Incremental Evaluation of Tabled Logic Programs

a dissertation presented

by

Diptikalyan Saha

to

The Graduate School

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

computer science

Stony Brook University

December 2006

Stony Brook University

The Graduate School

Diptikalyan Saha

We, the dissertation committee for the above candidate for the degree of Doctor of Philosophy, hereby recommend acceptance of this dissertation.

Professor C.R. Ramakrishnan, Advisor
Department of Computer Science

This dissertation is accepted by the Graduate School.

Dean of the Graduate School

Abstract of the Dissertation

Incremental Evaluation of Tabled Logic Programs

by

Diptikalyan Saha

Doctor of Philosophy

in

Computer Science

Stony Brook University

2006

Tabled logic programming has emerged as an important evaluation technique of logic programs. Tabling has enabled us to construct many practical applications - program analysis and verification systems in particular — by encoding them as high-level logic programs. Tabled resolution based systems evaluate programs by memoizing subgoals (referred to as calls) and their provable instances (answers). In this thesis we address the problem of efficiently updating the memoized information in tables with respect to the changes in programs due to addition/deletion of facts/rules. Such capability of incremental maintenance of memoized information facilitates the use of tabled logic programming for scalable program analysis, where the analysis information can be updated efficiently in response to small changes to the analyzed program.

Tabled resolution based systems process the addition of facts in semi-naive fashion and thus incremental by nature. We address the problem of maintaining tables for definite logic programs in response to deletion of facts/rules by maintaining an auxiliary and-or data structure called support graph. Each support in a support graph represents an immediate reason for derivation of an answer. Support graph thus maintains the dependency between answers and facts, and can be used to propagate the effect of deletion of facts/rules. We developed heuristics to reduce the over-approximation performed by existing algorithms for handling deletion.

Support graph based algorithms update tables extremely fast but impose considerable space overhead for large applications. Our general solution to space-overhead problem is based on a data structure called Partial Support Graph (PSG) which keeps bounded number of supports for every answer. For a special class of programs we present a more efficient algorithm than PSG, based on a compact data structure called Symbolic Support Graph which exploits the commonality between supports.

We also present an efficient algorithm for handling updates to the facts by carefully interleaving the insertion and deletion operations generated by updates. We present an incremental tabled maintenance algorithm for handling programs with negation, cuts, and aggregation operators. We demonstrate the efficiency of incremental computation to various problems such as pointer analysis, data-flow analysis, push-down model checking, parsing, dynamic programming, and XML validation.

To My Parents

Contents

List of Figures

List of Tables

Acknowledgements

I would like to sincerely thank all those who gave me support and encouragement to complete this thesis.

I met Prof. C. R. Ramakrishnan in the Computing with Logic course in the first semester. His great teaching skill and in-depth knowledge in the subject attracted me to the field of Logic Programming. Consequently, he became my thesis advisor. His quick thinking and enthusiasm have given me great insight on this research, and motivated me to work on this thesis. Along with his close guidance, he gave me freedom to work alone and explore different areas which helped me grow in confidence and maturity. Without his help and immense encouragement it was not possible to achieve the high quality of this research.

I started my research on formal methods and verification under close guidance of Prof. Scott Smolka. I am deeply indebted to him for his support on the verification research. I admire his lucid style of writing and I always try to reach that perfection. His advice on reading 'The Elements of Style' to improve my technical writing ability was very helpful.

I would like to extend my thanks to Prof. Annie Liu, who gave me great suggestions on the research of incremental evaluation. I first interacted with her while I was a teaching assistant to her and later on various meetings with her reading group. Her in-depth knowledge in the area of incremental evaluation has helped me improve the quality of this thesis. Over these years, she has given constructive comments on my teaching ability and presentation skills.

I am grateful to Prof. David Warren, who served in my thesis committee and gave many suggestions to improve the quality of this thesis. He has helped me to implement various incremental algorithms into XSB system.

I am also indebted to Dr. Kedar Namjoshi for taking time out of his busy schedule and agreeing to be my external committee member. His suggestions have made this thesis more readable and technically sound.

I would like to express my gratitude to Prof. R. Sekar who gave me the opportunity to work in the Model Carrying Code project. My thanks to Prof. Scott Stoller who served in my thesis committee. Discussion on various research areas with him has broaden my knowledge in different areas of Computer Science. Prof. I.V. Ramakrishnan was very helpful in giving useful tips to me about almost everything. His presence had made LMC lab a fun place to work in.

It was a pleasure for me to attend lectures and seminars taught by the excellent faculty of Computer Science Department. I especially enjoyed the year-long course work on Algorithms with Prof. Michael Bender. He has been a great influence on my teaching and presentation skills.

I had the opportunity to interact with Edwina Osmanski. She has been very kind to me all these years. I enjoyed those chit-chats with her in between her service to the graduate students. Because of those interesting conversations, I continued getting paycheck from Edwina for five years instead of opting for the direct deposit. I acknowledge the help and support from other administrative staffs, in particular Betty Knittweis and Kathy Germana. I extend my sincere thanks to entire staff in system administration.

It was my pleasure to interact and collaborate with some fine people outside Stony Brook: Prof. Haifeng Guo, Prof. Gopal Gupta, Dr. Prasad Rao, Dr. Raj Rajagopalan, Dr. Yow-Jian Lin, Matthias Kahl, Michael Eichberg, Prof. Mira Mezini, and Prof. Klaus Ostermann.

It was a great pleasure to meet some new and old friends in Stony Brook. They have been very helpful to me and because of them this journey has turned out to be enjoyable and eventful. I would like to sincerely thank my officemate and mentor Prof. Samik Basu with whom I did my research on software verification. Without my friend Giridhar Pemmasani it would have been hard for me to work in Linux. I am greatly indebted to Luis De Castro for his implementation of Support Graph in XSB. I sincerely acknowledge Wei Xu for sharing his implementation of C-to-Prolog converter. I sincerely thank Anu Singh who carefully went through this entire thesis and made it more readable. I am greatly indebted to my friends: Rahul Agarwal, Pradipta De, Arnab Ray, Bikram Sengupta, Saikat Mukherjee, Beata Sarna-Starosta, Ping Yang, Zan Sun, Wenxin Song, Ashish Raniwala, Srikant Sharma, Ningning Zhu, Amit Sasturkar, Alok Tongaokar, Paul Fodor, Prem Uppuluri, V.N. Venkatakrishnan, Ajay Gupta, Shruthi Murthy, Yunfan Bao, Sandeep Bhatkar, Pranav Nawani, Abhijit Sengupta, Puloma Mukherjee, Dominic Geiger, and many more.

Finally, this thesis would not have been completed without the moral support and encouragement of my parents. I dedicate my thesis to them.

Chapter 1

Introduction

Rule-based specifications are used in variety of application domains. Examples include security policy specification, program analysis, trust management, vulnerability analysis, model checking, business logic specification, etc. Typically in these kind of applications, known information about the domain is expressed in a knowledge base and rules are evaluated over this knowledge base to derive new information to accomplish the goals of the given application. For example, consider the following rule-based specification of reaching definition analysis from the domain of data-flow analysis.

```
gen(Stmt,(Var,Stmt)):-  assign(Stmt,Var,_Expr).
kill(Stmt,(Var,_AnyStmt)):- assign(Stmt,Var,_Expr).
in(Stmt,Def):- pred(Stmt,PrevStmt), out(PrevStmt,Def).
out(Stmt,Def):- gen(Stmt,Def).
out(Stmt,Def):- in(Stmt,Def), not kill(Stmt,Def).
```

(a)

```
assign(s1,a,b). % s1: a=b
assign(s2,c,a). % s2: c=a
pred(s2,s1).
```

(b)

Figure 1: Rule-based specification of reaching definition analysis (a), example input relations (b)

The above rules identify the variable definitions that may reach a statement, given a control-flow graph. The input control-flow graph is specified using the binary relations *pred* and *assign*. An example input relation is shown in Figure 1(b). The first rule in Figure 1(a) states that a definition (a variable, statement tuple) is generated (given by the relation *gen*) at a statement *Stmt* if the *Stmt* is an assignment statement which assigns value to the variable *Var*. The second rule specifies that due to the assignment to the variable *Var* all definitions of variable *Var* are killed at statement

Stmt. The binary relation *in* defines that all definitions coming *out* of a statement *PrevStmt* reach the statement *Stmt*, provided *PrevStmt* is a predecessor of *Stmt* in the control-flow graph. Definitions coming *out* of statement are either generated at that statement (fourth rule), or reaches the statement and not killed by it (fifth rule). The given rules, also known as intensional relations, derive the relations *gen*, *kill*, *in*, and *out* from the input relations (known as extensional relation) *assign* and *pred*.

Rules can be encoded using logic programming [Llo84]. In such a language rules are encoded as clauses and extensional relations are called facts. The logic program interpreter deduces the logical consequences of the clauses via the process of inference. Prolog is the most widely used logic programming languages. Note that the rules in the Figure 1 are encoded using Prolog syntax. The left hand side of a rule is called *head* and right hand side of a rule is called *body.* A body may contain conjunction (expressed using comma) of one of more predicates or negation (expressed using not) of predicates.

When it comes to query evaluation, logic programs can be evaluated in two ways: bottom-up and top-down algorithms. The bottom-up approach starts with known facts and extends of set of true derived facts using rules. Thus, it can derive new facts from the old facts and rules. The process repeats until no more facts can be derived. The query is then resolved against all the computed set of facts. On the other hand, top-down evaluation starts from a goal which is reduced to subgoals using the rules. The process goes on until the facts are reached.

Many debates exist on the comparative advantages of bottom-up and top-down evaluation techniques of query evaluation [Bra95]. Bottom-up evaluation guarantees computation of least Herbrand model [Llo84] for positive (definite) logic programs. Also due to the use of set oriented operations it performs well in terms of I/O operations. Thus it is suitable for large applications in deductive databases. In contrast, top-down evaluation is goal oriented and hence it generally computes less redundant facts, making it efficient. This makes top-down evaluation the standard method of computation in query evaluation framework like Prolog.

Prolog's query evaluation strategy uses a top-down theorem-proving approach, namely SLD resolution [Llo84]. It employs a backtracking search through the tree of SLD refutations using top-to-bottom rule selection strategy, and left-to-right subgoal selection strategy in the body of a rule. One of the main disadvantages of Prolog's search strategy for computing SLD tree is its susceptibility to infinite looping. The problem is evident for evaluation of left-recursive rules. Another disadvantage of Prolog's computational strategy is its tendency to recompute the same answers. Consider the left-recursive transitive closure program given in Figure 2(a) with the facts in Figure 2(b). When asked a query `reach(0,X)` to know all the nodes that are reachable from node 0, Prolog resolves the query against the first clause to produce answers `X=1` and `X=2` using the facts `edge(0,1)` and `edge(0,2)`, respectively. In search for getting more answers, backtracking through the second clause will generate the same query `reach(0,X)` which again produces the same answers and goes into the same loop. The problem becomes worse if the order the clauses are changed. In this case, Prolog's resolution will not produce any answer, and moreover, will not terminate.

One of the solution to the problem is based on SLG resolution [CW96], also called tabled resolution as it uses memoization or tabling to overcome the problem of weak termination, repeated subcomputation, and incomplete semantics for negation. These features of tabled resolution have been exploited to build variety of systems for finite- and infinite-state verification [RRR$^+$97, Ram00, BKPR02], program analysis [DRW96, SR05], security analysis [SSS04, OGA05], and implementation platform for more expressive logic such as F-logic [YK00].

Tabled resolution-based systems evaluate programs by memoizing subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a set of tables. Traditionally, the systems keep all calls in a *call table*. For each subgoal in the call table, its provable instances are kept in an *answer table*. During resolution, if a subgoal is present in the call table, it is resolved against the answers recorded in the corresponding answer table; otherwise the subgoal is entered into the call table, and its answers, computed by resolving the subgoal against program clauses (rules), are also entered in the answer table. In the above example, the `reach` predicate is defined as a tabled predicate. When asked a query `reach(0,X)`, tabled resolution will store the call `reach(0,X)` in the call table. Backtracking through the first clause will generate the answers `X=1` and `X=2` which are then entered into the answer table corresponding to call the `reach(0,X)`. While resolving the second clause, tabled resolution resolves the first subgoal `reach(0,Z)` with the answers already there in its

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

(a)

```
edge(0,1).
edge(0,2).
edge(1,1).
edge(1,2).
```

(b)

```
edge(0,1).
edge(0,2).
edge(1,1).
edge(1,2).
edge(2,3).
```

(c)

Figure 2: Example reachability program (a) and two `edge/2` relations (b & c).

answer table. This generates two subgoals `edge(1,X)` and `edge(2,X)` corresponding to the second conjunct in the body. The subgoals do not generate any new answer and the computation terminates.

1.1 Problem Addressed in this Thesis

When the logic program changes, by addition/deletion of facts/rules, the tables become stale: they may not have all the answers or the answers in the tables may be incorrect. For instance, consider the evaluation of the query `reach(0,X)` over the program in Figure 2(a) using the definition of `edge/2` relation shown in Figure 2(b). Tabled evaluation will create an answer table for call `reach(0,X)` with {`X=1`, `X=2`} as the answers. Subsequent invocation of `reach(0,Y)` will simply resolve the subgoal against the answers in the table, returning {`Y=1`, `Y=2`} as answers.

Now let a new tuple `edge(2,3)` be added to the `edge/2` relation resulting in the `edge/2` relation given in Figure 2(c). Note that the answer table for call `reach(0,X)` contains only answers {`X=1`, `X=2`} and hence is stale. Invocation of, say `reach(0,Z)`, will return only answers `Z=1` and `Z=2`, and miss the answer `Z=3`. The problem becomes worse if tuples can be deleted. If the tuple `edge(0,1)` is deleted from the `edge/2` relation in Figure 2(b), the query `reach(0,Z)` will still return answers `Z=1` and `Z=2`, even though `reach(0,1)` is no longer true!

Tabling systems currently provide no mechanism to refresh the tables after a change to the program. To handle such changes to the program, all tables are removed after an update to the program, and then the query is reissued. This approach is called *from-scratch* evaluation. This approach is clearly wasteful, especially if the changes to the program are small. For instance, in the above example, after the addition of `edge(2,3)`, the subgoal `reach(0,Z)` and its answer table must be removed and recomputed, deriving answers {`Z=1`, `Z=2`, `Z=3`}, in effect *rederiving* answers `Z=1` and `Z=2`.

An efficient way of maintaining “freshness” of the tables is to compute the changes in the tables in response to changes in facts and rules. This approach of computing changes to the result in response to changes in the input is known as *incremental computation*. The goal of efficient incremental computation is to compute the changes as efficiently as possible. In most cases small changes to the set of facts (hereafter called factbase) result in small changes to the tables, and in those cases incremental evaluation should be considerably faster than from-scratch evaluation strategy.

The goal of this thesis is to develop a general platform to perform efficient incremental rule-based evaluation. As many problems can be naturally expressed in terms of rules, they can benefit from incremental computation. Particularly, program analysis problems can be easily formulated as inference rules and the analysis information can be computed using logic program query evaluation. Hence, incremental rule evaluation will be particularly useful in iterative process of software development, where it is better to incrementally analyze a program as it changes, than recompute the analysis information from scratch after every change. In this thesis, program analysis problems is used for measuring the effectiveness of incremental evaluation.

The idea of incremental computation is used in various fields of computation including attribute grammar evaluation, functional programming, constraint logic programming, program analysis, data-flow analysis, truth maintenance systems, and model checking. In databases, the results of the query may be stored in the databases and are known as materialized views. When base relation changes, the materialized views need to be updated too. Thus the problem of maintenance of materialized views is closest to the problem of incremental maintenance of tables. Most of the earlier works on maintaining views in databases work only with non-recursive queries, or work with a cost model where disk access cost dominate all else. Hence they do not apply to the maintenance of in-memory tables in a rule-processing engine where rules are typically recursive. A discussion on these related works appears in Chapter 2 in this thesis.

1.2 Overview of Our Approach

1.2.1 Handling Addition

Bottom-up logic program evaluation strategy such as semi-naive [Ull89] is incremental with respect to addition of facts. Similarly, top-down goal-oriented techniques such as those based on SLG handle addition of facts incrementally. In SLG evaluation model, a subgoal that causes answers to be added to the tables is called a producer, and a subgoal which is resolved against answers already in the tables is called a consumer. The evaluation engine maintains auxiliary data structures to ensure that no consumer sees an answer more than once: e.g. environments to produce and consume answers and control structures linking answer producers to answer consumers. These data structures are torn down when all answers to a call have been derived, an operation that is crucial to memory efficiency of top-down evaluators. Owing to the deletion of such data structures, SLG evaluation strategy cannot perform incremental addition of facts after all answers to all calls have been derived.

In this thesis, we take three approaches for handling addition of facts. One of the approaches is to retain the data structures linking producers to consumers and perform incremental addition of facts as done in non-incremental evaluation. This process however imposes space overheads which make it unscalable unless specialized data structures are used to compactly represent such structures. The second approach is based on finite-differencing [PK82] where rules are generated to capture the new answers due to addition of facts. For instance, the changes to the `reach/2` relation can be computed by evaluating the predicate Δ`reach/2` defined as follows (where the additions to `edge/2` are given by the Δ`edge/2` relation): however, direct evaluation

```
Δreach(X,Y) :− Δedge(X,Y).
Δreach(X,Y) :− Δreach(X,Z), edge(Z,Y).
Δreach(X,Y) :− (reach(X,Z); Δreach(X,Z)), Δedge(Z,Y).
```

of these auxiliary rules will lead us two distinct tables for `reach` and Δ`reach`, and the two tables must be merged after the incremental phase. We describe a data structure that enables the two predicates to share the same table, eliminating most of the overheads of incremental evaluation. Our final approach of handling addition of facts is based on call dependency where only calls that are affected by the addition of facts are evaluated from scratch. In the following chapters we elaborate on the motivation, applicability and description of these approaches.

1.2.2 A Time-Efficient Technique for Handling Deletion

Among the techniques that handle recursive queries, the DRed algorithm [GMS93], is the most general one used to maintain materialized views. This bottom-up (or forward-chaining) algorithm considers two kinds of changes: addition to and deletions from the set of facts. The algorithm works in three phases. In the first phase, all the derived answers that are dependent on the deleted facts are marked as possibly deleted. In the second phase, some of these marked answers that have alternative derivations are rederived. In the third phase, new answers due to the added facts are computed. The algorithm, however, has unacceptable time overhead in practice since the dependencies between answers, rules and facts are computed afresh at each incremental step. Moreover, the first phase may eagerly mark a large number of answers, only to rederive them in the second phase. In the example given in Figure 2(a) and (b), if `edge(1,1)` is deleted after evaluation of the query `reach(0,X)`, the answers `reach(0,1)` and `reach(0,2)` are going to be deleted and subsequently rederived.

Our algorithm for handling deletion of facts and rules improves over DRed in two respects. First, in addition to the tables themselves, we maintain a structure called the *support graph* that keeps with each answer the proximal cause of its derivation. For instance, if an answer `reach(0,1)` was derived using the rule `reach(X,Y) :- edge(X,Y)` and the answer `edge(0,1)`, then the set $\langle edge(0,1)\rangle$ is maintained as a *support* of `reach(0,1)`. Unlike DRed, support graph materializes the dependency between answers and facts instead of computing them using auxiliary rules. Second, we identify supports that are *acyclic* and can be used to finitely construct a proof. For instance, let $\langle reach(0,1), edge(1,2)\rangle$ and $\langle edge(0,2)\rangle$ be two supports for `reach(0,2)`, and let $\langle edge(0,2)\rangle$ be the support generated when `reach(0,2)` was derived for the first time. Hence we know that the support $\langle edge(0,2)\rangle$ can be used to build a proof for the answer `reach(0,2)`. The first support for an answer is always acyclic and is called its *primary support*. When a fact is deleted, we mark all the supports that this fact participates in as deleted, and propagate the deletion marking through the support graph. An answer is not marked as long as it has at least one unmarked acyclic support. This stops the propagation of deletion marks through the support graph early. For example, deletion of fact `edge(1,1)` will not mark any answers as the primary support $\langle edge(0,1)\rangle$ of the answer `reach(0,1)` is unmarked. We find that using primary supports alone, we can reduce the number of answers deleted and later rederived by over two orders of magnitude. Since the support graph is explicitly maintained, the propagation of deletion marks and any necessary rederivation is done in constant time per answer. We find that the time for incremental computation of rules per source statement deletion is typically less than 1% of the time it takes for from-scratch evaluation in our experiments with pointer analysis. The support graph based deletion algorithms are discussed in Chapter 4.

1.2.3 Space-Time Tradeoff

Although the acyclic support-based algorithm is time efficient, the explicit storage of support graph imposes considerable space overheads. Our first approach to deal with this problem is based on maintaining a *partial support graph* which contains only a bounded number of supports for each answer. Although the size of partial support graph is linear to the table size, it increases the rederivation time as expensive program clause based rederivation is required instead of support graph based rederivation. Using partial support graphs, the time for incremental pointer analysis per statement deletion is about 8% of the time it takes for from-scratch evaluation.

Our second approach is based on a data structure called symbolic support graph which represents support information compactly. The method explores the commonality between supports originated in certain types of programs. For a variety of applications the size of symbolic support graphs grows no faster than the table size. Using symbolic support graphs, the time for incremental evaluation of pointer analysis per statement deletion is less than 1% of the from-scratch evaluation time. These algorithms are described in Chapter 5.

1.2.4 A Local Algorithm for Handling Updates

All recursive view maintenance algorithms treat addition and deletion as atomic changes, and an update as a deletion (of the old version) followed by an addition (of the new version). In some cases, the deletion may cause significant changes, most of which will be undone by the subsequent addition. This is also the case when multiple additions and deletions of facts occur. Based on this observation, 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 addition operations generated by the updates through this graph such that the overappoximation of deletion propagation can be nullified by the support generated by addition. 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. This algorithm is described in Chapter 6.

1.2.5 Handling Full Prolog

We implemented the incremental algorithms in XSB tabled logic programming system. We notice that the answer dependency based techniques cannot be readily applied for incremental computation of arbitrary tabled programs, especially those involving Prolog built-ins such as `findall`, other aggregation operations, or non-stratified negation. We explored a simpler incremental evaluation algorithm that, based on the dynamic call graph, invalidates and re-evaluates entire calls. The algorithm is agnostic to whether a change adds or deletes answers from tables, and hence can be applied uniformly to programs with negation, even when the negation is implicit (as is the case with certain aggregation operations). The call-graph based algorithm is described in Chapter 7.

Our incremental algorithms have been used in various real-life problems such as pointer analysis, push-down model checking, parsing, dynamic programming, and XML validation. We describe some of the major contributions of this thesis along with future avenues of research in Chapter 8.

Chapter 2

Related Work

The problem of incremental computation has been considered in various fields of research viz. materialized view maintenance and XML validation in databases, model checking, program analysis, logic programming, and functional programming. In this section we discuss some seminal works in these areas of research.

2.1 Materialized View Maintenance

Materialized views are pre-computed intensional relations stored in database for fast query response. When extensional database relations are updated, it is required to refresh the materialized views which depend on the updated base relations. The process of updating materialized views in response to the change in the underlying base relations is called *view maintenance*. The problem has been considered in databases community for many years (see, e.g. [GM95, MT99] for surveys). Although only few works have been proposed for recursive view maintenance.

Most of the works in recursive view maintenance generate rules that are similar in spirit to those of DRed [GMS93] and are subsumed by DRed (as compared in [GM95]). The DRed algorithm generates rules for computing deletions and additions of tuples in the views. Changes to the views are computed in three steps using bottom-up semi-naive computation of these rules. First, the algorithm computes an overestimate of deleted derived tuples: a tuple t is in this overestimate if the changes made to base relations invalidate *any* derivation of t. Second, this overestimate is pruned by removing (from the overestimate) those tuples that have alternative derivations in the new database. Finally, the new tuples that need to be added are computed using the partially updated materialized view and the changes made to the base relation.

