Incremental Analysis of Constraint Logic Programs

MANUEL HERMENEGILDO and GERMAN PUEBLA
Universidad Politécnica de Madrid
KIM MARRIOTT
Monash University
and
PETER J. STUCKEY
University of Melbourne

Global analyzers traditionally read and analyze the entire program at once, in a nonincremental way. However, there are many situations which are not well suited to this simple model and which instead require reanalysis of certain parts of a program which has already been analyzed. In these cases, it appears inefficient to perform the analysis of the program again from scratch, as needs to be done with current systems. We describe how the fixed-point algorithms used in current generic analysis engines for (constraint) logic programming languages can be extended to support incremental analysis. The possible changes to a program are classified into three types: addition, deletion, and arbitrary change. For each one of these, we provide one or more algorithms for identifying the parts of the analysis that must be recomputed and for performing the actual recomputation. The potential benefits and drawbacks of these algorithms are discussed. Finally, we present some experimental results obtained with an implementation of the algorithms in the PLAI generic abstract interpretation framework. The results show significant benefits when using the proposed incremental analysis algorithms.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming—*Automatic analysis of algorithms, Program transformation*; D.1.6 [**Programming Techniques**]: Logic programming; D.3.4 [**Programming Languages**]: Compilers; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about programs—*Logics of programs*

General Terms: Languages

Additional Key Words and Phrases: abstract interpretation, constraint logic programming, incremental computation, static analysis

---

This work was supported in part by ARC grant A49702580 and CICYT project TIC99-1151 "EDIPIA". Authors' addresses: M. Hermenegildo and G. Puebla, Facultad de Informática, Universidad Politécnica de Madrid, 28660-Boadilla del Monte, Madrid, Spain; email: {herme;german}@fi.upm.es; K. Marriott, School of Computer Science and Software Engineering, Monash University Clayton 3168, Australia; email: marriott@csse.monash.edu.au; P.J. Stuckey, Dept. of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia; email: pjs@cs.mu.oz.au.

1. INTRODUCTION

Global program analysis is becoming a practical tool in constraint logic program compilation in which information about calls, answers, and the effect of the constraint store on variables at different program points is computed statically [Hermenegildo et al. 1992; Van Roy and Despain 1992; Muthukumar and Hermenegildo 1992; Santos-Costa et al. 1991; Bueno et al. 1994]. The underlying theory, formalized in terms of abstract interpretation [Cousot and Cousot 1977], and the related implementation techniques are well understood for several general types of analysis and, in particular, for top-down analysis of Prolog [Debray 1989; 1992; Bruynooghe 1991; Muthukumar and Hermenegildo 1992; Marriott et al. 1994; Charlier and Van Hentenryck 1994]. Several generic analysis engines, such as PLAI [Muthukumar and Hermenegildo 1992; 1990], GAIA [Charlier and Van Hentenryck 1994], and the CLP($\mathcal{R}$) analyzer [Kelly et al. 1998b], facilitate construction of such top-down analyzers. These generic engines have the description domain and functions on this domain as parameters. Different domains give analyzers which provide different types of information and degrees of accuracy. The core of each generic engine is an algorithm for efficient fixed-point computation [Muthukumar and Hermenegildo 1990; 1992; Charlier et al. 1993]. Efficiency is obtained by keeping track of which parts of a program must be reexamined when a success pattern is updated. Current generic analysis engines are nonincremental—the entire program is read, analyzed, and the analysis results written out.

Despite the obvious progress made in global program analysis, most logic program and CLP compilers still perform only local analysis (although the &-Prolog [Hermenegildo and Greene 1991], Aquarius [Van Roy and Despain 1992], Andorra-I [Santos-Costa et al. 1991], and CLP($\mathcal{R}$) [Kelly et al. 1998a] systems are notable exceptions). We believe that an important contributing factor to this is the simple, nonincremental model supported by global analysis systems, which is unsatisfactory for at least three reasons:

—The first reason is that optimizations are often source-to-source transformations;[1] optimization consists of an analyze, perform transformation, then reanalyze cycle. This is inefficient if the analysis starts from scratch each time. Such analyze-transform cycles may occur for example when program optimization and multivariant specialization are combined [Winsborough 1992; Puebla and Hermenegildo 1995; 1999]. This is used, for instance, in program parallelization, where an initial analysis is used to introduce specialized predicate definitions with run-time parallelization tests, and then these new definitions are analyzed and those tests which become redundant in the multiply specialized program removed. It is also the case in optimization of CLP($\mathcal{R}$) in which specialized predicate definitions are reordered and then reanalyzed.

—The second reason is that incremental analysis supports incremental runtime compilation during the test-debug cycle. Again, for efficiency only those parts of the program which are affected by the changes should be reanalyzed. Incremental

[1]By source-to-source transformation we include transformations on the (high-level) internal compiler representation of the program source, which for (constraint) logic program compilers tend to be very close to the source.

compilation is important in the context of logic programs as traditional environments have been interpretive, allowing the rapid generation of prototypes. Incremental analysis is especially important when the system uses analysis information in order to perform compile-time correctness checking of the program [Puebla et al. 2000; Hermenegildo et al. 1999b].

—The third reason is to better handle the optimization of programs in which rules are asserted (added) to or retracted (removed) from the program at runtime.

Clearly, if we modify a program the existing analysis information for it may no longer be correct and/or accurate. However, analysis is often a costly task, and starting analysis again from scratch does not appear to be the best solution. In this article we describe how the fixed-point algorithm in the top-down generic analysis engines for (constraint) logic programs can be extended to support incremental analysis. Guided by the applications mentioned above, we consider algorithms for different types of incrementality. The first, and simplest, type of incrementality is when program rules are added to the original program. The second type of incrementality is rule deletion. We give several algorithms to handle deletion. These capture different trade-offs between efficiency and accuracy. The algorithms for deletion can be easily extended to handle the third and most general type of incrementality, arbitrary change, in which program rules can be deleted or modified in any way. Finally, we consider a restricted type of arbitrary change: local change in which rules are modified, but the answers to the rules are unchanged for the calling patterns they are used with. This case occurs in program optimization, as correctness of the optimization usually amounts to requiring this property. Local change means that changes to the analysis are essentially restricted to recomputing the new call patterns which these rules generate. We give a modification to the fixed-point algorithm which handles this type of incrementality. Finally we give a preliminary empirical evaluation. We argue that the experimental results show that our algorithms are practically important.

In the next section we present the formalization of a fixed-point algorithm which generalizes those used in generic analysis engines. In Section 3 we give an algorithm to handle incremental addition of rules. In Section 4 we give two algorithms to handle incremental deletion of rules. In Section 5 we modify these algorithms to handle arbitrary change of rules. We also give an algorithm to handle the special case of local change. In Section 6 we describe the implementation of the algorithms and our empirical evaluation. Section 7 discusses related work while Section 8 concludes.

2. A GENERIC ANALYSIS ALGORITHM

We start by providing a formalization of a fixed-point algorithm for analysis of (constraint) logic programs. We assume the reader is familiar with constraint logic programming (e.g., see Marriott and Stuckey [1998]) and abstract interpretation (see Cousot and Cousot [1977]). The aim of goal-directed top-down program analysis is, for a particular description domain, to take a program and a set of initial calling patterns and to annotate the program with information about the current environment at each program point whenever that point is reached when executing calls described by the calling patterns.

2.1 Program Analysis by Abstract Interpretation

Abstract interpretation [Cousot and Cousot 1977] is a technique for static program analysis in which execution of the program is simulated on a description (or abstract) domain ($\mathbf{D}$) which is simpler than the actual (or concrete) domain ($\mathbf{C}$). Values in the description domain and sets of values in the actual domain are related via a pair of monotonic mappings $\langle \alpha, \gamma \rangle$: *abstraction* $\alpha : \mathbf{C} \to \mathbf{D}$and *concretization* $\gamma : \mathbf{D} \to \mathbf{C}$ which form a Galois connection. A description $d \in \mathbf{D}$ *approximates* an actual value $c \in \mathbf{C}$ if $\alpha(c) \leq d$ where $\leq$ is the partial ordering on $\mathbf{D}$. Correctness of abstract interpretation guarantees that the descriptions computed approximate all of the actual values which occur during execution of the program.

Different description domains may be used which capture different properties with different accuracy and cost. The description domain that we use in our examples is the *definite Boolean functions* [Armstrong et al. 1994], denoted *Def*. The key idea in this description is to use implication to capture groundness dependencies. The reading of the function $x \to y$ is "if the program variable x is (becomes) ground, so is (does) program variable y." For example, the best description of the constraint $f(X,Y) = f(a, g(U,V))$ is $X \wedge (Y \leftrightarrow (U \wedge V))$. Groundness information is directly useful for many program optimizations such as constraint simplification, parallelization, and simplification of built-ins. It is also indirectly useful for almost all other optimizations of (constraint) logic programs, since it can be combined with many other analysis domains to give more precise analysis information.

We now recall some standard definitions in constraint logic programming. A *constraint logic program* or *program* is a set of *rules* of the form A :- $L_1, \ldots, L_n$, where $L_1, \ldots, L_n$ are literals and A is an atom said to be the *head* of the rule. A *literal* is an atom or a primitive constraint. We assume that each *atom* is normalized; that is to say, it is of the form $p(x_1, \ldots, x_m)$ where p is an m-ary predicate symbol and $x_1, \ldots, x_m$ are distinct variables. A *primitive constraint* is defined by the underlying constraint domain and is of the form $c(e_1, \ldots, e_m)$ where c is an m-ary predicate symbol and the $e_1, \ldots, e_m$ are expressions. For simplicity, in the examples we shall restrict ourselves to the Herbrand domain (Prolog) where primitive constraints are of the form $e_1 = e_2$ where e_1 and e_2 are terms.

As an example of goal-directed top-down program analysis, consider the following program for appending lists:

```
app(X,Y,Z) :- X=[], Y=Z.
app(X,Y,Z) :- X=[U|V], Z=[U|W], app(V,Y,W).
```

Assume that we are interested in analyzing the program for the call $app(X,Y,Z)$ with initial description Y indicating that we wish to analyze it for any call to app with the second argument definitely ground. We will denote this as the *calling pattern* $app(X,Y,Z) : Y$. In essence the analyzer must produce the *program analysis graph* given in Figure 1, which can be viewed as a finite representation of the (possibly infinite) set of (possibly infinite) AND-OR trees explored by the concrete execution [Bruynooghe 1991]. Finiteness of the program analysis graph (and thus termination of analysis) is achieved by considering description domains with certain characteristics (such as being finite, or of finite height, or without infinite ascending chains) or by the use of a *widening* operator [Cousot and Cousot 1977]. The graph has two sorts of nodes: those belonging to rules (also called "AND-nodes") and

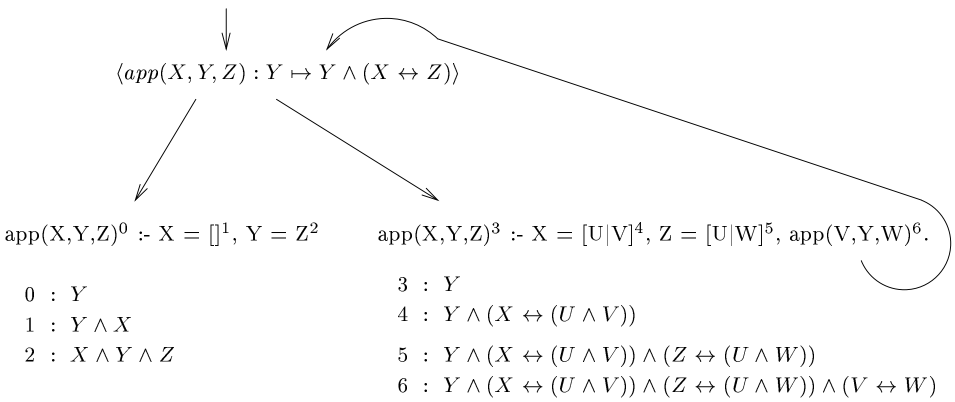

Fig. 1. Example program analysis graph

those belonging to atoms (also called "OR-nodes"). For example, the atom node $\langle app(X,Y,Z) : Y \mapsto Y \wedge (X \leftrightarrow Z)\rangle$ indicates that when the atom $app(X,Y,Z)$ is called with description Y the resulting description is $Y \wedge (X \leftrightarrow Z)$. This answer description depends on the two rules defining *app* which are attached by arcs to the node. These rules are annotated by descriptions at each program point of the constraint store when the rule is executed from the calling pattern of the node connected to the rules. The program points are at the entry to the rule, the point between each two literals, and at the return from the call. Atoms in the rule body have arcs to OR-nodes with the corresponding calling pattern. If such a node is already in the tree it becomes a recursive call. Thus, the analysis graph in Figure 1 has a recursive call to the calling pattern $app(X,Y,Z) : Y$. How this program analysis graph is constructed is detailed in Example 1.

It is implicit in this approach that *sets* of constraints are described by the description at each program point, rather than *sequences* of constraints. Although it is possible to base an analysis on sequences rather than sets, (e.g., see Charlier et al. [1994]) almost all generic (constraint) logic program analysis engines are set-based rather than sequence-based, so we shall focus on these.

