A Local Algorithm for Incremental Evaluation of Tabled Logic Programs

Diptikalyan Saha and C. R. Ramakrishnan

Dept. of Computer Science, University at Stony Brook, Stony Brook, NY 11794
E-mail: {dsaha, cram}@cs.sunysb.edu

Abstract. This paper considers the problem of efficient incremental maintenance of memo tables in a tabled logic programming system when the underlying data are changed. Most existing techniques for incremental evaluation (or materialized view maintenance in deductive databases) consider insertion and deletion of facts as primitive changes, and treat update as deletion of the old version followed by insertion of the new version. They handle insertion and deletion using independent algorithms, consequently performing many redundant computations when processing updates. In this paper, we present a local algorithm for handling updates to facts. We maintain a dynamic (and potentially cyclic) dependency graph between and among calls and answers in the memo tables. The key idea is to interleave the propagation of deletion and insertion operations generated by the updates through this graph. The dependency graph used in our algorithm is more general than that used in algorithms previously proposed for incremental evaluation of attribute grammars and functional programs. Nevertheless, our algorithm's complexity matches that of the most efficient algorithms built for these specialized cases. We demonstrate the effectiveness of our algorithm using data-flow analysis and parsing examples.

1 Introduction

Tabled resolution [4, 6, 29] and its implementations have enabled the development of applications in areas ranging from program analysis and verifi cation [8, 20, e.g.], to object-oriented knowledge bases [32, e.g.]. Since results of computations are cached, tabling also offers the potential to incrementally compute the changes to the results when a rule or fact in the program changes. Incremental evaluation of tabled logic programs will enable us to directly derive incremental versions of different applications.

The Driving Problem. The problem of incremental evaluation of logic programs is closely related to the view maintenance problem which has been extensively researched, especially in the context of deductive databases [11, 12, e.g.]. Most of these works, including our earlier algorithms [22–24] consider changes to the program only in terms of addition and deletion of facts. An update of a fact is treated as the deletion of the old version followed by the addition of the new version, which may lead to several unnecessary evaluation steps. In contrast, techniques originally from incremental attribute-grammar evaluation [9, 21], treat update as in-place change, and propagate this change. This approach is very restrictive for logic programs, since an update may lead to additions or deletions in general. In-place update techniques for logic programs work only with non-recursive programs, and restrict the changes to "non-key attributes" [27]: i.e. the control behavior of the program cannot change. However, these update propagation algorithms are optimal whenever they are applicable (i.e. when the restrictive conditions are met).

The interesting problem then is to devise an incremental technique for processing additions, deletions as well as updates, which applies to a large class of logic programs, and yet is optimal for the class of programs handled by the in-place update algorithms.

```
:- table r/2.
r(X,Y) :- e(X,Y).
r(X,Y) :- e(X,Z), r(Z,Y).

e(1,2).
e(2,3).
e(3,4).
e(4,3).
```

(a)

Calls	Answers
r(1,A)	r(1,2), r(1,3), r(1,4)
r(2,A)	r(2,3), r(2,4)
r(3,A)	r(3,4), r(3,3)
r(4,A)	r(4,3), r(4,4)

(b)

Fig. 1. Example tabled logic program (a), and its call and answer tables (b).

We present such a technique in this paper. We give an incremental evaluation algorithm that *interleaves* the processing of additions and deletions. When the conditions of in-place update algorithms are met, our algorithm generates matching pairs of additions and deletions which result in optimal change propagation. Our algorithm naturally generalizes incremental algorithms for attribute evaluation [21] and functional program evaluation [1].

An Illustrative Example. Consider the evaluation of query `r(1,A)` over the program in Figure 1(a). In the program, `r/2` defines the reachability relation over a directed graph, whose edge relation is given by `e/2`. The calls and answers computed by tabled resolution for this query are given in Figure 1(b).

Now consider the effect of changing fact `e(2,3)` to `e(2,4)` and treating this change as the deletion of `e(2,3)` followed by the addition of `e(2,4)`. First, when `e(2,3)` is deleted, nodes `3` and `4` are no longer reachable from `1` or `2`. Thus the answers `r(2,3)`, `r(2,4)`, `r(1,3)` and `r(1,4)` are deleted. Subsequently, the addition of `e(2,4)` makes nodes `3` and `4` again reachable from `1` and `2`. Incremental processing of this addition introduces answers `r(2,4)`, `r(2,3)`, `r(1,4)` and `r(1,3)`.

Updates may lead to deletions or additions in general making in-place update algorithms restrictive. For instance, in the above example, if fact `e(2,3)` is changed to `e(3,2)`, node `2` becomes reachable from `3` and `4`, and nodes `3` and `4` are no longer reachable from `1` or `2`. However, a judicious interleaving of additions and deletions can simulate the effect of in-place update wherever possible. For instance, consider the above example again when `e(2,3)` is changed to `e(2,4)`. Since `e(2,3)` is changed, we first inspect `r(2,X)`'s answer table and recalculate results. If we process all changes to `r(2,X)` first, we will stop the propagation there since there is no net change in `r(2,X)`'s answers. Hence `r(1,3)` and `r(1,4)` will not even be deleted in the first place.

Salient Features of Our Approach. We consider definite logic programs where facts as well as rules may be changed between two query evaluation runs. We consider an update as a delete and an insert, but select the order in which they will be processed based on the dependencies between and among the queries and computed answers. We describe data structures and algorithms to process additions bottom-up while using the information about the original queries. Section 2 introduces the data structures used for incremental processing of additions and deletions. Interleaving between the two operations is achieved by decomposing the processing of an addition or deletion into finer-grained operations, and assigning priorities to these operations. Section 3 describes the assignment of priorities and a scheduler to perform the operations in order.

The order in which the operations are performed generalizes the call-graph based orders used in previous incremental algorithms [13, e.g.] where changes are evaluated from topologically lower strongly connected components (SCCs) to higher SCCs in the call graph. As a result, our algorithm inspects the same number of answers in tables (which is a good measure of an incremental algorithm's performance) as algorithms that perform in-place updates. In particular, for non-recursive programs, the order in which operations are performed coincide with the topological order of the call dependencies. Hence our

algorithm is optimal for the cases for which optimal update algorithms are known. Moreover the algorithm handles inserts, deletes and updates efficiently, even when the changes affect the control behavior of the program. We explain how our algorithm naturally generalizes incremental evaluation of attribute grammars [21] and functional programs [1] in Section 4. We also present experimental results showing the effectiveness of the algorithm in that section.

In a more general setting when the dependencies may be recursive, our algorithm interleaves insertion and deletion operations even within an SCC in the call graph. It can be shown that our schedule of operations is uniformly better than inserts-first or deletes-first schedules. Our approach is closely related to those used in several incremental program analysis techniques where change propagation is controlled by considering SCCs in a dependency graph. A detailed discussion relating this algorithm to previous work appears in Section 5. Extensions and optimizations of our technique are discussed in Section 6. A more detailed version of this paper with formal aspects of the local algorithm is available as a technical report [26].

2 Data Structures for Incremental Evaluation

We restrict our main technical development to definite logic programs. We later describe how to extend these results to general logic programs evaluated under the well-founded semantics. As an optimization, we assume that definitions of only those predicates that are marked as *volatile* may be changed (i.e. with additions, deletions or updates).