Küchenhoff algorithm [K9̈1] derives rules to compute the difference between consecutive database states for a stratified recursive program. The rules generated are similar in spirit to those of DRed but some of these rules are not safe, and while dealing with positive rules they do not discard duplicate derivations.

Dong and Topor in [DT92] derived a nonrecursive program for addition of a single tuple for all views defined by a right linear chain Datalog programs.

Propagation/Filtration algorithm ([HD92]) is very similar to the DRed algorithm except that the changes made to the base relation propagate on a relation by relation basis. It computes changes in one derived relation due to changes in *one* base relation, looping over all derived and base relations to complete the view maintenance. However, rather than allowing the deletion step to complete before starting the pruning step, the deletion and pruning steps are alternated after each iteration of semi-naive evaluation. This allows the PF algorithm to avoid propagating some of the tuples that occur in the overestimate after the first iteration but do not actually change. However, the alternation of the steps after each semi-naive iteration also causes some tuples to be rederived several times.

The Urpi-Olive algorithm ([UO92]) computes the changes in the stratified deductive database. The views are defined as datalog rules and the changes to the views are computed using expressions derived from an analysis of deductive rules. The method takes into account key constraints of the base and derived relations. Updates to the non-key attributes are handled separately for improving efficiency. The paper defines several *transition* and *internal event rules*. *Internal event* represents the changes in the derived predicates and *external events* denote the changes in the base predicates. *Transition rules* capture the relation between old and new database states and the events that have occurred in the transition. *Internal event rules* define the conditions upon which an internal event occurs. It follows a three step algorithm — 1) generation of transition rules; 2) generation of event rules; and 3) SLDNF resolution to find the changes to the view.

Name	Addition	Deletion	Update	Sets of each	View Defn. Change
Küchenhoff[K̈91]	✓	✓	×	✓	✓
Gupta et. al.[GKM92]	✓	✓	×	✓	×
Dong, Topor[DT92]	✓	×	×	×	×
DRed[GMS93]	✓	✓	×	✓	×
Lu et. al.[LMSS95]	✓	✓	×	✓	×
Urpi, Olive[UO92]	✓	✓	✓	✓	×
Staudt, Jarke[SJ96]	✓	✓	×	✓	×

Figure 3: Types of changes

Lu et. al. ([LMSS95]) considered materialized mediated views as a set of constrained atoms. They extended the DRed ([GMS93]) algorithm to constrained database which can have non-ground tuples and presented a Straight Delete (StDel) algorithm which eliminates the rederivation phase of DRed. With every constraint atom the algorithm keeps track of the proof of truth of the atom. It does not address the problem of space overhead of keeping track of all possible *justifications* with every answer.

Staudt and Jarke ([SJ96]) presented a purely declarative encoding of DRed ([GMS93]) and transformed these rules with the help of supplementary magic set techniques so that the rules can be applied to cases where view caches are not accessible. The algorithm rederives the intensional relations on demand by firing these rules.

A top-down algorithm for incrementally checking integrity constraints (which can be seen as views) is presented in [SdS99]. This algorithm first computes the set of integrity constraints that are possibly affected by the changes to the facts. It then evaluates the integrity constraints top-down. The method works only for non-recursive predicates.

We present a tabular comparison of the type of changes handled by some of the view maintenance algorithm in Figure 3. Also note that apart from the algorithms in [LMSS95] none of the other algorithms maintain dependency graphs to propagate changes.

2.2 Model Checking

The first use of incremental computation in model checking is noted in [SS94], which considers model checking alternation-free fragment of modal mu-calculus. The algorithm, called MCI, takes as input a set Δ of added or deleted transitions to the labeled transition system (LTS) under investigation, and computes the new truth assignments of formula variables to LTS states. The main technique utilized by MCI is to first compute the immediate effects of Δ on the results of the previous computations and then restart the fix-point iteration. But before it starts the fix-point iteration, it makes adjustment to the current variable assignments - raising it sufficiently high in the lattice of all variable assignments when computing greatest fixed-point, and, dually, lowering it sufficiently when computing the least fixed-point. It uses a data structure called *product graph* for capturing all the dependencies between pairs of the form $\langle s, X_i \rangle$ for LTS state s and logical variable X_i of the given mu-calculus formula. The lowering of variable assignment in least-fix point computation follows the same heuristic of deletion phase of DRed [GMS93] - an assignment $\langle s, X_i \rangle$ is falsified if any of its immediate predecessor (a reason for derivation) is falsified. Its restarting fix-point iteration phase is again same as the rederivation phase of DRed. However, its use of product-graph for change computation and judicious use of counts distinguishes it from DRed.

In the context of formal verification of digital circuits Swamy et. al. presented several incremental algorithms ([SBS95, Swa96]). The algorithms employ similar fix-point adjustment algorithm as MCI but use rules as in DRed to compute the changes in the reachable state space in response to changes in the transition relation. The distinguishing feature of this work is its use of Binary Decision Diagrams (BDDs) [Bry86] to represent transition relation as well as reachable state space. The semi-naive evaluation of rules are expressed as operations on BDDs. To incrementally maintain reachable states in response to changes in the transition relation, a spanning graph (acyclic) is generated during reachability analysis as the evidence for all the reachable states. While computing the effect of deletion of edges, a state is considered unreachable if *all* of its immediate predecessors in the spanning graph are considered unreachable. Later an unreachable state is derived as reachable if it is reachable via a path which is not there in the spanning graph. This is possible as the predecessors of a node n in a spanning graph are reachable independent of whether n is reachable. Our careful observation reveals that this heuristic is better than the heuristics of MCI, DRed. MCI would mark a reachable state as unreachable if *any one* of its predecessors in the spanning graph is marked unreachable. As DRed does not keep any evidence of why a state is reachable, such heuristic to prevent over-propagation of deletion is not possible.

Note that Swamy's idea of keeping spanning tree and spanning graph can be considered as well-founded way of justifying reachable state space. This observation enabled us to devise a deletion algorithm which improves upon DRed, although the algorithm we use to compute such non-acyclic justifications (or supports) is different in the context of tabled logic programs. Note that we differ from Swamy's algorithm in the rederivation phase as we rederive based on the supports that are not marked in the deletion phase. However, Swamy's algorithm needs to perform reachability query for rederivation as it does not keep all justifications for reachable state space. We further extend our algorithms to confine change propagation by (i) using strongly connected components of dependency graph and ensuring that topologically lower components are stabilized before the effects of changes are propagated to a higher component, and (ii) using the effect of addition to restrict deletion propagation when both kinds of changes occur together.

[CNDE05] presents incremental version of inter-procedural analysis algorithm for verifying safety properties. This is in fact the first incremental algorithm for safety analysis of recursive state machines. The paper presents two algorithms to handle changes. The key to both the algorithms is the incremental maintenance of a data structure called the *derivation graph* which is the product of the control-flow graph (CFG) and automaton representing the finite state property. Both the algorithms handle modification, insertion and deletion of edges from the CFG. In general modifications are handled by deletion of the old control-flow edge followed by insertion of the modified edge. In IncrFwd algorithm insertion of CFG is handled by restarting the non-incremental evaluation algorithm from the point of insertion. Deletion is handled by checking the entire derivation graph and deleting edges in the derivation graph which is dependent on the deleted CFG edges. The second algorithm, called IncrBwd, processes the changes in bottom-up topological order of maximal strongly connected component (SCC) decomposition of the call graph. Thus it propagates changes to the topologically higher SCC once all changes are computed in lower SCCs. In each SCC it employs the IncrFwd algorithm. In contrast to IncrFwd algorithm, IncrBwd algorithm produces a derivation graph which is over-approximation of the graph generated by from-scratch evaluation. Our incremental local algorithm uses the idea of propagating changes using SCC-reduces dependency graphs. However, we interleave insertion and deletion propagation within each SCC.

2.3 Program Analysis

Incremental analysis is used to recompute global analysis information on programs in response to changes in the program. Many incremental algorithms have been developed for data-flow analysis problems. Some incremental analyses use the elimination method [Bur90, CR88, RP88]; some are based on the technique of restarting iterations [PS89] and some are combination of the two techniques [MR90]. A comparison of incremental iterative algorithms can be found in [BR90]. Effectiveness of incremental analysis has been shown for MOD analysis of C programs [YRLS97]. Pollock and Soffa [PS89] presented a precise incremental iterative algorithm using change classification and reinitialization for bitvector problems. The authors identified several cases where changes can be propagated without any overapproximation. However, in presence of cycles they employ two phases called *exaggerate* and *adjust* to compute the changes. These phases use the same heuristic of including a derived fact in the set of overapproximation of actual changes if *any one* of its predecessors (derivations) is already in the overaproximated set.

Yur et. al. in [YRL99] developed an incremental pointer analysis algorithm based on Landi-Ryders's flow- and context-sensitive alias analysis ([LR92]). They update points-to information after a program change rather than computing it from scratch. Their incremental algorithm is not *complete* in the sense that it may compute less precise solution than the exhaustive technique. This algorithm also has two phases of alias falsification (deletion) and alias introduction (rederivation) as DRed. Their selective falsification strategy degenerates to falsification strategy of DRed [GMS93] where all directly and indirectly generated aliases due to the deleted statement are falsified.

2.4 Attribute Grammar Evaluation

A lot of research has been done on incremental evaluation of attribute grammars. The categorized bibliography by Ramalingam et. al. [RR93] presents the work done in 80's and early 90's. Most of the algorithms in this area are based on updating values of attribute instances by change propagation through dependency graphs. As our technique also relies on change propagation using dependency graphs, we discuss here some of the relevant works on dependency graph based incremental attribute evaluation.

The work on incremental attribute evaluation is pioneered by Thomas Reps, Tim Teitelbaum, and Alan Demers [DRT81, Rep82, RTD83, Rep84] in their work motivated by its application on 'Synthesizer Generator', a system for creating specialized editors that are customized for editing some particular languages. All of these works concentrate only on non-circular attribute grammars. In their first paper ([DRT81]) on incremental evaluation of attribute grammars they presented a two pass algorithm based on nullification and re-evaluation. The algorithm presented in this paper is non-optimal. This paper was followed by a paper by Reps ([Rep82]) which presents an optimal solution on incremental evaluation of non-circular attribute grammars. This seminal paper has been elaborated in Reps' thesis ([Rep84]) and subsequently in the journal version ([RTD83]).

We present here an overview of the main result presented in [Rep82]. As mentioned before the optimal algorithm is based on keeping functional dependencies among various attribute instances. For a particular derivation of a string, the derivation tree represents the dependency between all nonterminal /terminal instances. Semantic tree is the derivation tree with their respective attribute instances (and their values). Dependency graph represents the functional dependencies between values of the attribute instances. This paper only considers acyclic dependency graphs, and provides a change propagation algorithm through dependency graphs. The change propagation is triggered by various editing operations on derivation tree- like pruning and grafting. Out of entire allocation of attribute instances of the tree, only certain ones require new values; these are denoted by the set $AFFECTED$ although the members of this set are not known a-priori. The total cost of the algorithm is $O(|AFFECTED|)$. The main idea of the algorithm is to do topological order evaluation through the static and acyclic dependency graphs.

2.5 Logic Program Analysis

Hermenegildo et. al. [HPMS95, PH96, HPMS00] discussed incremental algorithms for global analysis of logic programs. In [HPMS95] the authors presented incremental algorithms for re-analysis of logic programs and constraint logic programs respectively. They first presented an event based algorithm for non-incremental program analysis which builds a program analysis graph containing answer table (nodes) and dependency-arc-table (edges). Each entry in answer table shows a calling pattern and corresponding answer pattern. Each dependency arc shows the calling pattern of a subgoal in the body of a clause for a particular calling pattern of the head subgoal of the clause. Their incremental addition algorithm uses the non-incremental algorithm to propagate the changes due to addition of rules. The authors presented two deletion algorithms. The top-down algorithm first deletes the nodes and edges in the program analysis graph which is dependent (before the change) on the deleted rules and consequently re-evaluates the affected calls using non-incremental algorithm. Due to huge overapproximation present in the top-down algorithm they proposed a bottom-up deletion algorithm which uses SCC-reduced *predicate dependency graph* to propagate the changes to the analysis result from topologically lower predicate SCC to upper predicate SCC level only after lower SCC gets completely evaluated. Within each predicate SCC the algorithm reevaluates all possibly affected calling patterns and checks whether the new answer patterns are different from the old answer patterns in which case the changes are propagated to the topologically higher SCC. In presence of arbitrary changes (addition and deletion) similar bottom-up algorithm is followed except that the SCCs needs to be reevaluated due to addition of rules.

In [PH96] the authors optimized the non-incremental global analysis algorithm presented in [HPMS95]. The optimized algorithm is obtained using delayed dependency and new assignment of event priorities which assures that each topologically lower dynamic call SCC is completely evaluated before upper SCCs. The important contribution is to obtain this assurance of SCC-preserving computation without keeping any explicit SCC. As the incremental deletion algorithm of [HPMS95]) uses non-incremental algorithm as a subroutine, the efficiency of incremental deletion is also thereby increased. Incremental addition requires a special attention in the new non-incremental algorithm as meeting delayed dependency criteria is not always possible. Incremental addition uses application dependent strategy to make the event scheduling strategy SCC-preserving.

The idea of using SCC-reduced dependency graphs to optimize propagation of changes has been seen in various other works [Jon90, WJ88, CNDE05]. In the context of tabled logic programs we develop an incremental algorithm (called local algorithm; discussed in Chapter 6) which also propagates the changes due to insertion and deletion on topological order of SCC-reduced dependency graph. Infact our event based description for propagating changes in the local algorithm has been inspired by their Hermenegildo et. al.'s work. Apart from the fact that their incremental analysis is specialized towards logic program analysis, they only consider one answer pattern per call, and propagation is controlled based on the call graph. In contrast, our local algorithm consider two kinds of dependencies: one based on calls and one based on answers. Because of the answer dependencies we can achieve finer-grained interleaving between addition and deletion operations within call graph SCCs.

2.6 Functional Programming

The existing works on incremental functional programming can be divided into two categories. The literatures ([PT89, LST98, ABH03b]) on the first category use memoization to re-use results of an earlier call when a matching call occurs. The paper [ABH02] of the other category uses dependency graph to perform change propagation to update the solution. Only in recent times couple of technical reports [ABH03a, dMS03] try to combine these two approaches. In this thesis, we build dependency graph based approach on top of the memoization framework of tabling. We also extend the problem domain by considering cyclic dependency graphs.

In [PT89] Pugh described an incremental evaluator for functional languages based on function caching. The method is based on stable decomposition scheme that decomposes two subproblems in a similar way so that they share common subproblems. The data structures are designed in such a way that two similar values will have similar decomposition. Liu et. al. [LST98] presented systematic technique, called *cache-and-prune* to automatically determine which results need to be cached and how to cache and maintain them throughout incremental computation. Their method is based on static program analysis and semantic-preserving program transformation. Recently Acar. et. al. [ABH03b] presented *selective memoization* techniques to provide control over performance of memoization based facilities based on precise input-output dependencies, defining call-equality, and controlling space usage. The definition of call-equality is the most important aspect of re-using results of calls. The most commonly used approach is to deem two calls equal when the function as well as arguments match exactly. In this paper the authors considered two calls to be equal if the arguments on which the result is dependent matches. These arguments are found by keeping input-output dependencies.

The dependency graph based change propagation for incremental evaluation of functional programs has been recently explored by Acar et. al. [ABH02]. The paper describes how a pure functional program can be instrumented to obtain the Augmented Dependency Graph (ADG) while the program is executed. Change propagation is done through ADG to update the values of the expressions that are dependent on the changed value of the variable. The dependency graph in their work is acyclic as it represents functional dependencies between values of variables and expressions. The dependency graph also maintains an ordering among its edges to keep track of which edges are within the dynamic scope of other edges (containment hierarchy). The ordering among edges enables the re-evaluation of expressions in the same order as they were evaluated in the initial evaluation. The containment hierarchy also enables identification and removal of edges that become obsolete. Timestamps are used to represent this information, and dynamic topological order maintenance algorithm of Dietz and Sleator [DS87] is used to maintain timestamps incrementally.

Note that dependency graph based incremental attribute evaluation does not apply for evaluation of functional programming as the dependency graph considered in attribute evaluation is static. The dynamic nature of the augmented dependency graph (ADG) distinguishes their work from previous applications of incrementality. However, the acyclic property of ADG limits its use in set-expression evaluation which can potentially cause generation of cyclic dependency graph. Also note that, the algorithm only relies on dependency graph based evaluation and do not take advantage of memoization.

In context of incremental tabled evaluation, we maintain a dependency graph (known as subgoal dependency graph or call graph) which is potentially cyclic and dynamic. Handling cycles properly is a key challenge addressed in our algorithm. Nevertheless, we developed algorithm which naturally specializes to ADG based algorithm for incremental functional programming. Note that the stable decomposition strategy used by Pugh in [PT89] is catered specially towards incremental evaluation. Following such strategies requires that the input tabled logic program to be written in a specialized way that it may no longer be declarative in nature. Thus we do not follow such strategies in developing incremental algorithms for tabled evaluation.

2.7 Truth Maintenance Systems

Truth maintenance (also called belief revision or reason maintenance) is an area of AI concerned with revising sets of beliefs and maintaining the truth in the system when new information contradicts existing information. In this context, Doyle's [Doy79] truth maintenance system (TMS) determines current set of beliefs from current set of reasons and incrementally updates the current set of beliefs in accordance with the new reasons. It considers propositional reasoning and beliefs and uses general logical formula to describe them. The TMS maintains a four-valued propositional logic formula, in which all valid, invalid combinations of literal and its negated form can co-exist. The TMS maintains two data structures which represent beliefs, and justifications (reasons for beliefs). It maintains justifications for both valid (*in*) as well as invalid (*out*) beliefs. Justifications are divided into two groups: support-list justifications and conditional-proof justifications. Among all the supports of a belief it singles out one justification, called the supporting-justification or well-founded support which forms a non-circular argument for the belief in the *in* set. It also maintains the dependency between beliefs and justifications using a dependency graph.

The incremental algorithm employed by TMS propagates the changes through the dependency graph. The propagation algorithm employs a two phase algorithm consists of deletion and rederivation phases. The deletion phase however is better than deletion strategy of DRed as it prunes the overapproximation caused by DRed using well-founded supports.

Our incremental deletion algorithm considers Horn clause logic programs (except in Chapter 7) and thus is not general as TMS systems. Also we consider first-order logic whereas the TMS used propositional logic. We use dependency graph structures similar to support list justification of TMS to keep dependencies between answer and its supports (or justification). In our case the answer dependencies exist between only true/valid answers. We use the idea of having a non-circular support of TMS (referred to as primary support) to reduce the overapproximation of actually deleted answers. We exploit a property of least fixpoint computation to identify such supports in non-incremental evaluation without any extra cost. We also extended the deletion algorithm to include multiple such acyclic supports to further reduce the overapproximation of deletion. In addition to these when deletion and insertion changes are occurring together we devised a local algorithm which judiciously interleaves the insertion and deletion operations for better efficiency. TMS on the other hand does not employ such local strategy. It also does not consider the space problem potentially caused by the support justifications. We develop space efficient algorithm to address the space problem caused by support graph when incremental tabled evaluation is applied to program analysis.

Chapter 3

Preliminaries

In this section we describe operational description of tabled logic programming followed by a formal treatment of computational power of tabling. We also describe deductive formulation of a well known pointer analysis algorithm. In this thesis pointer analysis of C programs has been used as a practical evaluation framework for demonstrating effectiveness (and weaknesses too) of incremental algorithms.

3.1 Tabled Logic Programming

The use of tabling in logic programming has established itself as a powerful evaluation technique, since it allows bottom-up evaluation to be incorporated within the top-down framework, combining the advantages of both. At-least for loop-free, stratified programs with few redundant subcomputations top-down queries can be substantially faster than bottom-up deductive database systems. Thus, rather than adding goal orientation to a bottom-up evaluation, a natural approach to evaluate in-memory queries is to add bottom-up capabilities or tabling, to Prolog like top-down evaluation. The concept of tabled evaluation has been there for two decades ([TS86]) and in last decade we have seen some powerful implementation of tabled logic systems ([XSB, RSC00, ZSYY01, GG01]). For the purpose of this thesis we concentrate on XSB's ([XSB]) tabled logic programming system.

At a high level, top-down tabling systems evaluate programs by recording subgoal (referred to as *calls*) and their provable instances (referred to as *answers*) in a table. Predicates are marked *a-priori* as either *tabled* or *nontabled*. Clause resolution, which is the basic mechanism for program evaluation, proceeds as follows. For nontabled predicates the subgoal is resolved against program clauses. For tabled predicates, if the subgoal is "already present" in the table, it is resolved against the answers present in the table; otherwise the subgoal is entered into the table and its answers, computed by resolving the subgoal against the program clause, are also entered into the table. The process of resolving a subgoal against its program clauses will be henceforth called *program clause resolution* or PCR in short. And *answer clause resolution* will refer to resolving a subgoal against the tabled answers. For both tabled and nontabled predicates, program clause resolution is carried out using SLG-resolution [CW96].

The check for presence of a call in the call table can be done in two ways. In a *variant* based tabling system one checks whether a variant of the new goal already exists in the table. Two terms are variant to each other if one can be converted to other by renaming its variables. Alternatively in subsumptive based engine, a call can be deemed present in a table if another call *subsumes* the call already in the table; in that case the call can be resolved against the answers of this more general call. For the purpose of this thesis only variant based checking is considered. Due to sharing of call structures in subsumption based tabled engine, incremental algorithms that depend on call dependencies (Discussed in Chapters 6, 7 and Section 5.2) are not applicable to subsumptive tabling. But, insertion and support graph based deletion algorithm discussed in Chapter 4 and partial support graph based deletion algorithm discussed in Section 5.1 seamlessly works for subsumptive engine. Note that efficiency of memoization technique for the purpose of re-use of old answers is dependent on these methods of checking. It is important to note that the original goal of using tabulation was to incorporate bottom-up semi-naive technique to top-down evaluation framework thereby making sure of termination for definite logic programs with bounded term size property [RSS$^+$97]. In contrast in functional programming [PT89, ABH03b] memoization techniques have been used fundamentally for the purpose of incremental evaluation which focuses on interesting memoization schemes catered towards more re-usability of information. In this thesis we consider variant based simple call equality approach. Extending our incremental techniques to more fine grained memoization approaches such as [GG04] is an interesting open problem.

The XSB system uses trie-based data structures for storing terms in call and answer tables [RRS^{+}95]. For each tabled predicate a call table (trie) is maintained for various calls of that predicate. All answers to each entry of call-trie (call) in the call table are stored in an answer trie. Each answer trie is linked with the call trie entry with the intermediate structure called subgoal frame which keeps most of the information regarding a call. Tries permit efficient lookup and one-pass check-insert operations. However, tries do not maintain the terms in the order of insertion. When resolving answers against an incomplete table (where new answers may be added), XSB maintains and uses an *answer list*, which links leaf nodes in the trie in their order of insertion. When a table is complete, which means no new answers can be added to the table with respect to a given set of facts, answer resolution is done by backtracking through the trie top-down; the answer list is no longer needed and is deleted.

Most of the incremental algorithms in subsequent chapters are described with respect to definite logic programs which can be extended to stratified negation based on propagating changes strata-by-strata. In the following discussion we present a formal description of tabled evaluation for definite logic programs, focussing specially on the model computed by tabled evaluation.

The semantics of definite logic program is described in terms of existence of the least Herbrand model [Llo84]. However, Herbrand model is based on an interpretation where the domain is a set of variable-free terms. This assumption does not hold for the model inferred by the tabled evaluation of definite logic programs. Also the model generated by tabled evaluation is related to a goal (or query) to a program.

