Optimized Algorithms for Incremental Analysis of Logic Programs

Germán Puebla and *Manuel Hermenegildo*

{german,herme}@fi.upm.es
Department of Artificial Intelligence
Technical University of Madrid (UPM)

Abstract. Global analysis of logic programs can be performed effectively by the use of one of several existing efficient algorithms. However, the traditional global analysis scheme in which all the program code is known in advance and no previous analysis information is available is unsatisfactory in many situations. Incremental analysis of logic programs has been shown to be feasible and much more efficient in certain contexts than traditional (non-incremental) global analysis. However, incremental analysis poses additional requirements on the fixpoint algorithm used. In this work we identify these requirements, present an important class of strategies meeting the requirements, present sufficient a priori conditions for such strategies, and propose, implement, and evaluate experimentally a novel algorithm for incremental analysis based on these ideas. The experimental results show that the proposed algorithm performs very efficiently in the incremental case while being comparable to (and, in some cases, considerably better than) other state-of-the-art analysis algorithms even for the non-incremental case. We argue that our discussions, results, and experiments also shed light on some of the many tradeoffs involved in the design of algorithms for logic program analysis.

1 Introduction

Global program analysis is becoming a practical tool in logic program compilation and used to perform provably correct program optimizations [HWD92, RD92, MH92, SCWY91, BGH94, Bru91, Deb92]. Several generic analysis engines such as, for example, PLAI [MH92, MH90] and GAIA [CH94] facilitate construction of top-down analyzers. Different description domains and related functions render different analyzers which provide different types of information and degrees of accuracy. The core of such a generic engine is a specialized algorithm for efficient fixpoint computation, a subject that has received considerable attention [CC77, MH90, MH92, LDMH93, VWL94, Jor94].

Incremental analysis of logic programs has been shown to be feasible and much more efficient in certain contexts than traditional (non-incremental) global analysis [HMPS95, KB95]. In particular, [HMPS95] discussed the different types of changes that had to be dealt with in an incremental setting, provided overall solutions for dealing with such changes (in terms of which parts of the analysis graph need to be updated and recomputed), and proposed a basic set of solutions that showed the feasibility of the approach. It was also observed that incremental analysis poses additional requirements on the fixpoint algorithm used since some assumptions on the program that traditional algorithms make are no longer valid. The purpose of this work is to directly address this issue by identifying more concretely such requirements and to improve the performance of the fixpoint algorithm while meeting the requirements. We also aim to define, implement, and evaluate experimentally a novel algorithm for incremental analysis and compare it to some previously proposed algorithms for incremental as well as non-incremental analysis.

To the best of our knowledge, this is the first work dealing with the design and experimentation of fixpoint algorithms specially tailored for incremental analysis of logic programs. Additionally, our results imply performance improvements even in a non-incremental setting. Thus, we believe our discussions, results, and experiments may also clarify some of the many tradeoffs involved in the design of algorithms for logic program analysis in general.

2 Incremental Analysis Requirements

The aim of the kind of (goal oriented) program analysis performed by the analysis engines mentioned in the previous section is, for a particular description domain, to take a program and a set of initial *calling patterns* (descriptions of the possible calling modes into the program) 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. The program points considered are entry to the rule, the point between each two literals, and return from the call. In essence, the analyzer must produce a program analysis graph. For example, for a standard operational semantics based on AND-OR trees, the analysis graph can be viewed as a finite representation (through a "widening") of the set of AND-OR trees explored by the concrete execution [Bru91]. 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 provides the most precise information possible. This analysis graph corresponds to the least fixpoint of the abstract semantic equations.

The aim of *incremental* global analysis is, given a program, its least analysis graph, and a series of changes to the program, to obtain the new least analysis graph as efficiently as possible. A simple but inefficient way of computing the new least analysis graph is just to discard the previous analysis graph and start analysis from scratch on the new program. However, much of the information in the previous analysis graph may still be valid, and incremental analysis should be able to reuse such information, instead of recomputing it from scratch.

Unfortunately, traditional fixpoint algorithms for abstract interpretation of logic programs cannot be used directly (at least in general) in the context of incremental analysis for reasons of accuracy, efficiency, and even correctness. This is because such algorithms assume that once a local fixpoint has been reached for a calling pattern, i.e., an *answer pattern* for this calling pattern has been computed, this information will not change and can be used safely thereafter. This assumption is no longer valid in the incremental case, since an answer pattern may become inaccurate if some clauses are eliminated from the program (*incremental deletion*) or even incorrect if more clauses are added to the program (*incremental addition*). When performing *arbitrary change* on the program (i.e., when both additions and deletions are performed), the old answer pattern can be incorrect, inaccurate, or both.

We now discuss two requirements that incremental analysis poses on the fixpoint algorithm.

Detailed Dependency Information: Most practical fixpoint algorithms try to make iterations as local as possible by using some kind of dependency information. Thanks to this information it is possible to revisit only a reduced set of nodes of the graph when an answer pattern changes during analysis. Additionally, dependency information can also be used to detect earlier that a fixpoint has been reached. The more accurate such dependency information is, the more localized (and, thus, less costly) the fixpoint iterations can be.