In SLG resolution [6], derivations are captured as a proof forest, called *SLG forest* [5]. The SLG forest constructed when evaluating goal `r(1,X)` over the program in Figure 2(a) is given in Figure 2(e). Each tree in the forest corresponds to a call in the call table. For a given tree, the different branches correspond to derivations; the computed answer substitutions of successful derivations correspond to answers of the call. We informally describe the construction of an SLG forest using the example in Figure 2. The initial call, `r(1,X)` results in a root node `r(1,X) :- r(1,X)` in the forest (labelled p_1 in the figure). The call `r(1,X)` is also entered in the call table. The children of this node are obtained by resolving the selected literal in the body of the node (`r(1,X)`, in this case) with the program clauses. The node `r(1,X) :- e(1,X)` (labeled c_1) corresponds to the step in derivation of answers to `r(1,X)` based on the answers to `e(1,X)`. Since `e/2` is not tabled, children of this node are also obtained by program clause resolution. Note that since `e(1,2)` and `e(1,3)` are facts, this node has two children, `r(1,2)` and `r(1,3)` (labeled s_1 and s_2, resp.), corresponding to two answers of `r(1,X)`. These two answers are entered into the answer table corresponding to `r(1,X)`.

The other child of p_1, `r(1,X) :- r(1,Z),e(Z,X)` (labeled c_2) is the result of resolving `r(1,X)` with the second clause defining `r/2`. The selected literal in this node is `r(1,Z)` which is a variant (i.e. a renaming) of a call in the table, hence its children are obtained by resolving `r(1,Z)` with the answers in the corresponding answer table. For instance, using the answer `r(1,2)`, we get `r(1,X) :- e(2,X)` as a child of c_2. At any step, if the selected literal G of some node n is tabled but a variant of G is not already in the call table, we start a new tree in the forest with $G :- G$ as the root and add G to the call table. Children are added to the original node n as and when answers are computed for G. The construction process continues until the SLG forest is *complete*, i.e. it can no longer be expanded. The leaves of a complete SLG forest of the form $G_0 :- G_1, \ldots, G_n$ represent a failed derivation; the other leaves represent successful derivations of answers.

For program given in Figure 2(a) with facts in (b), the tabled call and answers are given in Figure 2(c); and the SLG forest in Figures 2(d) and (e).

Each tree in the SLG forest corresponds to a *generator*; the call associated with the root of a tree is said to be the call of that generator (denoted by $p.call$ where p is the generator). Each non-root node in the SLG forest whose selected literal is either tabled or volatile corresponds to a *consumer*, defined formally as follows:

Definition 1 (Consumer) *Let $\mathcal{P}$ be a definite logic program, and F be the SLG forest constructed when evaluating a query Γ over P. Then $c = \langle p, G_0, G_1, [G_2, \ldots, G_n] \rangle$ for some $n \geq 0$ is a consumer iff the SLG tree of generator p in F has a non-root node $G_0 :- G_1, G_2, \ldots, G_n$. The set of all consumers in F is denoted by C_F.*

Note that a consumer carries more information than its corresponding non-root node in the SLG forest. For the rest of this paper we refer to the 'non-root nodes' in the SLG forest and its corresponding 'consumer' interchangeably.

Each edge in the SLG forest arises due to program or answer clause resolution. For each edge (n_1, n_2) in the forest, n_1, as well as the program clause or answer used in that resolution step are called the *premises* of n_2. For instance, `r(1,X) :- r(1,Z),(Z,X)` (node c_2) and the answer `r(1,2)` are premises to `e(2,X)` (node c_3).

Definition 2 (Support) *A consumer $c \in C_F$ corresponding to a leaf of the SLG forest (i.e. $c = \langle \_, h, true, [] \rangle$) representing a successful derivation of an answer a (i.e. h is a variant of a) is called a* support *of the answer a, denoted as $a = c.answer$ and $c \in a.supports$.*

In Figure 2 (e), the nodes corresponding to the supports are shown as s_i. The various dependencies between the elements of SLG forests are defined below.

Generator-consumer dependencies: If p is a generator and $c = \langle \_, \_, g, \_ \rangle \in C_F$ is a consumer such that p is a variant of g, we say p is the generator of the consumer c, denoted by $p = c.generator$ and $c \in p.consumers$.
Consumer-consumer dependencies: For two consumers $c, c' \in C_F$ such that the node corresponding to c is a premise of the node corresponding to c' in the SLG forest, we say that $c' \in c.next\_consumer$ and $c = c'.prev\_consumer$.
Answer-consumer dependencies: If an answer a is a premise of a consumer $c \in C_F$ we say that a immediately affects c ($c \in a.imm\_affects$) and c depends on a ($c.depends\_on = a$). For example, s_6 depends on f_4 and f_4 immediately affects s_6.

We assume that the set of all consumers C_F is indexed on its third (goal) component and for constant time access the above defined relations such as *consumers*, *generator*, etc., are maintained explicitly.

Incremental changes to the facts/rules modify the SLG forest as follows. For instance, consider the insertion of fact `e(4,6)`. Since the goal field of consumer c_5 unifies with this fact, we can add in the SLG forest a child to c_5, say s_8: a support for answer `r(1,6)`. This is a new answer to generator p_1, which gets forwarded to its consumer c_2 [$c_2 \in p_1.consumers$]. The consumption of this answer by c_2 creates a child of c_2, say c_8 = `r(1,A) :- e(6,A)`. No further resolution steps are possible and the evaluation stops. Note that we perform only those operations that are needed to change the original forest to include the new fact and its effects.

Also consider the deletion of fact `e(3,4)` (f_4) from the program in Figure 2. Since the node s_6 in the SLG forest depends on f_4, that node should be deleted. Moreover, we now need to check if the corresponding answer a_3 (`r(1,4)`) is derivable using a different support *independent of* a_3. The Delete-Rederive (DRed) algorithm proposed for view maintenance in deductive databases [12] computes the changes in two phases. In

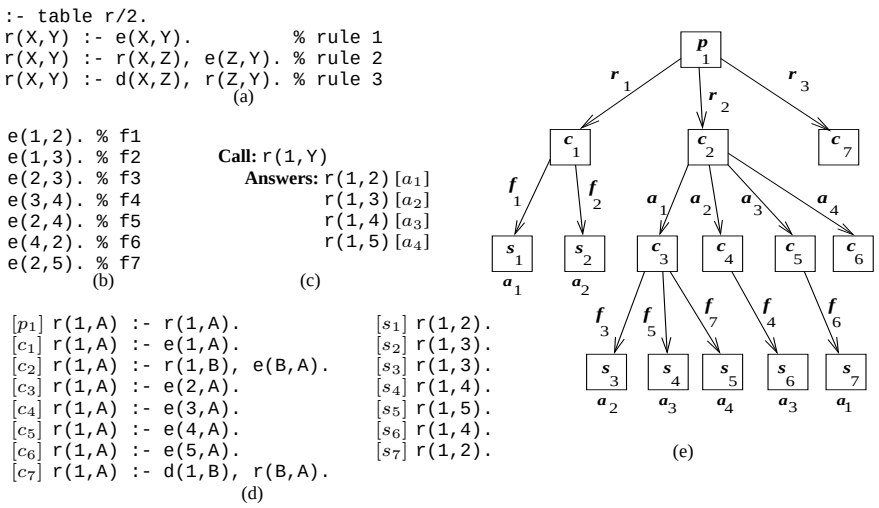


Fig. 2. Example program (a), facts (b), calls and answers (c), nodes in SLG forest (d), and SLG forest (e)

the first phase, answers that are derivable from the deleted fact are marked. In the second phase the marked answers that are derivable using only unmarked answers and facts are *rederived* and the marks on such answers are removed. This strategy is also followed in incremental evaluation techniques for program analysis [18, 33] and model checking [28]. Following this approach, we mark the support s_6 and hence the answer a_3. In the next step, node c_5 is marked since it depends on a_3. The mark on c_5 propagates to s_7, hence to answer a_1 (`r(1,2)`), ultimately marking nodes c_3–c_6, s_3–s_7 and answers a_1–a_4. In the second phase, since s_1 and s_2 are unmarked, we remove the marks on answers a_1 and a_2, and consequently nodes c_3–c_6, s_3–s_5, s_7 and answers a_3 and a_4.