Consider a definite logic program P be a tuple $\langle R, F\rangle$ where F is a set of facts and R is a set of rules. Let $EVAL_{tab}(P', Q')$ returns a tuple $\langle A', \Gamma'\rangle$ where A' is the union of set of answers generated by table evaluation while evaluating the queries in Q' and the set of facts in P', and Γ' is the set of calls generated by tabled evaluation. In the following we define the set of calls Γ' and set of answers A' generated by tabled evaluation.

The calls and answers generated by tabled evaluation can be defined using SLD resolution [Llo84].[1] The SLD resolution processes results in a finite or infinite sequence of goals starting from with the initial goal. At every step a program clause (with renamed variables) is used to resolve the subgoal selected by the computation rule $\mathcal{R}$ and an mgu (most general unifier) is created. Thus, the full record of a reasoning step would be a pair $\langle G_i, C_i \rangle,\ i \geq 0$, where G_i is a goal and C_i is a program clause with renamed variables. The computation rule $\mathcal{R}$ together with G_i and C_i determines (upto renaming of variables) the mgu (to be denoted θ_{i+1}) produced at $i+1$th step of the process. A goal G_{i+1} us said to be derived (directly) from G_i and C_i via $\mathcal{R}$. The computation rule is a selection function which for a given goal selects the subgoal for unification.

Definition 1 (SLD-derivation([NM00])) *Let G_0 be a definite goal, P is a definite logic program and $\mathcal{R}$ is a computation rule. An SLD-derivation of G_o (using P and $\mathcal{R}$) is a finite or infinite sequence of goals:*

$$G_0 \xrightarrow{C_0} G_1 \ldots G_{n-1} \xrightarrow{C_{n-1}} G_n \ldots$$

For the purpose of this discussion we only consider SLD-derivation in which computation rule selects a left-to-right subgoal selection strategy. Henceforth, unless stated we consider SLD-resolution which uses the above selection strategy.

We now define the calls evaluated by tabled evaluation based on SLD-derivations.

Definition 2 (Call) *Let a definite logic program P be a tuple $\langle R, F \rangle$ where F is a set of facts and R is a set of rules. Let G_0 be a query and $G_0 \xrightarrow{C_0} G_1 \ldots G_{n-1} \xrightarrow{C_{n-1}} G_n \ldots$ be an SLD-derivation. Let γ_i represents the first subgoal of goal G_i. Then the tabled resolution of P with respect to the query G_0 will have γ_i's as tabled calls if it corresponds to a subgoal of a table predicate. All calls generated by tabled resolution follows from all possible SLD-derivations.*

We also define the dependency between calls in terms of a graph called *call graph.*

Definition 3 (Call Graph) *Let a definite logic program P be a tuple $\langle R, F \rangle$ where F is a set of facts and R is a set of rules. Let G_0 be a query. The call graph (Γ,E) corresponding to the program P and query G_0 is defined as follows, Γ is the set of*

[1]The description on SLD-resolution has been taken from [NM00]

calls and $E \subseteq (\Gamma \times \Gamma)$ such that $(\gamma_i, \gamma_j) \in E$ iff there exists an SLD-derivation path between two goals G_i and G_j where γ_i and γ_j correspond to the first subgoal of G_i and G_j respectively, and the path from G_i to G_j does not contain any goal whose first subgoal is a tabled predicate.

Note that there can be infinitely long SLD-derivation paths possible. SLD-derivation is captured by SLD-derivation tree which has finite paths from root to leaf corresponding to finite derivations (either true or false) and infinite paths. Each finite SLD-derivation of the form: $G_0 \xrightarrow{C_0} G_1 \ldots G_{n-1} \xrightarrow{C_{n-1}} G_n$ yields a sequence $\theta_1, \ldots, \theta_2$ of mgu's. The composition of these mgu's us called the computed substitution of the derivation. SLD-derivations that end in a empty goal (and the binding of variables in the initial goal of such derivations) correspond to refutations of the initial goal. A finite SLD-derivation which ends in an empty goal is called SLD-refutation of the initial goal. The computed substitution if an SLD-refutation of G_0 is called *computed answer substitution* for G_0.

Definition 4 (Answers) *Let a definite logic program P be a tuple $\langle R, F \rangle$ where F is a set of facts and $\mathcal{R}$ is a set of rules. Let γ be a call generated by tabled evaluation. Then all computed answer substitution by SLD-resolution for γ_0 are exactly the answers of γ_0 computed by tabled resolution.*

The above discussion on tabled resolution is only meant for understanding this thesis. For further discussion on XSB's table engine refer to [CSW95, RRS$^+$95]. In the following section we show an example of tabled evaluation on logic programming encoding of pointer analysis.

3.2 A Deductive Formulation of Pointer Analysis

We consider Anderson's [And94] inclusion-based context-insensitive and flow-insensitive pointer analysis to experiment the effectiveness of various algorithms presented in this thesis. In this section we explain the rule-based encoding of pointer analysis. To simplify our presentation, we assume that a given input C program is decomposed to a set of primitive assignment statements of the following form:

$$u = \&v \mid u = v \mid u = *v \mid *u = v$$

$$\frac{}{u \longrightarrow v}\ \texttt{u = \&v} \qquad \frac{v \longrightarrow x}{u \longrightarrow x}\ \texttt{u = v}$$

$$\frac{v \longrightarrow x,\ x \longrightarrow y}{u \longrightarrow y}\ \texttt{u = *v} \qquad \frac{u \longrightarrow x,\ v \longrightarrow y}{x \longrightarrow y}\ \texttt{*u = v}$$

Figure 4: Anderson's rules for pointer analysis.

```
points_to(U,V) :- assign(plain(U),addr(V)).
points_to(U,X) :- assign(plain(U),plain(V)), points_to(V,X).
points_to(U,Y) :- assign(plain(U),star(V)), points_to(V,X), points_to(X,Y).
points_to(X,Y) :- assign(star(U),plain(V)), points_to(U,X), points_to(V,Y).
```

Figure 5: Logic program using Prolog notation corresponding to Anderson's rules

Figure 4 shows the points-to analysis rules for each assignment statements (from [HT01a]). In the figure $x \longrightarrow y$ denotes that x may point to y.

These rules can be readily written as a logic program which is shown in Figure 5. Following Prolog's notational convention, identifiers beginning with uppercase letters denote variables, and identifiers beginning with lowercase letters denote relation names and data constructors. For example, `points_to` denotes the binary may-points-to relation, and the term `points_to`(x,y) denotes that x may point to y. The terms `plain(`x`)`, `addr(`u`)` and `star(`u`)` represent pointer variable x, and pointer expressions $\&u$ and $*u$ respectively. Also `assign(`U`,`V`)` represents the assignment statement with left hand side and right hand side contain pointer expressions corresponding to the terms U and V. For example, the second rule should be read as "U may point to X if there exists an assignment statement of the form $U = V$ in the code and V may point to X."

Based on this formulation the set of all variables that a given variable v may point to can be computed as answers to the query[2] `points_to`(v, X). For instance, given the set of assignment statements {`u=&v`, `p=u`, `u=p`}, the query `points_to(p,X)` has one answer `X=v`, meaning that `p` may point to `v`. While the points-to analysis can be succinctly encoded in Prolog syntax, most Prolog systems will fail to evaluate the

[2]In this chapter, we use the terms "query", "goal", "subgoal" and "call" interchangeably.

program correctly. This results from Prolog's inability to compute the least models of programs with left recursion— even for Datalog programs (i.e. logic programs without data structures). For instance, consider the evaluation of the query `points_to(p,X)` w.r.t. the set of assignments {`u=&v`, `p=u`, `u=p`, `t=&u`}. In resolving the original goal, Prolog will issue the query `points_to(u,X)` (due to `p=u`), whose resolution gives an answer `X=v`. To find more answers for the latter goal, we again encounter the goal `points_to(p,X)`. This causes Prolog to loop.

Tabled resolution (discussed earlier in this chapter), on the other hand, removes this shortcoming of prolog for datalog programs by using memoization. Consider again the query `points_to(p,X)` w.r.t. the set of assignments {`u=&v`, `p=u`, `u=p`, `t=&u`}. Upon first encountering the original query, tabled resolution adds the goal to the call table and creates an empty answer table for it. Program clause resolution will then produce the goal `points_to(u,X)`, which, in turn is entered in the call table. Further resolution will produce one answer `X=v`, which is entered in both the answer tables. Continuing with resolution, we will once again get the goal `points_to(p,X)`. Instead of doing program clause resolution which makes Prolog loop, tabled resolution resolves this goal using answers in the `points_to(p,X)`'s answer table. In this example, this produces `X=v`, an answer that was already generated. No further answers can be generated, and hence the evaluation terminates.

By remembering the past resolution steps and avoid repeating them, memoization helps tabled resolution terminate for datalog programs, and moreover, evaluate queries with polynomial data complexity. Moreover, unlike the semi-naive algorithm used in the deductive database literature ([Ull89]), tabled resolution is goal-directed, and hence is naturally suited for demand-driven analysis.

Pragmatics. We apply the above analysis to C programs by transforming all assignment expressions into a set of primitive assignments. Nested uses of & and * are handled by introducing temporary variables. For each static call site we introduce assignment statements where formal parameter is assigned to actual parameter. Dynamic call sites are resolved with functions having same number and types of parameters. If the returned value is assigned to a variable then we generate an assignment statement which assigns a temporary variable (same as the function name) to that variable. For each function the return expression is assigned to the same temporary variable generated from function name. In this chapter we consider field-independent analysis which ignores field information for accessing structures and unions. Each array is treated as a single variable and index information is ignored. Relaxing these restrictions adds complexity to the analysis but does not reveal any more insight into the problem of incremental analysis, and hence we describe only an analysis with these restrictions.

Recall that the evaluation of `points_to(p,X)` w.r.t. the set of assignments {`u=&v`, `p=u`, `u=p`, `t=&u`} did not make use of the assignment statement `t=&u`. This is due to the goal-directedness of the evaluation technique: only calls and answers that are needed to resolve the given goal are used. However, note that implementations of tabled resolution follow Prolog's literal selection strategy: at each program clause resolution step, the left-most subgoal is selected for resolution. Hence, the order of literals in a clause affects the propagation of demand. For example, consider the same query `points_to(p,X)` with the set of assignments {`u=&v`, `p=u`, `u=p`, `t=&u`, `*s=r`}. Due to the statement `*s=r` and the fourth rule for points-to, tabled evaluation generates new queries `points_to(s,p)` and `points_to(r,X)`. Thus tabled engine evaluates the points-to relations of variables that are unrelated to `p`.

Consider the case when we are interested in evaluating, for different variables v, what variables v may point to. This can be done by issuing queries of the form `points_to(`v`, A)` for different v: queries where the first argument is bound and second argument is free. The pattern of boundedness of query arguments is known as the *calling mode* of the query; the calling mode of the above query is bound-free. Note that for bound-free queries, the fourth rule defining points-to uses the bound argument (`X`) only in the second literal. Thus, tabled evaluation will backtrack through every assignment of the form `*U=V`, without regard to whether it is related to the original query.

We can avoid this by reordering the literals in the fourth rule to as follows:

```
points_to(X,Y) :- points_to(U,X), assign(star(U),plain(V)), points_to(V,Y).
```

Due to the first literal in the body of the above rule, tabled evaluation will now generate queries to points-to with calling mode free-bound. Note that different calling modes require different literal orders. For instance, for free-bound queries, it is better to reorder the body of the fourth rule with `points_to(V,Y),` as the first literal.

```
points_to(X,Y):- assign(plain(X),addr(Y)).
points_to(X,Y):- assign(plain(X),plain(Z)), points_to(Z,Y).
points_to(X,Y):- assign(plain(X),star(Z1)), points_to(Z1,Z),points_to(Z,Y).
points_to(X,Y):- pointed_to_by(X,U), assign(star(U),plain(Z)), points_to(Z,Y).

pointed_to_by(X,Y):- assign(plain(Y),addr(X)).
pointed_to_by(X,Y):- pointed_to_by(X,Z), assign(plain(Y),plain(Z)).
pointed_to_by(X,Y):- pointed_to_by(X,Z), pointed_to_by(Z,Z1), assign(plain(Y),star(Z1)).
pointed_to_by(X,Y):- pointed_to_by(X,V), assign(star(U),plain(V)), points_to(U,Y).
```

Figure 6: Logic program for points-to analysis specialized with respect to calling modes

We hence specialize the points-to relation with respect to the two calling modes, as shown in Figure 6. In the figure, `points_to` handles bound-free queries to the original points-to relation; `pointed_to_by`, the inverse of `points_to`, handles free-bound queries to the original points-to relation. Note that if we issue only bound-free queries to `points_to` from the top-level, then all queries to `points_to` as well as `pointed_to_by` will be bound-free.

Note that the queries to `points_to` and `pointed_to_by` are distinct (bound-free queries are distinct from free-bound queries) and their answers are not shared in the current tabling infrastructure. Subsumptive tabling [RRR96] can be used to share the answers, but provided a more general (in this case, a free-free) query is issued first; however, the general query will mean that the analysis will no longer be demand-driven.

The rules in Figure 6 are similar to the inference rules for demand driven analysis in [HT01a]. The mode based specialization presented here is straightforward. However, since it involves goal reordering, it is not clear how this specialization can be automated for general logic programs. The program can be further optimized by grouping the literals on the right hand sides and tabling intermediate results. The details are omitted here.

Experimental Setup. We measured the performance of points-to analysis on programs taken from C benchmarks available with PAF [PAF] compiler suite and SPEC95 benchmarks. The benchmark characteristics are given in Figure 1. The first four benchmarks in the table are relatively small; to remove noise from the results,

Programs	LOC	Prim. Assign	Rep. Factor	From Scratch All Points-to					
				Avg. Size	Time(s)	Facts	Answers	Space(MB)	
								Table	Total
smail	3850	664	15	24.5	1.45	19.9K	398.0K	13	22
parser	11391	2190	15	5.8	1.20	65.7K	366.4K	17	32
vpr	17729	2708	15	1.8	0.37	81.2K	146.4K	15	33
m88ksim	19093	1406	15	6.0	0.25	42.2K	160.9K	10	22
twmc	24951	7065	1	16.7	0.87	14.1K	278.6K	9	14
nethack	33993	4875	1	35.0	2.55	9.7K	204.9K	6	21
vortex	67110	14387	1	69.8	12.90	28.8K	1,124.9K	31	57

Table 1: Benchmark characteristics

we replicated the programs, generating new variable names as appropriate. The remaining three benchmarks are large enough to permit stable measurements without replication. We show the replication factor for the each benchmark in the column named "RF".

We preprocessed the C source code using CIL [NMRW02] into Prolog facts representing the primitive assignment statements. Each library function was replaced by a stub representing the data flow between its formal parameters and return value and preprocessed in the same manner. Performance measurements were taken on a PC with 1.4Ghz Pentium M processor with 2GB of physical memory running Linux (Debian) 2.6.7. Similar setup is used for the pointer experiments given in this thesis.

We measured the effectiveness of the demand-driven analysis for resolving dynamic call sites in the benchmark programs. Function Pointer Analysis (FPA) uses the logic program encoding in Figure 6, and computes all answers to queries of the form `points_to(`f`, X)` for each function pointer f occurring in a given benchmark program. All Points-To Analysis (APA) also uses the same logic program, but computes the entire points to relation (i.e. over all program variables). The result of APA is given in Table 1.

Figure 7 shows the time taken and the size of the points-to relation computed by FPA relative to those of APA. Observe from the figure that for some benchmarks (`gzip`, `twmc`, and `nethack`) FPA takes less than 1.8% of the time taken by APA; and in others (`vpr`, `vortex`) the time taken for FPA is a significant fraction of that for APA. The latter benchmarks keep function pointers in structures, and the higher

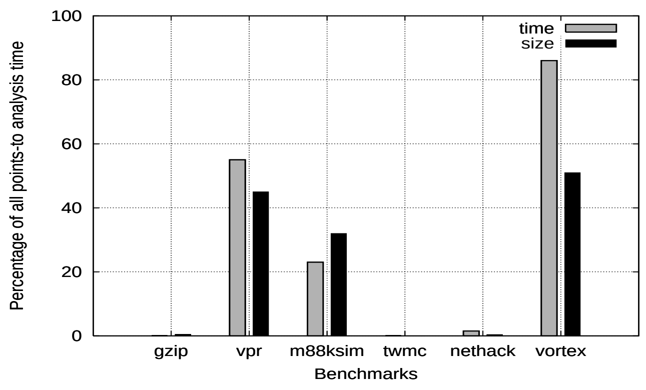


Figure 7: Relative performance of Function Pointer Analysis w.r.t. All Points-to Analysis

analysis times appear to be the artefact of performing field-insensitive analysis, which results in a large number of spurious points-to tuples for the function pointers.

Related Work. Points-to analysis has been studied extensively (see [Hin01] for a survey), and continues to attract significant attention (e.g. [FRD00, FFA00, GL03, WL04, LH01]). The aim of this work is to present techniques for making program analysis incremental. Although we do not directly address the accuracy-time tradeoffs that are at the core of much of the points-to analysis work, ability to perform incremental analysis will enable us to deploy more accurate analyses that may otherwise be deemed impractical.

Among the many works on points-to analysis, Heintze and Tardieu [HT01b, HT01a] encode flow-insensitive and context-insensitive subset-based pointer analysis due to Anderson [And94] using deductive rules. Our encoding in Section 3.2 follows [HT01a]. We derive demand-driven as well as incremental analyses directly based on these rules. Several graph based optimization techniques [FFSA98, SFA00] cannot be declaratively encoded showing the limitations of rule-based techniques. We believe that special purpose constructs can be introduced into table engine (such as XSB) to perform optimization such as cycle elimination - which collapses cycles in the dependency graph at that time of evaluation. However, we suspect that incremental evaluation in presence of such optimization may not be efficient as it is difficult to quickly regenerate dependency information once the cyclic dependency is broken due to deletion of facts.

Demand-driven program analysis using logic-programming based formulation has been studied before [Rep93, HT01a]. For instance, our encoding of demand driven context-insensitive Anderson's rules in terms of Horn clauses (Figure 6) is similar in nature to the rules obtained in [HT01a]. The main difference between the earlier works and the one presented in this section lies in the way the rules are evaluated. Heintze et. al. use CLA [HT01b] infrastructure to implement a set-based algorithm corresponding to the rules by using a technique similar to magic set transformation to bring goal-directedness to bottom-up evaluation [Rep93, HRS95]. Tabled resolution [TS86] is naturally goal directed, and as observed in [Rep93] this strategy ensures that it accesses only those assignment statements and generates only those intermediate queries which are relevant to answer the top level query [DRW96]. Although the implementation by Heintze et. al is considerably faster than our from-scratch APA analysis, our incremental algorithm can be incorporated into CLA framework for yielding better performance.

In [DGS97] the authors presented a method for the construction of precise demand-driven algorithms for the class of distributive finite data flow problems. As also explained in [HT01a] demand-driven pointer analysis falls outside the scope of [DGS97]. In [AG98] a demand-driven pointer algorithm has been presented for Steengard's Algorithm [Ste96] and program slicing is used to show effectiveness of demand driven analysis. Demand-driven call graph construction for the Java programs has been investigated in [ALS02].

Chapter 4

An Efficient Algorithm for Incremental Tabled Evaluation

In this chapter we present time efficient algorithms for incremental evaluation of definite tabled logic programs. In Section 4.1 we present a program transformation based technique for handling incremental insertion of facts. The algorithm uses top-down query evaluation technique to find the changes due to insertion of facts. In contrast we follow a dependency graph based bottom-up approach for handling deletion of facts (Section 4.2).

```
:- table r/2.
r(X,Y) :- b(X,Y).           % rule 1
r(X,Y) :- c(X,Z), r(Z,Y).   % rule 2

b(2,3).   %f1
b(5,3).   %f2
b(5,7).   %f3
c(1,2).   %f4
c(1,5).   %f5
c(4,1).   %f6
c(5,4).   %f7
c(6,4).   %f8
c(6,5).   %f9
```

Figure 8: Example program

4.1 Addition

Top-down goal-oriented evaluation systems (such as those based on the SLGWAM) inherently process answers incrementally. A subgoal that causes answers to be added to the tables is called a generator, and a subgoal which is resolved against answers already in the tables is called a consumer. The evaluation engine maintains auxiliary data structures to ensure that no consumer sees an answer more than once: e.g. environments to produce and consume answers and control structures linking answer producers to answer consumers. These data structures are torn down when all answers to a call have been derived, an operation that is crucial to memory efficiency of top-down evaluators. Retaining these after query evaluation to support incremental additions imposes unacceptable overheads; e.g., the space usage for evaluating left-recursive reachability queries increases by 2-6 times. An alternative, similar to the approach used in prior works such as [GMS93], is to generate rules to capture the new answers due to addition of facts. We formalize the notion below.

Incremental Evaluation after Additions: Let P be a definite logic program and γ be a query. We denote the answers to γ with respect to P by $ans_P(\gamma)$. Let δ_p be a set of facts and rules added to the program P. The problem of incremental evaluation of query γ then is one of computing the smallest set Δ of answers such that $ans_{P \cup \delta_p}(\gamma) = \Delta \cup ans_P(\gamma)$. That is, Δ is the set of *new* answers for γ.

Given a definite logic program P and an added program δ_p we derive a transformed program P' used for incremental evaluation as follows. For each predicate p/n defined in the program $P \cup \delta_p$, we introduce an incremental predicate p'/n. If γ is an atom with p at its root, we denote by γ' the atom obtained by replacing the p in γ by p'. The transformed program P' is such that $ans_P(\gamma) \cup ans_{P'}(\gamma') = ans_{P \cup \delta_p}(\gamma)$.

First of all, P' contains all the clauses in P. For each fact α in δ_p we add α' to P'. For every clause of the form $\gamma :- \beta_1, \beta_2, \ldots \beta_n$ in the program $P \cup \delta_p$, we add the clause $\gamma' :- (\beta_1; \beta_1'), \ldots, (\beta_{i-1}; \beta_{i-1}'), \beta_i', \beta_{i+1}, \ldots, \beta_n$ for each $i \in [1, n]$. The i-th clause computes new answers of γ due to new answers of β_i.

For instance, the changes to the `r/2` relation given in Figure 8 can be computed by evaluating the predicate `r'/2` defined as follows (where the additions to `b/2` and `c/2` are given by the `b'/2` and `c'/2` relations):

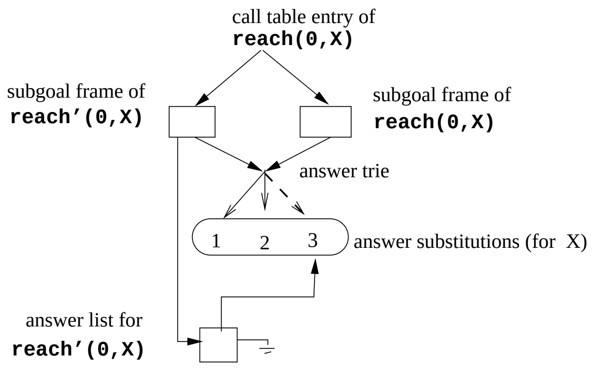


Figure 9: Example of the data structure to maintain tables for incremental predicates.

```
r'(X,Y) :- b'(X,Y).
r'(X,Y) :- c'(X,Z), r(Z,Y).
r'(X,Y) :- (c(X,Z); c'(X,Z)),r'(Z,Y).
```

The incremental predicate Δ`reach` defined in Chapter 1 is derived from the original definition of `reach` by the above transformation. The transformation is a straightforward application of finite differencing [PK82], and its variants have been widely used for materialized view maintenance [GMS93, LS03].

Direct evaluation of the transformed program has two sources of inefficiency. Firstly, the new answers of a query γ are actually added as answers to the new query γ'; consequently, we must merge the two answer tables after the incremental evaluation is complete. Secondly, to ensure that γ' computes only the new answers, each derived answer must be first checked against answers to the original query γ (e.g. using the goal $\neg\gamma$ [SJ96]), causing an extra table lookup.