In the context of incremental analysis, in addition to localizing the fixpoint and detecting termination earlier, dependencies are useful for a third reason: they help locate the parts of the analysis graph that may be affected by program changes and which thus need to be recomputed as required by such changes. Obviously, if more detailed dependency information is kept track of, a smaller part of the analysis graph will have to be recomputed after modifying the program.

Propagation of Incrementally Updated Answer Patterns: Incremental deletion and local and arbitrary change ([HMPS95]) do not pose extra requirements on the analysis algorithm, provided that detailed dependency information is available, since such changes only require the analysis algorithms to deal with new calling patterns. However, in incremental addition, i.e., new clauses are added to a program already analyzed, the new clauses may also generate unexpected changes to previously computed answer patterns, i.e., they may update any answer pattern in the analysis graph. Once a global fixpoint has been reached, there is usually no way to propagate this updated information to the places in the analysis graph that may be affected using traditional analysis algorithms. If an algorithm is to deal efficiently with incremental addition it needs to be able to deal incrementally with the update of any answer pattern.

3 A Generic Analysis Algorithm

We now present a generic analysis algorithm taken from [HMPS95], that we will use as the basis for describing the optimized analysis algorithms proposed in the paper. We will refer to this algorithm simply as "the generic algorithm."

We first introduce some notation. CP, possibly subscripted, stands for a calling pattern (in the abstract domain). AP, possibly subscripted, stands for an answer pattern (in the abstract domain). Each literal in the program is subscripted with an identifier or pair of identifiers. $A : CP$ stands for an atom (unsubscripted) together with a calling pattern. Rules are assumed to be normalized and each rule for a predicate p has identical sets of variables $p(x_{p_1}, \ldots x_{p_n})$ in the head atom. 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 atom (and constraint redundantly) e.g. $H_k \leftarrow B_{k,1}, \ldots, B_{k,n_k}$ 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. E.g., the append program is written:

```
app(X,Y,Z)₁ :- X=[]₁,₁, Y=Z₁,₂.
app(X,Y,Z)₂ :- X=[U|V]₂,₁, Z=[U|W]₂,₂, app(V,Y,W)₂,₃.
```

```
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_1 := Aproject(CP, B_{k,1})
      add_event(arc(
         A_k : CP ⇒ [CP] 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 dependency_arc_table
      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
   AP_0 := get_answer(B_{k,i} : CP_2)
   CP_3 := Acombine(CP_1, AP_0)
   if (CP_3 <> ⊥ and i <> n_k)
      CP_4 := Aproject(CP_3, 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 := Aproject(CP_3, H_k)
      insert_answer_info(H : CP_0 ↦ AP_1)

get_answer(L : CP)
   if (L is a constraint)
      return Aadd(L, CP)
   else   return lookup_answer(L, CP)

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. 1. Generic fixpoint algorithm

The program analysis graph is defined in terms of an initial set of calling patterns, a program, and four abstract operations on the description domain:

- $Aproject(CP, L)$ which performs the abstract restriction of a calling pattern CP to the variables in the literal L;
- $Aadd(C, CP)$ which performs the abstract operation of conjoining the actual constraint C with the description CP;
- $Acombine(CP_1, CP_2)$ which performs the abstract conjunction of two descriptions;
- $Alub(CP_1, CP_2)$ which performs the abstract disjunction of two descriptions.

The analysis graph has two sorts of nodes: those belonging to rules (also called "AND-nodes") and those belonging to atoms (also called "OR-nodes"). 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. The graph is implicitly represented in the algorithm by means of two data structures, the *answer table* and the *dependency arc table*. The answer table contains entries of the form $A : CP \mapsto AP$. A is always a base form. This represents a 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 calling pattern CP_0 then this causes literal $B_{k,i}$ to be called with calling pattern 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 included for efficiency. Dependency arcs represent the arcs in the program analysis graph from atoms in a rule body to an atom node.

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 fixpoint algorithms, a priority queue is used. Thus, the third, and final structure is a *prioritized event queue*. Events are of three forms. The first, $updated(A : CP)$, indicates that the answer pattern to atom A with calling pattern CP has been changed. The second, $arc(R)$, indicates that the rule referred to in R needs to be (re)computed from the position indicated. The third, $newcall(A : CP)$, indicates that a new call has been encountered. The priority mechanism for the queue is left as a parameter of the algorithm.

Figure 1 shows the generic analysis algorithm. Apart from the parametric 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 overwrites any other arcs of the form $H_k : CP \Rightarrow [\_]B_{k,i} : \_$ in the table and in the priority queue. 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 procedure remove_useless_calls traverses the dependency graph given by the dependency arcs from the initial calls S and marks those entries in the dependency arc and answer table which are reachable. The remainder are removed.