Note that when a support is marked, the answer may still have other independent derivations. We can significantly reduce the number of markings by identifying *acyclic* supports: the nodes whose existence is independent of the answer it supports. Using acyclic supports, we can mark an answer a only when all its acyclic supports are marked. Note that the *first* support for an answer constructed by tabled resolution is acyclic; we call this as the *primary* support. We can significantly improve on the DRed strategy using primary supports, as illustrated by the following example. Let us again consider the deletion of fact `e(3,4)` (f_4) from the program in Figure 2. Deletion of f_4 marks s_6. Note that supports in the figure are listed in chronological order. Marking of s_6 does *not* lead to marking a_3 since its primary support s_4 is still unmarked.

The effectiveness of this heuristic can be improved if we can identify acyclic supports other than the primary support. In [23] we used *derivation lengths* to determine the acyclicity of supports. In this paper we refine and generalize this measure. First, we maintain a *call graph* that captures the dependencies between the generators in an SLG forest, and identify strongly connected components (SCCs) in the graph. If p_1 is independent of p_2 in the call graph (i.e. p_1 does not call p_2), then consumers and answers of p_1 are independent of those of p_2. We number the SCCs in a topological order so that the independence of two generators can be determined based on their SCC numbers, denoted by $p.scc$ when p is a generator. This permits us to quickly identify independent consumers and answers irrespective of their derivation lengths. Consider again the example given in Figure 1. The call graph SCC consists of two trivial SCCs - `r(1,A)` and `r(2,A)`

and a non-trivial SCC consists of calls `r(3,A)` and `r(4,A)` with SCC `r(2,A)` topologically lower than SCC `r(1,A)`. This means that call `r(2,A)` is independent of call `r(1,A)` and hence we can process changes to `r(2,A)` before propagating any changes to `r(1,A)`. Note that in this example there is no net change in the answers of `r(2,A)` and thus we do not even process the call `r(1,A)`. Call-graph SCCs have been used before for localizing change propagation, and serve the same purpose in our algorithm.

Although processing changes within an SCC before propagating its net changes to topologically higher SCCs seems to be fruitful in some cases, it is clearly ineffective for change propagation *within* an SCC. To order change propagation within an SCC, we also associate an *ordinal* with all consumers— whether on a successful derivation or not— (analogous to the derivation length) in the evaluation graph. The ordinal and SCC number attributes (*ord* and *scc*, resp.) are defined as follows:

Entity (X)	SCC number (X.*scc*)	Ordinal (X.*ord*)
Answer (a)	$p.scc$ where a is an answer of p.call, where p is a generator	$\{s.ord \mid s$ is the primary support of $a\}$+1
Consumer (c)	$p.scc$ where $c = \langle p, h, g, G\rangle$	$max\{c'.ord, Ord, 0\}$, where $c' = c.$*prev_consumer*, $a = c.$*depends_on* and $Ord = a.ord$ if $a.scc = c.scc$ and 0 ow.

Note that a support s of an answer a is acyclic if $s.$*ord* $< a.$*ord*.

The ordinal and SCC numbers are used not only to control the propagation of markings during deletion, but also to interleave operations arising from addition of facts/rules with those from deletion. This is described in detail in the next section.

3 The Local Algorithm

In this section we present the algorithm for maintaining the SLG forest incrementally when facts/rules are added, deleted or updated. The goal of our algorithm is to confine the processing as closely as possible to the part of the SLG forest that is modified by the change. We will measure an algorithm's cost as the total number of answers taken up for processing. Updates are still treated as simultaneous deletes and inserts, but the algorithm interleaves the deletion phase of marking answers and processing of insertion such that it reduces (a) the number of answers marked for deletion and (b) the number of new answers computed only to be subsequently deleted. We illustrate some of the key features of the algorithm using the example[1] given in Figure 2.

Comparison of Inserts-first, Deletes-first Methods. We notice that neither inserts-first nor deletes-first strategies is uniformly better than the other. Consider the program in Figure 2 after updating fact `e(1,2)` ($f1$) to `e(1,5)`. This is treated as deleting $f1$ and inserting a new fact $f8$ =`e(1,5)`. If we process deletion before insertion, we would do the following: (i) mark $a1$ and $a4$ in the deletion phase; (ii) rederive $a1$ and $a4$; and finally (iii) generate $a4$ that can again be derived based on the inserted fact. On the other hand, if we process insertion before deletion we will (i) generate a new acyclic support for $a4$ (derivation based on the inserted fact is shorter than the earlier derivation of $a4$) (ii) mark $a1$ but do not mark $a4$ due to presence of the new acyclic support. Thus processing insertion first is better than processing deletion first for this example.

Now consider a different change to the program in Figure 2: deleting `e(1,2)` ($f1$) and adding `e(2,6)` ($f9$). Processing insertion before deletion, we will (i) derive a new answer `r(1,6)` based on `r(1,2)` and `e(2,6)`; (ii) mark this new answer along with answers $a1$ and $a4$ in the deletion phase (due to deletion of `e(1,2)`; and (iii) rederive

[1] Two additional examples in Appendix cover subtle aspects not covered by the main example.

all three answers since `r(1,2)` has an alternative derivation. Processing deletion before insertion will mark answers $a1$ and $a4$, and rederive both. Insertion of `e(2,6)` will generate a new answer `r(1,6)`. For this example, processing deletions first performs fewer operations, and is better.

The previous two examples indicate that interleaving the deletion and insertion may be better. In fact, if we delete $f1$ and insert $f8$ and $f9$, it is easy to see that the best change propagation strategy will be to process the insertion of $f8$ first, deletion of $f1$ next and insertion of $f9$ last. This key idea is encoded in our algorithm, where the ordering of operations upon a change is driven by associating events with each operation, priorities with each event, and processing the events in the order of their priorities.

The Event Model. Our algorithm is based on the event model where processing insertion of facts/answers is done using the event *consume_answer* and processing deletion of facts is done using three events called *mark*, *may_rederive* and *rederive*. We maintain two priority queues— *ready queue* and *delay queue* for processing events. Events are scheduled only from the ready queue in increasing order of their SCC numbers - thus all events of an SCC are scheduled before processing events of topologically higher SCC. This ensures that change propagation is processed from topologically lower to higher SCCs. The delay queue consists of events that were originally scheduled but later discovered to be needed only under certain conditions; events in the delay queue may be moved back into the ready queue when these conditions are satisfied.

