Cardinality Analysis of Prolog

C. Braem[1]
B. Le Charlier[2]
S. Modart[3]
P. Van Hentenryck[4]

Technical Report No. CS-94-17

April, 1994

[1]University of Namur 21 rue Grandgagnage, B-5000 Namur(Belgium)
[2]University of Namur 21 rue Grandgagnage,B-5000 Namur(Belgium)
[3]University of Namur 21 rue Grandgagnage, B-5000 Namur (Belgium)
[4]Brown University, Computer Science Department Box 1910, Providence, RI 02912

Cardinality Analysis of Prolog[1]

C. Braem, B. Le Charlier, and S. Modart
University of Namur
21 rue Grandgagnage, B-5000 Namur (Belgium)

P. Van Hentenryck
Brown University
Box 1910, Providence RI 02912 (USA)

Abstract

This paper proposes a novel analysis, called cardinality analysis, to approximate the number of solutions to a goal. The analysis is an instantiation of a new abstract interpretation framework for full Prolog (without dynamic predicates) based on sequences of substitutions. The abstract domain captures not only modes and types but also lower and upper bounds on the number of solutions and information about arithmetic and meta predicates, the cut, and termination. Experimental results show that the approach has a number of desirable properties wrt efficiency, accuracy, and simplicity not available in previous work. In particular, all aspects of the analysis are captured in a single framework, the mode/type analysis and the cardinality analysis are performed simultaneously and improve each other's accuracy, input/output patterns, cut, termination, and abstractions of arithmetic and meta-predicates are used to achieve great precision, and the overhead of the approach is small compared to traditional mode/type analyses.

1 Introduction

Don't know nondeterminism is an appealing feature of logic programming allowing concise statements of many problems. It complicates however sequential and parallel implementation [6, 8], partial evaluation [15], program transformation, and debugging. The number of solutions to a goal is thus useful for many purposes including indexing, cut insertion and elimination [6, 15], dead code elimination, and memory management and scheduling in parallel systems [8, 17].

The purpose of this paper is to propose a novel analysis, called cardinality analysis, to approximate the number of solutions to a goal. The analysis subsumes traditional determinacy analysis [6, 7, 15, 5] and applies to full Prolog (without dynamic predicates such as `assert/retract`). The theoretical foundation of the analysis is a novel generic abstract interpretation framework designed especially to deal with issues such as determinacy, termination, and the cut. The framework is presented is detail in a companion paper [12]: its key idea is to describe the result of a procedure as an abstract sequence of substitutions, representing a set of sequences of substitutions.

The technical contribution of this paper is to show how this framework can be instantiated with a novel abstract domain to produce an accurate and efficient cardinality analysis. The abstract domain manipulates abstract sequences of the form $\langle \beta, m, M, t\rangle$, where β is an abstract substitution characterizing all substitutions in the sequences, m and M are respectively a lower and an upper bound on the lengths of the sequences and t is a termination information. Traditional operations (e.g. projection and unification) and new operations (e.g. the concatenation of abstract sequences) are defined on this domain. The subdomain of abstract substitutions is designed to improve the accuracy of the cardinality analysis. Besides traditional abstractions for modes, types, and sharing,

[1]Partly supported by the Office of Naval Research under grant N00014-91-J-4052 ARPA order 8225 and the National Science Foundation under grant numbers CCR-9357704 and a NSF National Young Investigator Award.

it collects information on arithmetic predicates and subterm relationships. The resulting cardinality analysis has a number of desirable features.

Accuracy: The cardinality and the substitution domain complement each other to achieve high precision. On the one hand, the cut and the approximated number of solutions enable us to improve the mode and type analysis. To our knowledge, no existing analysis enjoys this property, which is desirable when libraries are used as shown later on. On the other hand, input and output abstract substitutions are used in the concatenation of abstract sequences to improve the cardinality analysis. The use of output patterns was proposed before for determinacy analysis (e.g. [7, 5]) but, to our knowledge, was never integrated or implemented in a complete framework.

Efficiency: The cardinality analysis introduces a small overhead compared to a traditional mode analysis. This is mainly due to the nature of abstract sequences and to an effective widening operation which ensures that the cardinality analysis essentially follows the mode analysis.

Uniformity: All aspects of the analysis are captured in a unique abstract interpretation framework, simplifying the proof of correctness and the design of the analysis. This is in contrast to existing determinacy analysis which either resort to special-purpose correctness proofs (e.g. [6, 15]) or exclude certain aspects of the analysis (e.g. how input and output patterns are used to detect determinacy) from their framework (e.g. [5]).

Wide Applicability: The analysis applies to full Prolog, excluding dynamic predicates such as `assert` and `retract`. In particular, it handles the cut and arithmetic and meta-level predicates.

The rest of this paper is organized as follows. Section 2 illustrates the functionality of the system on a set of examples. Section 3 gives an informal overview of the abstract interpretation framework. Sections 4, 5, and 6 describe respectively the abstract domain, the abstract interpretation algorithm, and the implementation of the abstract operations. Section 7 reports the experimental results on a number of benchmarks while Section 8 concludes the paper.

2 The Functionality of The System

This section illustrates the functionality of the system on a number of examples. The examples are small in general but they illustrate many aspects of our system. Results on medium-size programs are given in Section 7. Consider first the `delete_last` example:

```
delete_last(X,[X]).
delete_last(X,[_|T]) :- delete_last(X,T).
```

When given the input pattern `delete_last(v,g)`, where `v`, `g` denote respectively a variable and a ground term, our analysis returns, in 0.01 seconds, the output sequence $\langle \mathtt{delete\_last}(g,[g|g]),0,1,\mathtt{pt}\rangle$, where `pt` stands for "possible termination". Note that the analysis in [6] cannot infer determinacy in this case, since it does not use output patterns. Consider now the `partition` programs.

```
partition([],P,[],[]).
partition([S|T],P,[S|Ss],Bs) :-
    S ≤ P, !,
    partition(T,P,Ss,Bs).
partition([B|T],P,Ss,[B|Bs]) :-
    partition(T,P,Ss,Bs).
```

```
partition([],P,[],[]).
partition([S|T],P,[S|Ss],Bs) :-
    leq(S,P), partition(T,P,Ss,Bs).
partition([B|T],P,Ss,[B|Bs]) :-
    gt(B,P), partition(T,P,Ss,Bs).
leq(K1-V1,K2-V2) :- K1 ≤ K2.
gt(K1-V1,K2-V2) :- K1 > K2.
```

Note that the second program calls arithmetic predicates through an auxiliary predicate and is appropriate for a key sort. Given an input pattern `partition(g,g,v,v)`, our analysis returns in both examples the abstract sequence $\langle \mathtt{partition(g,g,g,g)}, 0, 1, \mathtt{pt}\rangle$ in 0.01 seconds. Input/output patterns are used to determine that the first clause and the two others are mutually exclusive in both programs, while the cut (in the first program) and the abstraction of arithmetic predicates (in the second program) determine the mutual exclusion of the second and the third clause. Note that [7, 5] do not deal with the cut and cannot handle the first program. In addition, we know of no implemented system which can handle the second program. The analysis in [6] cannot detect determinacy in this case due to the absence of abstraction for arithmetic predicates.