In [HMPS95] the following results are presented:

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 corollary of this is that the priority strategy does not affect the correctness of the analysis.

Corollary 2. *The result of the generic analysis algorithm does not depend on the strategy used to prioritize events.*

4 Optimizing the Generic Algorithm

The algorithm presented in Section 3 is parametric with respect to the event handling strategy in the priority queue, in order to capture the behavior of several possible algorithms. Correctness of the analysis does not depend on the order in which events are processed. However, efficiency does.

The cost of analysis can be split into two components. The cost of computing the arc events, which for a given program P and a queuing strategy q will be denoted $\mathcal{C}_a(P,q)$, and the cost associated with dealing with the event queue which will be denoted $\mathcal{C}_q(P,q)$. There is clearly a trade-off between $\mathcal{C}_a(P,q)$ and $\mathcal{C}_q(P,q)$ in that a more sophisticated event handling strategy may result in a lower number of arcs traversed but at a higher event handling cost. We now discuss some possible optimizations to the generic fixpoint algorithm.

4.1 General Simplifications

Dealing Only with Arc Events in the Priority Queue: The original fixpoint algorithm in Section 3 has to deal with three different kinds of events, namely *updated*, *arc* and *newcall*. This can make the priority mechanism for the queue rather complicated. Looking at the actions performed for each one of these events and the optimizations presented below, it can be seen that the effect of both updated and newcall can be reduced to that of the arc events. Additionally, newcall performs an initial guess of the answer pattern. However, we will always use $\perp$ as the trivial initial guess for newcall events. Therefore, the event queue only needs to deal with arc events. Whenever the generic analysis algorithm would add to the queue an updated or newcall event, the optimized algorithm will directly add to the queue the required arc events.

In what follows, the current event queue will be denoted as Q and will be a set of triples $\langle arc, q(arc), type\rangle$, where *type* can be either newcall or updated and indicates whether such arc was introduced due to a newcall or an updated answer pattern, and q will be a function called *queuing strategy* that will assign a priority (a natural number) to each arc event. $\top(Q)$ is a function that returns (and deletes) from a non-empty queue Q an element with highest priority. A *strongly connected component* (SCC) in a directed graph is a set of nodes S such that $\forall n_1, n_2 \in S$ there is a path from n_1 to n_2.

Only One Priority per Rule and Calling Pattern: The generic algorithm makes an intensive use of the event queue. Without loss of generality, we will assign priorities to arcs at a somewhat coarser level. Instead of assigning a (possibly) different priority to each arc event, we will always assign the same priority to all the arcs for the same rule and calling pattern.

Never Switching from an Arc to Another with the Same Priority: Once computation for an arc has finished (no other arc with a higher priority can be in the queue), the generic algorithm would add the rest of the arc (if any) to the queue and retrieve one of the arcs with highest priority. Instead, as there cannot be any other arc with higher priority, it is always safe to continue with the arc just added to the queue. Rather than adding the rest of the arc and retrieving it immediately, it is more efficient to process it directly. This optimization allows using the queue only once for each rule and calling pattern.

Indexing the Dependency Arc Table: Whenever a pattern is updated, all the arcs that used the old (incorrect) pattern must be found in order to generate arc events for them. This is done in procedure `add_dependent_rules` by checking all the entries in the dependency arc table against the pattern that has been updated. This process has linear complexity in the size of the analysis graph. The proposed optimization implies keeping a table that for each calling pattern contains the set of arcs that have used this information. In such a way, the set of arcs that depend on a given calling pattern can be found in constant time.

4.2 Restricting the Set of Queueing Strategies

Definition 3 Dynamic Call Graph. The *dynamic call graph* of a program P, denoted as $D(P)$ is the graph obtained from the answer table and the dependency arc table generated for P by the generic analysis algorithm as follows: for each entry $A : CP \mapsto AP_0$ in the answer table create the node $A : CP$ and for each entry $H : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2$ in the dependency arc table create a arc from node $H : CP_0$ to node $B : CP_B$, where $B : CP_B$ is the unique calling pattern for which there exists a renaming σ s.t. $B : CP_B = (B_{k,i} : CP_2)\sigma$.