Within an SCC, *mark* and *consume_answer* events have higher priority that *rederive* and *may_rederive* regardless of their ordinals. We process *mark* and *consume_answer* in ascending order of their ordinals. Among events with the same ordinal, a *mark* event has higher priority over a *consume_answer* event.

Before getting into more detailed description of our algorithm we provide here the key intuition behind interleaving of *mark* and *consume_answer* events. Note that a *mark* event overapproximates the actual answers that need to be deleted, and a *consume_answer* event can generate a support for a new/old answer. Marking of an answer can be avoided if we can generate an acyclic support for the answer using inserted and existed answers, provided the used answers are never going to be marked. The following two requirements guide the design of our local algorithm and choice of ordinals of events and entities.

Requirement 1 *The answers used in generating a new answer or a new support should not be marked in the same incremental phase.*

Requirement 2 *Marking and propagation of marking of an answer should be avoided if an acyclic support of the answer is generated due to insertion of facts.*

Insertion. Generation of a new answer or insertion of a fact/rule generates *consume_answer* events. For instance, when an answer a is added to p's table, we generate a *consume_answer* event for each consumer c of p. The event handler *consume_answer*(a, c) does the work needed to extend the SLG forest when an answer or a fact (a) is consumed by a consumer (c) (Figure 3(a)). If the consumer corresponds to the last subgoal of a rule, the consumption of the answer can generate a new answer (lines 1–9, 12–19), or a new support for an existing answer (lines 1–9, 12, 13, 20–25). Otherwise (i.e. the consumer has a non-empty continuation) it generates a new consumer corresponding to next literal of the clause.

The pseudo-code in Figure 4 describes processing of the new consumer. The *call_check_insert(*g*)* function returns the generator of g, creating a generator if one does not already exist. If a new generator were created, we perform program clause resolution by iterating through all the clauses of the program (lines 1–5). Otherwise we iterate

```
process_event(e=consume_answer(a,c))
1   c=⟨p,h,g,G⟩
2   θ=mgu(a,g)
3   g'=head(Gθ) // g'=true if G is empty
4   G'=tail(Gθ) // G'=null if G is empty
5   a'= hθ //answer generated
6   c'=⟨p,a',g',G'⟩ // new consumer
7   add c' in c.next_consumer, c'.prev_consumer=c
8   add c' in a.imm_affects, c.depends_on=a
9   if(!is_empty(G)) // last subgoal of a clause
10      resolve_goal(c')
11  else
12      is_newanswer=check_insert_answer(p,a')
//checks if a' is in p.answer_table,if not inserts a'
12    is_newanswer=check_insert_answer(p,a')
13    if(is_newanswer)
14      a'.ord=c'.ord+1;
15      ∀c"∈p.consumer
16        if(!marked(c"))
17          create_event(consume_answer(a',c"))
18        else
19          delay_event(consume_answer(a',c"))
20    else
21      if(∀c"∈(a'.supports−{c'})
          (c".ord<a'.ord→marked(c")))
22        if ((c'.ord<a'.ord) && (e.ord<a'.ord))
23          delete_from_ready_queue(mark(a'))
24        else
25          create_event(may_rederive(a'))
```

(a)

```
process_event(mark(a))
1   a.marked=true
2   ∀c', same_scc(a,c'),
3     move_to_delay(consume_answer(a,c'))
4   ∀c∈ a.affected ∧ same_scc(a,c)
5     if(justmarked(c))
6       ∀a', move_to_delay(consume_answer(a',c))
7       if(is_leaf(c) ∧ c.ord<c.answer.ord)
        // acyclic support
8         if(∀c'∈ c.answer.supports−{c}
9           (c'.ord< c.answer.ord→ marked(c')))
        // all other acyclic supports are marked
10            create_event(mark(c.answer))
11            create_event(may_rederive(c.answer))
```

(b)

```
event_loop()
1  while((SC=next_scc(CallSCC_Q))!=NULL)
2    while(!empty(READY_Q,SC))
3      process_event(next_event(READY_Q,SC))
4    ∀a such that a.scc=SC
5      if(a.marked)
         /* do same operation as in mark
          event but for different scc */
```

(c)

```
process_event(may_rederive(a))
1  if(∃c∈a.support s.t. !marked(c))
2    if(∀c'∈ a.support
       (c'.ord<a.ord→ marked(c')))
3      a.ord=max{c".ord | c"∈
         a.supports,!marked(c")}+1
4      create_event(rederive(a))
```

(d)

```
process_event(rederive(a))
1   a.marked=false
2   ∀c, s.t. !marked(c)
3       move_to_ready(consume_answer(a,c))
4   ∀c∈a.affected, same_scc(a,c) ∧ !marked(c)
5       ∀a' s.t. !a'.marked,
6         move_to_ready(consume_answer(a',c))
7   ∀c ∈ a.affected, same_scc(a,c)
8       c.ord = max(c.ord, a.ord)
      // update ordinal of consumers
```

(e)

Fig. 3. Algorithms for Processing Consumer Answer (a), Mark (b), Main (c), May Rederive (d), Rederive (e) events.

through all answers in answer table of g, creating *consume_answer* events for each of them (lines 7–11).

For example, insertion of fact `e(1,5)` [$f8$] generates the event *consume_answer*$(f8, c1)$ which when processed produces a new acyclic support for the already existing answer $a4$ (lines 1–9, 11–13, 21–25 of Figure 3(a)). On the other hand, insertion of `d(1,1)` [$f9$]

is consumed by the consumer $c7$ to generate a new consumer $\langle p1, \texttt{r(1,Y)}, \texttt{r(1,Y)}, [] \rangle$ ($c8$) (lines 1–10). Processing of consumer $c8$ by the function *resolve_goal* creates four *consume_answer* events for $c8$ and each of the answers $a1$, $a2$, $a3$, and $a4$ in generator $p1$'s answer table.

```
resolve_goal(c=⟨p,h,g,G⟩)
1  p'=call_check_insert(g)
2  if(is_newgenerator(p')) //g is a new call
3   ∀ rules α:-β1, β2 . . . , βn s.t. (θ=mgu(g,α)!=φ)
4       c'=⟨p',gθ,β1θ,[β2θ, . . . , βnθ]⟩
        // new consumer
5       resolve_goal(c')
6  else
7    ∀a∈p'.answer_table
8     if(!a.marked)
9          create_event(consume_answer(a,c))
10       else
11         delay_event(consume_answer(a,c))
12  add c in p'.consumer, p'=c.generator
13  calculate_call_graph_incrementally
```

Fig. 4. Algorithm for Function resolve goal

Most of the steps of *consume_answer* are common to traditional SLG resolution. The interesting aspects are the interaction between the effects of insertion and (possibly scheduled) deletion. For instance, when a new acyclic support c' is generated for an answer a' (line 22, first condition) whose other acyclic supports are already marked (line 21) and *mark(a')* event has been scheduled (line 22, second condition) we remove the *mark*(a') from the ready queue since a' cannot be deleted due to c', meeting Requirement 2.