Consider now the program `compress(L,Lc)` which holds if `Lc` is a compressed version of the list `L`. For instance, the compressed version of `[a,b,b,c,c,c]` is `[a,1,b,2,c,3]`. A library can contain the definition of a single procedure to handle both compression and decompression as follows.

```
comp([],[]).
comp([C],[C,1]).
comp([C1,C2|T],[C1,1,C2,N|Rest]) :-
    C1 <> C2,
    comp([C2|T],[C2,N|Rest]).
comp([C1,C1|T],[C1,N1|Rest]) :-
    comp([C1|T],[N|Rest]),
    N1 := N + 1.

compress(A,B) :-
    var(A), !, decomp(A,B).
compress(A,B) :-
    comp(A,B).
```

```
decomp([],[]).
decomp([C],[C,1]).
decomp([C1,C2|T],[C1,1,C2,N|Rest]) :-
    decomp([C2|T],[C2,N|Rest]),
    C1 <> C2.
decomp([C1,C1|T],[C1,N1|Rest]) :-
    N1 > 1,
    N := N1 - 1,
    decomp([C1|T],[N|Rest]).
```

Given input patterns `compress(g,v)` and `compress(v,g)`, our analysis returns in both cases the abstract sequence $\langle \mathtt{compress(g,g)}, 0, 1, \mathtt{pt}\rangle$ in about 0.2 seconds. The example illustrates many of the functionalities of our system, including input/output patterns, abstraction of arithmetic and meta predicates, and the cut, all of which are necessary to obtain the optimal precision. In addition, it shows that taking the cut into account during the mode analysis improves the output modes. A mode analysis ignoring the cut would return the output pattern $\mathtt{compress(novar,g)}$ for the input pattern `compress(v,g)`, losing the groundness information. None of the systems or proposals we know of can handle this example with an optimal result. Note also that if a program only uses the input pattern `compress(v,g)`, our analysis detects that the second clause of `compress` is dead code without any extra processing since no input/output pattern exists for `compress`. The second clause and the test `var` and the cut of the first clause can then be removed by an optimizer.

Consider now the `append` program. For the input patterns `append(g,g,a)`, `append(g,a,g)` and `append(a,g,g)`, the analysis returns the abstract sequence $\langle \mathtt{append(g,g,g)}, 0, 1, \mathtt{pt}\rangle$. The difficulty is in the third case which uses the subterm abstraction. For the input pattern `append(v,v,v)`, the system returns the abstract sequence $\langle \mathtt{append(novar,any,noground)}, 3, \infty, \mathtt{snt}\rangle$. Note that this result indicates that the goal always succeeds (it has at least 3 solutions) and does not terminate (`snt` stands for "sure non termination"). The number 3 comes from the widening on sequences which is applied when the abstract substitution part stabilizes.

Finally, consider the meta-program for unification from [16]. Given the input pattern `unify(a,a)`, our analysis returns the abstract sequence $\langle \mathtt{unify(a,a)}, 0, 1, \mathtt{pt}\rangle$ in 0.02 seconds. The key issue is to collect information on the meta-predicates to detect mutual exclusion.

```
unify(X,Y) :- var(X), var(Y), X = Y.
unify(X,Y) :- var(X), nonvar(Y), X = Y.
unify(X,Y) :- var(Y), nonvar(X), Y = X.
unify(X,Y) :- nonvar(X), nonvar(Y), constant(X), constant(Y), X = Y.
unify(X,Y) :- nonvar(X), nonvar(Y), compound(X), compound(Y), term_unify(X,Y).

term_unify(X,Y) :- functor(X,F,N), functor(Y,F,N), unify_args(N,X,Y).

unify_args(N,X,Y) :- N > 0, unify_arg(N,X,Y), N1 := N - 1, unify_args(N1,X,Y).
unify_args(0,X,Y).
unify_arg(N,X,Y) :- arg(N,X,ArgX), arg(N,Y,ArgY), unify(ArgX,ArgY).
```

Note also that Sahlin [15] defines a simpler cardinality analysis over a finite domain. His analysis ignores predicate arguments, since this appears to be satisfactory for partial evaluation. However, it cannot handle the examples discussed here and loses too much information for many applications.

3 Overview of the Abstract Interpretation Framework

In this section, we briefly review our abstract interpretation framework for the cardinality analysis. The framework is presented in detail in a companion paper [12].

Concrete Semantics As is traditional in abstract interpretation [4], the starting point of the analysis is a collecting semantics for the programming language. Our concrete semantics captures the top-down execution of Prolog using a left-to-right computation rule, a depth-first search, and the standard semantics for the cut. It manipulates three kinds of semantic objects: substitutions, substitution sequences, and substitution sequences with cut information. Substitution sequences are used to describe the result of procedures while substitution sequences with cut information describe the result of clauses; $cf = cut$ iff the clause has executed a cut. Substitutions are of the form $\{x_1 \leftarrow t_1, \ldots, x_m \leftarrow t_m\}$ for some $m \geq 0$. Substitution sequences are either *finite*, i.e. of the form $< \theta_1, \ldots \theta_n >$ $(n \geq 0)$, either *incomplete*, i.e. of the form $< \theta_1, \ldots \theta_n, \perp >$ $(n \geq 0)$, or *infinite*, i.e. of the form $< \theta_1, \ldots \theta_i, \ldots >$ $(i \in \mathbf{N})$. Incomplete sequences model non-terminating executions which provide a finite set of answers. Substitution sequences with cut information are of the form (S, cf), where S is a substitution sequence and $cf \in \{cut, nocut\}$. Three main operations are performed on sequences: unification, projection, and concatenation.