Definition 4 Reduced Call Graph. The *reduced call graph* of a program P, represented as $D_R(P)$ is the directed acyclic graph obtained by replacing each SCC in $D(P)$ by a single node in $D_R(P)$ labeled with the set of nodes in the SCC, and eliminating all arcs which are internal to the SCC.

Definition 5 SCC-preserving. A queuing strategy q is *SCC-preserving* if $\forall$ program P $\forall\langle A_1, q(A_1), type1\rangle$, $\langle A_2, q(A_2), type2\rangle \in Q$, where $A_1 = arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$ and $A_2 = arc(H_{k'} : CP_{0'} \Rightarrow [CP_{1'}]\ B_{k',i'} : CP_2')$: if there is a path in $D_R(P)$ from $H_k : CP_0$ to $H_{k'} : CP_{0'}$ then $q(A_1) < q(A_2)$.

Theorem 6. *For any program P and any queuing strategy q there is a queuing strategy q' which is SCC-preserving and $\mathcal{C}_a(P, q') \leq \mathcal{C}_a(P, q)$*

This theorem implies that if $\mathcal{C}_q(P, q')$ is low enough, the set of queuing strategies considered can be restricted to those which are SCC-preserving. Definition 5 (SCC-preserving) is not operational because $D_R(P)$ cannot be computed until analysis has finished. It is thus an "a posteriori" condition. Next we give sufficient "a priori" conditions that ensure that a queuing strategy is SCC-preserving.

Definition 7 Newcall Selecting. Let $Q \neq \emptyset$ be a queue with $\langle A, q(A), type\rangle = \top(Q)$ where $A = arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$. Let $A_1, \ldots, A_n$ be the set of arc events which the event $newcall(B_{k,i} : CP_2)$ will insert in the queue. A queuing strategy q is *newcall selecting* iff $\forall\langle A', q(A'), type'\rangle \in Q\ \forall i = 1 \ldots, n$: $q(A_i) > q(A')$.

The intuition behind a newcall selecting strategy is that analysis processes calling patterns in a depth-first fashion. Note also that if no recursive predicate appears in the program, the least fixpoint would be obtained in one iteration. If the queuing strategy is not newcall selecting, several iterations may be needed even for non-recursive programs.

Definition 8 Update Selecting. Let $Q \neq \emptyset$ be a queue with $\langle A, q(A), type\rangle = \top(Q)$. Suppose that after processing the last literal in A, an $updated(H : CP)$ event is generated. Let $A_1, \ldots, A_n$ be the set of arc events which the event $updated(H : CP)$ will insert in the queue and let $\langle A_k, q(A_k), newcall\rangle \in Q$ be such that $\forall\langle A', q(A'), newcall\rangle \in Q : q(A_k) \geq q(A')$. A queuing strategy q is *update selecting* iff $\forall\langle A', q(A'), type'\rangle \in Q\ \forall i = 1\ldots, n : (q(A_k) \geq q(A')) \rightarrow (q(A_i) > q(A'))$.

I.e., the arc events generated by an updated event must have higher priority than any existing arc in the queue except for the arcs of updated type that were introduced after the last newcall.

When update selecting strategies are used together with delayed dependencies introduced below, the analysis algorithm iterates whenever an answer pattern may not be final rather than using this possibly incorrect information in parts of the analysis graph outside the SCC the answer pattern belongs to.

Delaying Entries in the Dependency Arc Table: This modification to the generic algorithm consists in executing "$AP_0 :=$ get_answer$(B_{k,i} : CP_2)$" before the conditional "**if** ($B_{k,i}$ is not a constraint) add $(\ldots)$ to dependency_arc_table" in the procedure `process_arc` in the generic algorithm. The aim is not no introduce any dependency until an answer pattern is actually used.

Notice than in this case we are not restricting the set of considered queuing strategies but rather we are modifying the generic algorithm itself.

Theorem 9 Delaying Dependencies. *If the queuing strategy is newcall and update selecting then the algorithm obtained from the generic one by delaying dependencies produces the same analysis results as the generic algorithm.*

Theorem 10. *If dependencies are delayed and the queuing strategy is newcall and update selecting then all the arc events generated by an $updated(A : CP)$ event belong to the same SCC as $A : CP$.*

Suppose that when processing the event $arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$, the answer pattern for $B_{k,i} : CP_2$ is updated m times. In the worst case, the continuation of $arc(H_k : CP_0 \Rightarrow [CP_1]\ B_{k,i} : CP_2)$ would be computed m times. Additionally, this computation may generate an $updated(H_k : CP_0)$ event which may in turn generate update events for any calling pattern in the analysis graph. This theorem ensures that unless $B_{k,i} : CP_2$ and $H_k : CP_0$ are in the same SCC, the continuation of the arc will only be computed once due to updated values of $B_{k,i} : CP_2$, independently of the number of times the answer pattern for $B_{k,i} : CP_2$ is updated and the number of iterations needed to compute it.

Theorem 11 SCC-preserving. *If dependencies are delayed then if a queuing strategy q is newcall selecting and update selecting then q is SCC-preserving.*

This is a sufficient "a priori" condition to obtain SCC-preserving strategies.

4.3 Undefined Functions

Ordering Arcs from Newcall Events – the Newcall Strategy: Although SCC-preserving strategies are efficient in general, for any given program P different SCC-preserving strategies may have different values for $\mathcal{C}_a(P, q)$ and $\mathcal{C}_q(P, q)$. There are still several degrees of freedom associated with the event handling