As we have seen, a program analysis graph is constructed from an initial set of calling patterns and a program. It is defined in terms of five abstract operations on the description domain. As is standard these are required to be monotonic and to approximate the corresponding concrete operations; for more details see for example García de la Banda et al. [1998]. The abstract operations are

—$Arestrict(CP, V)$ which performs the abstract restriction of a description CP to the variables in the set V;
—$Aextend(CP, V)$ which extends the description CP to the variables in the set V;
—$Aadd(C, CP)$ which performs the abstract operation of conjoining the actual constraint C with the description CP;
—$Aconj(CP_1, CP_2)$ which performs the abstract conjunction of two descriptions;
—$Alub(CP_1, CP_2)$ which performs the abstract disjunction of two descriptions.

As an example, the abstract operations for the description domain *Def* are defined as follows. The abstraction operation α_{Def} gives the best description of a constraint. It is defined as

$$\alpha_{Def}(x = t) = (x \leftrightarrow \bigwedge\{y \in vars(t)\})$$

where x is a variable, t is a term, and the function $vars$ returns the set of variables appearing in some object. For instance, $\alpha_{Def}(X = [U|V])$ is $X \leftrightarrow (U \wedge V)$. We note that term constraints can always be simplified to conjunctions of this form. Extending to conjunctions, we have

$$\alpha_{Def}(e_1 \wedge \cdots \wedge e_k) = \alpha_{Def}(e_1) \wedge \cdots \wedge \alpha_{Def}(e_k)$$

where $e_1, \ldots, e_k$ are term equations.

The remaining operations are defined as follows:

$$\begin{aligned}
\mathsf{Arestrict}(CP, V) &= \exists_{-V}\, CP \\
\mathsf{Aextend}(CP, V) &= CP \\
\mathsf{Aadd}(C, CP) &= \alpha_{Def}(C) \wedge CP \\
\mathsf{Aconj}(CP_1, CP_2) &= CP_1 \wedge CP_2 \\
\mathsf{Alub}(CP_1, CP_2) &= CP_1 \sqcup CP_2
\end{aligned}$$

where $\exists_{-V} F$ represents $\exists v_1 \cdots \exists v_k F$ where $\{v_1, \ldots, v_k\} = vars(F) - V$, and $\sqcup$ is the least upper bound (lub) operation over the *Def* lattice (e.g., Armstrong et al. [1994]). The top ($\top$) of the the *Def* lattice is the formula $true$ while the bottom ($\bot$) is the formula $false$.

For a given program and calling pattern there may be many different analysis graphs. However, for a given set of initial calling patterns, a program and abstract operations on the descriptions, there is a unique *least analysis graph* which gives the most precise information possible.

For the reader with a formal bent, an alternative way of understanding the analysis graph is in terms of the recursive equations for the *general goal-dependent semantics* given in García de la Banda et al. [1998] The least analysis graph corresponds to their least fixed point.

2.2 The Generic Algorithm

We will now describe our generic top-down analysis algorithm which computes the least analysis graph. This algorithm captures the essence of the particular analysis algorithms used in systems such as PLAI [Muthukumar and Hermenegildo 1990; 1992], GAIA [Charlier and Van Hentenryck 1994], and the CLP($\mathcal{R}$) analyzer [Kelly et al. 1998b]. It will form the basis for our algorithms for incremental analysis. However there are several minor differences between the generic algorithm we present and these systems:

—First, the order in which rules for the same predicate are processed to compute the graph is not fixed, since the algorithm is parametric in the analysis strategy used to determine this order. The reasons for this are two-fold: the first reason is generality. The second reason is that the analysis strategy used for static analysis is not necessarily good for incremental analysis, and so we need to be able to explicitly refer to and reason about different strategies.

—Second, the algorithm keeps detailed information about dependencies for each literal in the graph. This is finer grained dependency information than that usually maintained in top-down analysis algorithms. We require this extra precision for efficiency in most of the incremental analysis algorithms.[2]

—Third, the algorithm is deliberately simplified. It does not include many minor optimizations, so as not to obscure the core behavior of the algorithm. Also it is only defined for pure CLP programs. However, standard analysis techniques for handling constructs such as cuts, `not`, and `->` and other built-ins can be added without difficulty [Bueno et al. 1996]; indeed the implementation actually handles (almost) full ISO-Prolog.

We first introduce some notation. CP, possibly subscripted, stands for a description (in the abstract domain). AP, possibly subscripted, stands for a description occurring as an answer description. Each literal in the program is subscripted with an identifier or pair of identifiers. The expression $A : CP$ denotes a *calling pattern*. This consists of an atom (unsubscripted or subscripted) together with a calling description for that atom.

As indicated earlier, rules are assumed to be normalized: only distinct variables are allowed to occur as arguments to atoms. Furthermore, we require that each rule defining a predicate p has identical sequence of variables $x_{p_1}, \ldots x_{p_n}$ in the head atom, i.e., $p(x_{p_1}, \ldots x_{p_n})$. We call this the *base form* of p. Rules in the program are written with a unique subscript attached to the head atom (the rule number), and dual subscript (rule number, body position) attached to each body literal, e.g.,

$$H_k \texttt{ :- } B_{k,1}, \ldots, B_{k,n_k}$$

where $B_{k,i}$ is a subscripted atom or constraint. The rule may also be referred to as rule k, the subscript of the head atom. For example, the append program of Section 2.1 is written

$\texttt{app}_1\texttt{(X,Y,Z) :- X=[]}_{1,1}\texttt{, Y=Z}_{1,2}\texttt{.}$
$\texttt{app}_2\texttt{(X,Y,Z) :- X=[U|V]}_{2,1}\texttt{, Z=[U|W]}_{2,2}\texttt{, app}_{2,3}\texttt{(V,Y,W).}$

The base form of `app` is $\texttt{app}(X, Y, Z)$, and each `app` atom only involves distinct variables as arguments.

The program analysis graph is implicitly represented in the algorithm by means of two data structures, the *answer table* and the *dependency arc table*. Given the information in these it is straightforward to construct the graph and the associated program point annotations. The answer table contains entries of the form $A : CP \mapsto AP$. A is always a base form. This corresponds to an OR-node in the analysis graph of the form $\langle A : CP \mapsto AP \rangle$. It is interpreted as the answer pattern for calls of the form CP to A is AP. A dependency arc is of the form $H_k : CP_0 \Rightarrow [CP_1]\, B_{k,i} : CP_2$. This is interpreted as follows: if the rule with H_k as head is called with description CP_0 then this causes literal $B_{k,i}$ to be called with description CP_2. The remaining part CP_1 is the program annotation just before $B_{k,i}$ is reached and contains information about all variables in rule k. CP_1 is not really necessary, but is

[2]In fact, as we shall see, the overhead of keeping more detailed information is compensated for by avoiding redundant recomputation when an answer pattern is changed.

included for efficiency. Dependency arcs represent the arcs in the program analysis graph from atoms in a rule body to an atom node. For example, the program analysis graph in Figure 1 is represented by

answer table: `app(X, Y, Z)` $: Y \mapsto Y \wedge (X \leftrightarrow Z)$
dependency arc table:
$\texttt{app}_2\texttt{(X,Y,Z)}: Y \Rightarrow [Y \wedge (X \leftrightarrow (U \wedge V)) \wedge (Z \leftrightarrow (U \wedge W))]\ \texttt{app}_{2,3}\texttt{(V,Y,W)}: Y$

Intuitively, the analysis algorithm is just a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixed-point algorithms, we use a priority queue. Thus, the third, and final, structure used in our algorithms is a *prioritized event queue*. Events are of three forms:

—*newcall*(A : CP) which indicates that a new calling pattern for atom A with description CP has been encountered.
—*arc*(R) which indicates that the rule referred to in R needs to be (re)computed from the position indicated.
—*updated*(A : CP) which indicates that the answer description to calling pattern A with description CP has been changed.

The generic analysis algorithm is given in Figure 2. Apart from the parametric description domain-dependent functions, the algorithm has several other undefined functions. The functions add_event and next_event respectively add an event to the priority queue and return (and delete) the event of highest priority.

When an event being added to the priority queue is already in the priority queue, a single event with the maximum of the priorities is kept in the queue. When an arc $H_k : CP \Rightarrow [CP'']B_{k,i} : CP'$ is added to the dependency arc table, it replaces any other arc of the form $H_k : CP \Rightarrow [\_]B_{k,i} : \_$ in the table and the priority queue. Similarly when an entry $H_k : CP \mapsto AP$ is added to the answer table, it replaces any entry of the form $H_k : CP \mapsto \_$. Note that the underscore (_) matches any description, and that there is at most one matching entry in the dependency arc table or answer table at any time.

The function initial_guess returns an initial guess for the answer to a new calling pattern. The default value is $\perp$ but if the calling pattern is more general than an already computed call then its current value may be returned.

The algorithm centers around the processing of events on the priority queue in main_loop, which repeatedly removes the highest priority event and calls the appropriate event-handling function. When all events are processed it calls remove_useless_calls. This procedure traverses the dependency graph given by the dependency arcs from the initial calling patterns S and marks those entries in the dependency arc and answer table which are reachable. The remainder are removed.

The function new_calling_pattern initiates processing of the rules in the definition of atom A, by adding arc events for each of the first literals of these rules, and determines an initial answer for the calling pattern and places this in the table. The function add_dependent_rules adds arc events for each dependency arc which depends on the calling pattern (A : CP) for which the answer has been updated. The function process_arc performs the core of the analysis. It performs a single step

```
analyze(S)
    foreach A : CP ∈ S
        add_event(newcall(A : CP))
    main_loop()

main_loop()
    while E := next_event()
        if (E = newcall(A : CP))
            new_calling_pattern(A : CP)
        elseif (E = updated(A : CP))
            add_dependent_rules(A : CP)
        elseif (E = arc(R))
            process_arc(R)
    endwhile
    remove_useless_calls(S)

new_calling_pattern(A : CP)
    foreach rule A_k :- B_{k,1}, ..., B_{k,n_k}
        CP_0 :=
            Aextend(CP, vars(B_{k,1}, ..., B_{k,n_k}))
        CP_1 := Arestrict(CP_0, vars(B_{k,1}))
        add_event(arc(
            A_k : CP ⇒ [CP_0] B_{k,1} : CP_1))
    AP := initial_guess(A : CP)
    if (AP ≠ ⊥)
        add_event(updated(A : CP))
    add A : CP ↦ AP to answer table

add_dependent_rules(A : CP)
    foreach arc of the form
            H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
            in graph
        where there exists renaming σ
            s.t. A : CP = (B_{k,i} : CP_2)σ
        add_event(arc(
            H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2))

process_arc(H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2)
    if (B_{k,i} is not a constraint)
        add H_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
        to dependency arc table
    W := vars(A_k :- B_{k,1}, ..., B_{k,n_k})
    CP_3 := get_answer(B_{k,i} : CP_2, CP_1, W)
    if (CP_3 ≠ ⊥ and i ≠ n_k)
        CP_4 := Arestrict(CP_3, vars(B_{k,i+1}))
        add_event( arc(
            H_k : CP_0 ⇒ [CP_3] B_{k,i+1} : CP_4))
    elseif (CP_3 ≠ ⊥ and i = n_k)
        AP_1 := Arestrict(CP_3, vars(H_k))
        insert_answer_info(H : CP_0 ↦ AP_1)

get_answer(L : CP_2, CP_1, W)
    if (L is a constraint)
        return Aadd(L, CP_1)
    else
        AP_0 := lookup_answer(L : CP_2)
        AP_1 := Aextend(AP_0, W)
        return Aconj(CP_1, AP_1)

lookup_answer(A : CP)
    if (there exists a renaming σ s.t.
        σ(A : CP) ↦ AP in answer table)
        return σ^{-1}(AP)
    else
        add_event(newcall(σ(A : CP)))
        where σ is a renaming s.t.
        σ(A) is in base form
        return ⊥

insert_answer_info(H : CP ↦ AP)
    AP_0 := lookup_answer(H : CP)
    AP_1 := Alub(AP, AP_0)
    if (AP_0 ≠ AP_1)
        add (H : CP ↦ AP_1) to answer table
        add_event(updated(H : CP))
```

Fig. 2. Generic analysis algorithm.

of the left-to-right traversal of a rule body. If the literal $B_{k,i}$ is an atom, the arc is added to the dependency arc table. The current answer for the call $B_{k,i} : CP_2$ is conjoined with the description CP_1 from the program point immediately before $B_{k,i}$ to obtain the description for the program point after $B_{k,i}$. This is either used to generate a new arc event to process the next literal in the rule if $B_{k,i}$ is not the last literal; otherwise the new answer for the rule is combined with the current answer in insert_answer_info. The function get_answer processes a literal. If it is a constraint, it is simply abstractly added to the current description. If it is an atom, the current answer to that atom for the current description is looked up; then this answer is extended to the variables in the rule the literal occurs in and conjoined with the current description. The functions lookup_answer and insert_answer_info lookup an answer for a calling pattern in the answer table, and update the answer table entry when a new answer is found, respectively. The function lookup_answer also generates *newcall* events in the case that there is no entry for the calling pattern in the answer table.

2.3 Example of the Generic Algorithm

The following example briefly illustrates the operation of the generic fixed-point algorithm. It shows how the `app` program would be analyzed, to obtain the program analysis graph shown in Figure 1.