Mark. The mark event for an answer marks a given answer and propagates the effect of marking (Figure 3(b)). If an answer is marked we move all *consume_answer* events (in the same SCC) which would consume the answer from the ready queue to the delay queue (line 2-3). This is due to the fact that if the consumer corresponding to a *consume_answer* event is dependent on a deleted answer then it should not be scheduled (following Requirement 1). The following definitions are used in the marking algorithm:

Definition 3 (Affected set of an answer) *A consumer c is said to be* affected *by an answer a (denoted by $c \in a$.affected) if $c.depends\_on = a$, or $c.prev\_consumer$ is affected by a.*

Definition 4 (Marked consumer) *A consumer c is marked (expressed as $marked(c)$) if either of its premises is marked. A consumer is justmarked (expressed as $justmarked(c)$) if there exists one and only one answer 'a' such that a is marked and $c \in a$.affected.*

Note that the consumers in an affected set of an answer are created due to the presence of the answer. Thus, when an answer is deleted, all its affected consumers must be deleted too. Thus when an answer a is marked we move any *consume_answer* event associated with an affected consumer c (in the same call graph component as a) from the ready queue to the delay queue (lines 4–6). When the last acyclic support of an answer gets marked, we mark the answer and also place a *may_rederive* event for it in the ready queue (lines 7–11).

Scheduling of Events. We now describe the assignment of event ordinals. Based on Requirement 1, we process a *consume_answer*(a, c) only after processing all *mark*(a') events which can affect the consumer c. To ensure this we make the ordinal of *consume_answer* event no less than the ordinal of its consumer. Additionally we need to ensure that a consumer does not consume an answer which can be potentially marked later on. First of all, if an answer belongs to topologically lower SCC than its consumer's SCC, the above condition is satisfied as we process all events in components according to their increasing SCC numbering. Secondly, we ensure that a new answer generated can never be marked

in the same incremental phase. The only remaining case is when the answer a in the same SCC existed before the incremental phase (function *resolve_goal*, lines 8–11), in which case it can be potentially marked. We ensure that the event is processed after a's marking is processed by making the ordinal of *consume_answer*(a, c) event no less than $a.ord$. The SCC number of *consume_answer(a,c)* is same as $c.scc$ and its ordinal is given by

$$\begin{cases} max\{c.ord, a.ord\} & \text{if (same\_scc(a,c)}\wedge \text{ existed\_answer(a))} \\ c.ord & \text{otherwise} \end{cases}$$

Also *mark*(a), *may_rederive*(a), and *rederive*(a) events have SCC number same as $a.scc$ and ordinal $a.ord$. This assignment of ordinals is critical for the following properties of the algorithm.

Property 1. If *consume_answer*(a, c) is a scheduled event, then a is never marked in the same incremental phase.

Property 2. If a' is a new answer and s is a support for an unmarked answer generated while processing *consume_answer*(a, c) (lines 14 and 21, Figure 3(a)) then a' and s are never marked in the same incremental phase.

The correctness of our local algorithm is based on the above two properties. For formal proof of these properties as well as the proof of correctness of the local algorithm refer to [26].

Consider deleting $f1$ =`e(1,2)`, and inserting $f9$ =`e(2,6)` and $f10$ =`d(1,1)` to the example in Figure 2. This generates events $e1$ = *consume_answer*$(f10, c7)$, $e2$ = *consume_answer*$(f9, c3)$, $e3$ = *mark*$(a1)$, and $e4$ = *may_rederive*$(a1)$. Note that although we can process event $e1$ before processing any other event (since $c7$ is not dependent on any answer), we cannot process event $e2$ before processing the mark event $e3$. This is because $c3$ is dependent on answer $a1$ which may be marked when $e3$ is processed. We ensure this by making a consumer's ordinal no less than that of any answer that affects it.

In the above example, using these ordinal assignments we get $e1.ord = c7.ord = 0$, $e2.ord = c3.ord = a1.ord = 1$, and $e3.ord = e4.ord = a1.ord = 1$. As all four events belong to the same SCC we process $e1$ first which generates four events $e5$ = *consume_answer*$(a1, c8)$, $e6$ = *consume_answer*$(a2, c8)$, $e7$ = *consume_answer*$(a3, c8)$, and $e8$ = *consume_answer*$(a4, c8)$. Processing the next event $e3$ moves event $e2$ and $e5$ in the delay queue and generates events $e9$ = *mark*$(a4)$ and $e10$ = *may_rederive*$(a4)$. Event $e6$ is processed next without any effect. Event $e9$ is processed next (since it has the lowest priority with ordinal 2) which moves event $e8$ to the delay queue, followed by event $e7$. The ready queue now contains two *may_rederive* events ($e4$ and $e10$) and the delay queue contains $e2$, $e5$, and $e8$.

Rederivation. When processing a *may_rederive*(a) event, we first check whether the answer a has any unmarked supports left. Subsequently, we make all existing unmarked supports acyclic by raising the ordinal of the answer a to the maximum ordinal of its unmarked supports (Figure 3(e)). We then create *rederive*(a) event which rederives a and propagates this further. The rederivation of a moves all *consume_answer*(a, c) events with an unmarked c from the delay queue to ready queue, thereby undoing the effect of marking in a's call-graph component. Also if any consumer c (in the same SCC that of a) got unmarked due to rederivation of a then all *consume_answer*(a', c) events are moved from the delay queue to the ready queue provided a' is unmarked (Figure 3(e)). Rederivation of

answer a also updates the ordinal of the support that contains a and belongs to the same call graph SCC as that of a.

In the above example, processing the next highest priority event $e4$ creates an event $e11 =$ *rederive*$(a1)$ as the answer $a1$ has an unmarked support $s7$ which is made acyclic by updating the ordinal of $a1$ to that of $s7.ord + 1 = 3$. Processing the next event $e11$ rederives $a1$, moves the events $e2$ and $e5$ to the ready queue, updates the ordinals of supports $s3$, $s4$, and $s5$ to 3. Subsequent processing of remaining events does not reveal any other interesting property of the algorithm and is not discussed here.

Figure 3(c) shows the pseudo code for the scheduling of events. After all events of a component are processed, we propagate the effect of marked answers in the component to topologically higher components. Note that the call graph can change during the evaluation. In our algorithm we permit only addition of edges to the call graph. Hence only two types of changes in the call graph are possible: (i) the topological order of components are changed without changes in any component (Example 3, Appendix); and (ii) components are merged into larger components (Example 4, Appendix). We employ incremental SCC maintenance algorithm of [3, 17] to maintain the call graph SCCs. The correctness of our algorithm depends on the maintenance of an invariant between ordinal numbers of answers and supports within an SCC: that the ordinal of the primary support for an answer is lower than that of the answer itself. Note that it is possible to have an answer $a1$ whose ordinal is lower than that of its premise answer $a2$ if $a2$ belongs to lower topological component than $a1$. Thus when multiple SCCs are merged, the ordinals of the answers and the supports needs to be redistributed within the merged component (details are omitted; see Example 4, Appendix).

4 Results

In this section we describe the optimality of the local algorithm for incremental attribute evaluation and incremental functional program evaluation, and present experimental results on its effectiveness.

