
    qsort(L2,R1,R2),
    qsort(L1,R,[X|R1]).
qsort([],R,R).

partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right):-
    E < C,
    partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
    E >= C,
    partition(R,C,Left,Right1).
```

Let the current abstraction at the point where atom `qsort(As,Bs,[])` is called be

$$(\{list(num, As), var(Bs)\}, true, \overline{1}, \overline{0}),$$

indicating that `qsort` is called with first argument a list of numbers, and second argument a free variable (and third argument an empty list), in a situation where the current clause is still non-failing and covered.
Upon entering the first clause defining `qsort/3`, the result of *call_to_entry* (restricted to the head variables) is

$$(\{num(X), list(num, L), var(R), [](R2)\}, true, \overline{1}, \overline{0}).^5$$

[5] To be more concise, we denote with $[](A)$ that the type of A is that of the empty lists.

plus, additionally, $\{var(R1), var(L1), var(L2)\}$ for the free variables in the clause. Once projected, this gives the call pattern for the first literal in that clause:

$$(\{list(num, L), num(X), var(L1), var(L2)\}, true, \overline{1}, \overline{0}).$$

For the sake of brevity, we omit the analysis of the partition predicate. After the fixpoint computation for this predicate, however, we will have a set of three abstract elements corresponding to the abstraction of the three clauses. For brevity, we express such set as a single abstraction where it is the c component that is a set.[6] Note that this is possible because all other components (types, modes, covering, non-failure) of the abstractions in the set are the same. Thus, we have:

$$\begin{array}{ll}(\{\ list(num, L), num(X), list(num, L1), list(num, L2)\ \}, & \\ \quad \{L = [] \wedge L1 = [] \wedge L2 = [], & \\ \quad L = [E|\_] \wedge E < X \wedge L1 = [E|\_], & \\ \quad L = [E|\_] \wedge E >= X \wedge L2 = [E|\_]\}, & \overline{1}, \overline{0}).\end{array}$$

This is now extended (by abstract function *extend*) to the corresponding program point of the clause of qsort. First, the covering test is performed, and it succeeds, since $list(num, L), num(X)$ covers indeed the global answer constraint projected onto the input variables:

$$(L = [E|_]) \wedge (E < X \vee E >= X)) \vee L = [].$$

Therefore, computation is still covered and non-failing. This, together with the projection of the c component onto the variables of the first clause of qsort, yields success abstraction for partition:

$$(\{num(X), list(num, L), var(R), [](R2), var(R1), list(num, L1), list(num, L2)\}, true, \overline{1}, \overline{0}).$$

where the c component is still *true* since the projection onto the clause variables factors out the previously computed global answer constraint. Now, analysis will proceed into call `qsort(L2,R1,R2)` with

$$(\{[](R2), var(R1), list(num, L2)\}, true, \overline{1}, \overline{0}).$$

Since this is basically the same call pattern that we started with, no new fixpoint computation is started in this case.[7] On the other hand, a new fixpoint computation is started for the second recursive call `qsort(L1,R,[X|R1])` with

$$(\{list(num, L1), var(R), num(X), list(num, R1)\}, true, \overline{1}, \overline{0}).$$

This is a new call pattern for the qsort predicate, which initiates a new fixpoint computation. The fixpoint value obtained in this computation is the same abstraction, except for the type of R which on output is a list.
Finally, *exit_to_success* now lifts this result to the original goal `qsort(As,Bs,[])` giving:

$$(\{list(num, As), list(num, Bs)\}, As = [\_|\_], \overline{1}, \overline{0}).$$

The analysis of the non-recursive clause immediately gives:

$$(\{[](As), [](Bs)\}, As = [] \wedge Bs = [], \overline{1}, \overline{0}),$$

and *extend* computes the covering test for the set of the above two abstractions with the initial input abstraction, in which the input types are $list(num, As)$. Certainly, this type covers the (projected) global answer constraint $As = [\_|\_] \vee As = []$. Thus, the goal is still covered and non-failing.
Finally, since the abstraction now computed is only the result of a first iteration of the fixpoint computation, a new iteration is started. The result in this case is the same, and fixpoint computation finishes with that very same result.

[6] This very same "trick" is used in the implementation.
[7] Here, we save the reader from some more fixpoint iterations that will be taking place. However, the results are as indicated.

5 Implementation Results

We have constructed a prototype implementation in (Ciao) Prolog which performs the analysis by adapting the framework of the PLAI implementation and defining the abstract operations for non-failure analysis that we have described in this paper. Most of these abstract operations have been implemented by reusing code of the implementation in [8], such as for example, the covering algorithm. We have incorporated the prototype in the Ciao/CiaoPP multiparadigm programming system [12, 11, 13] and tested it, first on a number of simple standard benchmarks, and then on more complex ones. All of them are taken from those used in the non-failure analysis of Debray *et al.* [8] and the latter are those used in the cardinality analysis of Braem *et al.* [1]. These two analyses are the closest related previous work that we are aware of. Some relevant results of these tests for non-failure analysis are presented in Table 1. **Program** lists the program names, **N** the number of predicates in the program, **F**, and **C** are the number of non-failing predicates detected by the cardinality analysis in [1], and the non-failure analysis in [8] respectively.