Example 1. Analysis begins from an initial set S of calling patterns. In our example S contains the single calling pattern `app(X,Y,Z)`:Y. The first step in the algorithm is to add the initial calling patterns as new calling patterns to the priority queue. After this the priority queue contains

newcall(`app(X,Y,Z)`:Y)

and the answer and dependency arc tables are empty. The *newcall* event is taken from the event queue and processed as follows. For each rule defining `app`, an arc is added to the priority queue which indicates the rule body must be processed from the initial literal. An entry for the new calling pattern is added to the answer table with an initial guess of $false$ ($\perp$ for *Def*) as the answer. The data structures are now

priority queue:	*arc*($\mathtt{app}_1\mathtt{(X,\ Y,\ Z)} : Y \Rightarrow [Y]\ \mathtt{X=[]}_{1,1}\ :\ true$)
	arc($\mathtt{app}_2\mathtt{(X,\ Y,\ Z)} : Y \Rightarrow [Y]\ \mathtt{X=[U\vert V]}_{2,1}\ :\ true$)
answer table:	$\mathtt{app(X,\ Y,\ Z)} : Y \mapsto false$
dependency arc table:	no entries

An arc on the event queue is now selected for processing, say the first. The routine get_answer is called to find the answer pattern to the literal `X=[]` with description *true*. As the literal is a constraint, the parametric routine *Aadd* is used. It returns the answer pattern X. A new arc is added to the priority queue which indicates that the second literal in the rule body must be processed. The priority queue is now

arc($\mathtt{app}_1\mathtt{(X,\ Y,\ Z)} : Y \Rightarrow [X \wedge Y]\ \mathtt{Y=Z}_{1,2} : X$)
arc($\mathtt{app}_2\mathtt{(X,\ Y,\ Z)} : Y \Rightarrow [Y]\ \mathtt{X=[U\vert V]}_{2,1}\ :\ true$).

The answer and dependency arc table remain the same.

Again, an arc on the event queue is selected for processing, say the first. As before, get_answer and *Aadd* are called to obtain the next annotation $X \wedge Y \wedge Z$. This time, as there are no more literals in the body, the answer table entry for $app(X, Y, Z) : Y$ is updated. *Alub* is used to find the least upper bound of the new answer $X \wedge Y \wedge Z$ with the old answer $false$. This gives $X \wedge Y \wedge Z$. The entry in the answer table is updated, and an *updated* event is placed on the priority queue. The data structures are now

priority queue:	*updated*($\mathtt{app(X,\ Y,\ Z)} : Y$)
	arc($\mathtt{app}_2\mathtt{(X,\ Y,\ Z)} : Y \Rightarrow [Y]\ \mathtt{X=[U\vert V]}_{2,1}\ :\ true$)
answer table:	$\mathtt{app(X,\ Y,\ Z)} : Y \mapsto X \wedge Y \wedge Z$
dependency arc table:	no entries

The *updated* event can now be processed. As there are no entries in the dependency arc table, nothing in the current program analysis graph depends on the answer to this call, so nothing needs to be recomputed. The priority queue now contains

$$arc(\ \texttt{app}_2\texttt{(X, Y, Z)} : Y \Rightarrow [Y]\ \ \texttt{X=[U|V]}_{2,1} : true).$$

The answer and dependency arc table remain the same.

Similarly to before we process the arc, giving rise to the new priority queue

$$arc(\ \texttt{app}_2\texttt{(X, Y, Z)} : Y \Rightarrow [Y \wedge (X \leftrightarrow (U \wedge V))]\ \ \texttt{Z=[U|W]}_{2,2} : true).$$

The arc is processed to give the priority queue

$$\begin{aligned} arc(\ \texttt{app}_2\texttt{(X, Y, Z)} : Y \Rightarrow\ & [Y \wedge (X \leftrightarrow (U \wedge V)) \wedge (Z \leftrightarrow (U \wedge W))] \\ & \texttt{app(V,Y,W)}_{2,3} : Y). \end{aligned}$$

This time, because $\texttt{app}_{2,3}\texttt{(V,Y,W)}$ is an atom, the arc is added to the arc dependency table. The call $\texttt{get\_answer}(\texttt{app(V,Y,W)}_{2,3} : Y,\ Y \wedge (X \leftrightarrow (U \wedge V)) \wedge (Z \leftrightarrow (U \wedge W))\ ,\ \{X, Y, Z, U, V, W\})$ is made. The answer table is looked up to find the answer to $\texttt{app(V,Y,W)}_{2,3} : Y$ and, appropriately renamed, gives $AP_0 = V \wedge Y \wedge W$. This description is extended to all variables (no change) and then conjoined with the second argument to give the next annotation $Y \wedge V \wedge W \wedge (X \leftrightarrow U) \wedge (Z \leftrightarrow U)$. As this is the last literal in the body, the new answer $Y \wedge (X \leftrightarrow Z)$ is obtained. We take the least upper bound of this answer with the old answer in the table, giving $Y \wedge (X \leftrightarrow Z)$. As the answer has changed, an *updated* event is added to the priority queue. The data structures are now

priority queue:	$\texttt{updated(app(X, Y, Z)} : Y)$
answer table:	$\texttt{app(X, Y, Z)} : Y \mapsto Y \wedge (X \leftrightarrow Z)$
dependency arc table:	$\texttt{app}_2\texttt{(X, Y, Z)} : Y \Rightarrow [Y \wedge (X \leftrightarrow (U \wedge V)) \wedge (Z \leftrightarrow (U \wedge W))]\ \texttt{app}_{2,3}\texttt{(V, Y, W)} : Y$

The *updated* event is processed by looking in the dependency arc table for all arcs which have a body literal which is a variant of $app(X, Y, Z) : Y$ and adding these arcs to the priority queue to be reprocessed. We obtain the new priority queue

$$\begin{aligned} arc(\texttt{app}_2\texttt{(X, Y, Z)} : Y \Rightarrow\ & [Y \wedge (X \leftrightarrow (U \wedge V)) \wedge (Z \leftrightarrow (U \wedge W))] \\ & \texttt{app}_{2,3}\texttt{(V, Y, W)} : Y) \end{aligned}$$

This arc is reprocessed, and gives rise to the answer $Y \wedge (X \leftrightarrow Z)$. Taking the least upper bound of this with the old answer, the result is identical to the old answer, hence no *updated* event is added to the priority queue. As there are no events on the priority queue, the analysis terminates with the desired answer and dependency arc table. □

2.4 Correctness

The generic algorithm provides a simple generic description of how top-down goal-directed analysis is performed. It is somewhat less abstract than the semantic equations, since we need to capture the use of dependency information during analysis. The algorithm captures the behavior of several implemented algorithms while at the same time is suitable for incremental analysis. Different top-down goal-directed analysis algorithms correspond to different event-processing strategies. In practice these algorithms also incorporate other optimizations. An example event-processing strategy would be to always perform *newcall* events first, to process nonrecursive rules before recursive rules, and to finish processing a rule before starting another. This strategy would produce an algorithm which is quite close to the one used in PLAI or GAIA (the differences between the proposed algorithm and that used in PLAI are presented in more detail in Section 6).

In essence, the algorithm defines a set of recursive equations whose least fixed point is computed using chaotic iteration [Cousot and Cousot 1977]. We note that even though the order in which events are processed is not fixed, the events themselves encode a left-to-right traversal of the rules, ensuring a unique result. For the least fixed point to be well-defined we require that the abstract operations are monotonic and that initial_guess returns a value below the least fixed point. Under these standard assumptions we have

THEOREM 1. *For a program P and calling patterns S, the generic analysis algorithm returns an answer table and dependency arc table which represents the least program analysis graph of P and S.*

The dependency arc table does not quite capture the annotations on rules in the analysis graph, since program points before constraint literals and the last program point do not correspond to stored arcs. This information can easily be recomputed from the dependency arc table, or indeed the algorithm can be simply modified to save it as it executes.

The corollary of the above theorem is that the priority strategy does not involve correctness of the analysis. This corollary will be vital when arguing correctness of the incremental algorithms in the following sections.

COROLLARY 1. *The result of the generic analysis algorithm does not depend on the strategy used to prioritize events.*

3. INCREMENTAL ADDITION

If new rules are added to a program which has already been analyzed, we have to compute the success patterns for each rule, use this to update the answer table information for the atoms defined by the rules, and then propagate the effect of these changes. Note that this propagation is not limited to the new rules, but rather a global fixed point has to be reached in order to ensure correctness of the analysis results. Existing analysis engines for (constraint) logic programming are unable to incrementally compute this new fixed point, and the only safe possibility is to start analysis from scratch. However, the generic algorithm we propose can do this rather simply. Computation of the success patterns for each rule is simply done by adding a set of arcs to the event queue before calling again main_loop. Propagation of the effects corresponds to processing, in the usual way, the *updated* events for entries in the answer table which are modified due to the newly added rules. When execution of main_loop ends, a new global fixed point has been reached.

The new routine for analysis of programs in which rules are added incrementally is given in Figure 3. The routine takes as input the set of new rules R. If these

```
incremental_addition(R)
   foreach rule A_k :- B_{k,1}, ..., B_{k,n_k} ∈ R
      foreach entry A : CP ↦ AP in the answer table
         CP_0 := Aextend(CP, vars(A_k :- B_{k,1}, ..., B_{k,n_k}))
         CP_1 := Arestrict(CP_0, vars(B_{k,1}))
         add_event(arc(A_k : CP ⇒ [CP_0] B_{k,1} : CP_1))
   main_loop()
```

Fig. 3. Incremental addition algorithm

match an atom with a calling pattern of interest, then requests to process the rule are placed on the priority queue. Subsequent processing is exactly as for the nonincremental case.

Example 2. As an example, we begin with the program for naive reversal of a list, `rev`, already analyzed for the calling pattern `rev(X, Y) :` *true* but without a definition of the append, `app`, predicate. The initial program is

$\texttt{rev}_1\texttt{(X, Y) :- X = []}_{1,1}\texttt{, Y = []}_{1,2}\texttt{.}$
$\texttt{rev}_2\texttt{(X, Y) :- X = [U|V]}_{2,1}\texttt{, rev}_{2,2}\texttt{(V, W), T = [U]}_{2,3}\texttt{, app}_{2,4}\texttt{(W, T, Y).}$

The answer table and dependency arc tables are (**State 1**)

answer table: $\texttt{rev(X, Y)} : true \mapsto X \wedge Y$
$\texttt{app(X, Y, Z)} : X \mapsto false$

dependency arc table:
$\texttt{rev}_2\texttt{(X, Y)} : true \Rightarrow [X \leftrightarrow (U \wedge V)]\ \ \texttt{rev}_{2,2}\texttt{(V, W)} : true$
$\texttt{rev}_2\texttt{(X, Y)} : true \Rightarrow [(X \leftrightarrow (U \wedge V)) \wedge V \wedge W \wedge (T \leftrightarrow U)]\ \ \texttt{app}_{2,4}\texttt{(W, T, Y)} : W$

We now add the rules for `app` one at a time. The first rule to be added is

$$\texttt{app}_3\texttt{(X, Y, Z) :- X = []}_{3,1}\texttt{, Y = Z}_{3,2}\texttt{.}$$

The incremental analysis begins by looking for entries referring to *app* in the answer table. It finds the entry $app(X,Y,Z) : X$ so the arc

$$\texttt{app}_3\texttt{(X,Y,Z)} : X \Rightarrow [X]\ \ \texttt{X = []}_{3,1} : X$$

is put in the priority queue. After processing this rule, the new answer $X \wedge (Y \leftrightarrow Z)$ for `app(X,Y,Z):`X is obtained. This is lubbed with the current answer to obtain $X \wedge (Y \leftrightarrow Z)$, and the answer table entry is updated (causing an *updated*(`app(X,Y,Z)` $: X$) event). Examining the dependency arc table, the algorithm recognizes that the answer from rule 2 must now be recomputed. This gives rise to the new answer $(X \leftrightarrow (U \wedge V)) \wedge V \wedge W \wedge (U \leftrightarrow Y)$ which restricted to $\{X, Y\}$ gives $X \leftrightarrow Y$. Taking the least upper bound of this with the current answer $X \wedge Y$ gives $X \leftrightarrow Y$. The memo table entry for `rev(X, Y) :` *true* is updated appropriately, and an *updated* event is placed on the queue. Again the answer to rule 2 must be recomputed. First we obtain a new calling pattern $\texttt{app}_{2,4}\texttt{(X, Y, Z)} : true$. This means that the dependency arc

$$\texttt{rev}_2\texttt{(X,Y)} : true \Rightarrow [(X \leftrightarrow (U \wedge V)) \wedge V \wedge W \wedge (T \leftrightarrow U)]\ \texttt{app}_{2,4}\texttt{(W,T,Y)} : W$$

in the dependency arc table is replaced by

$$\texttt{rev}_2\texttt{(X,Y)} : true \Rightarrow [(X \leftrightarrow (U \wedge V) \wedge (V \leftrightarrow W) \wedge (T \leftrightarrow U)]\ \texttt{app}_{2,4}\texttt{(W,T,Y)} : true$$

This sets up a new call `app(X, Y, Z)` : *true*. The current answer for the old call `app(X, Y, Z)` : X can be used as an initial guess to the new, more general, call. The algorithm examines rule 3 for the new calling pattern. It obtains the same answer $X \wedge (Y \leftrightarrow Z)$.