We overcome these problems by sharing the call table entry and the answer tables between the incremental goal and the original goal, although the calls access the answer table in different ways. Let γ be an original goal and γ' be its incremental counterpart. The first call to γ' creates a new subgoal frame. Answers to γ' are computed by program clause resolution, are added to the answer table of γ, and also kept in a separate answer list. Subsequent calls to γ' consume from this answer list (even after completion of γ')— exactly the same way answers are currently consumed from incomplete tables.

In order to prevent answers to γ' from being accessed when backtracking through the answers of γ, we mark all the newly added answers as "deleted". This exploits the current implementation of tries in XSB which provides a flag to mark terms as deleted without physically removing them. This flag is used in XSB for maintaining dynamic asserted data using tries. Finally, when the incremental evaluation is complete, we reset the deleted flag of all answers in the answer list of γ', thereby adding these answers to γ. Let us consider that we have added the fact `b(4,8)` to the set of facts given in Figure 8. Figure 9 shows the states of the answer tables of `r(4,X)` and its incremental version `r'(4,X)`, just before the completion of the incremental evaluation.

Discussion: The data structure described above enables us to incrementally evaluate queries without changing any other part of the tabling engine. Moreover, in contrast to bottom-up techniques, we refresh a table only on demand, when a query is made. Note that we maintain only two versions of answers, and hence cannot maintain tables with varying "staleness". For instance, if γ_1 and γ_2 are two queries, when γ_1 is incrementally evaluated after changes to facts, observe that γ_1 has consumed all the new facts while γ_2 has not. However, since after this evaluation we merge all new answers with the old, γ_2 will remain stale. A promising approach for solving this problem is to associate timestamps with the added facts and answers, and drive the incremental computation based on the timestamps. With the timestamp based solution, we can maintain multiple versions of tables within the same data structure and hence handle tables with different degrees of staleness. This is a topic of future research.

4.2 Deletion

Let P be a definite logic program, γ be a query and δ_p be a set of facts F and set of rules R_d to be deleted from the program. For notational purposes, we assume that every rule is associated with an unique identifier (e.g see Figure 8). Following the notation used in Section 4.1, the problem of incremental evaluation of query γ after the deletion is that of computing a set Δ of answers such that $ans_{P-\delta_p}(\gamma) = ans_P(\gamma) - \Delta$. We develop an algorithm to efficiently compute the set Δ in this section and describe

Calls	Answers		Supports			
r(1,X)	a1	r(1,3)	s1	{f4,a3}	s10	{f5,a6}
	a2	r(1,7)	s2	{f5,a7}		
r(2,X)	a3	r(2,3)	s3	{f1}		
r(4,X)	a4	r(4,3)	s4	{f6,a1}		
	a5	r(4,7)	s5	{f6,a2}		
r(5,X)	a6	r(5,3)	s6	{f2}	s11	{f7,a4}
	a7	r(5.7)	s7	{f3}	s12	{f7,a5}
r(6,X)	a8	r(6,3)	s8	{f8,a4}	s13	{f9,a6}
	a9	r(6,7)	s9	{f8,a5}	s14	{f9,a7}

Figure 10: Supports

its implementation in terms of tabling data structures.

Formulation: Clearly, the only answers that can be in Δ are those that depend on δ_p. The algorithms based on the delete-rederive approach [GMS93, SS94] first overapproximate Δ by the set of <u>*all*</u> answers that depend on δ_p (deletion phase). The second phase (called the rederivation phase) rederives the answers in the overapproximation of Δ without using δ_p. We use a better approximation using the notion of *support* for an answer defined below.

Definition 5 (Support) *Let P be a program, and let T be a set of answer tables obtained when evaluating a query γ over P. A tuple $s = \langle k, \{\beta_1, \beta_2, \ldots, \beta_n\}\rangle$ is called a support of an answer α of γ if there exists a clause k of the form $\alpha' :- \beta'_1, \beta'_2, \ldots, \beta'_n$ and a substitution θ, such that $\alpha'\theta = \alpha$, and for all $i \in [1, n]$ $\beta'_i\theta = \beta_i$ and β_i is an instance of an answer in T or a fact in P.*

For instance, the supports corresponding to the answers corresponding to all possible bound-free calls to the predicate `r/2` over the program given in Figure 8 are given in Figure 10. We refer to each fact, answer and supports with names of the form $f0$, $f1$, …; $a1$, $a2$, …; and $s1$, $s2$, … respectively.

Note that each support for an answer represents one step in some derivation of the answer. We can construct a derivation of an answer by picking a support and constructing derivations for all the atoms in that support. However, the choice of

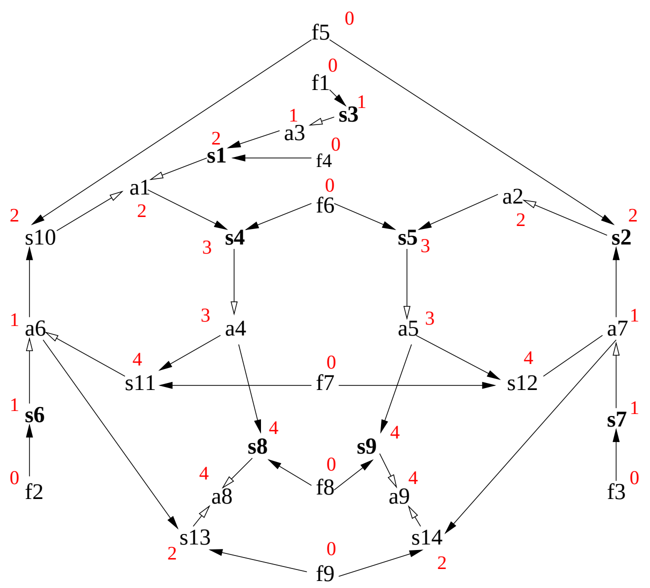


Figure 11: Support graph; the numbers corresponding to the nodes show the derivation lengths

support picked at each step is crucial to the construction of a valid (i.e. finite) derivation. For instance, picking support $s10$ each time to derive $a1$, and $s11$ each time to derive $a6$ would not lead to finite derivation of answer $a1$.

We maintain the relationship between answers and supports generated during query evaluation in form of a graph called support graph. We formally define support graph as follows.

Definition 6 (Support graph) *Let P be a definite logic program, and let T be a set of answer tables obtained when evaluating a query γ over P.*

The support graph for the evaluation of γ is a directed graph (V, E) where V contains the facts in P, answers in T and their supports. The set of edges E is such that

- $(b_i, s) \in E$ *for all supports* $s \in V$ *such that* $s = \langle k, \{b_1, b_2, \ldots b_n\}\rangle$, *and for all* $i \in [1, n]$. *We say that* $s \in b_i.uses\_of$ *and* $b_i \in s.part\_of$.
- $(s, a) \in E$ *for all* $a \in T$ *and* $s \in V$ *such that* s *is a support of* a. *We say that* $s.answer = a$ *and* $s \in a.support$.

The support graph corresponding to the supports of Figure 10 is given in Figure 11. Support graph makes it easy to determine the answers and supports that are affected by the deletion of facts. If we delete the fact $f5$ then we mark all the supports containing the fact, i.e. $s2$ and $s10$. Since $s2$ supports $a2$, following the traditional deletion strategy we mark $a2$ as affected. We continue the propagation of marks, marking support $s5$ and then answer $a5$. As answer $a5$ is in $s9$ and $s12$, they are also marked. As $s12$ supports $a7$, that answer will also be marked as affected. However, the answer $a7$ is rederived in the second phase due to the presence of fact $f3$. Note that $a7$ has two supports: $s12$ and $s7$, and that truth of support $s7$ is not dependent on the truth of $a7$. Thus support $s7$ has a derivation that is independent of answer $a7$. Since $s7$ is not marked, we can infer that $a7$ will be unaffected and hence need not be marked. Hence, *if we can quickly identify independence of supports and answers, we can limit the marking of affected answers*, and consequently reduce the rederivation effort also.

The key to quickly determine whether an answer is still derivable is to distinguish supports which can be selected without regard to the history and yet can build finite derivations. This is done using the notion of a acyclic support graph and primary support, defined below.

Definition 7 (Acyclic Support Sub-Graph (ASG)) *An acyclic support sub-graph (ASG) corresponding to a support graph (SG) is defined as a* directed acyclic subgraph *of SG which contains all the facts and answers of SG and at least one support for each answer.*

Definition 8 (Primary Support) *Primary Support Graph (PSG) is an acyclic support graph (ASG) where every answer has exactly one support. All supports in PSG are called primary supports.*

We compute the primary support for the non-incremental evaluation using the following procedure. Let $\alpha :- \beta_1, \beta_2 \ldots, \beta_n$ be the instance of a rule k that is used by tabled resolution to derive the answer α for the first time. Then $\langle k, \{\beta_1, \beta_2 \ldots, \beta_n\}\rangle$ is deemed as the primary support of α, and is denoted by $ps(\alpha)$. This method determines the primary support in a tabled evaluation at no extra cost.

In Figure 10 the first support listed for each answer is its primary support. We use primary supports to (over)approximate the set Δ of answers to be deleted as follows.

Definition 9 (Candidates for Deletion) *Let P be a program, γ be a query, and A be the answers computed during the evaluation of γ. Let $ps(\alpha) = \langle k, S \rangle$ be the primary support of $\alpha \in A$. The set of candidates for deletion due to the deletion of the program δ_p from P, denoted by $\Gamma(P, \delta_p)$ is the smallest set such that $\alpha \in \Gamma(P, \delta_p)$ whenever $\exists \beta \in S$ such that $\beta \in F \cup \Gamma(P, \delta_p)$ or rule $k \in R_d$.*

It is easy to establish that the set of candidates for deletion overapproximates the set of deleted answers. Formally,

Proposition 1 *The set of answers for a query γ over a program P, $ans_P(\gamma)$ is such that $ans_P(\gamma) - ans_{P-\delta_p}(\gamma) \subseteq \Gamma(P, \delta_p)$.*

Traditional two-phase delete-rederive algorithms such as [GMS93, SS94] use a coarser approximation. Also the effect of deletion of rules is not addressed in any view maintenance literature. The answers they delete in the first phase, which we denote by $\Gamma^\sharp$, can be characterized as follows. The set $\Gamma^\sharp$ is the smallest set such that $\alpha \in \Gamma^\sharp(P, \delta_p)$ if there is some support $s = \langle k, S \rangle$ of α such that $\exists \beta \in S$ and $\beta \in \delta_p \cup \Gamma^\sharp(P, \delta_p)$. It can be easily shown that $\Gamma(P, \delta_p) \subseteq \Gamma^\sharp(P, \delta_p)$. Note that the coarser approximation has a cascading effect: an answer marked incorrectly as a candidate, in turn, leads to (incorrect) marking of other answers. Our approximation reduces such propagation and hence is considerably less coarse. For instance, deletion of $f5$ marks $s10$ but does not mark $a1$, as its primary support $s1$ is not marked. This restricts the mark propagation to $s4$, $a4$, $s8$, $s11$ etc. Note that the notion of primary support is not specific to tabled evaluation and can be easily extended to any least fixed point computation.

Although only primary supports are used to obtain candidates for deletion, we still keep the set of all supports for an answer. First of all, note that due to the approximation, some candidates for deletion may be still derivable. We check this in the *rederivation* phase. Traditional algorithms in the view maintenance literature pose rederivation in terms of rule evaluation. In contrast, we avoid the proof search using an algorithm based on keeping counts and the set of all supports with each answer. Secondly, when a primary support is removed in the deletion phase, and the answer is still valid, we need to identify the new primary support; the new primary support can be easily generated from the set of all supports. Lastly, we can improve our approximation by finding all supports of an answer which are not dependent on the answer itself and falsify the answer when all such supports are falsified. Below we describe the data structures and primary support based algorithm for computing the candidates for deletion and for rederiving answers.

Data Structures: At first the direction of edges in the support graph may appear to be counter-intuitive. However, it coincides with the flow of information in our algorithm: we propagate deletion and rederivation from an answer-node a to all support-nodes s that contain a; and from a support-node s to answer-nodes a' for which s is a support. We also maintain an additional attribute called *primary_support* with each answer-node a denoting the primary support of a.

In the rederivation phase, we need to check if candidates for deletion identified in the first phase have alternative derivations. We do so efficiently by maintaining counts with each node in the support graph.

The meaning of the counts is different for answer-nodes and support-nodes. For an answer-node representing an answer a, its attribute *total_support_count* denotes the number of supports that a must *lose* before it becomes false. For example, in Figure 11 the count of answer-node $a1$ and $a2$ is 2. The count of an answer-node representing true fact or a rule is initially 1; it is decreased by 1 when the fact or the rule is deleted. For a support-node representing a support s, its attribute *false_count* denotes the number of false answers and facts in s; in other words, the count is the number of answers that must become true before the support itself becomes true. A support-node's attribute *false_count* enables us to quickly determine the truth value of a support without evaluating its constituents.

In Figure 11 the count of support-nodes $s1 - s14$ are all zero. Whenever an answer-node a becomes false, we increment the *false_count* of all support-nodes that contain a (given by $a.part\_of$). Similarly, when a support-node s becomes false, we decrement the count of the answer-node that is supported by s (given by $s.answer$).

The Algorithm: The algorithm for incremental evaluation after deletion of facts is shown in Figure 12 and has two phases as described below.

Deletion Phase: The algorithm starts in the marking phase which propagates the effect of deletion of facts and rules to answers and supports. We say an answer

```
mark()
  may_rederive_set = empty
  ∀ deleted facts f
    f.total_support_count=0;
    mark_answer(f)

mark_answer(a)
  a.marked = true
  if (a.total_support_count>0)
    add(may_rederive_set,a)
  ∀ s ∈ a.uses_of
    mark_support(s)

mark_support(s)
  s.false_count++
  if (s.false_count == 1)
    a=s.answer;
    a.total_support_count−−;
    if (a.primary_support==s)
      mark_answer(a);
```

```
rederive()
  ∀ a ∈ may_rederive_set
    if (a.total_support_count > 0 && a.marked)
      rederive_answer(a)

rederive_answer(a)
  a.marked = false
  let s ∈ a.support s.t. s.false_count = 0;
  a.primary_support = s;
  ∀ s ∈ a.uses_of
    rederive_support(s)

rederive_support(s)
  s.false_count−−
  if (s.false_count == 0)
    a=s.answer
    a.total_support_count++;
    if(a.total_support_count==1)
      rederive_answer(a);
```

Figure 12: Algorithm for primary support based incremental deletion.

is marked if its *marked* attribute is true and a support is said to be marked if its *false_count* is greater than zero. When the marking phase ends all the marked answers represent the candidates for deletion (Γ in Definition 9). The marking phase is as such straightforward, the only subtle part being its interaction with rederivation phase through the *may_rederive_set* which we explain while describing rederivation phase. We explain the algorithm using the support graph in Figure 10. Consider the deletion of fact $f1$ (`b(2,3)`). This marks the answer $a3$ as its primary support $s3$ is marked. Note that as $a3$ does not have any unmarked support, and thus it is not added to the *may_rederive_set*. At the end of marking phase we mark the answers $a3$, $a1$, $a4$ and $a8$ with only $a1$ and $a8$ in the *may_rederive_set*. It must be noted that traditional delete-rederive algorithms would have also picked $a6$ as a candidate for deletion.

Rederivation Phase: We say a support is to rederived if its *false_count* decreases from 1 to 0 and an answer to be rederived if its marked attribute is changed to false. There are two ways of rederiving a marked answer - if the answer has an unmarked support after the marking phase is over, or a support of the answer is rederived. For instance, $a1$ and $a8$ can be readily rederived due to presence of their respective unmarked supports $s10$ and $s13$. The *may_rederive_set* only contains these answers which may be potentially rederived in this manner. The other marked answers can be rederived due to rederivation of supports and thus does not require checking the existence of unmarked support. Note that it is also possible for an answer in the *may_rederived_set* to be rederived due to rederivation of its support. For instance $a8$ can also be rederived due to rederivation of support $s8$. The support $s8$ along with $s11$ is rederived due to rederivation of $a4$ which is rederived because of rederivation of $s4$ which in turn is caused by rederivation of $a1$.

Note that we assure that an answer is marked and rederived at most once. Below we present the complexity analysis of the algorithm.

Complexity Analysis: The complexity of the algorithm given in Figure 12 is given by the following expression:

$$O(|M_a|) + O(\sum_{a \in M_a} |a.uses\_of|) + O(\sum_{a \in M_a} |a.support|) + O(|M_s|)$$

where M_a and M_s denote the set of marked answers and supports, respectively. The total number of answers marked and rederived is given by $O(M_a)$. The total number of times the *total_support_count* of answers is adjusted is given by $O(M_s)$. The number of times falsecount of all supports can increase and decrease is given by the expression $O(\sum_{a \in M_a} |a.uses\_of|)$. To determine the primary support at the time of rederiving an answer the algorithm checks the support set of the answer- this takes time $O(\sum_{a \in M_a} |a.support|)$.

Rederivation phase also determines a primary support for each rederived answer. Note that any unmarked support of an answer before the answer is rederived can be chosen as primary support. We generalize the idea of primary supports in the following discussion.

Acyclic Supports: In the above example, $a1$ was marked since its primary support $s1$ was marked. Note that, its other support $s10$ is also independent of $a1$: i.e. if $a10$ is derivable, so is $a1$. We call these supports as acyclic supports. We formalize the notion in the following definition.

Definition 10 (Acyclic Supports) *Supports in an ASG are called acyclic supports.*

We can now generalize the primary support based algorithm by keeping acyclic supports with each answer, and marking an answer only if all its acyclic supports are marked. In the above example, if we keep $s10$ as an acyclic support of $a1$ then $a1$ will not be marked, and consequently, $a4$ and $a8$ are not marked and rederived.

Note that, there can be multiple ASGs corresponding to a support graph. Finding an ASG and corresponding acyclic supports becomes a key problem in this case. We now describe an algorithm to find ASG based on derivation length of answers. Given a vertex v in the support graph, $dl(v)$ is defined in Figure 13.

$$\begin{cases} 0 & \text{if } v \text{ is a fact} \\ \max\{dl(a) \mid v \in \texttt{uses\_of}(a)\} + 1 & \text{if } v \text{ is a support} \\ dl(s) \mid s \text{ is the primary support of } v & \text{if } v \text{ is an answer} \end{cases}$$

Figure 13: Derivation lengths

Definition 11 (Derivation Based Supports) *A support s is called derivation based support if $dl(s) \leq dl(\texttt{s.answer})$.*

Corollary: Primary support is also a derivation based support.

Definition 12 (Derivation length based Support Graph (DSG)) *Derivation based support graph DSG is defined as the subgraph of support graph SG which contains all the facts and answer of SG and all derivation based supports in SG.*

Theorem 2 *A DSG is an ASG.*

Proof sketch. We need to prove that DSG does not have a cycle. We prove this by contradiction. Consider there exists a cycle $a1, s1, a2 \ldots, a1$. From the definition of derivation length it follows derivation length of every node in the sequence $a1, s1, \ldots, a1$ is increasing (may not be strictly increasing) and $dl(s1) > dl(a2)$. Thus we lead to a contradiction that $dl(a1) > dl(a1)$. Proved.

For the following chapters we refer derivation based supports as acyclic supports. In what follows we present an acyclic support based algorithm derived from the primary support based algorithm. Instead of keeping *primary_support* associated with every answer, we keep the number of unmarked acyclic supports recorded for that answer in the attribute *acyclic_support_count*. This attribute has value 1 for a fact. As derivation length of an answer can change due to deletion of fact, we store the derivation length of answers and supports in an attribute dl associated with them. We present the changed algorithm in Figure 14. We show the derivation lengths of answers and supports in Figure 11.

The marking phase of the acyclic support based algorithm is similar to that of primary support based algorithm, the only difference being the condition which confirms that an answer of marked only if all its acyclic supports are marked. For example, deletion of fact $f2$ marks $s6$, $a6$, $s10$ and $s13$. The important point to notice in the rederivation phase is that an acyclic support for an answer may not remain acyclic after rederivation. This is because, due to the incremental deletion we change from one ASG to another ASG of the support graph. The new acyclic support of $a6$ is $a11$ whose derivation length is 4. Thus $a6.dl = 4$ and hence $s10.dl = 5$ and $s13.dl = 5$. Consequently, $s10$ and $s13$ are no longer acyclic. The ASGs before and after the incremental deletion of fact $f2$ are given in Figure 15.

Complexity Analysis: The complexity of the algorithm given in Figure 14 is given by the following expression:

$$O(|M_a|) + O(\textstyle\sum_{a \in M_a} |a.uses\_of|) + O(\textstyle\sum_{a \in M_a} |a.support|) + O(|M_s|)$$

where M_a and M_s denote the set of marked answers and supports, respectively.

Discussion: Supports for an answer are constructed based on the rule that generated the answer. In our implementation, when inserting an answer in a table we determine its supports by using the consumer choice points and other structures used to generate the answer. Hence the construction of the support graph does not increase the time complexity of query evaluation. However, the space requirement for the support graph typically exceeds that of the answer tables. For instance, the number of answers for query `reach(X,Y)` is $O(n^2)$ where n is the number of vertices in

```
mark()
  may_rederive_set = empty
  $\forall$ deleted facts f
    f.total_support_count=0;
    f.acyclic_support_count=0;
    mark_answer(f)

mark_answer(a)
  a.marked = true
  if (a.total_support_count>0)
    add(may_rederive_set,a)
  $\forall$ s $\in$ a.uses_of
    mark_support(s)

mark_support(s)
  s.false_count++
  if (s.false_count == 1)
    a=s.answer;
    a.total_support_count--;
    if (s.dl$\leq$a.dl)
      a.acyclic_support_count--;
     if (a.acyclic_support_count==0)
        mark_answer(a);
```

```
rederive()
  $\forall$ a $\in$ may_rederive_set
     if (a.total_support_count > 0
        && a.acyclic_support_count==0)
        rederive_answer(a)

rederive_answer(a)
  a.marked = false
  a.acyclic_support_count=a.total_support_count
  a.dl=max$\{$s.dl$|$s$\in$ a.support, s.false_count==0$\}$
  $\forall$ s $\in$ a.uses_of
     s.dl=max$\{$s.dl,a.dl+1$\}$
     rederive_support(s)

rederive_support(s)
  s.false_count--
  if (s.false_count == 0)
     a=s.answer
     a.total_support_count++
     if (a.acyclic_support_count==0)
        rederive_answer(a);
     else
        if (s.dl$\leq$a.dl)
           a.acyclic_support_count++
```

Figure 14: Algorithm for acyclic support based incremental deletion.

the graph. For each answer, there may be up to n supports, and hence the support graph has $O(n^3)$ nodes.

4.3 Experimental Results

We implemented our incremental algorithms by modifying XSB. In the following we present results of our preliminary experiments designed to measure (i) the effectiveness of our techniques, (ii) the effect of repeated incremental evaluation, and (iii) the effectiveness of using supports to control deletion, and (iv) overheads. We used left-recursive reachability and same generation predicates over trees and complete graphs, and pointer analysis as benchmarks.