Note that the multi–variant non-failure analysis presented in this paper can infer several variants (call patterns) for the same predicate, where some of them may be non-failing (resp. covered) and the other ones can be failing (resp. not covered). For instance, in the case of the program `mv` in Table 1 (also described in Example 1), which has 4 predicates (`mv`/3, `qsort`/2, `partition`/4 and `append`/3), the analysis infers one variant for `mv`/3, which is non-failing and covered, 2 variants for `qsort`/2 (one of them which is non-failing and covered, and the other one which is failing and not covered), one variant for `partition`/4, which is non-failing and covered, and 3 variants for `append`/3 (2 of them which are non-failing and covered, and the other one which is failing and not covered). For this reason, and in order to make the results comparable, column **AF** shows two figures (both corresponding to the analysis presented in this paper): the number of predicates such that **all of their variants** (call patterns) are detected as non-failing, and (between parenthesis) the number of predicates such that **some of their variants** are detected as non-failing (this second figure is omitted if it is equal to the first one).

Similarly, **ACov** shows two figures (both corresponding to the analysis presented in this paper): the number of predicates detected to cover **all** of their (calling) types (variants), and (between parenthesis), the number of predicates detected to cover **some** of their (calling) types. **Cov** is the number of predicates detected to cover their (calling) types by the analysis in [8].

$\mathbf{T}_{AF}$ and $\mathbf{T}_{F}$ are the total time (in milliseconds) required by the analysis presented in this paper and the analysis in [8] respectively (both of which include the time required to derive the modes and types). The timings were taken on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 1Gb of RAM memory, running Red Hat Linux 8.0, and averaging several runs and eliminating the best and worst values. Ciao version 1.9.111 and CiaoPP-1.0 were used.

Analysis time averages (per predicate) are also provided in the last row of the table. From these numbers, it is clear that the new implementation based on the abstract interpretation engine is more efficient than the previous one. It is also more precise, although this does not show in the benchmarks used in this preliminary evaluation (except for `mv`).

6 Conclusions

We have described a non-failure analysis based on abstract interpretation, which extends the previous proposal of Debray et al. Our analysis improves in precision, and enjoys a clear theoretical setting, and a simpler implementation. Also, the implementation is more efficient.

The abstract domain underlying the analysis can be easily modified to cater for a determinacy analysis. Such an analysis, provided with a depth-k abstraction, would be the abstract interpretation counterpart of determinacy analyses such as that of [5]. We are currently working on the verification of this proposition.

The implemented analysis we have described in this paper is currently integrated in CiaoPP, and is being used for lower-bounds cost analysis, granularity control, and program debugging. Arguably, although our presentation covers strictly constraint logic programming, the

Program	N	AF	F	C	ACov	Cov	T_{AF}	T_F	$\frac{T_{AF}}{T_F}$
Hanoi	2	2	2	N/A	2	2	33	242	0.14
Fib	1	1	1	N/A	1	1	17	22	0.77
Tak	1	1	1	N/A	1	1	9	11	0.82
Subs	1	1	1	N/A	1	1	5	33	0.15
Reverse	2	2	2	N/A	2	2	17	29	0.59
mv	4	2 (4)	1	N/A	2 (4)	2	54	102	0.53
zebra	6	2	1	N/A	5 (6)	4	1008	1100	0.92
family	3	3	1	N/A	3	2	10	18	0.56
blocks	7	1 (2)	0	N/A	4 (5)	4	30	59	0.51
reach	2	2	0	N/A	2	1	19	30	0.63
bid	20	5 (8)	5	N/A	14 (17)	14	3089	3369	0.92
occur	4	1 (3)	1	N/A	1 (3)	1	69	78	0.88
Plan	16	5 (8)	3	0	11 (13)	10	2626	4128	0.64
Qsort	3	3	3	0	3	3	29	65	0.45
Qsort2	5	3	3	0	3	3	33	76	0.43
Queens	5	2 (3)	2	0	3 (4)	3	60	74	0.81
Pg	10	2 (3)	2	0	6 (9)	6	412	477	0.86
Mean							38 (/p)	58 (/p)	0.67 (/p)

Table 1. Accuracy and efficiency of the non-failure analysis (times in mS).

technique could be easily extended to be applied to functional logic languages with similar results, as is indeed the case in the Ciao system, where the analysis presented works without modification for Ciao's functional subset or for combinations of functions and predicates.

7 Acknowledgments

This work has been supported in part by the European Union IST program under contract IST-2001-34717 "Amos" and IST-2001-38059 "ASAP," by MCYT projects TIC 2002-0055 "CUBICO" and HI2000-0043 "ADELA," and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. The Ciao system is a collaborative effort of members of several institutions, including UPM, UNM, U.Melbourne, Monash U., U.Arizona, Linköping U., NMSU, K.U.Leuven, Bristol U., Ben-Gurion U, and INRIA. The system documentation and related publications contain more specific credits.

References

1. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on Logic Programming*, pages 457–471, Ithaca, NY, November 1994. MIT Press.
2. M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.
4. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
5. Steven Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting determinacy in logic programs. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 424–438, Budapest, Hungary, 1993. The MIT Press.

6. S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.
7. S.K. Debray and N.-W. Lin. Static Estimation of Query Sizes in Horn Programs. In *Third International Conference on Database Theory*, Lecture Notes in Computer Science 470, pages 515–528, Paris, France, December 1990. Springer-Verlag.
8. S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.
9. S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.
10. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.
11. M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The Ciao Logic Programming Environment. In *International Conference on Computational Logic, CL2000*, July 2000.
12. M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
13. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.
14. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.
15. N. Jones and H. Sondergaard. A semantics-based framework for the abstract interpretation of prolog. In *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis-Horwood, 1987.
16. H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *Fourth IEEE Symposium on Logic Programming*, pages 205–214, San Francisco, California, September 1987. IEEE Computer Society.
17. C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in LNCS, pages 463–475. Springer-Verlag, July 1986.
18. K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.
19. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
20. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.
21. P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michael. The Impact of Granularity in Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, number 724 in LNCS, pages 1–14. Springer-Verlag, September 1993.
22. A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 700–710, Seattle,Washington, August 1988.
23. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.