This leads to a new answer for $\texttt{rev}_2$`(X, Y)`, $(X \leftrightarrow (U \wedge V)) \wedge (V \leftrightarrow W) \wedge W \wedge (T \leftrightarrow U) \wedge (T \leftrightarrow Y)$, which restricted to $\{X, Y\}$ gives $X \leftrightarrow Y$. This does not change the current answer, so the main loop of the analysis is finished. The reachability analysis removes the entry `app(X, Y, Z)` : $X \mapsto X \wedge (Y \leftrightarrow Z)$ from the answer table. The resulting answer and dependency arc table entries are (**State 2**)

answer table: `rev(X, Y)` : $\mathit{true} \mapsto X \leftrightarrow Y$
`app(X, Y, Z)` : $\mathit{true} \mapsto X \wedge (Y \leftrightarrow Z)$
dep. arc table:
$\texttt{rev}_2$`(X,Y)` : $\mathit{true} \Rightarrow [X \leftrightarrow (U \wedge V)]$ $\texttt{rev}_{2,2}$`(V,W)` : *true*
$\texttt{rev}_2$`(X,Y)` : $\mathit{true} \Rightarrow [X \leftrightarrow (U \wedge V) \wedge (V \leftrightarrow W) \wedge (T \leftrightarrow U)]$ $\texttt{app}_{2,4}$`(W,T,Y)` : *true*

If the second rule for `app`

$\texttt{app}_4$`(X,Y,Z) :- X = [U|V]`$_{4,1}$`, Z = [U|W]`$_{4,2}$`, `$\texttt{app}_{4,3}$`(V,Y,W).`

is added, the analysis proceeds similarly. The final memo and dependency arc table entries are (**State 3**)

answer table: `rev(X, Y)` : $\mathit{true} \mapsto X \leftrightarrow Y$
`app(X, Y, Z)` : $\mathit{true} \mapsto (X \wedge Y) \leftrightarrow Z$
dep. arc table:
(A) $\texttt{rev}_2$`(X, Y)` : $\mathit{true} \Rightarrow [X \leftrightarrow (U \wedge V)]$ $\texttt{rev}_{2,2}$`(V,W)` : *true*
(B) $\texttt{rev}_2$`(X,Y)`:*true* $\Rightarrow [X \leftrightarrow (U \wedge V) \wedge (V \leftrightarrow W) \wedge (T \leftrightarrow U)]$ $\texttt{app}_{2,4}$`(W,T,Y)`:*true*
(C) $\texttt{app}_4$`(X,Y,Z)` : $\mathit{true} \Rightarrow [X \leftrightarrow (U \wedge V) \wedge Z \leftrightarrow (U \wedge W)]$ $\texttt{app}_{4,3}$`(V,Y,W)` : *true*

□

Correctness of the incremental addition algorithm follows from correctness of the original generic algorithm. Essentially execution of the incremental addition algorithm corresponds to executing the generic algorithm with all rules but with the new rules having the lowest priority for processing. It therefore follows from Corollary 1 that:

THEOREM 2. *If the rules in a program are analyzed incrementally with the incremental addition algorithm, the same answer and dependency arc tables will be obtained as when all rules are analyzed at once by the generic algorithm.*

In a sense, therefore, the cost of performing the analysis incrementally can be no worse than performing the analysis all at once, as the generic analysis could have used a priority strategy which has the same cost as the incremental strategy. We will now formalize this intuition. Our cost measure will be the number of calls to the underlying parametric functions. This is a fairly simplistic measure, but our results continue to hold for reasonable measures.

Let $C_{noninc}(\bar{F}, R, S)$ be the worst-case number of calls to the parametric functions $\bar{F}$ when analyzing the rules R and call patterns S for all possible priority strategies with the generic analysis algorithm.

Let $C_{add}(\bar{F}, R, R', S)$ be the worst-case number of calls to the parametric functions $\bar{F}$ when analyzing the new rules R' for all possible priority strategies with the incremental addition algorithm after already analyzing the program R for call patterns S.

THEOREM 3. *Let the set of rules R be partitioned into $R_1, ..., R_n$ rule sets. For any call patterns S and parametric functions $\bar{F}$,*

$$C_{noninc}(\bar{F}, R, S) \geq \sum_{i=1}^{n} C_{add}(\bar{F}, (\bigcup_{j=1}^{j<i} R_j), R_i, S).$$

The theorem holds because the priority strategies which give the worst-case behavior for each of the R_1, ..., R_n can be combined to give a priority strategy for analyzing the program nonincrementally.

We note that our theorems comparing relative complexity of incremental and nonincremental analysis (Theorems 3, 5, and 8) are rather weak, since they relate only the worst-case complexity. Unfortunately, it is difficult to provide more insightful analytic comparisons; instead we will provide an empirical comparison in Section 6.

4. INCREMENTAL DELETION

In this section we consider deletion of rules from an already analyzed program and how to incrementally update the analysis information. The first thing to note is that, unlike incremental addition, we need not change the analysis results at all. The current approximation is trivially guaranteed to be correct, because the contribution of the rules are lubbed to give the answer table entry and so correctly describe the contribution of each remaining rule. This approach is obviously inaccurate but simple.

4.1 Refinement

More accuracy can be obtained by applying a strategy similar to *narrowing* [Cousot and Cousot 1979]. Narrowing is a generic fixed-point approximation technique in which analysis proceeds from above the least fixed point and iterates downward until a fixed point (not necessarily the least fixed point) is reached. We can use this approach because the current approximations in the answer table are greater than or equal to those in the answer table of the program analysis graph for the modified program. Applying the analysis engine as usual except taking the greatest lower bound (glb), written $\sqcap$, of new answers with the old rather than the least upper bound is guaranteed to produce a correct, albeit perhaps imprecise, result. We can let this process be guided from the initial changes using the dependency graph information. Care must be taken to treat new calling patterns that arise in this process correctly. Note this narrowing-like strategy is correct in part because of the existence of a Galois connection between the concrete and abstract domains, as this means that glb on the abstract domain approximates glb on the underlying concrete domain.

Example 3. Consider the program in Example 2 after both additions. The current answer table and dependency graph entries are given by **State 3**. Deleting rule 4 results in the following process.

First we delete any dependency arcs which correspond to deleted rules. In this case we remove the arc $\texttt{app}_4\texttt{(X, Y, Z)} : \mathit{true} \Rightarrow [\_]\ \texttt{app}_{4,1}\texttt{(V, Y, W)} : \mathit{true}$. In general we may subsequently delete other dependency arcs which are no longer required.

We recompute the answer information for all (remaining) rules for `app(X, Y, Z)` for all calling patterns of interest using the current answer information. We obtain $\texttt{app(X, Y, Z)} : \mathit{true} \mapsto X \wedge (Y \leftrightarrow Z)$.

Because this information has changed we now need to consider recomputing answer information for any calling patterns that depend on $\texttt{app(X, Y, Z)} : \mathit{true}$, in this case $\texttt{rev(X, Y)} : \mathit{true}$. Recomputing using rules 1 and 2 obtains the same answer information $X \leftrightarrow Y$. The result is **State 2** (with the useless entry for $\texttt{app(X, Y, Z)}{:}X$ removed).

Deleting rule 3 subsequently leads back to **State 1** as expected. In contrast removing rule 3 from the program consisting of rules 1 to 4 does not result in recovering **State 1** as might be expected. This highlights the possible inaccuracy of the narrowing method. In this case rule 4 prevents more accurate answer information from being acquired. □

The disadvantage of this method is its inaccuracy. Starting the analysis from scratch will often give a more accurate result. We now give two algorithms which are incremental yet are as accurate as the nonincremental analysis.

4.2 "Top-Down" Deletion Algorithm

The first accurate method we explore for incremental analysis of programs after deletion is to remove all information in the answer and dependency arc tables which depends on the rules which have been deleted and then to restart the analysis. Not only will removal of rules change the answers in the answer table, it will also mean that subsequent calling patterns may change. Thus we will also remove entries in the dependency arc table for those rules which are going to be reanalyzed.

Information in the answer table and dependency arc table allows us to find these no longer valid entries. Let D be the set of deleted rules and H be the set of atoms which occur as the head of a deleted rule, i.e., H is $\{A | (A \texttt{ :- } B) \in D\}$. We let $\mathit{depend}(H)$ denote the set of calling patterns whose answers depend on some atom in H. More precisely, $\mathit{depend}(H)$ is the smallest superset of

$$\{(A : \mathit{CP}) | (A : \mathit{CP} \mapsto \mathit{AP}) \in \text{answer table and } A \in H\}$$

such that if $A : \mathit{CP}$ is in $\mathit{depend}(H)$ and there is a dependency arc of the form $B_{\_} : \mathit{CP}_0 \Rightarrow [\_]A'_{\_} : \mathit{CP}'$ such that $A' : \mathit{CP}'$ is a renaming of $A : \mathit{CP}$ then $B : \mathit{CP}_0$ is also in $\mathit{depend}(H)$. After entries for these dependent calling patterns which are no longer valid are deleted, the usual generic analysis is performed. The routine for top-down rule deletion is given in Figure 4. It is called with the set of deleted rules D and a set of initial calling patterns S.

Example 4. Consider the program

$\texttt{q}_1 \texttt{ :- } \texttt{p}_{1,1}\texttt{(X, Y), } \texttt{r}_{1,2}\texttt{(X, Y, Z), } \texttt{s}_{1,3}\texttt{(Y, Z).}$
$\texttt{p}_2\texttt{(X, Y) :- X = } \texttt{a}_{2,1}\texttt{, Y = } \texttt{b}_{2,2}\texttt{.}$
$\texttt{p}_3\texttt{(X, Y) :- X = } \texttt{Y}_{3,1}\texttt{.}$
$\texttt{r}_4\texttt{(X, Y, Z) :- X = } \texttt{Z}_{4,1}\texttt{.}$

```
top_down_delete(D, S)
   H := {A|(A :- B) ∈ D}
   T := depend(H)
   foreach A : CP ∈ T
      delete entry A : CP ↦ _ from answer table
      delete each arc A_ : CP ⇒ [_] _ : _ from dependency arc table
   foreach A : CP ∈ S ∩ T
      add_event(newcall(A : CP))
   main_loop()
```

Fig. 4. Top-down incremental deletion algorithm

$\texttt{r}_5\texttt{(X, Y, Z) :- Y = Z}_{5,1}\texttt{.}$
$\texttt{s}_6\texttt{(Y, Z) :- Y = c}_{6,1}\texttt{.}$

After program analysis we obtain (**State 5**)

answer table:	$\texttt{q} : true \mapsto true$
	$\texttt{p(X, Y)} : true \mapsto X \leftrightarrow Y$
	$\texttt{r(X, Y, Z)} : X \leftrightarrow Y \mapsto (X \leftrightarrow Y) \wedge (Y \leftrightarrow Z)$
	$\texttt{s(Y, Z)} : Y \leftrightarrow Z \mapsto Y \wedge Z$
dependency arc table:	(D) $\texttt{q}_1 : true \Rightarrow [true]\ \ \texttt{p}_{1,1}\texttt{(X, Y)} : true$
	(E) $\texttt{q}_1 : true \Rightarrow [X \leftrightarrow Y]\ \ \texttt{r}_{1,2}\texttt{(X, Y, Z)} : X \leftrightarrow Y$
	(F) $\texttt{q}_1 : true \Rightarrow [(X \leftrightarrow Y) \wedge (Y \leftrightarrow Z)]\ \ \texttt{s}_{1,3}\texttt{(Y, Z)} : Y \leftrightarrow Z$

Now consider the deletion of rule $\texttt{r}_5$. $H = \{\texttt{r(X,Y,Z)}\}$ and $depend(H) = \{\texttt{r(X,Y,Z)} : X \leftrightarrow Y, \texttt{q} : true\}$. Hence all the dependency arcs are deleted, as well as the answer table entries for `r` and `q`. The initial state (**State 6**) when we start the main loop is

answer table:	$\texttt{p(X, Y)} : true \mapsto X \leftrightarrow Y$
	$\texttt{s(Y, Z)} : Y \leftrightarrow Z \mapsto Y \wedge Z$
dependency arc table:	no entries

The priority queue entry is $newcall(\texttt{q} : true)$. We start a new answer entry for $\texttt{q} : true \mapsto false$ and add an arc event for (D). This is selected; arc (D) is added again to the dependency arc table; and arc (E) is placed on the priority queue. This is selected; arc (E) is placed back in the dependency arc table, and the event $newcall(\texttt{r(X, Y, Z)} : X \leftrightarrow Y)$ is placed on the queue. This generates an answer entry $\texttt{r(X, Y, Z)} : X \leftrightarrow Y \mapsto false$ and arc $\texttt{r}_4\texttt{(X, Y, Z)} : X \leftrightarrow Y \Rightarrow [X \leftrightarrow Y]\ \texttt{X = Z} : true$. This in turn generates new answer information $(X \leftrightarrow Y) \wedge (Y \leftrightarrow Z)$ and the event $updated(\texttt{r(X, Y, Z)} : X \leftrightarrow Y)$. This is replaced with arc (E), which is replaced with arc (F), which results in arc (F) being added again to the dependency graph and new answer info $\texttt{q} : true \mapsto true$ and an event $updated(\texttt{q} : true)$ which is removed with no effect. The resulting state is identical to the starting state (**State 5**). □

Example 5. Consider again the `rev` and `app` program in Example 2. After analysis of the entire program we are in **State 3**. Now consider the deletion of rule 3 from the program consisting of rules 1 to 4. $T = depend(\texttt{app(X, Y, Z)})$ is all the calling patterns, so the answer table and dependency arc table are emptied. Reanalysis is complete starting from the initial calling pattern $\texttt{rev(X,Y)} : \mathit{true}$ and results in **State 1** as expected. Note that this is the case in Example 3 for which the refinement method yielded an inaccurate answer. □

Correctness of the incremental top-down deletion algorithm follows from correctness of the original generic algorithm. Execution of the top-down deletion algorithm is identical to that of the generic algorithm except that information about the answers to some call patterns which do not depend on the deleted rules is already in the data structures.