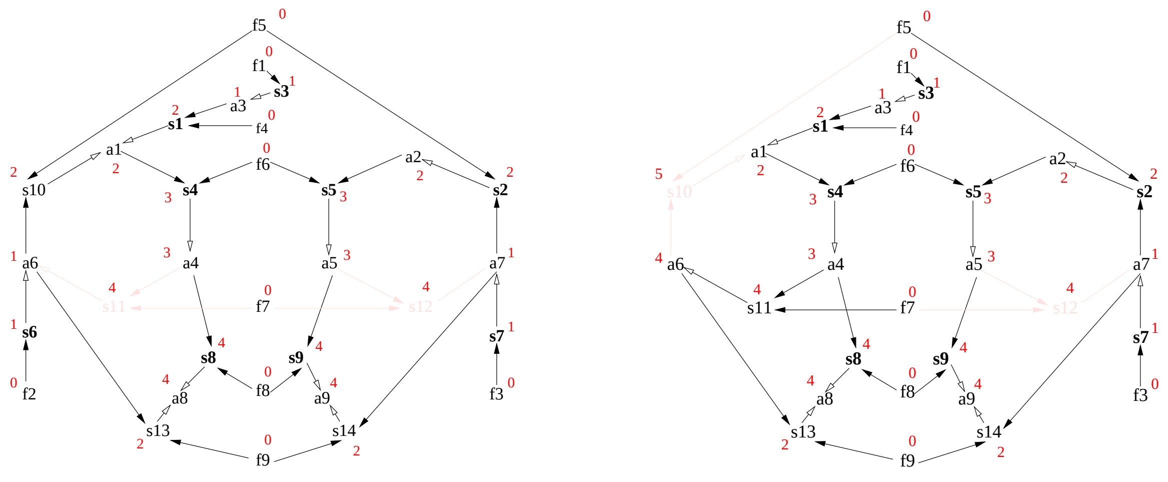


Figure 15: Acyclic support graphs before(a) and after(b) deletion of fact $f2$

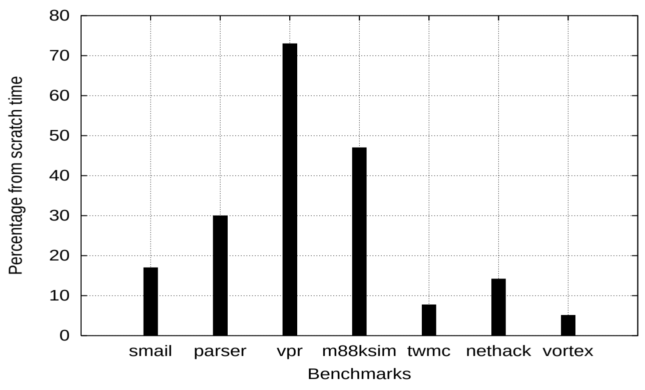
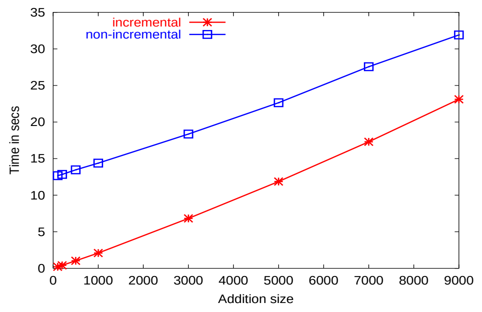
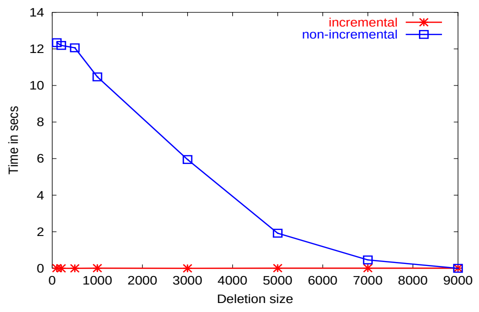


Figure 16: All Points-To Analysis: Incremental addition time as percentage of from-scratch time

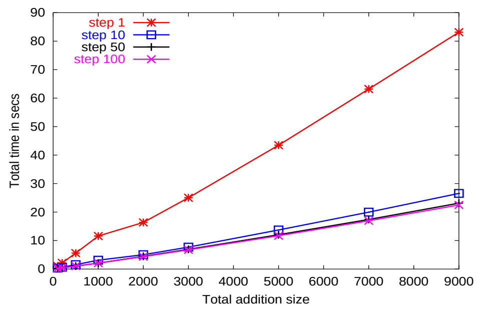
Incremental Addition We measured the performance of incrementally maintaining the points-to sets for APA when new statements are added, with the same random assignment statements used for incremental deletion. We computed the points-to relation after deleting a selected statement, then added back the deleted statement and ran the transformed program for incremental addition. The average time taken to run the incremental addition query as a percentage of the from-scratch re-evaluation time for each benchmark is shown in Figure 16. Observe that, for `m88ksim` and `vpr` incremental addition takes about 45-73% of the from-scratch time. However, for large



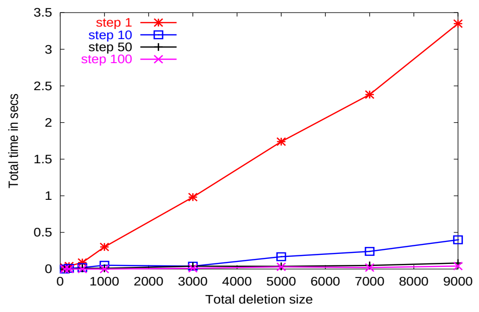
(a) Addition [reach(a,X);tree(9999 edges)]



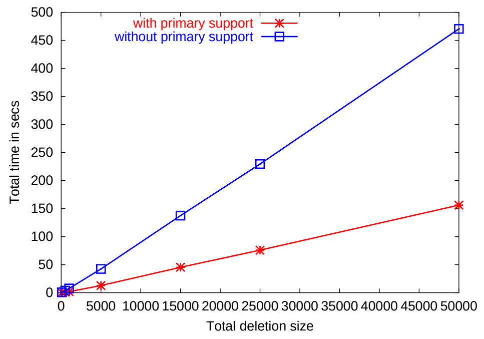
(b)Deletion [reach(a,X);tree(9999 edges)]



(c) Repeated additions
reach(a,X);tree(9999 edges)



(d) Repeated deletions
reach(a,X);tree(9999 edges)



(e) Deletions [reach(a,X); complete graph(89700 edges); step 1]

Figure 17: Experimental results for reachability analysis

benchmarks (`nethack`, `twmc` and `vortex`) it takes only 5% of the from-scratch evaluation time. Note that the time reflects the effect of addition on *all* derived relations.

Figures 17(a) compare the incremental and non-incremental evaluation time of one query after the *addition* of a set of facts to the edge relations for left-recursive reachability query. The figures do not include the times for evaluating the initial query. Observe that when the size of addition is small (less than 5% of the total size of the edge relation), incremental evaluation is 20–60 times faster than non-incremental evaluation. Moreover, as the size of the addition increases, incremental evaluation time approaches that of non-incremental evaluation, but remains lower even when the size of addition is over 90% of the original edge relation.

Incremental Deletion We performed three experiments to measure effectiveness of support graph based deletion algorithm. The results are shown in Table 2. The 'DRED' approach refers to the support graph based algorithm where we mark an answer as deleted if any one of its supports is marked. Unmeasurable small times and percentages (< 0.1) are denoted by '-'. The results show tremendous advantage of acyclic supports over 'DRED' approach. The explanation for this timing behavior can be well understood by comparing the number of answers marked in these approaches against the actual number of deleted answers. This is presented in Table 3.

Figure 17(b) compares the times of one query after the *deletion* of a set of facts from the edge relation for reachability analysis. Incremental deletion on this benchmark takes very little time (less than 0.1s for all deletion set sizes) and far outperforms its non-incremental counterpart. The two are comparable only when the input graph becomes very small due to deletion of a large number of edges.

Figure 17(e) compares the performance of the incremental engine with DRED and Primary Support based approach. Both versions used support graphs for rederivation. We measured the performance of the two versions for a sequence of deletions from a complete graph, issuing a query after each deletion. Observe from Figure 17(e) that use of primary supports results in a more than 3-fold reduction in evaluation time.

Effect of repeated evaluation: We now compare the performance of the incremental engine for sequences of query evaluations and changes (additions/ deletions) such that the changes are interspersed between evaluations. In all the runs, the total number of changes is the same; the number of changes between two query evaluations is the step size of the run. For instance, a run with step size 10 means that after every 10 changes we issue an incremental query to refresh the table. Figure 17(c) and (d) show the total evaluation times for additions and deletions, respectively, for runs with different step sizes. Observe from the figure that batching changes together (i.e. querying infrequently, and hence allowing tables to go stale) usually takes less time than maintaining the tables fresh all the time (i.e. step size 1). Nevertheless,

Benchmarks	DRED	Primary Spt.	Acyclic Spt.
smail	44	2	1
parser	32	-	-
vpr	3	-	-
m88ksim	1	-	-
twmc	21	-	-
nethack	33	-	-
vortex	45	-	-

Table 2: Incremental deletion times as percentage of from-scratch time

Benchmarks	DRED	Primary Spt.	Acyclic Spt.	Actual Deletion
smail	230.6K	13,282	8,812	8,012
parser	245.7K	1,095	965	446
vpr	20.8K	288	144	133
m88ksim	4.2K	1,086	633	618
twmc	51.3K	2770	615	478
nethack	54.5K	355	187	172
vortex	493.4K	2,882	545	455

Table 3: No. of answers marked in Deletion Phase

consistently maintaining freshness is only only 3 to 5 times slower than refreshing the table only after all changes are done. This reflects the low overheads for incremental query evaluation.

Overheads: We find that the initial query evaluation time for incremental evaluation is at most 8% greater than that for non-incremental evaluation. However, since the support graph size may be much larger than the answer set size, we do observe

Benchmarks	Support Count	Size in MB
smail	3,159.4K	92.2
parser	1,355.7K	44.2
vpr	213.1K	9.7
m88ksim	303.5K	11.8
twmc	5,728K	158.5
nethack	2,075K	59.4
vortex	33,444K	912.0

Table 4: Support graph sizes

significant memory overheads. Memory overheads range from a factor of 1.9 (49MB incremental vs. 26MB non-incremental) for same generation queries over binary trees, to as much as 131 (2.5MB incremental vs. 19KB non-incremental) for reachability over complete graphs.

We measured the support graph sizes for pointer analysis of various applications (Table 4). It is evident from the figures that for large applications support graphs are huge and may not fit into the main memory. This probed us to develop space efficient scalable incremental algorithms for practical use.

4.4 Related Work and Discussion

In this section we presented the first techniques for incrementally evaluating tabled logic programs. The relationship of our work to the DRed algorithm [GMS93] has been explained in sections 4.1 and 4.2. We can handle programs with stratified negation in the same way as DRed algorithm. It should be noted that the idea of counts has been used in other works such as [GKM92, SS94] but has not been used to avoid recomputing subgoals in recursive rules. The transformation rules for supporting addition are similar to those of [GMS93, SJ96] but we evaluate the incremental rules top-down and on-demand. We also use specialized data structures to efficiently generate new answers and avoid propagation of generation of old answers. Our techniques are also closely related to the incremental model checking algorithm (MCI) of [SS94]. We have adopted MCI's use of counts to efficiently compute truth values of nodes during incremental evaluation. Our idea of primary support is very similar to well-founded support of Doyle's TMS [Doy79]. Also the idea of acyclic support has a close resemblame to spanning graph algorithm in Swamy's work ([SBS95]) where a spanning graph is kept as a justification of the reachable state space. A reachable node is not marked unless all its predecessor node in spanning graph is marked. However, we exploit the fact that the first support generated in a non-incremental computation is acyclic and can be identified as primary support. This gives an efficient way of finding primary support. Our derivation length based heuristics to find acyclic supports is also different from Swamy's algorithm to determine spanning graph. Note that we differ from Swamy's algorithm in the rederivation phase as we rederive based on the unmarked supports. However, Swamy's algorithm needs to perform reachability query for rederivation.

Straight Delete (StDel) algorithm [LMSS95] eliminates the rederivation phase of DRed by keeping the entire proof with every answer. While such an approach may be feasible for constraint databases, it is prohibitively expensive for logic programs. For instance, while the support set size for a context free grammar parser is cubic in the length of the string, the number of distinct proofs may be exponential. Thus a succinct representation such as a support graph is essential. However, since we do not keep all the proofs, we cannot avoid rederivation.

Our implementation shows that incremental evaluation in the presence of addition of facts and rules can be added without any overhead whereas there is a tradeoff between memory overhead and performance in presence of deletion of facts and rules.

Chapter 5

Combating Space Issues

In the last chapter, we described a data structure, called support graph, for efficient incremental evaluation of tabled logic programs. The support graph records the dependencies between answers in the tables, and is crucial for efficiently propagating the changes to the tables when facts are deleted. As evident from the results shown in the last chapter, incremental computation with support graphs are hundreds of times faster than from-scratch evaluation for small changes in the program. However, the graph typically grows faster than the tables themselves, making it impractical to maintain the full support graph for large applications.

In this chapter we present two techniques for combating space requirement of support graph. The first technique (Section 5.1) is based on Partial Support Graph which keeps bounded number of supports along with every answer thereby has space complexity linear to the table size. Although the linear bound in space overhead in all cases makes this attractive for many applications of incremental table maintenance, we formulated a second technique which shows better time performance than the former and shows similar space efficiency for an useful set of tabled logic programs. The method explores the commonality between multiple supports originated in certain types of programs and represents the entire support graph symbolically (Section 5.2).

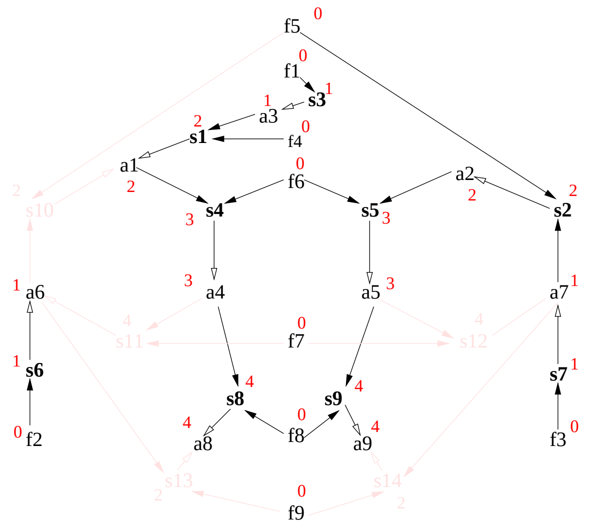


Figure 18: Primary support graph

5.1 Partial Support Graph

In this section we describe space-efficient techniques for incremental evaluation of logic programs. Below, we describe three new algorithms that keep the space usage bounded by keeping only a part of the support graph, yet exhibit acceptable time performance. The first is a memory efficient algorithm which keeps only the primary support for each answer; the subsequent algorithms expand on the earlier ones, trading off a bounded amount of space to obtain better time performance. We begin with the description of `PS`, the primary-support-based algorithm. We describe this technique using the example given in Figure 8.

5.1.1 Algorithm Primary Support

In the algorithm PS (Primary Support) we record only the primary support for each answer. The resultant support graph is called primary support graph or PSG. The PSG at the beginning of incremental phase is shown in Figure 18. With each answer vertex, we keep two additional fields: `call`, a pointer to the call to which it is an answer, and `all_support`, a boolean flag to record whether we have recorded all supports for the answer. In this case, as we are recording only the primary support

```
deletion_phase()
    ∀deleted fact f
        ∀support s ∈ f.uses_of
            mark_support(s);

mark_support(support s)
    s.false_count++;
    if(s.false_count==1)
        a=s.answer;
        mark_answer(a);

mark_answer(answer a)
    a.marked=true
    if(a.all_support==false)
        a.call.dirty_count++;
        ∀s ∈ a.uses_of
            mark_support(s);
```

Figure 19: Primary support based deletion

with each answer, true value of this boolean flag signifies that primary support is the only support for the answer. Since we have only incomplete support information, rederivation of an answer may be performed using program clause resolution (PCR) of the corresponding call (see below for details). For each call we keep a count of the number of marked answers in a field called `dirty_count`. This field is also initialized to zero before each incremental phase. We refer to a call as dirty if its `dirty_count` is greater than zero.

When an answer is derived for the first time we generate its primary support, and set its `all_support` to true. If the answer is derived again we discard the new support and make `all_support` false. The incremental deletion algorithm consists of two phases, described below:

Phase 1: Deletion phase. The algorithm for this phase is given in Figure 19. Consider the deletion of the fact $f4$. The deletion phase marks the answers $a1$, $a4$, and $a8$, and it makes `dirty_count=1` for the calls `r(1,X)` (call of $a1$) and `r(6,X)` (call of $a8$) but not for the call `r(4,X)` (call of $a4$) since the `all_support` field of $a4$ is true (see the rederivation phase, below).

Phase 2: PCR Rederivation phase. To check if an answer marked for deletion can be rederived, we can perform program clause resolution (PCR) for the corresponding call. PCR, however, is expensive since it may rederive answers not even marked for deletion in the first place. For instance, computing `r(1,X)`, `r(4,X)` and `r(6,X)` by PCR to rederive $a1$, $a4$ and $a8$ will also derive answers $a2$, $a5$ and $a9$. Hence we devise two ways to avoid PCR-based rederivation whenever possible.

The first is based on `all_support` flag of an answer. If this flag is true, then the recorded supports (in algorithm PS only the primary support) for an answer are the existing supports for this answer. Hence, in this case, the only way the answer can be true is if the deletion mark on the primary support is removed. Note that here we are considering only definite logic program where deletion of facts/rules can only result in falsification of supports. For instance, we do not have to execute call for `r(4,X)` to know whether `r(1,4)` (answer $a4$) is true, as we know that if `r(1,3)` (answer $a1$) is true then that will subsequently make support $s4$ and answer $a4$ true by removal of marks through support graph. Thus for a marked answer with `all_support` bit set, we do not increment the `dirty_count` for its call.

The second heuristic uses the same notion of *derivation length* (denoted by dl) discussed in Section 4.2. Intuitively, if an answer a_1 has a lower dl than answer a_2, then a_1 is independent of a_2, and support of a_2 may be dependent on a_1. Thus rederivation of a_1 may rederive a_2 by removal of deletion mark through partial support graph. Thus if answers with lower dls are rederived earlier, then PCRs for rederiving higher dls may be avoided altogether.

To ensure that a call is issued only when necessary, we keep the total number of its marked answers in `dirty_count`, and issue a call only if its `dirty_count` is non-zero. Whenever an answer is rederived using the support graph and its `all_support` is false, we decrement the `dirty_count` of its call. In the above example, since answer $a1$ has lower derivation length (2) compared to that of answer $a8$ (4), we issue the call `r(1,X)` first. This rederives the answer $a1$ along with generation of new support $s10$. By calling `rederive_answer(`$a1$`)` and `rederive_support(`$s4$`)` we rederive the answer $a4$ and, subsequently, calling `rederive_answer(`$a4$`)` and `rederive_support(`$s8$`)` we rederive $a8$. This decrements the `dirty_count` for the call `r(6,X)` to 0 and thereby we save the PCR rederivation for the call `r(6,X)`. The algorithm for rederivation

```
rederivation_phase()
    queue = set of all marked answers
    while(queue is not empty)
        remove ans from queue with minimum dl;
        call=ans.call;
        if((!considered_for_rederivation(call))
                    && dirty_count(call)>0)
            consider_for_rederivation(call)=true;
            execute_query(call);
            ∀newly generated answers ans
                if(marked_deleted(ans))
                    rederive_answer(ans);

rederive_answer(answer ans)
    ans.marked=false;
    if(ans.all_support==false)
        ans.call.dirty_count−−;
    ∀s ∈ ans.uses_of
        rederive_support(s)

rederive_support(support s)
    s.false_count−−;
    if(s.false_count==0)
        ans=s.answer;
        rederive_answer(ans);
```

Figure 20: PCR rederivation for algorithm PS

phase is shown in Figure 20.

5.1.2 Algorithm Acyclic Support

Our practical experience with algorithm PS showed that most of the time for incremental deletion is spent in PCR rederivation. One way to reduce expensive rederivation calls is to make less number of calls dirty which can be achieved by reducing the number of answers marked. In Chapter 4 we have generalized the primary support based algorithm by keeping all possible acyclic supports with every answer and mark an answer only if all its acyclic supports are marked; thus reducing the number of answer marked and rederived compared to primary support based algorithm. We follow similar generalization of Algorithm PS to obtain Algorithm AS (Acyclic Support) by storing only the acyclic supports with every answer, however we limit the maximum number of acyclic supports stored for an answer (called the Maximum Acyclic Support Count, or MASC) to keep the size of Acyclic Support Graph (ASG) bounded.

Algorithm AS is derived from Algorithm PS as follows. With each answer we now keep the number of unmarked acyclic supports recorded for that answer in `acyclic_support_count`. In `mark_support`, when `false_count` of a support becomes 1, we decrement the `acyclic_support_count` of its answer. We mark an answer (and

```
rederive_support(support s)
    $s.false\_count--;$
    $if(s.false\_count==0)$
        $ans=answer\_of(s);$
        $dl(s)= maximum\{dl(a)|\ s \in uses\_of(a)\}+1;$
        $if(ans.acyclic\_support\_count==0)$
            $ans.acyclic\_support\_count++;$
            $dl(ans) = dl(s);$
            $rederive\_answer(ans);$
        $else$
            $if(dl(s)\leq dl(ans))$
                $ans.acyclic\_support\_count++;$
```

Figure 21: PCR rederivation for algorithm AS

propagate this mark) only when its `acyclic_support_count` falls to zero. It is easy to see that our Algorithm PS is a special case of Algorithm AS with MASC=1. Consider the example in Figure 8. With MASC=2, we will store $s10, s13$ and $s14$ as additional acyclic supports (apart from the primary supports); When fact $f4$ is deleted, Algorithm AS will not mark any answer. The modified procedure `rederive_support` of Algorithm PS is shown in Figure 21.

5.1.3 Algorithm Mixed Support

Algorithm AS improves on PS by reducing the number of answers marked in the first phase. We now describe Algorithm MS (Mixed Support), which builds on AS, and aims to reduce the number of PCR rederivations. In AS we bound the number of acyclic supports for each answer by the constant MASC. For some answers we might not fill this "quota" as the number of acyclic supports may be less than MASC. Algorithm MS fills the rest of the "quota" for each answer with other (possibly cyclic) supports, while giving preference to acyclic supports.

For instance consider our running example. Let MASC=2, and assume that we keep supports $s11$ and $s12$ in the support graph (as supports to $a6$ and $a7$ respectively) even though they are do not meet the acyclicity criterion described before. Now consider the deletion of fact $f2$. In the deletion phase we will mark $s6, a6, s10$ and $s13$. Since $s11$ remains unmarked at the end of this phase, we know that the support

```
gred()
  $rederive\_list=\{\}$
  $\forall marked\ answers\ ans$
    $if\ (ans.total\_support\_count>0)$
      $ans.acyclic\_support\_count=ans.total\_support\_count;$
      $dl(ans) = max\ \{dl(s)|\ s\ is\ a\ support\ of\ ans, s.false\_count=0\}$
      $rederive\_list=rederive\_list + ans$
  $\forall ans \in rederive\_list$
      $rederive\_answer(ans);$
```

Figure 22: Support graph based rederivation

$s11$ has a derivation that is independent of $a6$. Thus $s11$ now qualifies as an acyclic support of $a6$. Hence we can remove the mark on $a6$ without PCR rederivation. We derive Algorithm MS from Algorithm AS by invoking `gred` (Figure 22) which does a support graph based rederivation before doing PCR rederivation. In addition to the data for Algorithm AS, this algorithm maintains the total number of current supports for an answer, and also a structure to access all supports for a given answer. This strategy does not affect the number of answers marked, but reduces the number of PCR rederivations.

Discussion For incremental deletion, the time taken by the deletion phase is proportional to the size of the partial support graph, and hence bounded by the time taken for from-scratch analysis of the original program. Rederivations done on the basis of the partial support graph is also proportional to the size of the graph. Now consider the rederivations that require program clause resolution. Note that PCR rederivation invokes calls that had been made for the original analysis, and hence the PCR rederivation time is bounded by the from-scratch analysis time. Therefore, incremental deletion, which comprises of deletion and rederivation phases, takes time proportional to that of from-scratch analysis of the original program.