The semantics associates with each of the predicate symbol p in the program a set of tuples of the form $(\Theta_{in}, p, \Sigma_{out})$ which can be interpreted as follows:

"Execution of $p(x_1, \ldots, x_n)\theta$, $\theta \in \Theta_{in}$, produces a sequence $\langle \theta_1, \ldots, \theta_n, \ldots \rangle \in \Sigma_{out}$."

Substitution sequences are denoted by the letter S and the set of all sequences by PSS. SUBST denotes the set of substitutions of a sequence S and NSUBST the number of substitutions in S.

Abstract Semantics The second step of the methodology is the abstraction of the concrete semantics. Our abstract semantics consists in abstracting a set of substitutions by a single abstract substitutions and a set of sequences by a single abstract sequence. As a consequence, the abstract semantics associates with each predicate symbol p a set of tuples of the form (β_{in}, p, B_{out}) which can be read informally as follows:

"Execution of $p(x_1, \ldots, x_n)\theta$ with θ satisfying the property described by β_{in} produces a sequence $\langle \theta_1, \ldots, \theta_n, \ldots \rangle$ satisfying the property described by B_{out}."

The abstract semantics assumes a number of operations on abstract sequences, in particular unification, projection, concatenation, and upper bound. The first three operations are simply consistent approximations of the corresponding concrete operations. The upper bound operation is a consistent abstraction of the union of sets of sequences.

The Abstract Interpretation Algorithm The last step of the methodology consists in computing a postfixpoint of the abstract semantics. The algorithm is described in Section 5.

4 Overview of the Abstract Domain

The main task to develop the cardinality analysis is the design of an abstract domain for sequences and the implementation of the abstract operations. In this section, we give an overview of the abstract domain in a bottom-up fashion, starting with abstract substitutions and ending with abstract sequences with cut. Section 6 describes the implementation of the abstract operations.

4.1 Abstract Substitutions

The precision of the cardinality analysis strongly depends on the information maintained on substitutions. Our abstraction of substitutions is best viewed as an instantiation of the generic pattern domain [3] with a mode, sharing, arithmetic, and subterm component. We describe each component successively. In the following, we denote abstract substitutions by β possibly subscripted and the set of abstract substitutions by AS.

The Generic Pattern Domain The key intuition behind `Pat(`$\mathcal{R}$`)` is to represent information on some subterms occurring in a substitution instead of information on terms bound to variables only. More precisely, `Pat(`$\mathcal{R}$`)` may associate the following information with each considered subterm: (1) its *pattern* which specifies the main functor of the subterm (if any) and the subterms which are its arguments; (2) its *properties* which are left unspecified and are given in the domain $\mathcal{R}$. In addition to the above information, each variable in the domain of the substitutions is associated with one of the subterms. Note that the domain can express that two arguments have the same value (and hence that two variables are bound together) by associating both arguments with the same subterm. It should be emphasized that the pattern information is optional. In theory, information on all subterms could be kept but the requirement for a finite analysis makes this impossible for almost all applications. As a consequence, the domain shares some features with the depth-k abstraction [10], although `Pat(`$\mathcal{R}$`)` does not impose a fixed depth but adjusts it dynamically through upper bound and widening operations. Note that the identification of subterms (and hence the link between the structural components and the $\mathcal{R}$-domain) is a somewhat arbitrary choice. In `Pat(`$\mathcal{R}$`)`, subterms are identified by integer indices, say $1, \ldots, n$ if n subterms are considered and we denote by I_p the set of indices $\{1, \ldots, p\}$.

More formally, the pattern and same-value component can be described as follows. The pattern component is a partial function $frm : I_p \to Pat_p$, from the set of indices I_p to the set of patterns over I_p, i.e. elements of the form $f(i_1, \ldots, i_n)$, where f is a functor symbol and $i_1, \ldots, i_n \in I_p$.

When the pattern is undefined for an index i, we write $frm(i) = undef$. The same-value component is a total function $sv : D \rightarrow I_p$, where $D = \{x_1, \ldots, x_n\}$ is the substitution domain.

The $\mathcal{R}$-domain is the generic part which specifies this information by describing properties of a set of tuples $< t_1, \ldots, t_p >$ where $t_1, \ldots, t_p$ are terms. As a consequence, defining the $\mathcal{R}$-domain amounts essentially to defining a traditional domain on substitutions and its operations. We now describe the various components of the $\mathcal{R}$-domain which can be built as an open product [3].

The Mode Component The mode component was described in [13] and associates a mode from the set $Modes$ = {`var`,`ground`,`novar`,`noground`,`ngv`,`gv`,`any`} with each subterm. Formally, it is a total function $mo : I_p \rightarrow Modes$ whose concretization function is given by

$$Cc(mo) = \{(t_1, \ldots, t_n) \ \mid \ t_i \in Cc(mo(i)) \ \ 1 \leq i \leq n\}.$$

The Sharing Component The sharing component maintains information about possible sharing between pairs of subterms and was also described in [13]. Formally, it is a symmetrical relation $ps : I_p \times I_p$ whose concretization function is given by

$$Cc(ps) = \{(t_1, \ldots, t_n) \ \mid \ var(t_i) \cap var(t_j) \Rightarrow ps(i,j) \ \ 1 \leq i, j \leq n\}.$$

The Arithmetic Component The arithmetic component is novel and aims at using arithmetic predicates to detect mutual exclusion between clauses. It approximates information about arithmetic relationships by rational order constraints, i.e. binary constraints of the form $i \ \delta \ j$ and unary constraints of the form $i \ \delta \ c$, where i, j are indices, $\delta \in \{>, \geq, =, \neq, \leq, <\}$ and c is an integer constant. For instance, a builtin $X \geq Y + 2$ is approximated by a constraint $X > Y$ in this domain. Formally, an element $arithm$ is a set of rational order constraints over indices whose concretization function is defined as follows (a constraint being satisfied only if the terms are numbers).

$$Cc(arithm) = \{(t_1, \ldots, t_n) \ \mid \ \forall \ i \ \delta \ j \ \in \ arithm: \ t_i \ \delta \ t_j \ \& \ \forall \ i \ \delta \ c \ \in \ arithm: \ t_i \ \delta \ c\}.$$

The Subterm Component The subterm component is also novel and maintains sure subterm relations between indices. Formally, it is a relation $subterm : I_p \times I_p$ whose concretization function is given by $Cc(subterm) = \{(t_1, \ldots, t_n) \ \mid \ subterm(i,j) \Rightarrow t_i \text{ is a subterm of } t_j \ \ 1 \leq i, j \leq n\}$.