```
analyze(S)
  foreach A : CP ∈ S
    new_calling_pattern(A : CP)

process_update(Updates)
  if Updates = A1 :: As
    UAs := process_arc(A1)
    NAs :=
    global_updating_strategy(As,UAs)
    process_update(NAs)

insert_answer_info(H : CP ↦ AP)
  AP0 := lookup_answer(H : CP)
  AP1 := Alub(AP,AP0)
  A := {}
  if (AP0 <> AP1)
    add (H : CP ↦ AP1) to answer_table
    foreach arc of the form
      Hk : CP0 ⇒ [CP1] Bk,i : CP2
      in dependency_arc_table
      where there exists renaming σ
      s.t. H : CP = (Bk,i : CP2)σ
      A := A ⋃
        {Hk : CP0 ⇒ [CP1] Bk,i : CP2}
  return:=local_updating_strategy(A)

lookup_answer(A : CP)
  if (there exists a renaming σ s.t.
    σ(A : CP) ↦ AP in answer_table)
    return σ^-1(AP)
  else
    return σ^-1(
      new_calling_pattern(σ(A : CP)))
    where σ is a renaming s.t.
    σ(A) is in base form
```

```
new_calling_pattern(A : CP)
  add A : CP ↦ ⊥ to answer_table
  A0 := {}
  foreach rule Ak ← Bk,1, ..., Bk,nk
    CP1 := Aproject(CP, Bk,1)
    A0 := A0 ⋃
    {Ak : CP ⇒ [CP] Bk,1 : CP1}
  Arcs := newcall_strategy(A0)
  process_newcall(Arcs)
  Let σ be a renaming s.t.
    σ(A : CP) ↦ AP in answer_table
    return σ^-1(AP)

process_newcall(NewCalls)
  if NewCalls = A1 :: As
    UArcs := process_arc(A1)
    process_update(UArcs)
    process_newcall(As)

process_arc(Hk : CP0 ⇒ [CP1] Bk,i : CP2)
  if (Bk,i is not a constraint)
    AP0 := lookup_answer(Bk,i : CP2)
    add Hk : CP0 ⇒ [CP1] Bk,i : CP2
    to dependency_arc_table
  else
    AP0 := Aadd(Bk,i, CP2)
  CP3 := Acombine(CP1, AP0)
  if (CP3 <> ⊥ and i <> nk)
    CP4 := Aproject(CP3, Bk,i+1)
    U := process_arc(
      Hk : CP0 ⇒ [CP3] Bk,i+1 : CP4)
  elseif (CP3 <> ⊥)
    AP1 := Aproject(CP3, Hk)
    U := insert_answer_info(
        H : CP0 ↦ AP1)
  return U
```

Fig. 2. Optimized SCC-preserving analysis algorithm

strategy. The first one, which we will call the *newcall strategy* refers to the priorities among the different arcs generated by a single newcall (there will be one arc event per clause defining the called predicate). We know that all of them should have a higher priority than the existing arcs, but nothing has been said up to now about their relative priorities.

Ordering Arcs from Updated Events – the Updating Strategy: The newcall selecting condition is in a sense stronger than the updating strategy condition. The newcall selecting condition requires the new arcs to be assigned priorities which are higher than any other existing one. Therefore, there is even more freedom to assign priorities to arcs generated by updated events. The approach taken will be to split the updating strategy into two components. One is the relative order of the arc events introduced by a single updated event (*local updating strategy*). The other one is the order of these new arc events with respect to the already existing updated type arc events in the queue that were

```
incremental_addition(R)
   A0 = {}
   foreach rule Ak ← Bk,1, ..., Bk,nk ∈ R
      foreach entry A : CP ↦ AP
         in the answer_table
         CP1 := Aproject(CP, Bk,1)
         A0 = A0 ⋃
            {Ak : CP ⇒ [CP] Bk,1 : CP1}
   A := inc_updating_strategy(A0)
   process_inc_update(A)