The correctness of our incremental addition and deletion algorithms follows from the correctness of DRed algorithm [GMS93]. Additionally our deletion algorithm identifies a set of acyclic supports for an answer. It is easy to construct a proof for an answer using an acyclic support, by recursively building proofs of answers in the

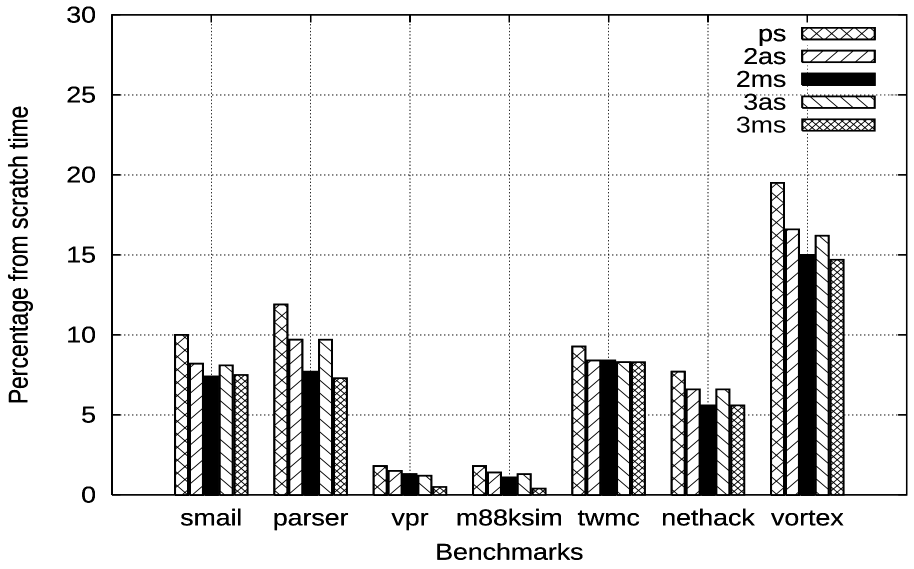


Figure 23: All Points-To Analysis: Incremental deletion time relative to from-scratch time

support. Since the support is acyclic, this recursive process will terminate, yielding a proof. Hence it follows that any answer with an unmarked acyclic support has a derivation and hence is not deleted.

5.1.4 Experimental Results

Incremental Analysis To measure the effectiveness of our incremental evaluation algorithms, we performed All Points-to Analysis as described in Chapter 3. For single statement-level changes to the benchmark programs, we measured the time and space taken to redo the analyses from scratch and to maintain the points-to relations using our incremental techniques. We report on the performance of incremental deletion based on partial support graph.

Effectiveness of incremental deletion. We compare the average time taken for single assignment statement deletion in source (over 105 deletions) for incremental and from-scratch APA. Note that each assignment statement in the source may correspond to multiple primitive assignment statements in the preprocessed code. Figure 23

	PS	2AS	2MS	3AS	3MS	ALL
smail	398K	435K	555K	450K	705K	3160K
parser	366K	435K	510K	480K	630K	1356K
vpr	147K	167K	174K	176K	191K	213K
m88ksim	161K	180K	210K	195K	225K	304K
twmc	279K	389K	397K	490K	506K	5728K
nethack	205K	239K	270K	252K	317K	2075K
vortex	1124K	1564K	1714K	1810K	2099K	33444K

Table 5: Support counts for different support graphs

shows the average time taken for incremental APA *per source statement deletion* as a percentage of the time taken for from-scratch APA. The figure shows the relative performance of the different incremental deletion algorithms PS, AS and MS, and for the latter, with different values of maximum support set size (MASC=2 and 3). Recall that PS is identical to AS and MS with MASC=1.

Observe from the figure that, even the simplest of our primary support-based algorithms, PS, takes 1.8%(m88ksim) to 19%(vortex) of the from-scratch time. Time decreases with increasing MASC, and the MS algorithm performs better than AS. This is because most of the time (more than 95%) is taken by the PCR rederivation phase and keeping extra supports considerably reduces the number of calls made for PCR rederivation.

Space Behavior. In Table 5 we compare the space overhead of our different incremental deletion algorithms by comparing the total number of supports recorded for APA. The last column in the table shows the total number of supports in complete support graph for each benchmark. Each support vertex takes at most 6 words, and the total number of supports is a very good measure of the space overheads due to incremental evaluation[1] . Note that the support graph space overheads are very small when the number of supports per answer is bounded. However, note that the size of full support graphs is prohibitively large for the bigger examples (e.g. 30M vertices, 912 MB memory for `vortex`), and hence the simpler incremental algorithm of Chapter 4 is impractical. It is interesting to observe that the support size for MS with MASC=2 is smaller than AS with MASC=3, but the average evaluation time is longer for the

[1]The amount of space each support takes varies from Algorithm PS (2 words), AS and MS (6 words).

	MASC=0	1	2	3
smail	230.6K	13,282	9,540	9,270
parser	245.7K	1,095	1,065	1,050
vpr	20.8K	288	150	150
m88ksim	4.2K	1086	690	680
twmc	51.3K	2,770	2172	1933
nethack	54.5K	355	208	208
vortex	493.4K	2,882	614	614

Table 6: No. of answers marked in Deletion Phase

latter (Figure 23), showing the advantage of keeping non-acyclic supports in reducing evaluation time.

The importance of supports. To measure the effectiveness of primary and acyclic supports in reducing the number of answers marked and later rederived, we collected the average number of answers marked in the first phase for APA. Table 6 shows the number of answers marked using the AS algorithm for various benchmarks (rows) and for different MASC values (columns). Note that AS and MS are identical w.r.t. number of marked answers, and that AS with MASC=0 is identical to the non-support-based (e.g. [GMS93, YRL99, SS94]) algorithms. The comparison of MASC=0 and MASC=1 columns shows the substantial advantage (4–160 times) of keeping primary support for restricting the marking and rederivation of answers. The number of marked answers decreases as we increase the number of acyclic supports stored. However, the decrease tapers off after MASC=2, and MASC=2 appears to be a good balance between evaluation time and space overhead.

Tables 5 and 6 show the importance of keeping a *partial* support graph for deriving scalable incremental analyses. Note that the PCR rederivation phase is needed only when the support set is partial. We observe that this phase takes more than 95% of the incremental evaluation time. Hence, if the entire support graph is kept, the incremental evaluation time will be less than 2% of the from-scratch evaluation time. The algorithms presented in this section hence trade off incremental evaluation time to lower the space requirements.

5.2 Symbolic Support Graphs

The algorithm presented in the last section stored only a limited number of supports for each answer, making its space requirement linear in the number of answers. The space savings and scalability however come at the price of increased rederivation time. Since all supports are not stored, an answer may have a derivation even when all of its stored supports are marked. Hence, we need to re-evaluate the program clauses to check if the answer can be rederived. The time penalty can be high: incremental evaluation time (for small changes) may be as much as 15% of that of from-scratch evaluation.

This raises an interesting question: *Can we store the entire support graph, which eases rederivation and significantly improves incremental evaluation time, without incurring a prohibitive space overhead*? We address this problem in this section. To explain various characteristics of our approach we take another example given in Figure 24(a). The corresponding support graph is shown in Figure 25.

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

b(1,2). %f1
b(6,2). %f2
b(6,4). %f3

c(1,6). %f4
c(3,6). %f5
c(3,1). %f6
c(6,3). %f7
```

(a)

Calls	Answers	Supports
`r(6,X)`		
	[a1] `r(6,2)`	[s1] **{**`b(6,2)`**}**, [s10] **{**`c(6,3),r(3,2)`**}**
	[a2] `r(6,4)`	[s2] {`b(6,4)`}, [s11] {`c(6,3),r(3,4)`}
`r(3,X)`		
	[a3] `r(3,2)`	[s3] **{**`c(3,6),r(6,2)`**}**, [s8] {`c(3,1),r(1,2)`}
	[a4] `r(3,4)`	[s4] {`c(3,6),r(6,4)`}, [s9] {`c(3,1),r(1,4)`}
`r(1,X)`		
	[a5] `r(1,2)`	[s5] {`b(1,2)`}, [s6] **{**`c(1,6),r(6,2)`**}**
	[a6] `r(1,4)`	[s7] {`c(1,6),r(6,4)`}

(b) (c)

Figure 24: Example program (a); calls and answers generated when evaluating query `r(6,X)` (b); and supports for the query evaluation (c).

The key to compactly storing the entire support graph is to make use of explicit sharing inherent in the supports. Consider the two answers to call `r(3,X)`, `r(3,2)` and `r(3,4)`, and their supports $s3$ and $s4$ respectively in Figure 24. Observe that the two supports share `c(3,6)`. Also notice that the literals which make the two supports

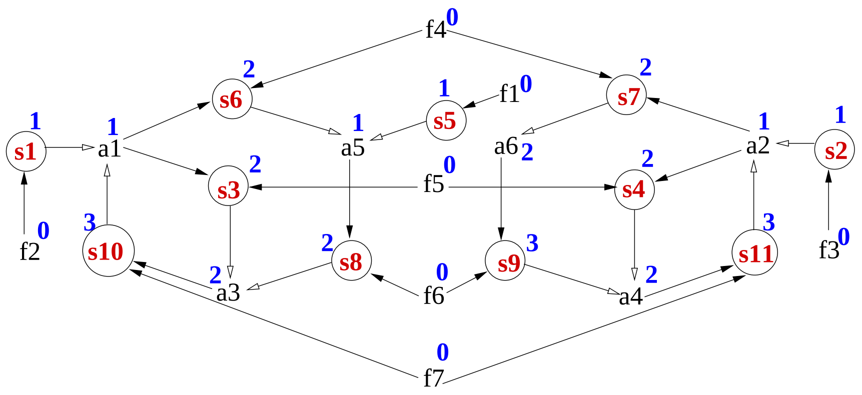


Figure 25: Support graph for answers to query `r(6,X)` over example program in Figure 24(a).

Call	**Symbolic Supports**
`r(6,X)`	⟨`r(6,X)`, {}, `b(6,X)`⟩, ⟨`r(6,X)`, {`c(6,3)`}, `r(3,X)`⟩
`r(3,X)`	⟨`r(3,X)`, {`c(3,6)`}, `r(6,X)`⟩, ⟨`r(3,X)`, {`c(3,1)`}, `r(1,X)`⟩
`r(1,X)`	⟨`r(1,X)`, {}, `b(1,X)`⟩, ⟨`r(1,X)`, {`c(1,6)`}, `r(6,X)`⟩

Figure 26: Symbolic supports for query evaluation over the example program in Figure 24(a).

different, i.e. `r(6,2)` and `r(6,4)`, are answers to the call `r(6,X)`. Thus two supports for answers to `r(3,X)` can be represented in intensional form as: `c(3,6), r(6,X)`. This intensional form is represented in a *symbolic support*, which consists of three parts, namely, the set of answers supported (e.g. `r(3,X)`, the common part of all the supports (e.g. `c(3,6)`, and the call whose answers distinguish the supports (e.g. `r(6,X)`). Now, when an answer to `r(6,X)`, say `r(6,2)` is deleted, we can compute, using the symbolic support, that `r(3,2)` may be affected. A symbolic support captures dependencies between certain calls while our earlier notion of supports captured dependencies between answers. By lifting this to the level of calls, a symbolic support compactly represents multiple supports.

The symbolic supports for the evaluation of query `r(6,X)` over the program in Figure 24(a) appears in Figure 26. Marking can be readily done using the symbolic supports. Given a marked answer (e.g. `r(6,2)`), we first compute the substitution for the variables in a support corresponding to it (e.g. `r(6,X), X = 2`), and use this substitution to find the supported answer (e.g. `r(3,2)`). When the intensional form does not contain any join operations, we can compute the answer dependencies from the symbolic support in time proportional to the answer size.

We now formally define the notion symbolic supports, and describe the data structure to represent symbolic support graphs.

Definition 13 (Symbolic Support) *Let P be a definite logic program with a set of facts F, and let C and A be a set of call and answer tables respectively, obtained when evaluating a query ξ over P. The triple $S = \langle h, s, d \rangle$ is a symbolic support for a call $\gamma \in C$ if there is a clause in P of the form α :– $\beta_1, \beta_2, \ldots, \beta_{n-1}, \beta_n$ and a substitution θ such that*

1. *h, called the* head *of S, is such that $\alpha\theta = h$;*
2. *s, called the* static part *of S, is such that $s = \{b_1, b_2, \ldots b_{n-1}\}$, and $\forall i \in [1, n-1]$, $b_i = \beta_i\theta$ and $b_i \in A \cup F$;*
3. *d, called the* dynamic part *of S, is such that $\beta_n\theta = d$.*

Note that a symbolic support is shared between a non-empty set of answers of a call. All non-symbolic supports represented by a symbolic support S are called *embedded* supports of S, defined below.

Definition 14 (Embedded Supports) *Let P be a definite logic program with set of facts F, and let A be answers in the tables obtained when evaluating a query γ over P. A non-symbolic support s is embedded in a symbolic support $S = \langle h, s', d \rangle$ if there is a substitution σ such that $s.answer = h\sigma \in A$, $s = s' \cup \{d\sigma\}$, and $d\sigma \in A \cup F$.*

Given a symbolic support $S = \langle h, s, d \rangle$, the answer a' is said to be a *supported answer* for an answer/fact a w.r.t. S if there is a substitution σ such that $a = d\sigma$ and $a' = h\sigma$. In that case, we also say that a is a *supporting answer* w.r.t. S.

When the mark on an answer is propagated, we need to find an embedded support that contains this answer. For instance, consider our running example and its symbolic supports in Figures 24(a) and 26 respectively. If `r(6,2)` is marked, since it is an instance of `r(6,X)`, we need to mark supports embedded in $\langle$`r(1,X)`$, \{$`c(1,6)`$\},$ `r(6,X)`$\rangle$ and $\langle$`r(3,X)`$, \{$`c(3,6)`$\},$ `r(6,X)`$\rangle$. This lookup can be efficiently done if the dynamic part of a symbolic support is a tabled call. Moreover, we can maintain, for each tabled call, the set of symbolic supports that contain it. If the dynamic part of the symbolic support is not a tabled call, then we need to maintain additional indexing structures to find the embedded supports. *Hence we do not use a symbolic support when its dynamic part is not a tabled call, and use non-symbolic supports instead.*

Symbolic support graphs (SSG) are an extension of the support graphs that has calls, answers, symbolic as well as non-symbolic supports as vertices and the relationships between them as edges. The edges in an SSG are described below.

- *uses_of*, *part_of*, *support*, *answer*: as in Definition 6.
- *set_uses_of*: If a fact or an answer a is in the static part of a symbolic support SS then there is a *set_uses_of* edge from a to SS.
- *set_uses_of_call* and *dynamic_call*: If a call C is the dynamic part of a symbolic support SS then there is a *set_uses_of_call* edge from C to SS. There is also a *dynamic_call* edge from SS to C.
- *supported_call* and *symsupport*: If a symbolic support SS supports a nonempty set of answers of a call C then there is a *supported_call* edge from SS to C, and a *symsupport* edge from C to SS.
- *answers* and *subgoal*: If A is the set of answers for call C, then there is a *answers* edge from C to elements of A and a *subgoal* edge from each element of A to C.

The SSG corresponding to the supports in Figure 25 is shown in Figure 27. Note that any tabling engine will give unique identities to each tabled call (e.g. the *subgoal frame* in the SLG-WAM [CW96]) and tabled answers. We use these identifiers in our implementation of the SSG to denote calls and answers (we use terms to represent calls in examples, for clarity). The information about the variables in the head and the dynamic part, needed to compute the embedded supports, is also kept in a symbolic support. This implementation detail is not shown in the examples.

The *set_uses_of*, *set_uses_of_call*, and *supported_call* edges in an SSG are required for propagation of marks and rederivation. They are analogues of *uses_of* and *answer* in a support graph. The *symsupport* edges are used to adjust the derivation length of an answer after rederivation. Finally, *dynamic_call* is used to compute the embedded supports of a symbolic support.

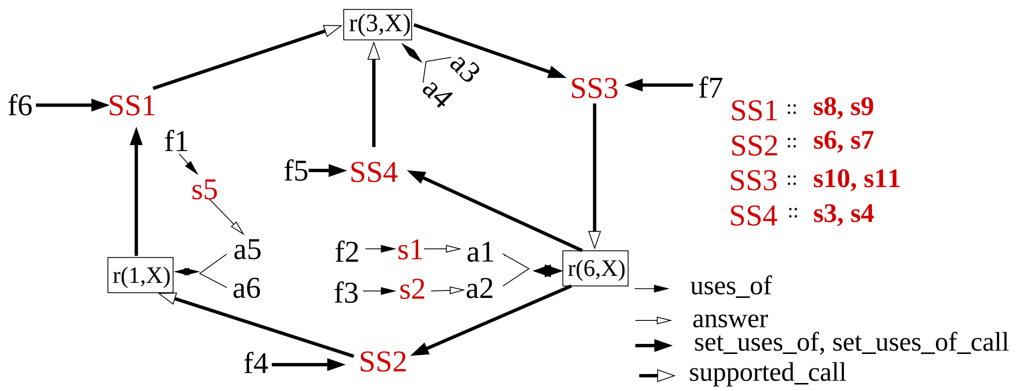


Figure 27: Symbolic support graph for answers to query `r(6,X)`.

In addition, for each answer we maintain the total number of unmarked supports in attribute *total_support_count* and the number of unmarked acyclic supports in attribute *acyclic_support_count*. These attributes counts the number of embedded supports represented by a symbolic support. In Figure 27, *total_support_count* of $a1$ is 2 and *acyclic_support_count* of $a1$ is 1.

5.2.1 Symbolic Support Based Incremental Algorithm

We now describe the incremental algorithm for maintaining the tables using symbolic supports. The algorithm extends the one in Figure 14 and handles graphs with a mixture of symbolic and non-symbolic supports. We have already seen in the previous section how to compute the embedded (non-symbolic) supports for each symbolic support. Note that information such as derivation length and marking are specific to the non-symbolic embedded supports; computing this information based on symbolic supports is the key issue in the algorithm. Note also that the static part of a symbolic support is common to all its embedded supports. Hence we associate the information due to the static part in the symbolic support. For each symbolic support node we maintain an attribute *static_maxdl* that stores the maximum of derivation lengths of the answers and facts in its static part. We use this information to compute the derivation length of each embedded support. Similarly, corresponding to the attribute *false_count* with each non-symbolic support which counts the number of marked answers/facts in the support, we associate with each symbolic support the attribute *static_false_count* which counts the number of marked answers and facts in

```
mark()
  mark_queue = empty
  ∀ deleted facts f
    mark_fact(f)
  while (mark_queue != empty)
    a = dequeue(mark_queue)
    mark_answer(a)

mark_fact(f)
  ∀ Support s ∈ f.uses_of
    mark_support(s)
  ∀ Sym.Support S ∈ f.set_uses_of
    mark_static(S)

mark_answer(a)
  a.marked = true
  ∀ s ∈ a.uses_of
    mark_support(s)
  ∀ S ∈ a.set_uses_of
    mark_static(S)
  subg = a.subgoal
  ∀ S ∈ subg.set_uses_of_call
    mark_dynamic(S, a)

mark_support(s)
  s.false_count++
  if (s.false_count == 1)
    propagate_mark(s.answer, s.dl)

mark_dynamic(S, sourceans)
  (* Propagate via dynamic part *)
  if (S.static_false_count == 0)
    targetans = supported answer of sourceans
                          w.r.t. S
    support_dl =
          max(S.static_maxdl, sourceans.dl) + 1
    propagate_mark(targetans, support_dl)

mark_static(S)
  (* Propagate via static part *)
  S.static_false_count++
  if (S.static_false_count == 1)
    ∀ sourceans ∈ answers(S.dynamic_call)
      if (! sourceans.marked)
        targetans = supported answer of sourceans
                              w.r.t. S
        support_dl =
              max(S.static_maxdl, sourceans.dl)+1
        propagate_mark(targetans, support_dl)

propagate_mark(ans, support_dl)
  ans.total_support_count−−
  if (ans.dl ≥ support_dl)
    ans.acyclic_support_count−−
    if (ans.acyclic_support_count == 0)
      enqueue(mark_queue, ans)
      marked_set = marked_set ∪ { ans }
```

Figure 28: Algorithm for Marking

its static part. The algorithm has two phases analogous to the two phases of DRed and other incremental recursive-view maintenance algorithms.

Marking Phase. The algorithm for the marking phase is shown in the Figure 28. The *false_count* attributes of symbolic and non-symbolic supports are initialized to zero before the marking phase. An answer is marked by setting its *marked* flag to true; this attribute is initialized to false. The answers to be marked are placed in a queue, and the marking phase ends when the queue is empty. The marked answers are placed in a set *marked_set* for processing in the rederivation phase.

The functions *mark_answer* and *mark_fact* propagate the effect of marking an answer/fact to the supports containing it. The function *mark_fact* propagates the effect of deleting a fact to the supports containing it. In addition *mark_answer* places a mark on the answer. Function *mark_support* marks a support and propagates this mark to the answer supported by it; functions *mark_static* and *mark_dynamic* mark a symbolic support and if needed propagate this mark to the answer(s) supported by it. Note that a (symbolic) support is marked if its (*static_*) *false_count* is nonzero.

We illustrate the working of the marking phase using the deletion of $f2$ and $f4$ from Figure 27 as an example. A call to mark_fact($f2$) will call mark_support($s1$), and subsequently propagate_mark($a1$, 1). This will decrement $a1$'s total and acyclic support counts (to 1 and 0, resp.), and place $a1$ in the queue. We will call mark_fact($f4$) next. Since $f4$ is in the static part of symbolic support *SS2*, we call mark_static(*SS2*). This sets *static_false_count* of *SS2* to 1, iterates over the answers of the dynamic part of *SS2*, i.e. `r(6,X)`. The supported answers of $a1$ and $a2$ w.r.t *SS2* are $a5$ and $a6$, resp., and $a6$ is added to the queue as $a5$ has an unmarked acyclic support $s5$. Note this is equivalent to propagation of marking through $s6$ and $s7$ in support graph based algorithm. Continuing further, we pick up $a1$ for processing from the queue. Since $a1$ appears in the dynamic parts of *SS2* and *SS4* *mark_dynamic* is called for both the symbolic supports. However, *mark_dynamic*(*SS2*,$a1$) has no effect as its *static_false_count* is already 1; *mark_dynamic*(*SS4*,$a1$) will call *propagate_mark*($a3$, 2) which reduces the total and acyclic support counts of $a3$ to 1 (due to acyclic embedded support $s8$ in *SS1*). Similarly, processing $a6$ from the queue does not mark $a4$ as it has an acyclic embedded support in *SS4*. Thus at the end of marking phase $a1$ and $a6$ are marked.

Rederivation Phase. Each marked answer that has some unmarked support at the end of the marking phase is known to have a proof not involving its previously known acyclic supports. In addition to resetting its mark, we need to compute its new derivation length (due to the new proofs). In our running example we compute the new derivation length of $a1$ by computing derivation length of its unmarked support ($s10$ in SG) embedded in *SS3*. This is done by finding the supporting answer for $a1$ w.r.t. *SS3*, i.e. answer $a3$ ($dl = 3$), and computing the dl of the embedded support. When some of the marked answers are rederived, we propagate rederivation using the function *rederive_answer*. Figure 29 gives the rederivation algorithm, which is very