Operations on Abstract Substitutions Traditional operations (e.g. unification, projection, and upper bound) are required on abstract substitutions. Most of these operations are presented elsewhere (e.g. [13, 3]) and their generalizations for the arithmetic and subterm components are omitted for space reasons. A novel operation, called `EXCLUSIVE`, is required on abstract substitutions to detect mutual exclusion and is discussed in Section 6.

4.2 Abstract Sequences

Abstract Substitution Sequences Our cardinality analysis uses abstract sequences of the form $\langle \beta, m, M, t \rangle$ to represent sets of sequences, where $\beta \in AS$, $m \in \mathbf{N}$, $M \in \mathbf{N} \cup \{\infty\}$, and $t \in \{snt, st, pt\}$ (st stands for sure termination). Informally, m and M are lower and upper bounds on the number of substitutions in the sequences, β describes all substitutions in the sequences, and t is an information on termination. The concretization function can be defined as follows:

$Cc(\langle \beta, m, M, t \rangle) = Sseq_1(\beta) \cap Sseq_2(m, M) \cap Sseq_3(t)$

where $Sseq_1(\beta) = \{S : SUBST(S) \subseteq Cc(\beta)\}$,

$Sseq_2(m, M) = \{S : m \leq NSUBST(S) \leq M\}$,

$Sseq_3(snt) = \{S : S \text{ is incomplete or infinite}\}$,

$Sseq_3(st) = \{S : S \text{ is finite}\}$,

$Sseq_3(pt) = PSS$

In the following, abstract sequences are denoted by the letter B possibly subscripted. The set of abstract sequences is denoted by *ASS*. The ordering on abstract sequences is defined as follows:

$$B_1 \leq B_2 \text{ iff } \beta_1 \leq \beta_2 \wedge m_1 \geq m_2 \wedge M_1 \leq M_2 \wedge t_1 \leq t_2$$

Note that $t_1 \leq t_2$ iff $t_1 = t_2$ or $t_2 = pt$.

It is important to point out that our termination information is not intended to compete with existing termination analysis. Its goal here is to improve the precision of the analysis. In this context, it is more important to detect non-termination than termination and our domain gives reasonable results for this purpose. It gives rather poor results to detect termination. See [12] for a more general discussion of this issue.

Abstract Sequences with Cut Information To achieve reasonable accuracy, the cardinality analysis needs to preserve information on the cut which motivated the concept of abstract sequences with cut information which are of the form $\langle B, cf \rangle$ where $cf \in \{cut, nocut, weakcut\}$. Informally, *cut* indicates that a cut has been executed in all sequences, *nocut* that no cut has been executed in any sequence, and *weakcut* that a cut has been executed for all sequences producing at least one solution. More formally, the concretization function is defined as follows:

$Cc(\langle B, cut \rangle) = \{\langle S, cut \rangle : S \in Cc(B)\}$,
$Cc(\langle B, nocut \rangle) = \{\langle S, nocut \rangle : S \in Cc(B)\}$,
$Cc(\langle B, weakcut \rangle) = \{\langle S, cut \rangle : S \in Cc(B)\} \cup \{\langle S, nocut \rangle : S \in Cc(B) \cap \{<>, <\perp>\}\}$.

Abstract sequences with cut are denoted by the letter C possibly subscripted. The set of abstract sequences with cut information is denoted by *ASSC*. Note also that our abstract domain is in fact more complex, since it also contains information about meta-predicates. We omit the description of this last part for space reasons as it follows the same spirit.

Abstract Operations Traditional operations (i.e. unification, projection, and upper bound) as well as two novel operations `AI_CUT` and `CONC`, to handle the cut and the concatenation of sequences needs to be defined on sequences. Section 6 discusses their new implementations.

5 The Abstract Interpretation Algorithm

The algorithm of the cardinality analysis is a top-down chaotic iteration algorithm using dynamic dependencies to speed up convergence. The overall strategy as well as the use of dependencies is now well-understood and is used in Bruynooghe's framework [2], in Hermenegildo's `PLAI` [14], and in `GAIA` [11, 13]. For space reasons, we will not repeat these results and the reader is referred to these papers for a comprehensive presentation. Our algorithm is derived from `GAIA`, uses normalized program, and is depicted in Figure 1. Procedure `solve_goal` is standard. The only important

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

procedure solve_call(in β,p,suspended; inout sat,dp)
   if (β,p) ∉ (dom(dp) ∪ suspended) then begin
      if (β,p) ∉ dom(sat) then sat := EXTEND(β,p,sat);
      repeat
         EXT_DP(β,p,dp);
         solve_procedure(β,p,proc(p),suspended ∪ {(β,p)},B,sat,dp);
         (sat,modified) := ADJUST(β,p,B,sat);
         REMOVE_DP(modified,dp)
      until (β,p) ∈ dom(dp)
   end

procedure solve_procedure(in β,p,pr,suspended; out B; inout sat,dp)
   begin
      SetofC_nocut := { ⟨⊥,0,0,st⟩ };
      SetofC_cut := ∅;
      let pr be c_1,...,c_n in begin
         i := 0;
         while i ≠ n & SetofC_nocut ≠ ∅ do begin
            i := i + 1;
            solve_clause(β,p,c_i,suspended,C,sat,dp);
            ⟨SetofC,SetofC_nocut⟩ := CONC(β,SetofC_nocut,C);
            SetofC_cut := SetofC_cut ∪ SetofC
         end;
         B := UNIONS(SetofC_nocut,SetofC_cut)
      end
   end