process_inc_update(Updates)
   if Updates = A1 :: As
      U := process_arc(A1)
      NAs :=
         inc_updating_strategy(As,U)
      process_inc_update(NAs)
```

Fig. 3. Incremental Addition Algorithm

introduced in the queue later than any newcall type arc event (*global updating strategy*).

5 An Optimized Analysis Algorithm

Figure 2 presents an optimized analysis algorithm in which dependencies are delayed. It also ensures that the newcall selecting and updating selecting conditions will hold, thus always providing SCC-preserving strategies (Theorem 11). It is parametric with respect to the newcall strategy and local and global updating strategies introduced above. Different choices of these strategies will provide different SCC-preserving instances of the algorithm with possibly different efficiency.

The two different types of arc events are treated separately by procedures `process_newcall` and `process_update`. Also, rather than having an external data structure for the queue, we will use explicit parameters to store the arc events that have to be processed. The run-time stack of procedure and function calls will isolate and store the arcs. Assuming the the pseudo-code used to describe the algorithm is sequential, the newcall selecting condition is satisfied because if no entry is stored for a calling pattern in the answer table, the procedure `look_up_answer` will have to wait for `new_calling_pattern` to finish before returning control to the calling `process_arc` procedure. The update selecting condition is also satisfied because in the procedure `process_newcall`, `process_update` is called after processing each arc and before executing the recursive call to `process_newcall` for the remaining arcs from the same newcall. Therefore, as dependencies are delayed, according to Theorem 11, the algorithm is SCC preserving for any newcall and local and global updating strategies.

5.1 Augmenting the Algorithm for Incremental Addition

In order to cope with incremental addition, i.e., a set of rules R is added to a program, analysis should process each rule in R with all the existing calling patterns in the answer table for the predicate the rule belongs to. This is done by procedure `incremental_addition` in Figure 3. Note that a specialized version of procedure `process_update` which is called `process_inc_update` is used to start incremental analysis of the new arcs. However, in this case delaying dependencies is not possible because before incrementally analyzing the new clauses, a fixpoint will have been reached and all dependencies will have been introduced. Therefore, for any node $A : CP$ which existed in the analysis graph before incremental analysis started the arc events generated by an event *updated*$(A : CP)$ will not necessarily belong to the same SCC as $A : CP$ and analysis may no longer be SCC-preserving. Thus, it makes sense to use a more involved updating strategy for this case than for the non-incremental one in order to avoid unneeded recomputations. This strategy will be called the *inc_updating_strategy*. Incremental addition will be SCC-preserving or not depending on this strategy. However, for any new calling pattern in the analysis graph it is possible to delay dependencies and thus the algorithm in Figure 2 will be SCC-preserving for them. Thus, for such calling patterns it is profitable to use `process_update` whenever possible rather than `process_inc_update`.

6 Experimental Results

A series of experiments has been performed for both the incremental and non-incremental case. The fixpoint algorithms we experiment with have been implemented as extensions to the PLAI generic abstract interpretation system. We argue that this makes comparisons between the new fixpoint algorithms and that of PLAI meaningful, since on the one hand PLAI is an efficient, highly optimized, state-of-the-art analysis system, and on the other hand the algorithms have been implemented using the same technology, with many data structures in common. They also share the domain dependent functions, which is *sharing+freeness* [MH91] in all the experiments.

Three analysis algorithms, as well as PLAI.[1] have been considered. **DD** is the algorithm for incremental analysis used in [HMPS95] (**Incr** or **I** in the experimental results). Both **DI** and **DI**$_S$ are instances of the algorithm presented in Figure 2, with the extensions for incremental addition presented in Figure 3. The difference between **DI** and **DI**$_S$ is the newcall strategy used. **DI**$_S$ uses the more elaborated strategy of computing the SCC of the static graph in order to give higher priority to non-recursive clauses. **DI** simply uses the lexical order of clauses to assign them different priorities. Both use the same updating strategy: the local strategy is to process arcs in the order they were introduced in the dependency arc table, and the global strategy is to use a LIFO stack and eliminate subsumed arcs , i.e., other arcs in the queue exist which ensure that their computation is redundant. Due to lack of space, subsumed arcs are not studied here [PH96]. The incremental updating strategy is to use a FIFO queue and eliminate subsumed arcs. **DD** uses *depth-dependent* and both **DI** and **DI**$_S$ *depth-independent* propagations ([PH96]).

6.1 Analysis Times for the Non-Incremental Case