THEOREM 4. *If a program P is first analyzed and then rules R are deleted from the program and the remaining rules are reanalyzed with the top-down deletion algorithm, the same answer and dependency arc tables will be obtained as when the rules $P \setminus R$ are analyzed by the generic algorithm.*

The cost of performing the actual analysis incrementally can be no worse than performing the analysis all at once. Let $C_{del-td}(\bar{F}, R, R', S)$ be the worst-case number of calls to the parametric functions $\bar{F}$ when analyzing the program R with rules R' deleted for all possible priority strategies with the top-down deletion algorithm after already analyzing the program R for call patterns S.

THEOREM 5. *Let R and R' be sets of rules such that $R' \subseteq R$. For any call patterns S and parametric functions $\bar{F}$,*

$$C_{noninc}(\bar{F}, R \setminus R', S) \geq C_{del-td}(\bar{F}, R, R', S).$$

4.3 "Bottom-up" Deletion Algorithm

The last theorem shows that the top-down deletion algorithm is never worse than starting the analysis from scratch. However, in practice it is unlikely to be that much better, as on average deleting a single rule will mean that half of the dependency arcs and answers are deleted in the first phase of the algorithm. The reason is that the top-down algorithm is very pessimistic—deleting everything unless it is sure that it will be useful. For this reason we now consider a more optimistic algorithm. The algorithm assumes that calling patterns to changed predicate definitions are still likely to be useful. In the worst case it may spend a large amount of time reanalyzing calling patterns that end up being useless. But in the best case we do not need to reexamine large parts of the program above changes when no actual effect is felt.

The algorithm proceeds by computing new answers for calling patterns in the lowest strongly connected component[3] (SCC) of the set $\mathit{depend}(H)$ of calling patterns which could be affected by the rule deletion. After evaluating the lowest SCC, the algorithm moves upward to higher SCCs. At each stage the algorithm recomputes or verifies the current answers to the calls to the SCC without considering dependency arcs from SCCs in higher levels. This is possible because if the answer changes, the *arc* events they would generate are computed anyway. If the answers

[3]The set of nodes in a graph can be partitioned into *strongly connected components* $S_1, \ldots, S_n$ $n \geq 0$ so that no node in S_i can reach a node is $S_j, \forall j > i$. Two nodes n_1, n_2 are in the same strongly connected component S_i if and only if both n_1 can reach n_2 and n_2 can reach n_1.

```
bottom_up_delete(D, S)
  H := ∅
  foreach rule A_k :- B_{k,1}, ..., B_{k,n_k} ∈ D
    foreach A : CP ↦ AP in answer table
      H := H ∪ {A : CP}
      delete each arc A_ : CP ⇒ [_]_ : _ from dependency arc table
  while H is not empty
    let B : _ ∈ H be such that B is of minimum predicate SCC level
    T := calling patterns in program analysis graph for predicates in
        the same predicate SCC level as B
    foreach A : CP ∈ T
      delete each arc A_ : CP ⇒ [_]_ : _ from dependency arc table
    foreach A : CP ∈ external_calls(T, S)
      move entry A : CP ↦ AP from answer table to old answer table
      foreach arc B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 in dependency arc table
        where there exists renaming σ s.t. (A : CP) = (B_{k,j} : CP_2)σ
        move B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 to old dependency table
      add_event(newcall(A : CP))
    main_loop()
    foreach A : CP ∈ external_calls(T, S)
      foreach arc B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 in old dependency arc table
        where there exists renaming σ s.t. (A : CP) = (B_{k,j} : CP_2)σ
        if answer pattern for A : CP in old answer table and answer table agree
          move B_k : CP_0 ⇒ [CP_1] B_{k,j} : CP_2 to dependency arc table
        else
          H := H ∪ {B : CP_0}
    H := H − T
    empty old answer table and old dependency arc table

external_calls(T, S)
  U := ∅
  foreach A : CP ∈ T
    where exists arc B_k : CP_0 ⇒ [CP_1] B_{k,i} : CP_2
    and B : CP_0 ∉ T
    and there exists renaming σ s.t. (A : CP) = (B_{k,i} : CP_2)σ
    %% this means there is an external call
    U = U ∪ {A : CP}
  return U ∪ (T ∩ S)
```

Fig. 5. Bottom-up incremental deletion algorithm

are unchanged then the algorithm stops; otherwise it examines the SCCs which depend on the changed answers (using the dependency arcs). For obvious reasons we call the algorithm Bottom-Up Deletion. It is shown in Figure 5.

Rather than using the program analysis graph to determine SCCs, an object which changes as the analysis proceeds, the algorithm uses the predicate call graph of the program P before the rule deletions to determine the calling patterns that can affect one another. This is static, and gives overestimates on the SCCs of the program analysis graph (at any stage), and hence this approach is safe. The *predicate call graph* has predicate names as nodes, and an arc from node `p` to `q` if there exists a rule with predicate `p` in the head and an atom for predicate `q` appearing in the body. We can use this graph to define a *predicate SCC level* to each predicate (and atom) where predicates in lower levels cannot reach any predicate in a higher level.

The algorithm begins by collecting all the calling patterns for atoms with deleted rules, and deleting all dependency arcs for deleted rules, since they can play no part in the reanalysis. It calculates as H the initial calling patterns which must be reevaluated since they have had a rule removed.

It then chooses the minimum predicate SCC level for predicates in H and collects all the affected calling patterns in this level in T. All the dependency arcs for pairs in T are deleted; the current answers for pairs in T are moved to the old table for later comparing with the new answers.

Next the external calls to calling patterns in T are calculated by external_calls(T,S). These are the calling patterns for which there is dependency arc from a higher SCC levels, or an external call from S. All the dependency arcs which call T from higher SCC are moved temporarily into the old dependency arc table. During analysis of this SCC, *updated* events may be generated which do not need to be propagated outside the SCC, as the answer computed after deleting some rules may (finally) be equal to the one computed for the original program. In effect this isolates the calls T from the rest of the higher SCC levels of the program analysis graph. Each external call is then treated as a *newcall* event.

After the SCC has been fully analyzed, a comparison is made for each external calling pattern $A : CP$. If the new and old answers in the table agree then no recomputation of calling patterns which depend on this $A : CP$ is required. Otherwise the dependent calling patterns are added to H, the calling patterns which need recomputation. Finally the calling patterns T are deleted from H, and the process proceeds.

Example 6. Consider the same deletion as in Example 4. Initially the set H is $\{$`r(X,Y,Z)` $: X \leftrightarrow Y\}$, and T is the same set. There are no dependency arcs for pairs in T, and the single pattern is an external call because of arc (E). The answer table entry `r(X,Y,Z)` $: X \leftrightarrow Y \mapsto (X \leftrightarrow Y) \wedge (Y \leftrightarrow Z)$ is moved to the old answer table. The dependency arc (E) is moved to the old dependency table. The event *newcall*(`r(X,Y,Z)` $: X \leftrightarrow Y$) is placed on the queue. This (re)generates new answer information `r(X,Y,Z)` $: X \leftrightarrow Y \mapsto (X \leftrightarrow Y) \wedge (Y \leftrightarrow Z)$ and an event *updated*(`r(X,Y,Z)` $: X \leftrightarrow Y$). As the dependency arc table has no arc that needs to be recomputed we stop. Because the answer for `r(X,Y,Z)` $: X \leftrightarrow Y$ is unchanged the arc (E) is moved to the dependency arc table and the algorithm terminates, without recomputing `q` $: true$. □

Example 7. Consider the `rev` and `app` program. After analysis of the entire program we are in **State 3**. Now consider the deletion of rule 3 from the program. H is initially $\{$`app(X,Y,Z)` $: true\}$. So is T. We remove the arc (C). We move the answer pattern for `app(X,Y,Z)` $: true$ and the arc (B) to the old tables. The event *newcall*(`app(X,Y,Z)` $: true$) is placed in the queue. The analysis proceeds to compute answer `app(X,Y,Z)` $: true \mapsto false$. Since this has changed, `rev(X,Y)` $:$ $true$ is added to H. `app(X,Y,Z)` $: true$ is removed from H. In the next iteration $H = \{$`rev(X,Y)` $: true\}$. The answer pattern for `rev(X,Y)` $: true$ is moved to the old table, and the arc (A) is removed. Reanalysis proceeds as before including building a new call to `app(X,Y,Z)` $: X$. This gives the answer `rev(X,Y)` $: true \mapsto X \wedge Y$. The resulting state is **State 1** as expected. Note that the reanalysis of `app(X,Y,Z)` :

true was unnecessary for computing the answers to the call to `rev` (this was avoided by the top-down deletion). □

Proving correctness of the incremental bottom-up deletion algorithm requires an inductive proof on the SCCs. Correctness of the algorithm for each SCC follows from correctness of the generic algorithm.

THEOREM 6. *If a program P is first analyzed for calls S, and then rules R are deleted from the program, while the remaining rules are reanalyzed with the bottom-up deletion algorithm, then the same answer and dependency arc tables will be obtained as when the rules $P \setminus R$ are analyzed by the generic algorithm for S.*

Unfortunately, in the worst case, reanalysis with the bottom-up deletion algorithm may take longer than reanalyzing the program from scratch using the generic algorithm. This is because the bottom-up algorithm may do a lot of work recomputing the answer patterns to calls in the lower SCCs which are no longer made. In practice, however, if the changes are few and have local extent, the bottom-up algorithm will be faster than the top-down.

5. ARBITRARY CHANGE

In this section we consider the most general case of program modification, in which rules can both be deleted from and added to an already analyzed program and how to incrementally update the analysis information. Given the above algorithms for addition and deletion of rules we can handle any possible change of a set of rules by first deleting the original and then adding the revised version. This is inefficient, since the revision may not involve very far reaching changes, while the deletion and addition together do. Moreover we compute two fixed points rather than one.

Instead, we can use the top-down and bottom-up deletion algorithms in order to perform incremental analysis while only computing a single fixed point. For the case of top-down deletion it suffices to execute main_loop with the updated set of rules. For using the bottom-up deletion algorithm, and unlike in the case of incremental deletion, we must recompute the SCCs in order to ensure correctness, as new cycles may have been introduced in the call dependency graph due to the newly added rules. Then, we can use the bottom-up deletion algorithm as usual with the updated set of rules.

Example 8. Consider the following program:

$\mathtt{q}_1\mathtt{(X, Y) :- p}_{1,1}\mathtt{(X, Y).}$
$\mathtt{p}_2\mathtt{(X, Y) :- X=a}_{2,1}\mathtt{, Y=b}_{2,2}\mathtt{.}$

The complete analysis information for the initial call `q(X, Y)`: X is

answer table:	`q(X, Y)` $: X \mapsto X \wedge Y$
	`p(X, Y)` $: X \mapsto X \wedge Y$
dependency arc table:	$\mathtt{q}_1$`(X, Y)` $: X \Rightarrow \mathtt{p}_{1,1}$`(X, Y)` $: X$

Consider replacing the rule for `p` by $\mathtt{p}_3$`(X, Y) :- U =` $\mathtt{a}_{3,1}$`,` $\mathtt{q}_{3,2}$`(U, Y)`. If we do not recompute the SCCs, the bottom-up algorithm would analyze the rule $\mathtt{p}_3$`(X, Y) :- U =` $\mathtt{a}_{3,1}$`,` $\mathtt{q}_{3,2}$`(U, Y)` with entry description X. Using the (no longer correct) entry `q(X, Y)` $: X \mapsto X \wedge Y$ in the answer table, analysis would compute the

```
local_change(S, R)
   let R be of the form A_k :- D_{k,1}, ..., D_{k,n_k}
   T := ∅
   foreach A : CP ↦ AP in answer table
       T := T ∪ {A : CP}
   T := T plus all B : CP_0 in same SCCs of program analysis graph
   delete each arc of the form A_k : _ ⇒ [_ ]_ : _ from the dependency arc table
   foreach A : CP ∈ external_calls(T, S)
       CP_0 := Aextend(CP, vars(D_{k,1}, ..., D_{k,n_k}))
       CP_1 := Arestrict(CP_0, vars(D_{k,1}))
       add_event(arc(A_k : CP ⇒ [CP_0] D_{k,1} : CP_1))
   main_loop()
```

Fig. 6. Local change algorithm

incorrect entry `p(X, Y)` : $X \mapsto X \wedge Y$ which is consistent with the old table and thus terminate. However, if we had recomputed the SCCs, since `p` and `q` are in the same SCC, the entry `q(X, Y)` : $X \mapsto X \wedge Y$ would not be in the answer table but rather in the old table. □

5.1 Local Change

One common reason for incremental modification to a program is optimizing compilation. Changes from optimization are special in the sense that usually the answers to the modified rule do not change. This means that the changes caused by the modification are local in that they cannot affect SCCs above the change. Thus, changes to the analysis are essentially restricted to computing the new call patterns that these rules generate. This allows us to obtain an algorithm for local change (related to bottom-up deletion) which is more efficient than arbitrary change.

The algorithm for local change is given in Figure 6. It takes as arguments the original calling patterns S and a modified rule R, which we assume has the same number as the rule it replaces.

The local change algorithm resembles bottom_up_delete in that only (part of) an SCC is reanalyzed. An important difference is that in local change it is guaranteed that we do not need to reanalyze the SCCs above the modified one. First all possibly affected calling patterns are collected in T. Because it is a local change we do not need to consider calling patterns outside the SCC of the program analysis graph. The arcs corresponding to the deleted rule are then deleted, and new arc events are added to process the new version of the changed rule.