procedure solve_clause(in β,p,c,suspended;  out C; inout sat,dp)
   begin
      C := EXTCS(c,β);
      for i := 1 to m with l_1,...,l_m body-of c do
         if l_i is !  then C := AI-CUT(C)
         else begin
            β_aux := RESTRG(l_i,SUBST(C));
            switch (l_i) of
               case x_j = x_k:  B := AI_VARS(β_aux)
               case x_j = f(...):  B := AI_FUNCS(β_aux,f)
               case q(...):
                  solve_call(β_aux,q,suspended,sat,dp);
                  B := sat(β_aux,q);
                  if (β,p) ∈ dom(dp) then ADD_DP(β,p,β_aux,q,dp)
               end;
            C := EXTGS(l_i,C,B)
         end;
      C := RESTRCS(c,C)
   end
```

Figure 1: The Generic Abstract Interpretation Algorithm

points to mention is that EXTEND adds $(\beta, p, \langle \perp, 0, 0, snt\rangle)$ in *sat* (i.e. it assumes that the procedure loops) while ADJUST update *sat* by performing the widening of the old and the new result. The main originality appears in procedure `solve_procedure` which computes an approximation of procedure p given an input substitution β. The key idea is to separate the set of abstract sequences resulting from the execution of previous clauses in two sets: executions in which a cut has been executed ($SetofC_{\texttt{cut}}$) and those in which no cut has been executed ($SetofC_{\texttt{nocut}}$). The result produced by a clause execution needs to be combined with $SetofC_{\texttt{nocut}}$ only, since other executions have been cut. Operation CONC is used for this concatenation. Procedure `solve_procedure` starts by initializing $SetofC_{\texttt{nocut}}$ to $\{\langle \perp, 0, 0, st\rangle\}$ (i.e. an empty sequence) and $SetofC_{\texttt{cut}}$ to $\emptyset$. It then considers each clause of the procedure and concludes by taking an upper bound of the elements of $SetofC_{\texttt{nocut}}$ and $SetofC_{\texttt{cut}}$. Note also that the loop may terminate as soon as $SetofC_{\texttt{nocut}}$ is empty, since this means that a cut is surely executed in all possible executions. Procedure `solve_clause` is rather standard. The only differences with previous algorithms comes from the operation to handle the cut and the fact that many operations are generalized to work and return abstract sequences. Note however that operation EXTCS initializes the cut information to *nocut* and that SUBST(C) returns the abstract substitution component of sequence C. Procedure `solve_procedure` does not follow exactly the abstract semantics presented in [12] because the domain Pattern is not able to keep track of a finite number of possible functors for a given index. Hence, we have to keep all incompatible abstract sequences separately until the procedure has been fully executed. No such treatment would be necessary if a more expressive type domain (e.g. type graphs [9]) would be used.

6 Implementation of the Abstract Operations

We now turn to the definition of the abstract operations on abstract sequences. For space reasons, we only consider a subset of the operations, in particular, unification, upper bound, widening, cut, and the concatenation operation. Operations on abstract sequences use operations on abstract substitutions and we consider those as black-boxes in the following. Consistency of the abstract operations is specified in [12].

Unification We first consider operation AI_VARS: AS $\rightarrow$ ASS which, given an abstract substitution β with domain $\{x_1, x_2\}$, returns an abstract substitution sequence which represents a set of sequences of length 0 or 1 (depending upon the success or failure of the unification). The terms bound to x_1 and x_2 are unified in all these sequences.

AI_VARS(β) = $\langle \beta_{out}, m, M, t\rangle$ where
    $\langle \beta_{out}, \mathit{flag}\rangle$ = AI_VAR(β)
    m = if flag = $\mathit{success}$ then 1 else 0
    M = if flag = $\mathit{failure}$ then 0 else 1
    t = st.

Operation AI_VAR on abstract substitutions (see [11, 13]) is slightly modified and returns, in addition to the substitution, a flag indicating if the unification always succeeds, always fails, or can both succeed and fail. For instance, unification of two variables will always succeed.

Upper Bound The upper bound operation UNIONS: ASS $\times$ ASS $\rightarrow$ ASS is straightforward:

UNIONS($\langle \beta_1, m_1, M_1, t_1\rangle$, $\langle \beta_2, m_2, M_2, t_2\rangle$) = $\langle$UNION$(\beta_1, \beta_2), min(m_1, m_2), max(M_1, M_2),$ LUB$(t_1, t_2)\rangle$.

Widening Since the domain of abstract sequences is infinite (even if the substitution domain was finite), widening is necessary to ensure the finiteness of the analysis. To achieve a good tradeoff between accuracy and efficiency, the widening uses the following strategy: it iterates like an analysis on substitutions (possibly applying widening on substitutions) until the substitution part stabilizes. The widening on sequences is then applied by taking the least upper bound of the termination components, the minimum of the lower bounds and setting the upper bound to infinity. More precisely, the operation $\bigtriangledown$: $\texttt{ASS} \times \texttt{ASS} \rightarrow \texttt{ASS}$ is defined as follows, assuming that $B_{old} = \langle \beta_{old}, m_{old}, M_{old}, t_{old} \rangle$ and $B_{new} = \langle \beta_{new}, m_{new}, M_{new}, t_{new} \rangle$:

$$B_{new} \bigtriangledown B_{old} = \begin{cases} \langle \texttt{WIDEN}(\beta_{new}, \beta_{old}), m_{new}, M_{new}, t_{new} \rangle & \text{if } \beta_{new} \not\leq \beta_{old} \\ \langle \beta_{old}, m_{new}, M_{new}, \texttt{LUB}(t_{new}, t_{old}) \rangle & \text{if } \beta_{new} \leq \beta_{old} \;\&\; t_{new} \not\leq t_{old} \\ \langle \beta_{old}, min(m_{new}, m_{old}), \infty, t_{old} \rangle & \text{if } \beta_{new} \leq \beta_{old} \;\&\; t_{new} \leq t_{old} \;\&\; B_{new} \not\leq B_{old} \\ B_{old} & \text{if } B_{new} \leq B_{old}. \end{cases}$$

The first case makes sure that the algorithm iterates until the abstract substitution stabilizes by applying a widening on this subcomponent. When it is stable, the widening is applied on sequences.

Cut Abstract Interpretation of the cut $\texttt{AI\_CUT}$: $\texttt{ASSC} \rightarrow \texttt{ASSC}$ is defined as follows:

$$\begin{array}{l} \texttt{AI\_CUT}(\langle\langle \beta, m, M, t\rangle, cf\rangle) \texttt{ = } \langle\langle \beta, min(1,m), min(1,M), t'\rangle, cf'\rangle \texttt{ where} \\ \begin{array}{rcll} t' & = & snt & \text{if } M = 0 \text{ and } t = snt, \\ & = & st & \text{if } t = st \text{ or } m \geq 1, \\ & = & pt & \text{otherwise} \\ cf' & = & cut & \text{if } cf = cut \text{ or } m \geq 1, \\ & = & nocut & \text{if } cf = nocut \text{ and } M = 0, \\ & = & weakcut & \text{otherwise} \end{array} \end{array}$$

Concatenation Operation $\texttt{CONC} : AS \times 2^{ASSC} \times ASSC \rightarrow 2^{ASSC} \times 2^{ASSC}$ is the most complex operation on sequences; it receives three inputs: the input abstract substitution to the procedure, the set of abstract sequences produced by clauses $1, \ldots, i-1$ which have not been cut, and the abstract sequence resulting from the execution of clause i. It produces two results: a set $\texttt{S}_{\texttt{nocut}}$ of abstract sequences which were not cut and a set $\texttt{S}_{\texttt{cut}}$ of abstract sequences which were cut.

To simplify the case analysis, the core of operation CONC manipulates only abstract sequences with sure success ($m \geq 1$) or abstract sequences with no solution ($M = 0$). Moreover, CONC only manipulates abstract sequences with sure cut information (i.e. *cut* ot *nocut*). To ensure this property, operation CONC first splits the result sequence to obtain one or several sequences satisfying the above properties. Operation SPLIT is defined as follows:

$$\texttt{SPLIT}(\langle\langle \beta, m, M, t\rangle, cf\rangle) \texttt{ = } Set_1 \cup Set_2 \cup Set_3 \cup Set_4 \texttt{ where}$$

$$Set_1 = \begin{cases} \{\langle\langle \bot, 0, 0, t\rangle, cut\rangle\} & \texttt{if } m = 0 \;\&\; cf \neq nocut \\ \{\} & \texttt{otherwise} \end{cases}$$

$$Set_2 = \begin{cases} \{\langle\langle \bot, 0, 0, t\rangle, nocut\rangle\} & \texttt{if } m = 0 \;\&\; cf \neq cut \\ \{\} & \texttt{otherwise} \end{cases}$$

$$Set_3 = \begin{cases} \{\langle\langle \beta, max(1,m), M, t\rangle, cut\rangle\} & \texttt{if } M \geq 1 \;\&\; cf \neq nocut \\ \{\} & \texttt{otherwise} \end{cases}$$

$$Set_4 = \begin{cases} \{\langle\langle \beta, max(1,m), M, t\rangle, nocut\rangle\} & \texttt{if } M \geq 1 \;\&\; cf \neq cut \\ \{\} & \texttt{otherwise} \end{cases}$$

Note also that an invariant of our algorithm is that any element of $SetofC_{\texttt{cut}}$ and $SetofC_{\texttt{nocut}}$ is in split form (i.e. `SPLIT(`C`) = `$\{C\}$). Operation `CONC` can now be defined as follows:

```
CONC(in β, SetofC_nocut, C; out Set_nocut, Set_cut)
begin
   Set_nocut := ∅; Set_cut := ∅;
   Set := SPLIT(C);
   for each C1 in SetofC_nocut do
      for each C2 in Set do
         SCONC(β,C1,C2,Set_nocut, Set_cut);
end
```

`CONC` splits the new result and concatenates each resulting sequence with each sequence in $SetofC_{\texttt{nocut}}$. Operation `SCONC` is the core of the operation and performs a sophisticated case analysis.

```
SCONC(in β, C1, C2; inout Set_nocut, Set_cut)
begin
   if C1 = ⟨⟨⊥,0,0,st⟩,nocut⟩ then
      if cf2 = cut then Set_cut := Set_cut ∪ { C2 }
                   else  Set_nocut := Set_nocut ∪ { C2 }
   else if C1 = ⟨⟨⊥,0,0,snt⟩,nocut⟩ then
      Set_nocut := Set_nocut ∪ { C1 }
   else if C1 = ⟨⟨⊥,0,0,pt⟩,nocut⟩ then begin
      Set_nocut := Set_nocut ∪ { C1 };
      if cf2 = cut then Set_cut := Set_cut ∪ { C2 }
                   else  Set_nocut := Set_nocut ∪ { C2 }
      end
   else begin
      if t1 = snt ∨ t1 = pt then
         Set_nocut := Set_nocut ∪ { C1 };
      if t1 = st ∨ t1 = pt then
         if β2 = ⊥ ∨ ¬ EXCLUSIVE(β,β1,β2) then
            let C = ⟨⟨UNION(β1,β2),m1 + m2,M1 + M2,t2⟩,cf2⟩ in
               if cf2 = cut then Set_cut := Set_cut ∪ { C }
                            else  Set_nocut := Set_nocut ∪ { C }
   end
end
```

The definition assumes that that $C_i = \langle\langle\beta_i, m_i, M_i, t_i\rangle, cf_i\rangle$. The first three cases handle the case where C_1 is an empty sequence. In the first case, C_1 is a failure ($s_1 = st$), in which case the new result C_2 is added to the appropriate set. In the second case, C_1 is non-terminating ($s_1 = snt$), in which case C_1 must be added to Set_{nocut}. The third case can be both a failure and a loop, in which case the actions of the previous two cases must be executed. The fourth case corresponds to a non-empty sequence (i.e. it has a least one solution). If C_1 may loop (i.e. $t_1 = snt$ or $t_1 = pt$), then it must be added to Set_{nocut}. The most interesting case appears when C_1 may terminate. The algorithm first checks whether the output substitutions β_1 and β_2 are mutually exclusive wrt the input substitution β. If they are, no action is taken, since no actual sequence of concrete substitutions may result from the concatenation of a sequence from C_1 and a sequence from C_2 since they contain substitutions which are *not* instances of the same input substitution. Otherwise, a new sequence is computed with the `UNION` of the abstract substitutions, the sum of the bounds, and the termination and cut information of C_2. The test $\beta_2 = \perp$ is needed to insert C_1 when C_2 fails. Note that, when a clause result has its lower bound at 0 and its upper bound greater than 0, it is split into two parts. When another result with a mutually exclusive abstract substitution is considered, it will produce a new result when concatenated with the empty sequence but no new result when concatenated with the non-empty sequence.

It is important to point out that operation `CONC` is also the most complex computationally, since it may have to compare all clause results pairwise. This operation is mainly responsible for the difference in efficiency between `GAIA` over `Pattern` and the cardinality analysis presented here.

Mutual Exclusion We conclude this section by showing a partial implementation of operation `EXCLUSIVE`. The implementation only uses the pattern, same-value, and mode components but it gives the idea behind the complete implementation. The specification of `EXCLUSIVE` is as follows:

$$\texttt{EXCLUSIVE}(\beta, \beta_1, \beta_2) \Leftrightarrow \neg(\exists \theta_1 \in Cc(\beta_1) \& \theta_2 \in Cc(\beta_2) \& \theta \in Cc(\beta) : \theta_1, \theta_2 \leq \theta).$$

where $\theta \leq \theta'$ if θ' is more general than θ.

To define mutual exclusion, we need to find the indices which correspond to each other in the input and in the output substitutions. Assume that $\{x_1, \ldots, x_m\}$ is the domain of β, I_p, I_{p_1}, I_{p_2} be the indices in β, β_1, β_2, and sv, sv_1, sv_2 be their same-value components. Then, because β_1, β_2 are "instances" of β, there exist two functions $t_i : I_p \rightarrow I_{p_i}$ $(1 \leq i \leq 2)$ such that

- $sv_i(x_k) = t_i(sv(x_k))$ $(1 \leq k \leq m)$;
- $frm(j) = f(j_1, \ldots, j_n) \Rightarrow frm_i(t_i(j)) = f(t_i(j_1), \ldots, t_i(j_n))$.

Definition 1 (i, j) is *directly exclusive* wrt (frm_1, frm_2) iff $frm_1(i) = f(i_1, \ldots, i_p)$, $frm_2(j) = g(i_1, \ldots, i_q)$ and either $f \neq g$ or $p \neq q$. (i, j) is *exclusive* wrt (frm_1, frm_2) iff (i, j) is directly exclusive wrt (frm_1, frm_2) or $frm_1(i) = f(i_1, \ldots, i_p)$, $frm_2(j) = f(j_1, \ldots, j_q)$ and there exists $k : 1 \leq k \leq p$ such that (i_k, j_k) is exclusive wrt (frm_1, frm_2).

The implementation of `EXCLUSIVE` can now be defined as follows.