Optimal Incremental Attribute Grammar Evaluation. In [21] Reps has presented an optimal algorithm for evaluation of non-circular attribute grammars. The dependency between attribute instances are maintained using an acyclic dependency graph which remains static during the change propagation. In such cases topological evaluation is sufficient to produce an optimal change propagation which means that the number of evaluated attributes is of the order of number of changed attributes. The local algorithm presented in this paper shows the same optimal behavior for non-circular attribute grammar evaluation. In this case each update to an attribute instance is performed using a pair of *consume_answer* and *mark* event. When the dependency between attribute instances (essentially acyclic) is represented by the call graph, topological evaluation of call graph SCCs produces optimal change propagation. Otherwise, when the call graph is cyclic (for left-recursive encoding of grammar), the dependencies between answers represent attribute dependencies. In this case topological scheduling of `consume_answer` and `mark` events allows us to obtain the desired optimal behavior.

Optimal Incremental Evaluation of Functional Programs. We now discuss the optimality of our algorithm when the call graph is acyclic but dynamic. We encounter such graphs when evaluating functional programs (hence non-recursive dependencies) incrementally [1]. We can build an incremental functional program evaluator by writing an interpreter for pure functional programs and evaluating it using our incremental algorithm. Since the call graph is acyclic, topological evaluation suffices. However, since the graph may change over time (due to different outcomes for the conditionals), [1] employs an

optimal dynamic topological order maintenance algorithm using Dietz and Sleator data structures [10]. When the call graph considered in our algorithm is acyclic, our incremental topological SCC maintenance algorithm converges to dynamic topological graph maintenance [1]. Thus we obtain the optimal change propagation algorithm for functional programs.

The complexity of the local algorithm for interpreting a pure functional program is no worse than the complexity of adaptive algorithm for pure functional program evaluation given in [1]. A formal statement and proof about the correspondence between the local algorithm and adaptive functional program evaluation is beyond the scope of this paper. However, we formally state and prove this correspondence in [26].

Experimental Results. The goals of our experiments are to determine the effectiveness of the local algorithm to confine propagation of changes and to determine the runtime overhead of maintaining additional data structures required for scheduling of events. To determine the effectiveness of our algorithm, we measured its performance for flow-sensitive data-flow analysis for C programs. The measurement of overhead is demonstrated by the experiments with flow-insensitive pointer analysis where acyclic supports are sufficient for confining propagation of deletion [24]. The local algorithm is implemented by extending the XSB logic programming system [31] (ver. 2.7.1). Our implementation, experimental setup, benchmark characteristics, and detailed experimental results on parsing and pointer analysis are available in [26].

We evaluated the effectiveness of the local algorithm by performing reaching definition analysis [2] which can be easily encoded in logic programs. The experiments are performed by deleting each assignment statement from the source which effectively deletes all incoming and outgoing flow-edges to the statement and inserts flow-edges from its previous statements to the next statements. The local algorithm is expected to perform well in this case as the effect of deletion of all reaching definitions resulted due to deletion of flow-edges would be nullified by the insertion of flow-edges. We compared the performance of the local algorithm with the performance of from-scratch evaluation and that of deletes-first strategy. The implemented deletes-first strategy first performs all marking and rederivations due to deletion of facts followed by insertion of facts. The marking and rederivation of answers are performed in each call-graph component (in topological order) before the effect of marking is propagated to answers and supports in other components. Thus comparison of the local and the deletes-first strategy demonstrates the effectiveness of insertion of facts to confine the changes due to deletion.

For each benchmark we deleted and restored 250 random assignment statements from the source. The results presented in Figure 5 are averaged over each such deletion of source statement. Note that, the number of inserted and deleted answers is considerably less (8–20 times) for local algorithm compared to deletes-first strategy. Despite the extra overhead of maintaining event priority queues, our preliminary implementation achieves 50-70% reduction in time compared to deletes-first strategy.

In cases where deletes-first strategy is extremely fast, such as pointer analysis, we notice a maximum run-time overhead of 70% for maintaining the extra data structures for topological evaluation in the local algorithm compared to deletes-first strategy. However, we notice that the incremental time of local algorithm varies from 10% to 105% (when all answers are recomputed) of from-scratch time depending on the position of changes. Thus, in special cases for which it is optimal, the local algorithm demonstrates only 5-10% run-time overhead.

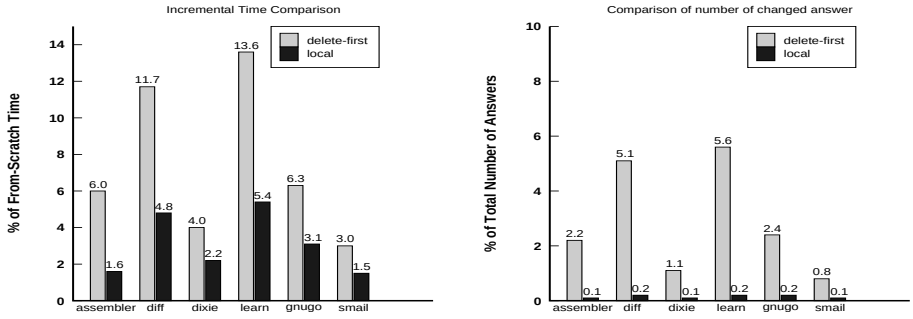


Fig. 5. Incremental Reaching Definition Analysis; Time comparison (a); Change comparison (b).

5 Related Work

The problem of incremental evaluation of table logic program is closely related to the problem of materialized view maintenance which has been extensively researched (see, e.g. [11, 15] for surveys) in database community. Most of the works in recursive view maintenance generate rules that are similar in spirit to those of DRed [12] and are subsumed by DRed (as compared in [11]). DRed computes the dependencies between answers using rules derived from the original program and does not maintain any dependency structure. The algorithm processes multiple changes by first considering all deleted facts followed by all inserted facts. Deletion of facts is handled in two phases - the deletion phase marks an answer if *any* of its supports is deleted, thereby over-propagating the effects of a deletion, and subsequently the rederivation phase rederives an answer if it has an unmarked support. The DRed-like strategy is also used in incremental analysis such as model checking [28], pointer analysis [33], MOD analysis [34], and data-flow analysis [18].

Our primary-support-based algorithm [22] improved on the DRed strategy by significantly reducing the need to propagate deletions. In [23] we extended the concept of primary support by identifying acyclic supports for every answer, all of which should be deleted before the answer can be marked. The space overhead due to the support graphs is mitigated by representing them symbolically [24]. The local algorithm presented in this paper further optimizes and extends the deletion mark propagation: (i) using the effect of insertion of new facts and answers which is very useful in updates where insertion and deletion occur hand-in-hand; and (ii) by scheduling rederivation of answers in each call graph component, ensuring that topologically lower components are stabilized before the effects are propagated to a higher component. In our earlier works, incremental insertion was done by evaluating difference rules [16] (obtained by program transformation) which are evaluated top-down. In contrast, in this paper we presented a combined bottom-up algorithm to handle both insertions and deletions.

Recently, we developed an algorithm for incremental evaluation for arbitrary tabled Prolog programs including those that use Prolog built-ins, cuts, aggregation and non-stratified negation [25]. That algorithm maintained a much coarser dependency structure based on calls. The algorithm presented in this paper maintains finer grained dependency structures based on answers. The results of [25] show that answer-based approaches perform significantly better, but cannot be easily extended beyond pure programs. More-

over, as this paper shows, fine-grained dependencies are needed to achieve or at least approach optimal performance for incremental evaluation. Integrating the fine-grained local algorithm so that it can be deployed wherever applicable within the more general setting of [25] is an interesting open problem.