Correctness of the local change algorithm essentially follows from correctness of the bottom-up deletion algorithm. Let A `:-` B and A `:-` B' be two rules. They are *local variants* with respect to the calls S and program P if for each call pattern in S the program $P \cup \{A$ `:-` $B\}$ has the same answer patterns as $P \cup \{A$ `:-` $B'\}$.

THEOREM 7. *Let P be a program analyzed for the initial call patterns S. Let R be a rule in P which in the analysis is called with call patterns S', and let R' be a local variant of R with respect to S' and $P \setminus \{R\}$. If the program P is reanalyzed with the routine* local_change(S,R') *the same answer and dependency arc tables will be obtained as when the rules $P \cup \{R'\} \setminus \{R\}$ are analyzed by the generic algorithm.*

The cost of performing the actual analysis incrementally can be no worse than performing the analysis all at once. Let $C_{local}(\bar{F}, P, R, R', S)$ be the worst-case number of calls to the parametric functions $\bar{F}$ when analyzing the program P with rule R changed to R' for all possible priority strategies with the local change algorithm after already analyzing the program P for call patterns S.

THEOREM 8. *Let P be a program analyzed for the initial call patterns S. Let R be a rule in P which in the analysis is called with call patterns S', and let R' be a local variant of R with respect to S' and $P \setminus \{R\}$. For any parametric functions $\bar{F}$,*

$$C_{noninc}(\bar{F}, P \cup \{R'\} \setminus \{R\}, S) \geq C_{local}(\bar{F}, P, R, R', S).$$

6. EXPERIMENTAL RESULTS

We have conducted a number of experiments using the PLAI generic abstract interpretation system in order to assess the practicality of the techniques proposed in the previous sections. PLAI can be seen as an efficient restricted instance of the generic algorithm of Section 2 specialized for the case of analyzing the whole program at once. As mentioned in Section 2, PLAI uses the concrete strategy of always performing *newcall* events first, processing nonrecursive rules before recursive rules, and finishing processing a rule before starting another. Prior to the invocation of the fixed-point algorithm a step is performed in which the set of predicates in the program is split into the SCCs based on the call graph of the program found using Tarjan's algorithm [Tarjan 1972]. This information, among other things, allows determining which predicates and which rules of a predicate are recursive. PLAI (and its incremental extensions) also incorporates some additional optimizations such as dealing directly with nonnormalized programs and filtering out noneligible rules using concrete unification (or constraint solving) when possible.

In one way, however, the original PLAI differed from the algorithm given in Section 2: in order to simplify the implementation, the original fixed-point algorithm did not keep track of dependencies at the level of literals, but rather, in a coarser way, at the level of rules. As a result, when an updated event takes place, the dependent arcs have to be recomputed from the head of the rule. However, all subsequent iterations do not affect those rules which are detected not to be recursive. Since keeping track of dependencies at the literal level allows the analysis to only recompute from the point where the changed answer can first affect the analysis information for the rule, it can avoid a significant amount of recomputation. This idea has been applied as an optimization technique (in a nonincremental analysis setting) in the *prefix* version [Englebert et al. 1993] of GAIA. The technique proved quite relevant in practice for GAIA, as it allowed avoiding recomputation of nonrecursive rules and starting reanalysis of a rule from the first literal possibly affected. This prefix version of GAIA can also be taken as a starting point for implementing an instance of the generic algorithm. We modified PLAI in order to keep track of dependencies at the literal level and to store such dependencies explicitly, i.e., in such a way that it computes the dependency arc table of the generic algorithm. PLAI, as most analysis engines not designed for incremental analysis, does compute the answer table, but loses the dependency information when analysis terminates. As a result, the modified version of PLAI, which uses the dependency arc table for guiding recomputations, constitutes precisely an instance of the generic algorithm. As we will see below, and unlike for the prefix version of GAIA, such modification

Table I. Summary of Benchmark Statistics

Benchmark	Mean vars/rule	Max vars/rule	# of preds	# of rules	% direct recursion	% mutual recursion
aiakl	4.58	9	7	12	57	0
ann	3.17	14	65	170	20	36
bid	2.20	7	19	50	31	0
boyer	2.36	7	26	133	3	23
browse	2.63	5	8	29	62	25
deriv	3.70	5	1	10	100	0
fib	2.00	6	1	3	100	0
grammar	2.13	6	6	15	0	0
hanoiapp	4.25	9	2	4	100	0
mmatrix	3.17	7	3	6	100	0
occur	3.12	6	4	8	75	0
peephole	3.15	7	26	134	7	46
progeom	3.59	9	9	18	66	0
qplan	3.18	16	46	148	32	28
qsortapp	3.29	7	3	7	100	0
query	0.19	6	4	52	0	0
rdtok	4.20	13	24	54	12	33
read	3.07	7	22	88	27	40
serialize	4.18	7	5	12	80	0
tak	7.00	10	1	2	100	0
warplan	2.47	7	29	101	31	17
witt	4.57	18	77	160	35	22
zebra	2.06	25	6	18	33	0

does not speed up PLAI much, since the original PLAI algorithm already avoids reanalysis of nonrecursive rules.

A relatively wide range of programs have been used as benchmarks. These benchmarks are the de-facto standard for logic program analysis. Some statistics on their size and complexity are given in Table I. Dead code, i.e., code which is unreachable from the original calling patterns, has been removed from the benchmarks. We give the average and maximum number of variables in each rule analyzed; total number of predicates and rules in the program; the percentage of directly and mutually recursive predicates.

All the analysis algorithms we experiment with have been implemented as extensions to the PLAI generic abstract interpretation system, which is an efficient, highly optimized, state-of-the-art analysis system and which is part of a working compiler.[4] We argue that this makes the comparisons meaningful, since the algorithms have been implemented using the same technology, with many data structures in common. They also share the domain-dependent functions, which are those of the *sharing+freeness* domain [Muthukumar and Hermenegildo 1991] in all the experiments. The whole system is implemented in Prolog and has been compiled using Ciao Prolog 0.9 [Bueno et al. 1997] with compilation to bytecode.[5] All of

[4]PLAI is currently integrated in CiaoPP [Hermenegildo et al. 1999a], the preprocessor of the Ciao Prolog System. The analysis information produced by PLAI is used for static debugging [Puebla et al. 2000], program specialization, and program parallelization.
[5]See `http://www.clip.dia.fi.upm.es/Software` for downloading instructions.

our experiments have been performed on a Pentium II at 400mH and 512MB RAM running RedHat linux 5.2. Execution times are given in milliseconds and memory usage in kilobytes. In order to have an accurate measure of memory usage, garbage collection is turned off during analysis. Memory usage has been computed as follows. We measure the size of the heap[6] (where dynamic terms are built) plus the size of dynamic code[7] (where asserted information is stored) both before and after analyzing each benchmark. The difference between those two values is taken as the memory used in the process. Other memory structures such as the environment stack are not considered, since their size has been measured to be irrelevant when compared to the heap space. Though the resulting memory usage is high, it is important to note that run-time garbage collection actually reduces memory consumption by a very large amount.

6.1 Efficiency of the Generic Algorithm

Our first experiment compares the original PLAI fixed-point approach, where dependencies are kept at the level of rules, versus the approach with dependency tracking at the level of literals. This also serves to show that our baseline, the PLAI instance of the generic algorithm, is competitive with other top-down analysis engines. The results are shown in Table II.

In this table, as in the other tables in this section, there are basically two kinds of columns. Those which present the cost of analysis both in terms of time and memory usage for a given algorithm, and those which compare the costs of two analysis algorithms. Columns of the first kind are labeled with the name of the algorithm, say *Alg*, while columns of the second kind are labeled by Alg_1 / Alg_2. The values in these columns are computed as the cost of analysis using Alg_1 divided by cost using Alg_2. In addition, below columns which compare algorithms we summarize the results for the different benchmarks using two different measures: the arithmetic mean of the values in the column above and a weighted mean which place more importance on those benchmarks with relatively larger analysis times / memory usage figures. We use as the weight for each program the actual analysis time or memory usage for it using Alg_2. We believe that the weighted mean is more informative than the arithmetic mean, as, for example, doubling the speed in which a large and complex program is analyzed is more important than doing this for small, simple programs.

As shown in Table II, it turns out that the additional cost of keeping track of more detailed dependencies is offset by savings on recomputation. The value of 0.97 for the weighted mean for both time and memory consumption indicates that the original algorithm is slightly faster and uses slightly less memory than the modified algorithm. In any case, the differences in analysis times are due to the difference in dependency tracking rather than the cost of supporting incrementality. Hence modifying the analysis algorithm in order to support incremental analysis has not significantly slowed nonincremental analysis nor posed additional memory requirements. All subsequent experiments use the modified PLAI as the basic analysis algorithm, both for the incremental and nonincremental case. In Tables III

[6]Obtained using `statistics(global_stack,[G,_])`.
[7]Obtained using `statistics(program,[P,_])`.

Table II. Cost of Literal Based Dependency Tracking

Benchmark	Modified PLAI		Original PLAI		Original / Modified	
	Time	Memory	Time	Memory	Time	Memory
aiakl	867	3304	1003	3712	1.16	1.12
ann	1655	6324	1743	6347	1.05	1.00
bid	178	668	171	602	0.96	0.90
boyer	573	2190	761	2659	1.33	1.21
browse	87	294	85	270	0.98	0.92
deriv	105	408	177	683	1.69	1.67
fib	2	22	4	21	2.00	0.95
grammar	25	98	26	96	1.04	0.98
hanoiapp	113	475	115	432	1.02	0.91
mmatrix	57	244	51	222	0.89	0.91
occur	68	258	69	262	1.01	1.02
peephole	1453	5639	1611	5843	1.11	1.04
progeom	39	144	37	137	0.95	0.95
qplan	433	1285	413	1218	0.95	0.95
qsortapp	64	273	61	256	0.95	0.94
query	19	46	21	45	1.11	0.98
rdtok	137	399	143	379	1.04	0.95
read	10695	38396	9475	35086	0.89	0.91
serialize	147	621	115	468	0.78	0.75
tak	11	79	14	73	1.27	0.92
warplan	823	2970	979	3452	1.19	1.16
witt	491	1651	509	1662	1.04	1.01
zebra	591	2172	579	2144	0.98	0.99
				Arithmetic mean	1.10	1.01
				Weighted mean	0.97	0.97

and VI we denote by *Orig* the cost of analyzing the whole program at once using this modified PLAI algorithm.

6.2 Incremental Addition

The next experiment compares the relative performance of incremental and nonincremental analysis in the context of addition. To do so for each benchmark we measured the cost of analyzing the program adding one rule at a time, rather than analyzing the whole program at once. That is, the analysis was first run for the first rule only. Then the next rule was added and the resulting program (re-)analyzed. This process was repeated until all rules had been added. Table III shows the cost of this process using the incremental approach of Section 3, and using a nonincremental approach, where analysis starts from scratch whenever a new rule is added. For the nonincremental case the same implementation was used but the tables were erased between analyzes. This factors out any differences in fixed-point algorithms. Since for the nonincremental approach the analysis tables are erased after each analysis, we can reuse memory space. Thus, we take as memory usage the maximum amount of memory used when analyzing rules 1 to i of the benchmark for $i \in \{1 \dots n\}$, where n is the number of rules in that benchmark. Usually, the maximum is reached when all the rules are analyzed, i.e., i is n. In the next column we compare the relative costs of nonincremental and incremental analysis. Finally we compare the cost of analyzing the entire program one rule at a time using the

Table III. Incremental vs. Non-incremental Addition

Benchmarks	Incremental		Non-incremental		Non / Inc		Inc / Orig	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
aiakl	973	1399	1750	3361	1.80	2.40	1.12	0.42
ann	5136	6470	23688	7333	4.61	1.13	3.10	1.02
bid	861	260	3617	831	4.20	3.20	4.84	0.39
boyer	3632	1054	59867	2554	16.48	2.42	6.34	0.48
browse	395	145	2881	433	7.29	2.99	4.54	0.49
deriv	681	1482	369	442	0.54	0.30	6.49	3.63
fib	9	29	13	34	1.44	1.17	4.50	1.32
grammar	84	40	223	136	2.65	3.40	3.36	0.41
hanoiapp	189	392	358	494	1.89	1.26	1.67	0.83
mmatrix	153	191	216	266	1.41	1.39	2.68	0.78
occur	74	263	151	285	2.04	1.08	1.09	1.02
peephole	12930	1752	136145	6118	10.53	3.49	8.90	0.31
progeom	69	159	168	206	2.43	1.30	1.77	1.10
qplan	7970	1147	35492	1898	4.45	1.65	18.41	0.89
qsortapp	127	175	263	298	2.07	1.70	1.98	0.64
query	365	31	839	139	2.30	4.48	19.21	0.67
rdtok	609	227	4112	725	6.75	3.19	4.45	0.57
read	42195	25673	246754	51361	5.85	2.00	3.95	0.67
serialize	322	793	719	667	2.23	0.84	2.19	1.28
tak	22	84	25	94	1.14	1.12	2.00	1.06
warplan	9147	2151	26429	3335	2.89	1.55	11.11	0.72
witt	3808	1543	13536	2523	3.55	1.64	7.76	0.93
zebra	1685	1426	5836	2240	3.46	1.57	2.85	0.66
				Arithmetic mean	4.00	1.97	5.40	0.88
				Weighted mean	6.16	1.83	4.91	0.69

incremental approach (column 1) with analyzing the entire program once using the nonincremental approach (column 2 of Table II).