Definition 2 $\texttt{EXCLUSIVE}(\beta, \beta_1, \beta_2) \Leftrightarrow \exists k \in I_p$ such that

- $mode(i) \in \{ngv, novar\}$ and $(t_1(i), t_2(i))$ is directly exclusive wrt (frm_1, frm_2);
- $mode(i) = ground$ and $(t_1(i), t_2(i))$ is exclusive wrt (frm_1, frm_2).

7 Experimental Results

Benchmarks Our experiments use our traditional benchmarks (available by anonymous ftp from Brown university) except that cuts have been reinserted as in the original versions. In addition, some new program have been added. `Boyer` is a theorem-prover from the DEC-10 benchmarks, `Credit` is an expert system from `Sterling86`, while `Unify` and `Unify2` are meta programs for unification from the same book. There are two versions of `qsort` which differs in procedure `partition` which uses or does not use auxiliary predicates for the arithmetic builtins.

	OR		PC				PCA				PCAM			
Programs	I	T	I	T	IR	TR	I	T	IR	TR	I	T	IR	TR
Qsort	13	0.08	17	0.12	1.31	1.50	13	0.08	1.00	1.00	13	0.09	1.00	1.13
Qsort2	15	0.08	19	0.12	1.27	1.50	15	0.09	1.00	1.13	15	0.10	1.00	1.25
Queens	15	0.07	18	0.08	1.20	1.14	18	0.10	1.20	1.43	18	0.10	1.20	1.43
Press1	532	11.77	581	13.11	1.09	1.11	581	13.45	1.09	1.14	581	13.83	1.09	1.18
Press2	197	3.27	200	3.56	1.02	1.09	200	3.56	1.02	1.09	200	3.67	1.02	1.12
Gabriel	78	0.90	84	1.00	1.08	1.11	84	0.98	1.08	1.09	84	1.00	1.08	1.11
Peep	132	3.21	131	18.85	0.99	5.87	131	19.08	0.99	5.94	131	20.60	0.99	6.42
Read	432	23.91	458	25.32	1.06	1.06	458	25.37	1.06	1.06	458	26.40	1.06	1.10
Kalah	115	1.90	121	2.09	1.05	1.10	120	2.11	1.04	1.11	120	2.19	1.04	1.15
Cs	79	2.19	91	3.05	1.15	1.39	90	3.02	1.14	1.38	90	3.00	1.14	1.37
Plan	36	0.21	38	0.30	1.06	1.43	38	0.27	1.06	1.29	37	0.28	1.03	1.33
Disj	64	1.95	68	2.14	1.06	1.10	68	2.12	1.06	1.09	68	2.13	1.06	1.09
Pg	38	0.32	40	0.36	1.05	1.13	39	0.35	1.03	1.09	39	0.37	1.03	1.16
Boyer	56	0.76	56	1.15	1.00	1.51	56	1.17	1.00	1.54	56	1.18	1.00	1.55
Credit	63	0.57	64	0.81	1.02	1.42	64	0.80	1.02	1.40	64	0.82	1.02	1.44
Unify	9	0.03	14	0.05	1.56	1.67	14	0.06	1.56	2.00	9	0.02	1.00	0.67
Unify2	14	0.03	17	0.06	1.21	2.00	17	0.08	1.21	2.67	19	0.10	1.36	3.33
Mean					1.13	1.60			1.09	1.61			1.07	1.64

Table 1: Efficiency of the Cardinality Analysis

Efficiency The efficiency results are reported in Table 1. Several algorithms are compared: `OR` is the `GAIA` algorithm on `Pattern` [13], `PC` is the cardinality analysis with `Pattern`, `PCA` is `PC` with the abstraction for arithmetic predicates, and `PCAM` is `PCA` with the abstraction for meta-level predicates. The first interesting point to notice is the slight increase (about 7% on `PCAM`) in iterations when moving from abstract substitutions to abstract sequences, showing the effectiveness of our widening operator. Even more important perhaps is the fact that the time overhead of the cardinality analysis is small wrt the traditional analysis: `PCAM` is 1.64 slower than `OR`. Note that in fact most programs enjoys an even smaller overhead but `peep` is about 6.5 slower than `OR` in `PCAM`. This comes from many procedures with many clauses, most of which being not surely cut. Much time is spent in the concatenation operation in this program. Finally, note that adding more functionality in the domain did not slow down the analysis by much.

Accuracy The accuracy results are reported in Table 2. Several versions of the algorithm are compared wrt their ability to detect determinacy of procedures, which was our primary motivation. `P` is using only the domain `pattern` (i.e. cuts are ignored), `C` is only using the cut (i.e. `EXCLUSIVE` always returns false), and `PC, PCA, PCAM` are defined as previously. There are several interesting points to notice. First, `PCAM` detects that 60% of the procedures are deterministic, although many of these programs in fact use heavily the nondeterminism of Prolog. Most of the results are in fact optimal and a nice example is program `Kalah`. Second, the cut and input/output patterns are really complementary to improve the analysis. Input/output patterns alone give 35% of the deterministic procedures, while the cut detects 48%. of the deterministic procedures. The abstractions of arithmetic and meta predicates both add 10% of deterministic procedures. The main lesson here is that all these components are of primary importance to obtain precise results. Note also that the subterm abstraction only marginally improves the overall precision.

Other results are also given for `PCAM`. `F` is the number of procedures with finitely many solutions,

		P		C		PC		PCA		PCAM						
Programs	NP	D	%D	D	%D	D	%D	D	%D	D	%D	F	%F	SS	%SS	ST
Qsort	3	0	0.00	0	0.00	0	0.00	3	1.00	3	1.00	3	1.00	0	0.00	0
Qsort2	5	2	0.40	2	0.40	2	0.40	5	1.00	5	1.00	5	1.00	0	0.00	2
Queens	5	2	0.40	0	0.00	2	0.40	2	0.40	2	0.40	2	0.40	0	0.00	0
Press1	47	8	0.17	19	0.40	19	0.40	19	0.40	19	0.40	23	0.49	1	0.02	9