Table 1 shows the analysis times for a series of benchmark programs using the algorithms mentioned above. Times are in milliseconds on a Sparc 10 (SICStus 2.1, fastcode). A relatively wide range of programs has been used as benchmarks. They can be obtained from `http://www.clip.dia.fi.upm.es`. However, the number of clauses is included in the table (column **Cl**) for reference. **DD_SU**, **DI**$_S$**_SU** and **DI_SU** are the speed-ups obtained in analysis time by each fixpoint algorithm with respect to PLAI. As already observed in [HMPS95], the performance of **DD** is almost identical to that of PLAI (it introduces no relevant

[1] The algorithm used for PLAI is the one in the standard distribution which has been augmented to keep track of detailed dependencies that are later used in multiple specialization [PH95]. This introduces a small overhead over the original algorithm.

Bench.	Cl	PLAI	DD	$\mathbf{DI}_S$	DI	DD_SU	$\mathbf{DI}_S$_SU	DI_SU
aiakl	12	3526	3532	2563	2483	1.00	1.38	1.42
ann	170	6572	6593	6615	6906	1.00	0.99	0.95
bid	50	783	779	769	789	1.01	1.02	0.99
boyer	133	2352	2346	2339	2475	1.00	1.01	0.95
browse	29	329	339	343	393	0.97	0.96	0.84
deriv	10	420	436	421	406	0.96	1.00	1.03
fib	3	29	36	29	33	0.81	1.00	0.88
grammar	15	132	128	129	119	1.03	1.02	1.11
hanoiapp	4	579	565	619	539	1.02	0.94	1.07
mmatrix	6	309	306	312	326	1.01	0.99	0.95
occur	8	296	299	316	273	0.99	0.94	1.08
peephole	134	5855	5919	4870	5090	0.99	1.20	1.15
progeom	18	199	199	199	219	1.00	1.00	0.91
qplan	148	1513	1499	1422	1383	1.01	1.06	1.09
qsortapp	7	346	332	323	402	1.04	1.07	0.86
query	52	108	116	109	89	0.93	0.99	1.21
rdtok	54	2528	2509	1316	1209	1.01	1.92	2.09
read	88	44362	44259	14123	11765	1.00	3.14	3.77
serialize	12	629	629	663	616	1.00	0.95	1.02
tak	2	98	99	102	103	0.99	0.96	0.95
warplan	101	3439	3352	2789	2803	1.03	1.23	1.23
witt	160	1902	1902	1762	1738	1.00	1.08	1.09
zebra	18	3376	3356	3362	3259	1.01	1.00	1.04
					Overall	(1.00) 1.00	(1.13) 1.75	(1.12) 1.84

Table 1. Analysis Times for the Non-Incremental Case

overhead) but has the advantage of being able to deal with incremental addition. On the other hand, both **DI** and $\mathbf{DI}_S$ show significant advantage with respect to **DD** (and PLAI). **DI** is the most efficient of the three, but the margin over $\mathbf{DI}_S$ is small. Two overall speed-ups appear in the table for each algorithm. The one in brackets represents the overall speed-up after eliminating the `read` benchmark, because of the atypical results. The relative advantage of **DI** and $\mathbf{DI}_S$ is inverted in this case. The peculiarity in `read` stems from the fact that the dynamic call graph has many cycles with lengths that are as high as 13. However, even when taking `read` out **DI** and $\mathbf{DI}_S$ are both still somewhat better that **DD** and PLAI.

6.2 Analysis Times for the Incremental Case

Among the different types of incremental change identified in [HMPS95] the one which is really relevant for experimentation is incremental addition. The performance of the fixpoint algorithms in the other types of changes will be directly related to the efficiency of the algorithms in the non-incremental case, as no incremental update propagation is needed. Table 2 shows the analysis times for the same benchmarks but adding the clauses one by one. I.e., the analysis was first run for the first clause only. Then the next clause was added and the resulting program (re-)analyzed. This process was repeated until the last clause of the program. The total time involved in this process is given by **DD**, $\mathbf{DI}_S$, and **DI**. Columns $\mathbf{SU}_{DD}$, $\mathbf{SU}_{DI_S}$, and $\mathbf{SU}_{DI}$ contain the speed-up obtained with respect to analyzing with the same algorithm the program clause by clause but erasing the analysis graph between analyses. Thus, it is a measure of the incrementality of each algorithm. An important speed-up is observed in $\mathbf{SU}_{DD}$

Bench.	**DD**	**DI**$_S$	**DI**	**SU**$_{DD}$	**SU**$_{DI_S}$	**SU**$_{DI}$	**SD**$_{DD}$	**SD**$_{DI_S}$	**SD**$_{DI}$
aiakl	3860	3527	3237	1.52	1.38	1.29	1.09	1.38	1.30
ann	41680	25686	8120	12.66	22.82	72.83	6.32	3.88	1.18
bid	4220	2240	1433	3.82	6.54	9.80	5.42	2.91	1.82
boyer	20029	9039	3870	13.00	29.21	69.35	8.54	3.86	1.56
browse	1110	652	556	5.61	3.91	4.82	3.27	1.90	1.41
deriv	3083	1570	1126	0.54	1.63	2.07	7.07	3.73	2.77
fib	57	49	49	1.68	1.96	1.84	1.58	1.69	1.48
grammar	510	300	209	2.41	4.17	5.34	3.98	2.33	1.76
hanoiapp	990	779	816	1.37	1.86	1.46	1.75	1.26	1.51
mmatrix	709	360	343	1.37	2.67	3.12	2.32	1.15	1.05
occur	456	396	322	1.32	3.73	3.97	1.53	1.25	1.18
peephole	59899	15333	8533	8.66	28.19	52.05	10.12	3.15	1.68
progeom	389	360	283	2.63	2.87	3.44	1.95	1.81	1.29
qplan	39890	11303	2342	3.69	12.42	56.94	26.61	7.95	1.69
qsortapp	623	506	466	1.81	2.17	2.73	1.88	1.57	1.16
query	2296	919	277	2.23	7.14	20.32	19.79	8.43	3.11
rdtok	24176	3822	2363	1.66	6.96	10.06	9.64	2.90	1.95
read	176779	35760	22160	5.57	8.16	11.28	3.99	2.53	1.88
serialize	1496	1290	973	2.23	2.63	3.25	2.38	1.95	1.58
tak	139	120	113	1.31	1.75	1.77	1.40	1.18	1.10
warplan	41999	9436	5479	2.69	10.71	17.32	12.53	3.38	1.95
witt	19336	18606	2523	3.08	3.37	17.57	10.17	10.56	1.45
zebra	8580	2716	2480	4.87	15.32	16.44	2.56	0.81	0.76
Overall	6.64	2.13	1	6.15	13.74	28.36	5.69	3.18	1.57

Table 2. Incremental Addition Times

(as already noted in [HMPS95]), but the incrementality of **DI**$_S$ is twice as high, and that for **DI** in turn twice as high as that of **DI**$_S$.