The idea of using SCC-reduced dependency graphs to optimize propagation of changes has been seen in various past works [7, 13, 14, 19, 30]. Among these, Hermenegildo et. al.'s works [13, 19] on re-analyzing (constraint) logic programs are closest to our work. Our event based description for modeling the main aspects of memoized logic program has been inspired by their work. These papers consider one answer pattern per call, and propagation is controlled based on the call graph. In [19] insertion events are processed in such a way that lower components are stabilized before their effect is propagated to higher ones without explicitly computing the SCCs. However, since the SCCs are themselves dynamic, the event ordering only approximates the SCC ordering. In our approach we maintain call graph SCCs explicitly, similar to [13]. However, we use event ordering to control propagation of changes *within* an SCC, leading to finer-grained interleaving between insertion and deletion operations.

6 Concluding Remarks

We presented an efficient algorithm for incrementally evaluating definite logic programs with the rules/facts of the program being changed: added, deleted, or updated. The key to the algorithm is the interleaving of insertion and deletion operations based on an order that generalizes those based on call dependency graphs. The algorithm naturally generalizes techniques that were developed in settings where dependencies are non-recursive (e.g. attribute grammars, functional programs).

The algorithm maintains dependencies between calls, answers and intermediate structures used for resolution and propagates insertions and deletions bottom-up through this graph. This enables us to adapt our algorithm to handle programs with stratified negation, processing one stratum at a time, and processing lower strata completely before propagating its effects to the higher ones.

We can also extend our technique to handle programs with non-stratified negation under the well founded semantics as follows. With each consumer, we can keep the delay list containing negative literals [5], mark a negative literal if an answer is added to its corresponding positive literal and resolve the negated literal if all answers are deleted from the positive literal. Note that interleaving of insertion and mark propagation is essential to handle programs with non-stratified negation.

The focus of this paper has been on developing a uniform algorithm to treat all forms of changes— additions, deletions and updates— to facts as well as rules in a logic program. Although the algorithm maintains extensive dependency information, we believe that techniques such as those used in symbolic support graphs [24] can be used to compactly store the dependencies.

References

1. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *POPL*, volume 37, pages 247–259, New York, NY, USA, 2002. ACM Press.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, pages 585–718. Addison-Wesley, 1986.
3. B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Symposium on Discrete algorithms*, pages 32–42, 1990.
4. R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *ILPS*, 1993.
5. W. Chen, T. Swift, and D. S. Warren. Efficient implementation of general logical queries. *JLP*, 1995.

6. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1):20–74, 1996.
7. C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, volume 3576 of *LNCS*, pages 449–461, Edinburgh, Scotland, July 2005.
8. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems —a case study. In *ACM PLDI*, pages 117–126, 1996.
9. A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, pages 105–116. ACM Press, 1981.
10. P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC*, pages 365–372, New York, NY, USA, 1987. ACM Press.
11. A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and applfications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
12. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.
13. M. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental analysis of constraint logic programs. *ACM Trans. Program. Lang. Syst.*, 22(2):187–223, 2000.
14. L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Trans. Program. Lang. Syst.*, 12(3):429–462, 1990.
15. E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *ER Workshops*, pages 62–73, 1999.
16. R. Paige and S. Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, 1982.
17. D. J. Pearce and P. H. J. Kelly. Online algorithms for topological order and strongly connected components. Technical report, Imperial College, London, 2003.
18. L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng.*, 15(12):1537–1549, 1989.
19. G. Puebla and M. V. Hermenegildo. Optimized algorithms for incremental analysis of logic programs. In *SAS*, pages 270–284, 1996.
20. C. R. Ramakrishnan et al. XMC: A logic-programming-based verification toolset. In *CAV*, number 1855 in LNCS, pages 576–580, 2000.
21. T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *POPL*, pages 169–176, New York, NY, USA, 1982. ACM Press.
22. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *ICLP*, volume 2916 of *LNCS*, pages 389–406, 2003.
23. D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP*. ACM Press, 2005.
24. D. Saha and C. R. Ramakrishnan. Symbolic support graph: A space-efficient data structure for incremental tabled evaluation. In *ICLP*, volume 3668 of *LNCS*, pages 235–249, 2005.
25. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled prolog: Beyond pure logic programs. In *PADL*, volume 3819 of *LNCS*, pages 215–229. Springer, 2006.
26. D. Saha and C. R. Ramakrishnan. A local algorithm for incremental evaluation of logic programs, 2006. Available at `http://www.lmc.cs.sunysb.edu/~dsaha/local`.
27. R. Seljee and H. de Swart. Three types of redundancy in integrity checking; an optimal solution. *Journal of Data and Knowledge Enigineering*, 30:135–151, 1999.
28. O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, volume 818 of *LNCS*, pages 351–363, 1994.
29. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, pages 84–98, 1986.
30. J. A. Walz and G. F. Johnson. Incremental evaluation for a general class of circular attribute grammars. In *PLDI*, pages 209–221, New York, NY, USA, 1988. ACM Press.
31. XSB. The XSB logic programming system. Available at `http://xsb.sourceforge.net`.
32. G. Yang and M. Kifer. FLORA: Implementing an efficient DOOD system using a tabling logic engine. In *International Conference on Computational Logic*, volume 1861 of *LNCS*, pages 1078+, 2000.
33. J. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *ICSE*, pages 442–451, 1999.
34. J. Yur, B. G. Ryder, W. Landi, and P. Stocks. Incremental analysis of side effects for C software system. In *ICSE*, pages 422–432, 1997.

A Appendix

The content of this section is not essential for reviewing this paper. The examples presented in the section describes certain subtleties of the algorithm developed in Section 3 which could not be explained by the main example presented in the paper.

Example 3. We present another example to illustrate the handling of dynamic call graph SCC by the algorithm. In Figure 6 we present the example of right recursive transitive closure. Note that each producer creates a trivial SCC in the call graph. The initial topological ordering of the components is shown along with each call.

Consider a scenario of updating `e(1,3)` to `e(1,2)` and `e(2,3)` to `e(4,3)`. This effectively creates the following six events in the ready_queue given below.

Event	Ordinal	SCC
$[e1]$ consume_answer(f4,c1)	0	2
$[e2]$ consume_answer(f4,c2)	0	2
$[e3]$ mark(a1)	1	2
$[e4]$ consume_answer(f5,c10)	0	3
$[e5]$ consume_answer(f5,c11)	0	3
$[e6]$ mark(a2)	1	4

Below we describe processing of each event step-by-step. An event is deleted from the ready_queue when it is chosen for processing.

Step 1: Processing event $e1$ generates a new answer `r(1,2)` (say $a4$) with a new support $s4 = \langle e(1,2)\rangle$.