Press2	47	12	0.26	19	0.40	28	0.60	28	0.60	28	0.60	31	0.66	2	0.04	10
Gabriel	17	0	0.00	4	0.24	4	0.24	4	0.24	4	0.24	4	0.24	0	0.00	1
Peep	24	4	0.17	7	0.29	16	0.67	16	0.67	16	0.67	19	0.79	0	0.00	4
Read	46	11	0.24	27	0.59	31	0.67	31	0.67	31	0.67	33	0.72	8	0.17	13
Kalah	46	16	0.35	20	0.43	33	0.72	40	0.87	40	0.87	40	0.87	6	0.13	13
Cs	32	11	0.34	7	0.22	11	0.34	13	0.41	13	0.41	13	0.41	7	0.22	7
Plan	13	1	0.08	0	0.00	1	0.08	1	0.08	3	0.23	6	0.46	0	0.00	2
Disj	28	13	0.46	11	0.39	13	0.46	13	0.46	13	0.46	14	0.50	10	0.36	12
Pg	10	2	0.20	3	0.30	5	0.50	6	0.50	6	0.50	6	0.50	0	0.00	0
Boyer	24	0	0.00	20	0.83	20	0.83	20	0.83	20	0.83	24	1.00	2	0.08	16
Credit	26	14	0.58	11	0.42	14	0.54	16	0.62	16	0.62	21	0.81	10	0.38	19
Unify	4	0	0.00	0	0.00	0	0.00	0	0.00	4	1.00	4	1.00	0	0.00	0
Unify2	6	0	0.00	0	0.00	0	0.00	0	0.00	2	0.33	2	0.33	0	0.00	0
Mean			0.21		0.29		0.40		0.51		0.60		0.66		0.08	

Table 2: Accuracy of the Cardinality Analysis

`SS` is the number of procedures that surely succeed, and `ST` is the number of procedures that surely terminate. 66% of the procedures were shown to be finite and some results indicated places where the programs could be improved (by inserting cuts or tests). 8% of the procedures are detected as sure success. This would be further improved if a more sophisticated type domain was used. Note also that our analysis detected a single procedure with `snt`, $M = \infty$, and $m = 3$ (which is an indication that the predicate has infinitely many solutions) and two procedures that always fail.

8 Conclusion

This paper introduced a cardinality analysis for full Prolog (without dynamic predicates). The analysis is based on a new abstract interpretation framework and a novel abstract domain for substitution sequences. The overall approach has significant practical and theoretical advantages compared to previous approaches. In particular, all aspects of the analysis are captured in a single framework, the mode/type analysis and the cardinality analysis are performed simultaneously and improve each other's accuracy, the analysis uses input/output patterns, cut, termination, and abstractions of arithmetic and meta-predicates to achieve great precision, and the overhead of the approach is small compared to a tradition mode/type analysis.

There are many directions for extending this work. First, termination analysis can be improved by introducing *a priori* knowledge about termination [1] or by introducing symbolic values in the abstract domain to replace the postfixpoint computation by a mathematical induction argument. Second, sure success information can be improved by using a more sophisticated type domain (e.g. [9]). Finally, the mode analysis can be improved by generalizing the domain to preserve all abstract substitutions which surely succeed and an additional one to represent all remaining cases as suggested in [12]. The tradeoff accuracy/efficiency of these extensions need to be studied carefully.

References

[1] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog control. In *POPL-92*, pages 95–104. ACM Press, 1992.

[2] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

[3] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *POPL-94*, Portland, OR, January 1994.

[4] P Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252, Los Angeles, CA, January 1977.

[5] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.C. Sekar. Extracting Determinacy in Logic Programs. In *ICLP-93*, Budapest (Hungary), June 1993.

[6] S. Debray. Functional Computations in Logic Programs. *ACM TOPLAS*, 11(3):451–481, 1989.

[7] R. Giacobazzi. Detecting Determinate Computations by Bottom-up Abstract Interpretation. In *ESOP'92*, pages 167–181, 1992.

[8] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel execution of logic programs. In *ICLP'96*, pages 25–39, London, July 1986.

[9] G. Janssens and M. Bruynooghe. Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.

[10] T. Kanamori and T. Kawamura. Analysing Success Patterns of Logic Programs by Abstract Hybrid Interpretation. Technical report, ICOT, 1987.

[11] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and Its Complexity Analysis (Extended Abstract). In *ICLP-91*, pages 64–78, Paris (France), June 1991.

[12] B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework for Almost Full Prolog. Research Paper RP-94/?, Department of Computer Science, University of Namur, March 1994. (Submitted to ILPS'94).

[13] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM TOPLAS.*. January 1994

[14] K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):315–347, August 1992.

[15] D. Sahlin. Determinacy Analysis for Full Prolog. In *PEPM'91*, 1991.

[16] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Ma, 1986.

[17] D.H.D Warren. The Andorra Model. Presented at Gigalips Project Workshop, University of Manchester, 1987.

Ø