The results are quite encouraging: using an incremental analysis was 6.16 times faster than using nonincremental analysis on average, while the cost of incrementally analyzing a program rule by rule as opposed to all at once was only 4.91 times worse on average (5.40 arithmetic mean). Though not explicitly shown in the table, using a nonincremental approach is over 30 times worse than analyzing the program at once.

An important observation is that incremental analysis performed when adding rules one by one (Inc), although slower than when performing the analysis nonincrementally all at once (Orig), on average requires significantly less memory (only 69% as much). This indicates that those programs which are too large to be analyzed at once (because the system runs out of memory) could be tackled by splitting them in several parts and performing incremental addition, albeit at the price of increasing analysis time somewhat.

6.3 Incremental Deletion

In order to test the relative performance of incremental and nonincremental analysis in the context of deletion, we timed the analysis of the same benchmarks where each rule was deleted one by one. Starting from an already analyzed program, the last rule was deleted and the resulting program (re-)analyzed. This process

Table IV. Top-Down Incremental Deletion

Benchmark	Top-down incr. Time	Top-down incr. Mem	Non-incr. Time	Non-incr. Mem	Non / Top-down Time	Non / Top-down Mem
aiakl	650	1407	869	1579	1.34	1.12
ann	4955	28	21542	1009	4.35	36.04
bid	1124	241	3425	824	3.05	3.42
boyer	17478	1400	59731	2544	3.42	1.82
browse	529	238	2843	428	5.37	1.80
deriv	230	336	264	360	1.15	1.07
fib	1	4	4	8	4.00	2.00
grammar	80	39	194	125	2.42	3.21
hanoiapp	149	255	144	242	0.97	0.95
mmatrix	139	178	142	193	1.02	1.08
occur	9	10	27	31	3.00	3.10
peephole	30761	2454	134902	6073	4.39	2.47
progeom	19	4	128	202	6.74	50.50
qplan	27380	629	34954	1893	1.28	3.01
qsortapp	83	100	173	234	2.08	2.34
query	399	37	818	137	2.05	3.70
rdtok	1300	340	3927	720	3.02	2.12
read	201059	38748	234405	51356	1.17	1.33
serialize	114	112	564	664	4.95	5.93
tak	2	18	7	23	3.50	1.28
warplan	18737	2032	25977	3330	1.39	1.64
witt	2432	152	12974	2513	5.33	16.53
zebra	2040	2021	5233	2224	2.57	1.10
			Arithmetic mean		2.98	6.42
			Weighted mean		1.75	1.51

was repeated until no rules were left. This experiment has been performed using a nonincremental approach and using the top-down deletion algorithm of Section 4.2 and the bottom-up deletion algorithm of Section 4.3. Table IV presents the results for top-down deletion and the nonincremental approach, while Table V contains the figures for the bottom-up algorithm together with a comparison of the two incremental deletion algorithms.

The results are also very encouraging. The improvement of analyzing rule by rule in an incremental fashion gave an average speedup with respect to the nonincremental algorithm of 1.75 for the top-down deletion algorithm and 7.30 for the bottom-up deletion algorithm. The results favor the bottom-up algorithm in this experiment, as shown in the comparison column in Table V, where bottom-up deletion is 4.16 times faster than top-down deletion.

Examining memory usage, we see that both the top-down and bottom-up deletion algorithms require less memory than the nonincremental approach (by a factor of 1.51 and 1.40 respectively). Between the two incremental deletion algorithms, top-down deletion, though slower, requires somewhat less memory than the bottom-up algorithm. In our experiments, top-down requires (only) 92% of the amount of memory used by the bottom-up algorithm. As for the incremental addition experiment, we take as memory usage for the nonincremental approach the maximum amount of memory when analyzing rules 1 to i of the benchmark with $i \in \{1 \ldots n\}$

Table V. Bottom-Up Incremental Deletion

Benchmark	Bottom-up incr.		Non / Bottom-up		Top-down / Bottom-up	
	Time	Mem	Time	Mem	Time	Mem
aiakl	514	1110	1.69	1.42	1.26	1.27
ann	979	469	22.00	2.15	5.06	0.06
bid	331	138	10.35	5.97	3.40	1.75
boyer	4535	1372	13.17	1.85	3.85	1.02
browse	196	155	14.51	2.76	2.70	1.54
deriv	230	347	1.15	1.04	1.00	0.97
fib	1	4	4.00	2.00	1.00	1.00
grammar	47	35	4.13	3.57	1.70	1.11
hanoiapp	147	248	0.98	0.98	1.01	1.03
mmatrix	39	57	3.64	3.39	3.56	3.12
occur	17	18	1.59	1.72	0.53	0.56
peephole	8754	2562	15.41	2.37	3.51	0.96
progeom	14	5	9.14	40.40	1.36	0.80
qplan	623	392	56.11	4.83	43.95	1.60
qsortapp	57	64	3.04	3.66	1.46	1.56
query	110	13	7.44	10.54	3.63	2.85
rdtok	301	224	13.05	3.21	4.32	1.52
read	55517	45162	4.22	1.14	3.62	0.86
serialize	84	122	6.71	5.44	1.36	0.92
tak	6	18	1.17	1.28	0.33	1.00
warplan	1135	1437	22.89	2.32	16.51	1.41
witt	528	446	24.57	5.63	4.61	0.34
zebra	259	707	20.20	3.15	7.88	2.86
	Arithmetic mean		11.35	4.82	5.11	1.31
	Weighted mean		7.30	1.40	4.16	0.92

where n is the number of rules in that benchmark. Note that we start with the program already analyzed, and thus analysis of the complete program (i.e., with rules 1–n) is not considered.

6.4 Local Change

Although we have implemented it, we do not report explicitly on the performance of arbitrary change because of the difficulty in modeling in a meaningful way the types of changes that are likely to occur in the circumstances in which this type of change occurs (as, for example, during an interactive program development session). We have studied, however, the case of local change in a context in which it occurs in a way which is more amenable to systematic study: within the &-Prolog compiler.[8] The &-Prolog system is capable of executing, in parallel, goals which are independent [Conery and Kibler 1981; Hermenegildo and Rossi 1995]. The &-Prolog compiler includes an automatic parallelizer which replaces some conjunctions of literals in the body of rules which are possibly independent with parallel expressions [Bueno et al. 1999b]. The conditions for independence can often be proved at compile-time by the use of global analysis [Bueno et al. 1999a]. However, there are cases in which run-time tests have to be introduced in the program in order to ensure independence. If the run-time tests fail, the goals are executed sequentially.

[8] Currently also integrated into the Ciao System preprocessor.

Table VI. Local Change

Benchm	CGEs	Incremental		Non-incr.		Non / Inc		Non/(Orig+Inc)	
		Time	Mem	Time	Mem	Time	Mem	Time	Mem
aiakl	2	97	347	823	3404	8.48	9.81	0.85	1.02
ann	12	739	2605	2035	8193	2.75	3.15	0.85	1.24
bid	6	111	363	257	948	2.32	2.61	0.89	1.27
boyer	2	181	710	975	3876	5.39	5.46	1.29	1.68
browse	4	93	331	179	697	1.92	2.11	0.99	1.96
deriv	4	211	821	243	980	1.15	1.19	0.77	1.16
hanoiapp	1	42	187	135	567	3.21	3.03	0.87	1.12
mmatrix	2	81	337	120	501	1.48	1.49	0.87	1.44
occur	2	81	328	131	558	1.62	1.70	0.88	1.63
peephole	2	267	938	1579	6286	5.91	6.70	0.92	1.10
progeom	1	11	30	47	156	4.27	5.20	0.94	1.04
qplan	2	51	203	415	1430	8.14	7.04	0.86	1.05
query	2	13	62	31	79	2.38	1.27	0.97	1.20
read	1	13	35	10979	38425	844.54	1097.86	1.03	1.00
serialize	1	10	32	161	666	16.10	20.81	1.03	1.06
warplan	8	721	2496	4693	17359	6.51	6.95	3.04	5.14
zebra	1	379	1325	627	2382	1.65	1.80	0.65	1.05
				Arithmetic mean		53.99	69.31	1.04	1.48
				Weighted mean		7.56	7.76	1.12	1.28

Though the system does not introduce redundant tests, the parallelized programs can be further optimized if a *multivariant* analysis (of which both PLAI and its incremental versions are capable) is performed on such programs. The additional optimization (e.g., recursive invariant extraction) is possible because more precise information can be propagated beyond the program points in which tests have been introduced [Puebla and Hermenegildo 1995; 1999]. The interesting point in our context is that in this process the program is analyzed once, parallelized (introducing conditionals), and then reanalyzed before performing the multiple specialization step, and that between these two analysis phases changes are made to the program that correspond to our local change scenario.

We have applied the process to our set of benchmarks, and the results are shown in Table VI. Benchmarks for which no run-time tests were added in the parallelized program during the first step of the process have been omitted, since the specialization step is not performed for them, and no reanalysis is needed in that case. The calling patterns S used for these tests were just program entry points with the most general calling pattern, i.e., no information about constraints in calls. This represents the likely situation where the user provides little information to the analyzer. The case is also interesting in our context because it produces more run-time tests and thus more specializations and reanalyzes, which allows us to study more benchmarks (note that if very precise information is given by the user then many benchmarks are parallelized without any run-time tests, and then no specialization—and thus no reanalysis—occurs).

Table VI presents the results for this experiment. The first column shows the number of conditional graph expressions (CGEs), which are parallel expressions "guarded" by run-time tests. It is an indicator of the number of changes introduced during parallelization. The following columns present the cost of performing the second analysis using the incremental algorithm for local change and by simply reanalyzing from scratch. The next column compares the incremental and nonincremental approaches, giving the ratio of nonincremental reanalysis costs to incremental reanalysis costs. The final column compares the cost of analyzing the parallelized program either by a nonincremental analysis of the parallelized program, or by analyzing the original program and then after parallelization, incrementally reanalyzing.

The results of incremental analysis of local change are even more encouraging than the previous ones. The improvement in analysis time and memory usage is very high overall (7.56 and 7.76 respectively). For large programs with little modifications, such as `read`, the incremental approach is three orders of magnitude better than reanalyzing from scratch. In addition, for a number of benchmarks, in particular `boyer` and `warplan`, it is actually less costly both in terms of time and memory to analyze first the original program and then update the analysis information after the modifications introduced by the parallelizer than analyzing the specialized program alone. This is because parallelization has introduced parallel expressions, and thus complicated even further, rules for highly recursive and complex predicates for which analysis requires several iterations until a fixed point is reached. Because the incremental approach already starts with an answer for such predicates which is already a fixed point (which has been computed with a simpler but equivalent version of the predicates), the algorithm for local change requires much less work to complete the analysis than analyzing the entire specialized program from scratch. As a result, analysis plus local change is faster by a factor of 1.12. The results for memory usage are even better (1.28). This is because we do not have to add the memory used in both analyses, as we can reuse the heap space. All the information required by the second (incremental) analysis is asserted. Thus, memory usage is the maximum heap space usage for the two analysis phases plus the sum of the program space used by each phase.

7. RELATED WORK

Surprisingly, there has been little research into incremental analysis for (constraint) logic programs apart from previous work by the authors Hermenegildo et al. [1995; Puebla and Hermenegildo [1996]. Also, Krall and Berger [1995a; Krall and Berger [1995b] define a compiled analysis approach for logic programs that use the Vienna abstract machine model. They briefly mention that there approach can be used for incremental analysis, though the only kind of incremental change they consider is incremental addition. Several researchers have looked at compositional analysis of modules in (constraint) logic programs [Codish et al. 1993; Bossi et al. 1994], but this does not consider incremental analysis at the level of rules. Also, preliminary studies of full modular analysis of (constraint) logic programs [Puebla and Hermenegildo 2000] indicate that the techniques presented in this article may be very valuable in that context.

There has, however, been considerable research into incremental analysis for other programming paradigms (see for example the bibliography of Ramalingam and Reps [1993]). The need for incremental analysis techniques was realized very early in research on data flow analysis of traditional languages. The first algorithm was proposed by Rosen [1981], and since then a bewildering array of algorithms for the incremental data flow analysis of traditional languages has been suggested.

Generally, these algorithms can be separated into two approaches which reflect the underlying mechanism used to solve the data flow equations. *Elimination-based* methods use variable elimination, much like in Gauss-Jordan elimination, to solve the equations, while *iterative* methods find the least fixed point[9] essentially by computing the Kleene sequence. Incremental analysis algorithms based on each of these approaches have been suggested. Those based on elimination include Burke [1990], Carroll and Ryder [1988], and Ryder [1988]; those based on iteration methods include Cooper and Kennedy [1984] and Pollock and Soffa [1989]; while a hybrid approach is described in Marlowe and Ryder [1990].

Our algorithms are formulated in terms of the standard top-down abstract interpretation framework for (constraint) logic programs. Like iteration-based data flow analysis algorithms this framework also computes the Kleene sequence in order to find the least fixed point of the recursive equations. Thus our algorithms are most closely related to those using iteration. Early incremental approaches such as Cooper and Kennedy [1984] were based on *restarting iteration*. That is, the fixed point of the new program's data flow equations is found by starting iteration from the fixed point of the old program's data flow equations. This is always safe, but may lead to unnecessary imprecision if the old fixed point is not below the least fixed point of the new equations [Ryder et al. 1988]. *Reinitialization approaches* such as Pollock and Soffa [1989] improve the accuracy of this technique by reinitializing nodes in the data flow graph to bottom if they are potentially affected by the program change. Thus they are as precise as if the new equations had been analyzed from scratch.