Step 2: Processing event $e2$ generates a new consumer $c12 = \langle p2, r(1,Y), r(2,Y), []\rangle$. This adds a new call edge from $p3$ to $p2$ which reassigns topological ordering of calls $p3$ and $p5$ to 1.5 and 0.5 respectively. This makes the priority of the event $e4$, $e5$, and $e6$ greater than that of $e3$. As answers $a2$ and $a3$ already existed in the answer tabled of the call `r(2,Y)` (producer p3), two new events $e7$=`consume_answer(`$a2$`,`$c12$`)` and $e8$=`consume_answer(`$a2$`,`$c12$`)` are created. The changed ready_queue is shown below.

```
:- table r/2.
r(X,Y) :- e(X,Y).
r(X,Y) :- e(X,Z), r(Z,Y).
e(1,3). % f1
e(2,3). % f2
e(2,4). % f3
```

(a)

```
Calls: Producers
factbase [p1]
r(1,Y)   [p2]
r(2,Y)   [p3]
r(3,Y)   [p4]
r(4,Y)   [p5]
    Answers
r(1,3).  [a1]
r(2,3).  [a2]
r(2,4).  [a3]
```

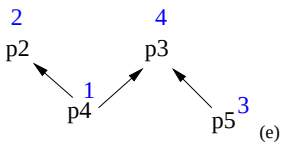
(b)

```
[c1] <p2,r(1,Y),e(1,Y),[]>
[c2] <p2,r(1,Y),e(1,Z),[r(Z,Y)]>
[c3] <p2,r(1,Y),r(3,Y),[]>
[c4] <p3,r(2,Y),e(2,Y),[]>
[c5] <p3,r(2,Y),e(2,Z),[r(Z,Y)]>
[c6] <p3,r(2,Y),r(3,Y),[]>
[c7] <p3,r(2,Y),r(4,Y),[]>
[c8] <p4,r(3,Y),e(3,Y),[]>
[c9] <p4,r(3,Y),e(3,Z),[r(Z,Y)]>
[c10]<p5,r(4,Y),e(4,Y),[]>
[c11]<p5,r(4,Y),e(4,Z),[r(Z,Y)]>
```

(c)

```
e(1,2) [f4]    e(4,3) [f5]
```

(d)



(e)

Fig. 6. Example program (a), calls and answers (b), consumers (c), new facts (d), and call graph (e).

Event	Ordinal	SCC
$[e4]$ consume_answer(f5,c10)	0	0.5
$[e5]$ consume_answer(f5,c11)	0	0.5
$[e6]$ mark(a2)	1	1.5
$[e7]$ consume_answer(a2,c12)	0	2.0
$[e8]$ consume_answer(a3,c12)	0	2.0
$[e3]$ mark(a1)	1	2

Step 3: Processing event $e4$ generates new answer $a5$=`r(4,3)` (a5.ord=1) with a new support $s5$=`e(4,3)` (s5.ord=1). This generates a new event $e9$=`consume_answer(`$a5$`,`$c7$`)` (ordinal=0, SCC=1.5) which gets higher priority than $e6$.

Step 4: Processing of event $e5$ generates new consumer $c13 = \langle p5, r(4,Y), r(3,Y), [] \rangle$. Since the goal r(3,Y)'s producer is $p4$, this adds a new call-graph edge $p4$ to $p5$. By applying incremental SCC maintenance the SCC of $p4$ is reduced from 1 to 0.25 (a number lower than p5's SCC number 0.5).

Step 5: The next event processed is $e9$. This generates a new support $s6 = \langle e(2,4), r(4,3) \rangle$ for an already existed answer $a2 = r(2,3)$. As both the parts of the support $s6$ belong to different SCC from that of $s5$, s6.ord is 1. As $a2.ord = 1$, $s6$ is an acyclic support of $a2$. As $a2$ can be derived using $s6$ we delete $e6 = mark(a2)$ event from the ready_queue (Lines 16-20 Figure 3(a)).

Step 6: Processing event $e7$ generates a new support
$s7 = \langle e(1,2), r(2,3) \rangle$ for $a1 = r(1,3)$ with ordinal 1. This removes the event $e3 = mark(a1)$ from the ready_queue as in previous step.

Step 7: Finally processing event $e8$ generates a new answer $r(1,4)$ with its primary support $s8 = \langle e(1,2), r(2,4) \rangle$. The incremental phase stops here having no other events in the ready queue to process. □

Example 4.

We present another example to illustrate the handling of dynamic call SCC (where SCCs are merged) by the algorithm. In Figure 7 we present the example of right recursive transitive closure. The initial topological ordering of the components is shown along with each call.

Consider a scenario of adding fact `e(2,1)`(f3). This effectively creates the following two events in the ready_queue given below.

Event	Ordinal	SCC
$[e1]$ consume_answer(f3,c4)	0	2
$[e2]$ consume_answer(f3,c5)	0	2

Below we describe processing of each event step-by-step. An event is deleted from the ready_queue when it is chosen for processing.

Step 1: Processing event $e1$ generates a new answer `r(2,1)` (say $a4$) with a new support $s4 = \langle e(2,1) \rangle$.

Step 2: Processing event $e2$ generates a new consumer $c9 = \langle p3, r(2,Y), r(1,Y), [] \rangle$. This adds a new call edge from $p2$ to $p3$. As there exists an edge from $p3$ to $p2$, adding the new edge combines the two trivial SCCs of producer $p2$ and $p3$ to non-trivial SCC containing $p2$ and $p3$. This changes the `SCC` attribute of c1-c6, a1-a3 and s1-s3 to 3. The important point is to note that the attribute `ordinal` needs to be changed due to the merging of the SCCs. For example, as $s2$ and $a3$ now belongs to the same SCC the ordinal of $s2$ is increased to 1 and consequently the ordinal of $a2$ is also changed to 2. This is done in the function *calculate_call_graph_incrementally*. The attribute adjustment

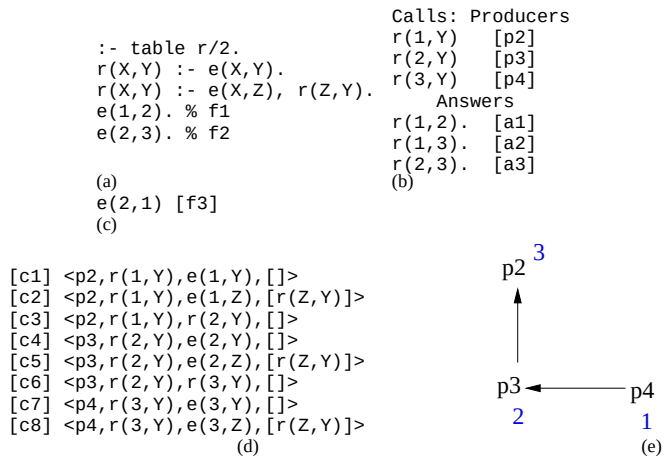


Fig. 7. Example program (a), calls and answers (b), new facts (c), consumers (d), and call graph (e).

process increases the ordinals of a support s which belongs topologically higher SCC than an answer a such that $s.ord < a.ord$ and $s \in a.\mathit{affected}$. If the increase in $s.ord$ makes $s.ord \geq s.answer.ord$ and s is the only acyclic support of $s.answer$ then we increase s.answer.ord to $s.\mathit{ord}+1$, otherwise the ordinal of $s.\mathit{answer}$ is not changed which means s changes to non-acyclic support. Also note that the propagation of this increase of attribute due to merging of SCCs can at most propagate to the boundary of new merged SCC. The example shows this case. All the other steps are not further discussed as they do not illustrate any further aspect of the algorithm.

□