```
rederive()
  ∀ ans ∈ marked_set
    if (ans.total_support_count > 0)
      ans.acyclic_support_count
        = ans.total_support_count
      recalculate_dl(ans)
      enqueue(rq, ans)
  ∀ Answer ans ∈ rq
    rederive_ans(ans)

recalculate_dl(targetans)
  spt_max=max{s.dl |
          s = targetans.support ∧
          s.false_count == 0}
  espt_max=
      max{max(S.static_maxdl,ans.dl)+1
    | S ∈ targetans.subgoal.symsupport
    ∧ S.static_false_count=0
    ∧ ans is a supporting answer
    of targetans w.r.t S
    ∧ !ans.marked}
  targetans.dl=max(spt_max,espt_max)

rederive_answer(ans)
  ans.marked = false
  ∀ s ∈ ans.uses_of
    rederive_support(s, ans.dl)
  ∀ S ∈ ans.set_uses_of
    rederive_static(S, ans.dl)
  subg=get_subgoal(ans)
  ∀ S ∈ subg.set_uses_of_call
    rederive_dynamic(S, ans)

rederive_support(s, dlen)
  s.dl = max(s.dl, dlen+1)
  s.false_count−−
  if (s.false_count == 0)
    propagate_rederive(s.answer_of, s.dl)

rederive_dynamic(S, sourceans)
  if (S.static_false_count == 0)
    targetans = supported answer of
                sourceans w.r.t. S
    dlen =
      max(S.static_maxdl, sourceans.dl)+1
    propagate_rederive(targetans, dlen)

rederive_static(S, dlen)
  S.static_maxdl = max(S.static_maxdl, dlen)
  S.static_false_count−−
  if (static_false_count(S) == 0)
    ∀ sourceans ∈ S.supported_call.answers
      if (!sourceans.marked)
        targetans = supported answer of
                    sourceans w.r.t. S
        dlen' =
        max(S.static_maxdl, sourceans.dl)+1
        propagate_rederive(targetans, dlen')

propagate_rederive(ans, dlen)
  ans.total_support_count++
  if (ans.acyclic_support_count==0)
    ans.acyclic_support_count = 1
    ans.dl = dlen
    enqueue(rq, ans)
  else
    if (ans.dl ≥ dlen)
      ans.acyclic_support_count++
```

Figure 29: Algorithm for Rederivation

similar to the marking algorithm.

Complexity Analysis: The complexity of the symbolic support graph based algorithm is given by the following expression:

$$O(|M_a|) + O(\sum_{a \in M_a} |a.uses\_of|) + O(\sum_{a \in M_a} |a.support|) + O(|M_s|)$$

where M_a denotes the set of marked answers and M_s is the set of marked static and embedded supports. Note the complexity is same as the acyclic support graph (ASG) based deletion algorithm presented in Chapter 4. This algorithm has some extra overhead of finding embedded supports. Finding each embedded support from a symbolic support takes a trie lookup time. The complexity of trie lookup for a term is proportional to the length of the term which is considered here as constant.

5.2.2 Space Complexity of Symbolic Support Graphs

In this section we compare the asymptotic size of SSGs with respect to table size and the size of non-symbolic support graphs for a number of useful tabled programs. For purposes of this comparison, we assume that all supports in the SSG are symbolic. The selected programs and the complexity measures are shown in Figure 30. The apparently simple transitive closure programs (`lreach/2` and `rreach/2`) lie at the heart of a remarkable number of applications of tabled logic programming. For instance, verification of safety properties of systems and implementation of inheritance in object-oriented logics reduce to reachability problem. Context-free language reachability, which is the basis for the verification of push-down systems, has rules that resemble the definition of the simpler same-generation (`sg/2`) predicate. A class of useful tabled logic programs not in the figure are those involving negation and aggregation (e.g. dynamic programming problems). In principle, symbolic supports can be used in these cases also, but other aspects of our implementation (e.g. handling of addition/updates) need extension. Hence we do not include this class in the comparison.

For the graph traversal examples, we assume that the `edge/2` relation defines a graph with v vertices and e edges. We consider a bound-free query to right-recursive transitive closure, say `rreach(`a`,X)`. Tabled evaluation makes $O(v)$ distinct tabled calls to answer this query. Each of these calls can have $O(v)$ answers, and hence the table size is $O(v^2)$. Each answer `rreach(`b`,`c`)` has supports of the form {`edge(`b`,`Y`)`, `rreach(`Y`,`b`)`} where Y ranges over neighbors of b. The number of supports for this answer is bounded by the out-degree of b. Since there are $O(v^2)$ answers, the total number of supports is $O(v * e)$. The symbolic supports associated with call `rreach(`a`,X)` are {`edge(`a`,X)`} and those of the form {`edge(`a`,`Y`)`, `rreach(`Y`,X)`}.

Example programs	Query Modes	Space Complexity		
		Table	SG	SSG
lreach(X,Y):- edge(X,Y).	bb, bf	$O(v)$	$O(e)$	$O(v)$
lreach(X,Y):- lreach(X,Z), edge(Z,Y).	fb, ff	$O(v^2)$	$O(v*e)$	$O(v^2)$
rreach(X,Y):- edge(X,Y).	bf, ff	$O(v^2)$	$O(v*e)$	$O(e)$
rreach(X,Y):- edge(X,Z), rreach(Z,Y).	bb, fb	$O(v)$	$O(e)$	$O(e)$
sg(X,X). *sg(X,Y):-* *edge(X,Y1),sg(Y1,Y2),edge(Y2,Y).*	all	$O(v^2)$	$O(e^2)$	$O(v*e)$
sg_opt(X,X). *sg_opt(X,Y) :- aux(X,Z),edge(Y,Z).* *aux(X,Y):- edge(X,Z),sg_opt(Z,Y).*	all	$O(v^2)$	$O(v*e)$	$O(v^2)$
Context-Free Language Reachability *N=|nonterms|,G=grammar size*	all	$O(N*v^2)$	$O(G*v^3)$	$O(G*v^2)$

Figure 30: Space complexity of symbolic support graphs

Thus there are two symbolic supports for each edge and hence the number of symbolic supports is $O(e)$. Note that SSG grows slower than the tables for this example.

The asymptotic space complexity for the other examples and queries in Figure 30 are computed along the same lines. The figure shows two versions of the same generation predicate: the naive *sg/2*, and an optimized version *sg_opt/2* obtained by supplementary tabling (i.e. tabling an intermediate join). The latter has better time complexity; observe from the figure that the size of SSG for this program is proportional to table size. For such programs, the space needed for SSG is less than three times the table space in the worst case. In practice the constant factor is close to 1.5.

5.2.3 Experimental Results

The aim of symbolic support graphs is to make incremental evaluation scale to large applications.

Space. We performed All Points-to Analysis (APA), which computes the points-to relation for all program variables (Chapter 3). Table 7 shows the number of supports, and space (in MB) taken by, support graphs, partial support graphs with maximum of 2 supports per answer, and symbolic support graphs for each benchmark. Observe from the table that the symbolic support graph takes the least space among the three. Note that the symbolic support graph may contain non-symbolic supports; while it is possible to make all supports symbolic, we find that it usually increases

Benchmark	Avg. size	Support Graph supports	Support Graph mem	Partial Su. Gr. supports	Partial Su. Gr. mem	Symbolic Support Gr. support	Symbolic Support Gr. symspt	Symbolic Support Gr. mem	mem $\frac{sym}{com}\%$
smail	24.5	3,159.4K	92.2	560K	22.8	42.2K	163.0K	15.5	16.8
parser	5.8	1,355.7K	44.2	518K	21.8	130.0K	159.2K	17.9	40.5
vpr	1.8	213.1K	9.7	172K	8.7	56.2K	51.9K	8.5	86.9
m88ksim	6.0	303.5K	11.8	206K	9.2	34.3K	47.9K	7.1	60.5
twmc	16.7	5,727.8K	158.5	396K	16.2	90.5K	105.0K	12.6	8.0
nethack	35.0	2,074.8K	59.4	269K	11.2	34.9K	60.4K	8.1	13.6
vortex	69.8	33,334.5K	912.0	1,714K	65.2	215.3K	361.4K	46.1	5.1

Table 7: Comparison of support graph sizes for pointer analysis

Benchmark	complete	partial	symbolic
smail	1.2	7.4	3.0
parser	-	7.7	0.9
vpr	-	1.3	-
m88ksim	-	1.1	-
twmc	-	8.4	0.5
nethack	-	5.6	-
vortex	-	15.0	0.2

Table 8: Incremental deletion time as a percentage of from-scratch time

space requirements by 20%. Finally, the table shows that the symbolic support graph can be considerably smaller than the (non-symbolic) support graph. Since symbolic supports keep dependencies between calls instead of answers, the reduction in space is proportional to the number of answers per call (the average points-to size).

Table 9 shows the sizes of non-symbolic (SG) and symbolic (SSG) support graphs for performing automata-based dead variable analysis of C programs using the push down model checker of [BKPR02]. The model checker has few answers per call, consequently we see a reduction in space due to SSGs, but not as much as in the

Benchmark	Table	SG	SSG
smail	0.8	0.9	0.8
allroots	0.5	0.5	0.4
assembler	47.7	67.6	48.6
compiler	51.1	154.0	53.7
compress	7.0	9.2	7.0
loader	5.9	6.9	6.0

Table 9: Support graph sizes (in MB) for push-down model checking

Programs	Graphs								
	chain			complete			tree		
	2000 nodes			100 nodes			10000 nodes		
	Table	SG	SSG	Table	SG	SSG	Table	SG	SSG
lreach	0.3	0.2	0.2	0.3	0.7	0.4	1.6	0.8	0.9
rreach	47.0	96.0	40.2	0.7	28.5	0.9	5.4	5.7	3.2
sg_opt	1.1	0.4	0.4	1.3	56.6	1.9	5.7	1.8	1.9

Table 10: Support graph sizes (in MB) for synthetic benchmarks from Figure 30

Bench-mark	from scratch	Incremental	
		SG	SSG
smail	0.520	.0011	.0013
allroots	0.0320	.0004	.0005
assembler	6.0280	.0253	.0333
compiler	27.1290	.0161	.0330
compress	0.3440	.0104	.0165
loader	0.4480	.0070	.0072

Table 11: Comparison of running times (in Seconds) for model checking

points-to analysis.

Recall from Section 5.2.2 the size of the symbolic support graph grows at or near the same rate as the table size for bound-free queries to left-recursive and right-recursive transitive closure and same generation programs (from Figure 30). Table 10 shows that not only the growth rates, but the total space requirements of symbolic supports are also close to those of the tables themselves.

Time. The effectiveness of the incremental techniques were evaluated by removing one (source-level) statement from the benchmark programs, and measuring the time and space taken to redo the analysis from scratch and to maintain the points-to relation incrementally. Deleting one source level assignment statement may delete multiple primitive assignment statements and hence multiple facts.

Table 8 shows the incremental evaluation times for pointer analysis as a percentage of time of from-scratch analysis using the non-symbolic support graph (complete), the partial support graph with at most 2 supports per answer (partial), and the symbolic support graph (symbolic). We found SSG-based algorithm is on average 5 times slower than the complete support graph based one, but is still two orders of magnitude faster than the from-scratch analysis for small changes. Incremental percentage which are less than .1% are shown using '-'. For model checking benchmarks SSG-based analysis is on average 1.5 times slower than complete support graph based analysis (Table 11). The incremental times for synthetic benchmarks are non-measurable and thus omitted here.

5.3 Related Work and Discussion

In this chapter we presented space-efficient data structures and incremental algorithms for maintaining tables in the presence of deletion of facts. The techniques can be readily extended to handle deletion of rules by keeping rule information in supports as done in the Chapter 4. The partial support graph grows linearly with the table size but it suffers from high rederivation time. In contrast, symbolic support graph in general is not bounded by table size but for a useful set of programs it showed similar space efficiency as partial support graph.

In-spite of the fact that the chief hindrance of using dependency graph based incremental computation is its space complexity, it is surprising to note that among all the works we are aware of only Rep's thesis [Rep84] addresses this problem. The first method discussed in his thesis reduces the storage by reducing the number of attribute values retained at any stage of attribute evaluation, although it compromise on optimality of the algorithm. The second method shares the space used for attribute values that are complex data structure. More specifically, it uses 2-3 tree for sharing ordered sets of values. Symbolic support graph is also based on sharing common part of support graph although the technique is dependent on dependency between calls and answers which is exploited in such a way that the dependencies between answers can be readily computed from the dependency between calls.

The ease of computation is ensured by keeping the symbolic supports in a "join-free" form, keeping only the last literal of a support as a call. This can be easily extended to keeping calls as the right-most literal in a clause that is followed by simple computations (such as comparison operations). This extension of the notion of symbolic supports will permit us to represent programs with aggregation operations using symbolic supports, thereby enabling incremental evaluation of dynamic programming problems.

In this chapter we considered only definite logic programs, where all predicates are either tabled or defined by facts. We can extend this to programs containing a mixture of tabled and non-tabled predicates along the same lines as in Chapter 4: accumulating support information from non-tabled predicates and storing them with answers.

Chapter 6

A Uniform Approach To Change Propagation

Most existing techniques for incremental evaluation consider addition and deletion of facts as primitive changes, and treat update as deletion of the old version followed by addition of the new version. They handle addition and deletion using independent algorithms, consequently performing many redundant computations when processing updates. Our first step (described in Chapter 4) towards designing algorithms for incremental update to tables also followed the above approach.

In this chapter, we deviate from these existing approaches to present a local algorithm for handling updates to facts or a set of addition and deletion of 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 addition 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.

```
:- 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)

Figure 31: Example tabled logic program (a), and its call and answer tables (b).

6.1 Motivation

Most of these works consider changes to the program 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 and processing addition and deletion using independent algorithms. Treating updates in this manner may lead to unnecessary evaluation steps, as illustrated by the following example.

Consider the evaluation of query `r(1,A)` over the program in Figure 31(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 for this query evaluation are given in Figure 31(b).

Now consider the effect of updating the 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, since `e(2,3)` is deleted, the answers `r(2,3)` and `r(2,4)` for call `r(2,X)` are deleted. Then answers `r(1,3)` and `r(1,4)` are deleted since they depend on answers `r(2,3)` and `r(2,4)`. In summary, nodes `3` and `4` are no longer reachable from `1` or `2`.

Having propagated the effect of deleting `e(2,3)`, we now process the addition of `e(2,4)`. This introduces two answers to call `r(2,X)`: `r(2,4)` (from the first clause of `r/2`), and `r(2,3)` (from the second clause of `r/2`, since `e(2,4)` and `r(4,3)` hold). The addition of these two answers to call `r(2,X)` results in the addition of `r(1,3)` and `r(1,4)` to call `r(1,X)`.

Several works, especially those on incremental evaluation of attribute grammars [DRT81, Rep82], treat update as in-place change, and propagate this change. This approach is very restrictive for logic programs. In order to use this approach the program has to be non-recursive and the change can only be on "non-key attributes" [SdS99]: i.e. the control behavior of the program cannot be changed.

However, these update propagation algorithms, whenever applicable (i.e. when the restrictive conditions are met), are optimal.

Updates may lead to deletions or additions in general. 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`. Hence any algorithm that handles only in-place updates is bound to be restrictive in the context of logic programs.

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. We present such a technique in this chapter.

Our Approach: We give an incremental evaluation algorithm that *interleaves* the processing of additions and deletions. The interleaving reduces the number of intermediate steps in the incremental evaluation. For instance, consider the example in Figure 31. Since `e(2,3)` is changed, inspect `r(2,X)`'s answers table and recalculate results. However, if we process all additions and deletions to `r(2,X)` before moving further, we will stop the propagation there since there is no net change to `r(2,X)`'s answers. Hence `r(1,3)` and `r(1,4)` will not be deleted in the first place.

Interleaving addition and deletion requires that additions be performed bottom-up (i.e. in a forward-chaining manner). Our earlier algorithms in the previous chapters for incrementally processing additions was based on top-down query evaluation. In this chapter we describe data structures and algorithms to process additions bottom-up while using the information about the original queries. Section 6.3 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 6.4 describes the assignment of priorities and a scheduler to perform the operations in order.

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 deletion and an addition, but select the order in which the deletion and addition will be processed based on the dependencies between the queries and computed answers. The order in which the operations are performed generalizes the call-graph based orders used in previous incremental algorithms [HPMS00, e.g.]. 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 dependencies. Hence our algorithm is optimal for the cases for which optimal update algorithms (e.g. incremental attribute evaluation) are known. Moreover the algorithm handles adds, deletes and updates efficiently, even when the changes affect the control behavior of the program. We describe the properties of the algorithm, including its behavior on parsing, attribute evaluation in Section 6.6. In a more general setting when the dependencies may be recursive, our algorithm interleaves addition and deletion operations even with a strongly connected component (SCC) in the call graph. It can be shown that our schedule of operations is uniformly better than adds-first or deletes-first schedules.

Although the technical development of this chapter deals only with definite logic programs, it is easy to extend the results to programs with (dynamically) stratified negation. The data structures used by this algorithm can be modified to reduce its space requirements. These issues along with results of our experiments on incremental data-flow analysis are discussed in Section 6.10.

6.2 Preliminaries

We assume familiarity with the standard logic programming definitions of terms, formulas, predicates, Horn clauses, rules, facts, and unification [Llo84].

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 [CW96], derivations are captured as a proof forest, called *SLG forest* [CSW95]. Since we are treating SLG evaluation for definite logic program we defined here SLG forest with respect to definite logic programs as follows. This definitions follows from [Swi99].

Definition 15 (SLG Forest) *An SLG forest is a set of SLG trees. Nodes of an SLG tree either have the form*

$$Answer\_Template \ :- \ Goal\_List.$$

where the answer template, Answer_template, is an atom and the Goal_list is a sequence of literals.

The SLG forest constructed when evaluating goal `r(1,X)` over the program in Figure 32(a) is given in Figure 32(b). 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 32. 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 answers.

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),e(Z,X)` (node c_2) and the answer `r(1,2)` are premises to `e(2,X)` (node c_3).

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 16 (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 in F iff $G_0 :- G_1, G_2, \ldots, G_n$ is a non-root node in the SLG tree of generator p in F. The set of all consumers in F is denoted by C_F.*

6.3 Data Structures

Note that the SLG forest itself does not explicitly represent the set of dependencies between the nodes of the forest. Our incremental algorithm maintains these dependencies using two auxiliary structures, namely, the evaluation graph and the call graph, defined below.

Definition 17 (Evaluation graph) *Let $\mathcal{P}$ be a definite logic program, F be the SLG forest constructed when evaluating query Γ over $\mathcal{P}$, and A_F, P_F and C_F be the answers, generators and consumers in F, resp. The evaluation graph corresponding to F_Γ is a directed graph (V, E) where $V = \mathcal{P} \cup A_F \cup P_F \cup C_F$ (i.e. the program clauses, answers, generators and consumers) and E is the smallest set such that*

1. *$\forall c = \langle \_, \_, g, \_ \rangle \in C_F, p \in P_F$, if p is a variant of g then (c, p) and (p, c) are in E. We say that $c \in p.consumers$ and $p = c.generator$.*

```
:- table r/2.
r(X,Y) :- e(X,Y).          % rule 1
r(X,Y) :- r(X,Z), e(Z,Y). % rule 2
r(X,Y) :- d(X,Z), r(Z,Y). % rule 3
```
(a)

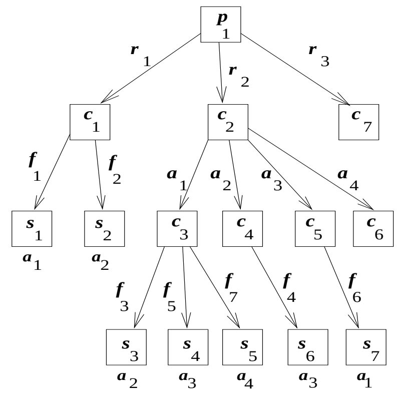
```
e(1,2). % f1
e(1,3). % f2
e(2,3). % f3
e(3,4). % f4
e(2,4). % f5
e(4,2). % f6
e(2,5). % f7
```
(b)

Call: `r(1,Y)`
Answers: `r(1,2)` $[a_1]$
`r(1,3)` $[a_2]$
`r(1,4)` $[a_3]$
`r(1,5)` $[a_4]$

(c)

$[p_1]$	`r(1,A) :- r(1,A).`	$[s_1]$	`r(1,2).`
$[c_1]$	`r(1,A) :- e(1,A).`	$[s_2]$	`r(1,3).`
$[c_2]$	`r(1,A) :- r(1,B), e(B,A).`	$[s_3]$	`r(1,3).`
$[c_3]$	`r(1,A) :- e(2,A).`	$[s_4]$	`r(1,4).`
$[c_4]$	`r(1,A) :- e(3,A).`	$[s_5]$	`r(1,5).`
$[c_5]$	`r(1,A) :- e(4,A).`	$[s_6]$	`r(1,4).`
$[c_6]$	`r(1,A) :- e(5,A).`	$[s_7]$	`r(1,2).`
$[c_7]$	`r(1,A) :- d(1,B), r(B,A).`		

(d)



(e)

Figure 32: Example program (a), facts (b), calls and answers (c), nodes in SLG forest (d), and SLG forest (e)

2. *$\forall c, c' \in C_F$, if c' is a premise of c then (c, c') and (c', c) are in E. We say that $c' \in c.next\_consumer$ and $c = c'.prev\_consumer$.*

3. *$\forall c \in C_F$ and $\forall a \in \mathcal{P} \cup A_F$, if a is a premise of c then (a, c) and (c, a) are in E. We say that, c is dependent on a, $c.dependent\_on = a$ and $c \in a.holds$.*

4. *$\forall c \in C_F$ such that $c = \langle \_, h, true, [] \rangle$ (i.e. leaves of successful derivations) and $a \in A_F$ such that h is a variant of a, (c, a) and (a, c) are in E. We say that c is a support of a, $a = c.answer$ and $c \in a.supports$.*

For program given in Figure 32(a) with facts in (b), the tabled call and answers are given in Figure 32(c); and the SLG forest in Figures 32(d) and (e).

The edges in the evaluation graph are used as follows. An edge from a generator to its consumer (type 1 in the above definition) is used to notify the consumer about insertion of an answer in the generator's answer table. The consumer then follows the reverse edge to access the inserted answer(s). An edge from premises to their conclusion (types 2 and 3) are used to ensure that when a premise is deleted or otherwise invalidated all its consequences are re-examined. Edges connecting supports and their answers are used to find whether an answer is supported after a deletion, and to process unsupported answers. In addition to the evaluation graph, we assume that the set of all consumers C_F is indexed on its third (goal) component. This ensures that when a rule or fact is inserted, we can locate the affected consumers quickly.

The properly indexed set of consumers and the evaluation graph are sufficient for incrementally adding nodes to the SLG forest (and hence the call and answer tables) when new rules/facts are inserted. 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: `r(1,6)`. This is a new answer to generator p_1, which gets forwarded to its consumer c_2 [using a type-1 edge in the evaluation graph]. 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 needed to change the original forest to include the new fact and its effects.

The evaluation graph can also be used to modify the SLG forest when a fact/rule is deleted. For instance, consider the deletion of fact `e(3,4)` (f_4) from the program in Figure 32. 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 consumer *independent of* s_6*)*.

In Chapter 4 we used *support graph* to propagate marks in supports and answers. We preserve all the information of support graph in the evidence graph which enables us to process deletion through evidence graph. The type-4 edges in definition of evidence graph preserves the relation between an answer and its supports. The constituent answers of a support can be found by following the leaf-to-root path in an SLG tree using type-2 edges and collecting the answers premises (using type-3 edges) for all the consumers (including the support itself) in the path. This corresponds to the relation *part-of* in Definition 6. The *uses-of* (Definition 6) of an answer can be found following the type-3 edges to find a consumer and following *next_consumer* edges to find the supports.

In Chapter 4 we presented heuristics to limit the propagation of deletion mark using the notion of *primary supports* and subsequently *derivation length* based *acyclic* *supports*. In this chapter we refine and generalize this measure further. 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 (total order) so that the independence of two generators can be determined based on their SCC numbers. This permits us to quickly identify independent consumers and answers irrespective of their derivation lengths. Consider again the example given in Figure 31. 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)` is 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 call `r(1,A)`. Call-graph SCCs have been used for localizing the change propagation in existing works and we build our algorithm on top of it.