The last three columns in the table contain the slow-downs for clause by clause incremental analysis with respect to the time taken by the same algorithm when analyzing the file all at once. If we use the **DD** algorithm in an incremental way, the overhead resulting from analyzing clause by clause is greatly reduced with respect to the non-incremental case. However, the time required if we use **DI** incrementally is only about 3/2 of the time required to analyze the program all at once. There is even one case (the `zebra` benchmark) in which using the **DI** algorithm clause by clause is somewhat faster than analyzing the program all at once. However, we believe this is related to working set size and cache memory effects, as the number of arc events processed in both cases (presented in Table 3) is almost the same. In the **Overall** row we give the average analysis times for each algorithm, taking as unit the time for analysis clause by clause using the **DI** algorithm. At least for the benchmark programs **DI** is more than twice as fast as **DI**$_S$ and more than 6 times faster than **DD** ([HMPS95]).

6.3 Measuring $\mathcal{C}_a(P, q)$: Number of Arc Events

Table 3 shows the number of arc events needed to analyze each benchmark program in both the non-incremental and incremental case using the **DI** algorithm. This is equivalent to counting the number of times the function `process_arc` in the algorithm in Figure 2 is called (including any recursive calls) from (**N**) `process_newcall`, (**U**) `process_update`, and (**UI**) `process_inc_update`. **T** is the total number of arc events processed. $_I$ is used for the incremental case. The

Bench.	N	U	T	U/T	N_I	U_I	UI_I	T_I	UI_I/T_I
aiakl	50	19	69	0.28	52	8	76	136	0.56
ann	570	179	749	0.24	496	203	101	800	0.13
bid	191	14	205	0.07	144	10	165	319	0.52
boyer	248	70	318	0.22	82	34	330	446	0.74
browse	41	19	60	0.32	21	3	78	102	0.76
deriv	24	1	25	0.04	0	0	52	52	1.00
fib	14	3	17	0.18	6	3	8	17	0.47
grammar	24	0	24	0	2	0	28	30	0.93
hanoiapp	21	15	36	0.42	18	11	26	55	0.47
mmatrix	10	9	19	0.47	2	3	14	19	0.74
occur	15	14	29	0.48	12	12	4	28	0.14
peephole	255	170	425	0.40	180	23	440	643	0.68
progeom	41	9	50	0.18	38	9	3	50	0.06
qplan	384	41	425	0.10	205	31	235	471	0.50
qsortapp	44	15	59	0.25	23	4	41	68	0.60
query	59	0	59	0	0	0	62	62	1.00
rdtok	332	33	365	0.09	145	24	328	497	0.66
read	840	155	995	0.16	720	22	1398	2140	0.65
serialize	43	15	58	0.26	16	1	102	119	0.86
tak	27	5	32	0.16	17	5	10	32	0.31
warplan	330	38	368	0.10	169	13	362	544	0.67
witt	389	39	428	0.09	352	36	44	432	0.10
zebra	51	2	53	0.04	28	2	24	54	0.44
Overall	4003	865	4868	0.18	2728	457	3931	7116	0.45

Table 3. Number of *arc* Events Processed

last row in the table shows the number of arc events of each type needed to analyze all the benchmarks. The remaining two columns (**U/T** and $\mathbf{UI}_I/\mathbf{T}_I$) give respectively the ratio of the total arc events that were due to update events in the non-incremental case and those due to the newly introduced clauses in the incremental case. **U/T** gives an idea of how much analysis effort is due to fixpoint computation for recursive calls. These figures show that using a good analysis algorithm, less than 20% of the effort is due to iterations. $\mathbf{UI}_I/\mathbf{T}_I$ gives the ratio of the computation performed by `process_inc_update` (which may use a more complex updating strategy). The ratio between the total number of arcs computed in the incremental and non-incremental case explains the slow-down associated to the analysis clause by clause. It is $7116 \div 4868 = 1.46$ in number of arc events processed and 1.57 in analysis times for the **DI** algorithm. The table also seems to imply that, for the strategies used, counting arc events is a good (and architecture independent) indicator of analysis time.

7 Acknowledgments

This work has benefited greatly from discussions with K. Marriott and P. Stuckey.

References

[BGH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CH94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[Deb92] S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[HMPS95] M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*. MIT Press, June 1995.

[HWD92] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[Jor94] N. Jorgensen. Finding Fixpoints in Finite Function Spaces Using Neededness Analysis and Chaotic Iteration. In *International Static Analysis Symposium*. Springer-Verlag, 1994.

[KB95] A. Krall and T. Berger. Incremental global compilation of prolog with the vienna abstract machine. In *International Conference on Logic Programming*. MIT Press, June 1995.

[LDMH93] B. Le Charlier, O. Degimbe, L. Michel, and P. Van Henteryck. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *International Workshop on Static Analysis*. Springer-Verlag, 1993.

[MH90] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH91] K. Muthukumar and M. Hermenegildo, Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming*, MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[PH95] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*. ACM, June 1995.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for Incremental Analysis of Logic Programs. Technical Report CLIP3/96.0, Facultad de Informática, UPM, 1996.

[RD92] P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

[VWL94] B. Vergauwen, J. Wauman, and J. Levi. Efficient Fixpoint Computation. In *International Static Analysis Symposium*. Springer-Verlag, 1994.