Our first algorithm, that for incremental addition, works by restarting iteration from the old fixed point. However, because the contribution of each rule is lubbed together, if rules are added to the program, the least fixed point will always increase. Thus restarting iteration is guaranteed to be as precise as starting from scratch in the case rules are incrementally added. Of course, this "monotonicity" of rule addition does not apply to traditional programming languages. The top-down deletion algorithm can be viewed as a variant of the reinitialization approach in which dependency arcs are used to keep track of which information will be affected by deletion of a rule. The bottom-up deletion and local change algorithms are to the best of our knowledge novel.

Despite the similarities between our work and previous research into incremental analysis there are a number of important differences. First, other research has concentrated on traditional programming languages. To our knowledge this is the first article to identify the different types of incremental change which are useful in (constraint) logic program analysis and to give practical algorithms which handle these types of incremental change. Second, our research is formalized in terms of the generic data flow analysis technique, abstract interpretation. This means that our algorithms are usable with any abstract domain approximating constraints.

[9] For consistency with the rest of this article, we have reversed the usual data flow terminology so as to accord with abstract interpretation terminology in which the description lattice is ordered by generality, with the most general element at the top of the lattice.

This contrasts to earlier work in which the algorithms applied only to a single type of analysis or at best to a quite restricted class of analyses. Another contribution of the paper is a simple formalization of the nonincremental fixed-point algorithms used in generic analysis engines. We formalize the analysis as a graph traversal and couch the algorithm in terms of priority queues. Different priorities correspond to different traversals of the program analysis graph. This simple formalization greatly facilitates the description of our incremental algorithms and their proofs of correctness. Finally, we have given a detailed empirical evaluation of our algorithms.

8. CONCLUSIONS

We have described extensions to the fixed-point algorithms used in current top-down generic analysis engines for constraint logic programming languages in order to support incremental analysis. We have classified the possible changes to a program into addition, deletion, arbitrary change, and local change, and proposed, for each one of these, algorithms for identifying the parts of the analysis that must be recomputed and for performing the actual recomputation. We have also discussed the potential benefits and drawbacks of these algorithms. Finally, we have presented some experimental results obtained with an implementation of the algorithms in the PLAI generic abstract interpretation framework.

Our empirical evaluation shows that the incremental analysis algorithms have little overhead compared with the standard nonincremental analysis algorithms but offer considerable benefits both in terms of analysis times and memory usage over these algorithms if the program must be reanalyzed after modification. Thus optimizing compilers for constraint logic programming which rely on a repeated source-to-source transformation and reanalysis cycle should make use of incremental algorithms, such as those presented in this article. Indeed we have successfully employed them [Puebla and Hermenegildo 1999] in an automatic parallelizer for logic programs [Bueno et al. 1999b] and an optimizing compiler for CLP($\mathcal{R}$) [Kelly et al. 1998a]. Modifying PLAI to support our algorithms for incremental analysis was relatively straightforward. The only real difficulty was the addition of dependency arc tracking, which also enabled handling updated events. We believe it would also be straightforward to modify other analysis engines for (constraint) logic programs in a similar way.

We believe that our work contributes to opening the door to practical, everyday use of global analysis in the compilation of (constraint) logic programs, even in the interactive environment which is often preferred by the users of such systems. Furthermore, while current analyzers can deal correctly with dynamic program modification primitives, this implies having to give up on many optimizations not only for the dynamic predicates themselves but also for any code called from such predicates. The ability to update global information incrementally (and thus with reduced overhead) down to the level of single rule additions and deletions makes it possible to deal with these primitives in a much more accurate way.

Throughout the article we have concentrated on the analysis of pure CLP programs. There is nothing precluding the use of standard analysis techniques for handling logical and nonlogical built-ins within the incremental framework. Indeed the implementation [Bueno et al. 1996] actually handles (almost) full ISO-Prolog, and many such built-ins occur in the benchmarks. For simplicity we have also ignored the abstract operation of widening [Cousot and Cousot 1979] in our analysis framework. However, it is straightforward to modify the algorithms to include widening of call and answer patterns.

Although we have focussed on top-down goal-dependent analysis of constraint logic programs, our results are also applicable to goal-independent analysis. Again the key is to keep track of dependencies between the head of a rule and the literals in the rule and answers to each atom. It is straightforward to modify the generic algorithm of Figure 2 and our incremental algorithms to define an incremental top-down goal-independent analysis (by restricting the possible calling patterns in the answer table). Similar ideas can also be used to give incremental algorithms for bottom-up goal-independent analysis, which are far simpler, since information flow is only in one direction (upward).

ACKNOWLEDGEMENTS

The authors would like to thank María García de la Banda, Harald Søndergaard, and the anonymous referees for useful comments.

REFERENCES

Armstrong, T., Marriott, K., Schachte, P., and g H. Søndergaard. 1994. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In *Static Analysis Symposium, SAS'94*. Number 864 in LNCS. Springer-Verlag, Namur, Belgium, 266–280.

Bossi, A., Gabbrieli, M., Levi, G., and Meo, M. 1994. A compositional semantics for logic programs. *Theoretical Computer Science 122,* 1,2, 3–47.

Bruynooghe, M. 1991. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming 10*, 91–124.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and Puebla, G. 1997. The Ciao Prolog system. reference manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM). August.

Bueno, F., Cabeza, D., Hermenegildo, M., and Puebla, G. 1996. Global analysis of standard Prolog programs. In *European Symposium on Programming.* Number 1058 in LNCS. Springer-Verlag, Sweden, 108–124.

Bueno, F., García de la Banda, M. G., and Hermenegildo, M. 1999a. Effectiveness of abstract interpretation in automatic parallelization: A case study in logic programming. *ACM Transactions on Programming Languages and Systems 21,* 2 (March), 189–238.

Bueno, F., García de la Banda, M. G., Hermenegildo, M., and Muthukumar, K. 1999b. Automatic compile-time parallelization of logic programs for restricted, goal-level, independent and-parallelism. *Journal of Logic Programming 38,* 2, 165–218.

Bueno, F., García de la Banda, M., and Hermenegildo, M. 1994. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *International Symposium on Logic Programming.* MIT Press, Cambridge, Massachusetts, 320–336.

Burke, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems 12,* 3, 341–395.

Carroll, M. and Ryder, B. 1988. Incremental data flow analysis via dominator and attribute updates. In *15th ACM Symposium on Principles of Programming Languages (POPL).* ACM Press, San Diego, 274–284.

Charlier, B. L. and Van Hentenryck, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems 16,* 1, 35–101.

Charlier, B. L., Degimbe, O., Michael, L., and Van Hentenryck, P. 1993. Optimization techniques for general purpose fixpoint algorithms: Practical efficiency for the abstract interpretation of Prolog. In *Workshop on Static Analysis.* Springer-Verlag, 15–26.

Charlier, B. L., Rossi, S., and Van Hentenryck, P. 1994. An abstract interpretation framework which accurately handles Prolog search–rule and the cut. In *International Symposium on Logic Programming.* MIT Press, 157–171.

Codish, M., Debray, S., and Giacobazzi, R. 1993. Compositional analysis of modular logic programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93.* ACM, Charleston, South Carolina, 451–464.

Conery, J. S. and Kibler, D. F. 1981. Parallel interpretation of logic programs. In *Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture.* (1981). ACM Press, 163–170.

Cooper, K. and Kennedy, K. 1984. Efficient computation of flow insensitive interprocedural summary information. In *ACM SIGPLAN Symposium on Compiler Construction (SIGPLAN Notices* **19***(6)).* ACM Press, 247–258.

Cousot, P. and Cousot, R. 1977. 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.* ACM Press, Los Angeles, 238–252.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *Sixth ACM Symposium on Principles of Programming Languages.* ACM Press, San Antonio, Texas, 269–282.

Debray, S., Ed. 1992. *Journal of Logic Programming, Special Issue: Abstract Interpretation.* Vol. 13(1–2). North-Holland, Amsterdam.

Debray, S. K. 1989. Static inference of modes and data dependencies in logic programs. *ACM Transactions on Programming Languages and Systems 11,* 3, 418–450.

Englebert, V., Le Charlier, B., Roland, D., and Van Hentenryck, P. 1993. Generic abstract interpretation algorithms for Prolog: Two optimization techniques and their experimental evaluation. *Software Practice and Experience 23,* 4 (Apr.), 419–459.

García de la Banda, M., Marriott, K., Søndergaard, H., and Stuckey, P. 1998. Differential methods in logic program analysis. *Journal of Logic Programming 35,* 1, 1–37.

Hermenegildo, M. and Greene, K. 1991. The &-Prolog system: Exploiting independent and-parallelism. *New Generation Computing 9,* 3,4, 233–257.

Hermenegildo, M. and Rossi, F. 1995. Strict and non-strict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *Journal of Logic Programming 22,* 1, 1–45.

Hermenegildo, M., Bueno, F., Puebla, G., and López-García, P. 1999a. Program analysis, debugging and optimization using the Ciao system preprocessor. In *1999 International Conference on Logic Programming.* MIT Press, Cambridge, MA, 52–66.

Hermenegildo, M., Puebla, G., and Bueno, F. 1999b. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: a 25–Year Perspective*, K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, Eds. Springer-Verlag, 161–192.

Hermenegildo, M., Puebla, G., Marriott, K., and Stuckey, P. 1995. Incremental analysis of logic programs. In *International Conference on Logic Programming.* MIT Press, 797–811.

Hermenegildo, M., Warren, R., and Debray, S. K. 1992. Global flow analysis as a practical compilation tool. *Journal of Logic Programming 13,* 4 (August), 349–367.

Kelly, A., Macdonald, A., Marriott, K., Søndergaard, H., and Stuckey, P. 1998a. Optimizing compilation for CLP($\mathcal{R}$). *ACM Transactions on Programming Languages and Systems 20,* 6, 1223–1250.

Kelly, A., Marriott, K., Søndergaard, H., and Stuckey, P. 1998b. A practical object-oriented analysis engine for CLP. *Software: Practice and Experience 28,* 2, 199–224.

Krall, A. and Berger, T. 1995a. Incremental global compilation of Prolog with the Vienna abstract machine. In *International Conference on Logic Programming.* MIT Press, Tokyo, 333–347.

Krall, A. and Berger, T. 1995b. The $\mathrm{VAM_{AI}}$—an abstract machine for incremental global dataflow analysis of Prolog. In *ICLP'95 Post-Conference Workshop on Abstract Interpretation* *of Logic Languages*, M. G. de la Banda, G. Janssens, and P. Stuckey, Eds. Science University of Tokyo, Tokyo, 80–91.

Marlowe, T. and Ryder, B. 1990. An efficient hybrid algorithm for incremental data flow analysis. In *17th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, San Francisco, 184–196.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints: an Introduction*. MIT Press.

Marriott, K., Søndergaard, H., and Jones, N. 1994. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems 16,* 3, 607–648.

Muthukumar, K. and Hermenegildo, M. 1990. 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.

Muthukumar, K. and Hermenegildo, M. 1991. Combined determination of sharing and freeness of program variables through abstract interpretation. In *1991 International Conference on Logic Programming*. MIT Press, Paris, 49–63.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming 13,* 2/3 (July), 315–347. Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

Pollock, L. and Soffa, M. 1989. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering 15,* 12, 1537–1549.

Puebla, G. and Hermenegildo, M. 1995. Implementation of multiple specialization in logic programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*. ACM Press, La Jolla, California, 77–87.

Puebla, G. and Hermenegildo, M. 1996. Optimized algorithms for the incremental analysis of logic programs. In *International Static Analysis Symposium*. Number 1145 in LNCS. Springer-Verlag, 270–284.

Puebla, G. and Hermenegildo, M. 1999. Abstract multiple specialization and its application to program parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs 41,* 2&3 (November), 279–316.

Puebla, G. and Hermenegildo, M. 2000. Some issues in analysis and specialization of modular Ciao-Prolog programs. *Electronic Notes in Theoretical Computer Science 30,* 2 (March). Special Issue on Optimization and Implementation of Declarative Programming Languages.

Puebla, G., Bueno, F., and Hermenegildo, M. 2000. A generic preprocessor for program validation and debugging. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. Hermenegildo, and J. Maluszynski, Eds. Springer-Verlag. To appear.

Ramalingam, G. and Reps, T. 1993. A categorized bibliography on incremental computation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*. ACM, Charleston, South Carolina.

Rosen, B. 1981. Linear cost is sometimes quadratic. In *Eighth ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, Williamsburg, Virginia, 117–124.

Ryder, B. 1988. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems 10,* 1, 1–50.

Ryder, B., Marlowe, T., and Paull, M. 1988. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming 11,* 1, 1–15.

Santos-Costa, V., Warren, D., and Yang, R. 1991. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In *1991 International Conference on Logic Programming*. MIT Press, Paris, 443–456.

Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1,* 140–160.

Van Roy, P. and Despain, A. 1992. High-performace logic programming with the aquarius Prolog compiler. *IEEE Computer Magazine 25,* 1 (January), 54–68.

Winsborough, W. 1992. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming 13,* 2 and 3 (July), 259–290.

Received July 1998; revised July 1999; accepted November 1999