Although processing changes within an SCC before propagating its net changes to topologically higher SCC seems to be fruitful in some cases, it is clearly ineffective for change propagation *within* an SCC. To order events 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 in Figure 33.

Entity (X)	SCC number ($X.scc$)
Answer	$p.scc$ where a is an answer of p
Consumer(c)	$p.scc$ where $c = \langle p, h, g, G \rangle$

Entity (X)	Ordinal ($X.ord$)
Answer (a)	$\{s.ord \mid s$ is the primary support of $a\}+1$
Consumer (c)	$max\{c'.ord, Ord, 0\}$, where $c' = c.prev\_consumer, a = c.premise$ and $Ord = a.ord$ if $a.scc = c.scc$ and 0 otherwise

Figure 33: Ordinal definitions.

Corresponding to this definition of ordinals of answers and consumers we say 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.

6.4 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 adds, but the algorithm interleaves the deletion phase of marking answers and processing of addition such that (a) it reduces 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 given in Figure 32. Two additional examples in the next section cover subtle aspects not covered by the main example.

Consider the program in Figure 32 after updating fact `e(1,2)` ($f1$) to `e(1,5)`. This is treated as deleting $f1$ and adding a new fact $f8$ =`e(1,5)`. If we process deletion before addition, 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 added fact. On the other hand, if we process addition before deletion we will (i) generate a new acyclic support for $a4$ (derivation based on the added 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 (iii) rederive $a1$. Thus processing addition first is better than processing deletion first for this example.

Now consider a different change to the program in Figure 32: deleting `e(1,2)` ($f1$) and adding `e(2,6)` ($f9$). Processing addition 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 all three answers since `r(1,2)` has an alternative derivation. Processing deletion before addition will mark answers $a1$ and $a4$, and rederive both. Addition of `e(2,6)` will generate a new answer `r(1,6)`. For this example, processing deletions first performs fewer operations, and is better. These two examples show that neither add-first nor delete-first strategies is uniformly better than the other.

An interesting question remains: *is there a change propagation strategy which is uniformly 'no worse' than either delete-first or add-first strategies?* The previous two examples indicate that interleaving the deletion and addition may be better. In fact, if we delete $f1$ and add $f8$ and $f9$, it is easy to see that the best change propagation strategy will be to process the addition of $f8$ first, deletion of $f1$ next and addition 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 addition of facts 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 is first scheduled before processing events of topologically higher SCC. This makes sure 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 than *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 a more detailed description of our algorithm we provide here the key intuition behind interleaving of *mark* and *consume_answer* events. Note that *mark* operation overapproximates the actual answers that needs to be deleted. Marking of an answer can be avoided if we can generate an acyclic support for the answer using added and existing answers, provided the used answers are never going

```
process_event(e=consume_answer(a,c))
    c=⟨p,h,g,G⟩
    θ=mgu(a,g)
    g'=head(Gθ) // g'=true if G is empty
    G'=tail(Gθ) // G'=null if G is empty
    a'= hθ //answer generated
    c'=⟨p,a',g',G'⟩ // new consumer
    insert c' in c.next_consumer, c'.prev_consumer=c
    insert c' in a.holds, c.dependent_on=a
    if(is_empty(G)) // last subgoal of a clause
        is_newanswer=check_insert_answer(p,a')
// checks whether a' is in p.answer_table, if not inserts a'
        if(is_newanswer)
            a'.ord=c'.ord+1;
            ∀c"∈p.consumer
                if(!marked(c"))
                    create_event(consume_answer(a',c"))
                else
                    delay_event(consume_answer(a',c"))
        else
            if(∀c"∈(a'.supports−{c'}) s.t. (c".ord<a'.ord→marked(c")))
                if ((c'.ord<a'.ord) && (e.ord<a'.ord))
                    delete_from_ready_queue(mark(a'))
                else
                    create_event(may_rederive(a'))
    else
        resolve_goal(c')
```

Figure 34: Algorithm for Processing consume_answer event.

to be marked. Hence we choose the ordinals of events and entities such that the added answers and the supports generated by newly added and existing answers are not marked in the same incremental phase.

Addition. Generation of a new answer or addition 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 34). If the consumer

```
resolve_goal(c=$\langle p,h,g,G\rangle$)
1    p'=call_check_insert(g)
2    if(is_newgenerator(p')) //g is a new call
3        for each rule $\alpha$:-$\beta_1, \beta_2 \ldots, \beta_n$ s.t. ($\theta$=mgu(g,$\alpha$)!=$\phi$)
4            c'=$\langle$p',g$\theta$,$\beta_1\theta$,[$\beta_2\theta, \ldots, \beta_n\theta$]$\rangle$ // new consumer
5            resolve_goal(c')
6    else
7        $\forall$a$\in$p'.answer_table
8            if(!a.marked)
9                create_event(consume_answer(a,c))
10           else
11               delay_event(consume_answer(a,c))
12   insert c in p'.consumer, p'=c.generator
13   calculate_call_graph_incrementally
```

Figure 35: Algorithm for resolving goal.

corresponds to the last subgoal of a rule, the consumption of the answer can generate a new answer (lines 1–17), or a new support for an existing answer (lines 1–11, 19–23). Otherwise (i.e. the consumer has a non-empty continuation) it generates a new consumer corresponding to next literal of the clause (lines 1–8, 25).

The pseudo-code in Figure 35 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 through all answers in answer table of g, creating *consume_answer* events for each of them (lines 7–11).

For example, addition 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 1–11, 18 of Figure 34. On the other hand, addition 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 1–9, 25). 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.

Most of the steps of *consume_answer* are common to traditional SLG resolution.

The interesting aspects are the interaction between the effects of addition and (possibly scheduled) deletion. For instance, when a new acyclic support c' is generated for an answer a' (line 20, first condition) whose all other acyclic supports are already marked (line 19) and *mark(a')* event has been scheduled (line 20, second condition) we remove the $mark(a')$ from the ready queue since a' cannot be deleted due to c'.

Mark. The mark event for an answer marks a given answer and propagates the effect of marking (Figure 36(a)). 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). The following definitions are used in the marking algorithm:

Definition 18 (Affected set of an answer) *A consumer c is said to be* affected *by an answer a (denoted by $c \in a.affected$) if the answer a is a premise of c (i.e. $c \in a.holds$), or a premise of c is affected by a.*

Definition 19 (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. Consider deleting $f1$ =`e(1,2)`, and adding $f9$ =`e(2,6)` and $f10$ =`d(1,1)` to the example in Figure 32. 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 process the mark event $e3$. This is because $c3$ is dependent on answer $a1$ which may be marked when $e3$ is

```
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)
        // c is 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))
```

(a)

```
event_loop()
1  while((SC=next_scc(CallSCC_Q))!=NULL)
2     while(!empty(READY_Q,SC))
3       process_event(get_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 */
```

(b)

Figure 36: Mark event (a), Main event loop (b)

processed. We ensure this by making a consumer's ordinal no less than that of any answer that affects it (Figure 33). To process a *consume_answer*(a, c) only after processing all *mark*(a') events which can affect the consumer c, 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 then the above condition is satisfied as we complete components according to their increasing SCC numbering. Secondly, we ensure that a new answer generated (lines 9–11, Figure 34 can never be marked in the same incremental phase. The only

Events(e)	Ordinal (e.ord)	scc (e.scc)
consume_answer(a,c)	$max\{c.ord, a.ord\}$ $if(same\_scc(a,c) \wedge existed\_answer(a))$; $c.ord$ $ow.$	c.scc
mark(a)	a.ord	a.scc
may_rederive(a)	a.ord	a.scc
rederive(a)	a.ord	a.scc

Figure 37: Priorities of Events

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 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 is no less than $a.ord$.

The assignment of ordinals to the different events is summarized in Figure 37. This assignment of ordinals is critical for the following properties of the algorithm which are proved in Section 6.8.

Property 3 *If $consume\_answer(a, c)$ is a scheduled event, then a is never marked in the same incremental phase.*

Property 4 *If a' is a new answer and s is a support for an unmarked answer generated while processing $consume\_answer(a, c)$ (lines 12 and 19, Figure 34 then a' and s are never marked in the same incremental phase.*

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 $e5 = consume\_answer(a1, c8)$ ($e5.ord = a1.ord = 1$), $e6 = consume\_answer(a2, c8)$ ($e6.ord = a2.ord = 1$), $e7 = consume\_answer(a3, c8)$ ($e7.ord = a3.ord = 2$), and $e8 = consume\_answer(a4, c8)$ ($e8.ord = a4.ord = 2$) ($c8 = \langle p1, r(1, Y), r(1, Y), []\rangle$). 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 followed by event $e9$ 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 $e8$.

```
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))
                                        (a)

process_event(rederive(a))
 1  a.marked=false
 2  ∀c, s.t. !marked(c) move_to_ready(consume_answer(a,c))
 3  ∀c∈a.affected, same_scc(a,c) ∧ !marked(c)
 4      ∀a' s.t. !a'.marked, move_to_ready(consume_answer(a',c))
 5  ∀c ∈ a.affected, same_scc(a,c)
 6      c.ord = max(c.ord, a.ord) // update ordinal of consumers
 8  ∀c ∈ a.affected, same_scc(a,c)
 9      if(is_leaf(c) ∧ !marked(c) ∧ marked(c.answer))
10          ans=c.anwswer
11          ans.ord=max{c".ord | c"∈ ans.supports,!marked(c")}+1
12        create_event(rederive(ans))
                                        (b)
```

Figure 38: Algorithms for Processing May Rederive (a) and Rederive (b) events.

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 38(a)). 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 38). Rederivation of answer a also updates the ordinal of the support that contains a and in 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. Subsequently processing of remaining events does not reveal any other interesting property of the algorithm and is not discussed here.

Figure 36(b) shows the pseudo code for scheduling 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 is changed without change in any component (Example 3, Section 6.5); and (ii) components are merged into larger components (Example 4, Section 6.5). We employ incremental SCC maintenance algorithm of [PK03, AHR$^+$90] 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 a topologically lower component than that of $a1$. Thus when multiple SCCs are merged the ordinals of answers and supports need to be redistributed within the merged component (see examples in next section).

6.5 Handling Dynamic Call Graph

We present two more examples to illustrate the handling of dynamic call graph SCC by the algorithm. In Figure 39 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

```
:- table r/2.
r(X,Y) :- e(X,Y).          % rule 1
r(X,Y) :- e(X,Z), r(Z,Y). % rule 2
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]
```

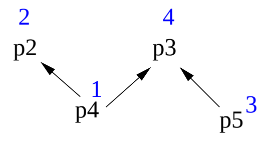
(b)

```
[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)]>
[s1] <p2,r(1,3),true,[]>
[s2] <p3,r(2,3),true,[]>
[s3] <p4,r(2,4),true,[]>
```

(c)

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

(d)



(e)

Figure 39: Example program (a), calls and answers (b), consumers (c), new facts (d), and call graph (e).

$a3$ already existed in the answer tabled of the call `r(2,Y)` (producer p3), two new events $e7 = \mathit{consume\_answer}(a2, c12)$ and $e8 = \mathit{consume\_answer}(a3, c12)$ are created. The changed *ready_queue* is shown below.

Event	Ordinal	scc
$[e4]$ $\mathit{consume\_answer}(f5, c10)$	0	0.5
$[e5]$ $\mathit{consume\_answer}(f5, c11)$	0	0.5
$[e6]$ $\mathit{mark}(a2)$	1	1.5
$[e7]$ $\mathit{consume\_answer}(a2, c12)$	0	2
$[e8]$ $\mathit{consume\_answer}(a3, c12)$	0	2
$[e3]$ $\mathit{mark}(a1)$	1	2

Step 3: Processing event $e4$ generates new answer $a5$=`r(4,3)` (a5.ord=1) with a new support $s5$ ($\langle e(4,3)\rangle$) (s5.ord=0). This generates a new event $e9$=*consume_answer(*$a5$,$c7$*)* (ordinal=0, scc=1.5) which gets higher priority than $e7$ as well as $e6$.

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 0. As $a2.ord = 1$, $s6$ is an acyclic support of $a2$. As $a2$ can be derived using $s6$ we delete $e6 = \mathit{mark}(a2)$ event from the ready_queue (Lines 19-23 Figure 34.

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 $e3 = \mathit{mark}(a1)$ 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. □

We present another example to illustrate the handling of dynamic call scc (where SCCs are merged) by the algorithm. In Figure 40 we present the example of right

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

(a)

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

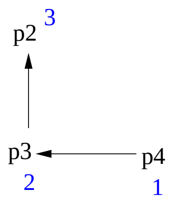
(b)

```
e(2,1) [f3]
```

(c)

```
[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(2,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] <p4,r(3,Y),e(3,Y),[]>
[c8] <p4,r(3,Y),e(3,Z),[r(Z,Y)]>
```

(d)



(e)

Figure 40: Example program (a), calls and answers (b), new facts (c), consumers (d), and call graph (e).

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 update due to the merging of SCCs. For example, as $s2$ and $a3$ now belong to the same SCC the ordinal of $s2$ should be 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 process increases the ordinals of a support s which belongs topologically higher SCC than an answer a such that $s.ord < a.ord$ and $s \in a.affected$. 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 $s.ord + 1$, otherwise the ordinal of $s.answer$ is not changed which means s is changed to a 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. □

6.6 On the Optimality of the Algorithm

In this section we describe the optimality of the local algorithm for certain classes of problems. Although the interleaved scheduling of *consume_answer* and *mark* operations is not optimal in general, we show that its performance is always no worse than, but sometimes better than fixed schedules.

Attribute Grammars: Consider the attribute grammar for evaluating simple expressions given below:

```
:- table expr/3.
expr(S1,S2,X):- expr(S1,S3,X1), c(S3,S4,'+'), c(S4,S2,X2), X is X1+X2.
expr(S1,S2,X):- c(S1,S2,X).
```

We represent the string to be parsed and evaluated as a set of `c/3` facts. A fact of the form `c(`i`, `$i+1$`, S)` represents the symbol at i-th position in the string. Let the input string be of length $2n-1$ (indexed 1 to 2n) containing n numbers and separated by $n-1$ '+' symbols, and let the query be `expr(1,S,X)`. The answers to the query are of the form `expr(1,`$2*i$`,`k`)` for $i \in [1,n]$ and k is the sum of the first i numbers in the string.

If we update the string by exchanging the numbers at two positions say $2i-1$ and $2j-1$ for some $1 \leq i < j \leq n$, our algorithm will update (i.e. delete and add) at most $j-i$ answers of the form `expr(1,2*x,_)` for $x \in [i, j-1]$. It is easy to see that this is optimal. This particular example is an instance of non-circular attribute grammar evaluation where the dependency graph is acyclic and static. As noted by Reps in [Rep82], 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 changed attributes. Note that, although our left recursive encoding of expression grammar only produces one call SCC, we obtain the topological ordering using event ordinals. Thus topological scheduling of *consume_answer* and *mark* events allows the desired optimal behavior.

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 [ABH02]. 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), [ABH02] employ an optimal dynamic topological order maintenance algorithm using Dietz and Sleator data structures [DS87]. When the call graph considered in our algorithm is acyclic, our incremental topological SCC maintenance algorithm converges to dynamic topological graph maintenance [ABH02]. Thus we obtain the optimal change propagation algorithm for functional programs.

The complexity of the local algorithm for incremental interpretation of a pure functional program is no worse than the complexity of the adaptive algorithm for pure functional program evaluation given in [ABH02]. The following theorem expresses the complexity of our local algorithm for incremental evaluation of pure functional program. We assume that we have logarithmic time priority queue operations, and insert, delete, find_next, and compare operations in order maintenance algorithm can be performed in constant time (as in [ABH02]).

Theorem 5 *The complexity of local algorithm for interpreter based incremental evaluation of pure functional program is*

$$O(\sum_{e \in E} |e| + Elog(q) + C)$$

where E is the number of consume_answer events processed due to update, $|e|$ is the complexity of each such consume_answer events and C denotes the total number of changes in SCC numbering because of the update.

Proof: Each update can generate a *mark* and a *consume_answer* event. For functional case, each call component has only single call and thus each *mark* event has constant time complexity. The rest of the proof is simple, the time taken by local algorithm can be divided into (i) time for processing *consume_answer* and *mark* events ($O(\sum_{e \in E} |e|)$); (ii) re-assignment of SCC numbering due to change in the topological ordering; (iii) time for handling events in the priority queue.

The complexity of adaptive functional programming as given in [ABH02] is

$$O(\textstyle\sum_{e \in I_u}(|e| + ||e||) + Ilog(q))$$

where I is the set of invalidated edges (each edge corresponds to the one *consume_answer* and one *mark* event) and I_u is the set of updates edges. The algorithm presented in [ABH02] does not take advantage of memoization, thus it re-evaluates certain calls whereas our local algorithm fetches the answers from already re-evaluated calls. Also $||e||$ denotes the number of timestamps created for evaluation of $e \in I_u$. Thus $I_u = E$ and $O(\sum_{e \in I_u}(||e||) = C$. Thus complexity of our algorithm is no less than complexity of adaptive functional program evaluation algorithm in [ABH02].

Scheduling: Note that given the four events that arise in incremental evaluation, there are many possible schedules, yielding algorithms with varying performance. We first define a measure to compare the relative efficiency of two deterministic scheduling strategies:

Definition 20 *Let $m_{P,\gamma}(S)$ denote the number of answers marked when performing incremental evaluation of program P after changes γ using scheduling strategy S. We say that $S1$ is better than $S2$ (denoted by $S1 \geq S2$) iff (i) for all programs P and for all changes γ $m_{P,\gamma}(S1) \leq m_{P,\gamma}(S2)$; and (ii) there is some program P_0 and a set of changes γ_0 $m_{P_0,\gamma_0}(S1) < m_{P_0,\gamma_0}(S2)$ is true.*

Note that we only compare scheduling strategies based on the number of marked answers. This is due to the fact that the number of net added/deleted answers are the same for all scheduling strategies, and the number of rederived answers is proportional to the number of marked answers.

The following properties of our scheduling strategy follow from Property 12: that we do not schedule a *consume_answer* event if it is dependent on a consumer or an answer which can be potentially marked.

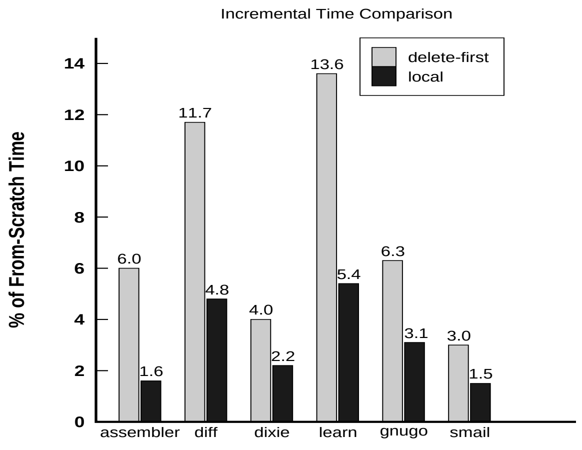
Proposition 6 *Let S_0 be the scheduling strategy used in the local algorithm. Let S_{di} be the strategy that schedules all mark events before consume_answer events; and S_{id} be the strategy that schedules all consume_answer events before mark events. Then*

- $S_0 \geq S_{di}$
- $S_0 \geq S_{id}$

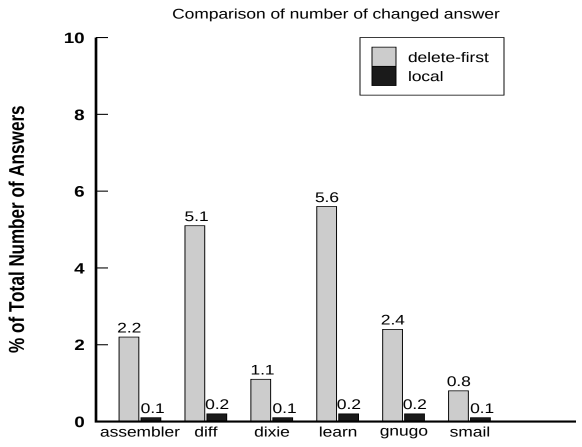
6.7 Experimental Results

We now present results of experiments aimed at measuring the effectiveness of the local algorithm as well as its overheads. The local algorithm was implemented by extending the XSB logic programming system [XSB] (ver. 2.7.1). Our implementation, experimental setup, benchmark characteristics, and detailed experimental results on parsing and pointer analysis are available in [SR06].

We evaluated the effectiveness of the local algorithm by performing reaching definition analysis [ASU86] of C programs which can be easily encoded as a logic program. The incremental change used in these experiments is the deletion of a statement from the C program source. In the data-flow graph, this translates to deletion of all incoming and outgoing edges from the deleted statement, and addition of flow edges from the statement's predecessors to its successors. The local algorithm is expected to perform well in this case, by confining the changes to reaching definitions to the affected region of the graph. We compare the performance of the local algorithm with that of from-scratch evaluation and that of deletes-first strategy. The deletes-first strategy first performs all marking and rederivations due to deletion before processing insertions. Marking and rederivation phases are performed in each strongly connected component of the call-graph (in topological order) before their effect is propagated to other components in the call graph. Thus comparison of the local and the deletes-first



(a)



(b)

Figure 41: Reaching Definition Analysis; Time comparison (a); Change comparison (b).

strategy demonstrates the effectiveness of interleaving the processing of deletions and insertions.

For each benchmark we deleted 250 randomly chosen assignment statements from the source. The ratios of the average time taken and the average number of answers processed by the local algorithm and the deletes-first strategy, compared to the from-scratch strategy is shown in Figure 41. 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 flow-insensitive pointer analysis, we notice a maximum run-time overhead of 70% for the local algorithm compared to deletes-first strategy. The local algorithm is optimal for incremental evaluation of parsing problems. In the parsing problem, the time taken for incremental evaluation depends on the position where change occurs in the input string, and some changes may require the entire parse tree to be regenerated. In such cases, when from-scratch evaluation is optimal, we observe that the local algorithm incurs a time overhead of 5% compared to from-scratch evaluation.

6.8 Properties of Local Algorithm

In this section we prove several properties about our algorithm and subsequently prove its correctness. Unless stated all statements should be considered for a single incremental phase.

Since the call graph changes with time we associate the subscript time to denote the call graph and its components at a particular instant. The parameter time advances by 1 unit after processing each event and before processing the next event. Note that while processing an event the call graph changes. Thus at a particular time t there can be two different call graphs — the call graph at the beginning of processing an event and call graph at the end of processing the event. For purpose of describing the properties we use $S_t(v)$ to denote a strongly connected call-graph component (hereafter referred to as *component*) containing a vertex v at time instant t such that $S_t(v)$ is computed at the beginning of processing an event. As such, any expression, which is dependent on time, is evaluated at the beginning of processing an event. We drop the subscript t when it is clear from the context. We use arithmetic comparison operators with subscript *top* to compare topological order (total order) of two call graph components at same time instant.

6.8.1 Foundational Properties of Local Algorithm.

Definition 21 *A call graph component $S_t(v)$ is* complete *iff there exists no event E in ready_queue and delay_queue at time t such that $E.scc = S_t(v)$.*

Lemma 7 *If an event E' is generated while processing an event E then $E'.scc \geq_{top} E.scc$.*

Proof: Follows from the definition of the events.

Lemma 8 *If a component $S_t(v)$ is complete then any component $S_t(v')$, such that $S_t(v') <_{top} S_t(v)$, is also complete at time t.*

Proof: Since the component $S_t(v)$ is complete at time t, there exists no event in the event queues for the component. As we process events in increasing order of their *scc* thus there exist no events for $S_t(v')$ in the event queues. Thus by definition $S_t(v')$ is complete.

Lemma 9 *If E be an event to be considered for processing at time instant t and $S_t(v)$ be a component such that $S_t(v) <_{top} E.scc$ then $S_t(v)$ is complete.*

Proof: When the event E is considered for processing there exists no event E' in the event queues such that $E'.scc <_{top} E.scc$ because we process events in increasing topological order of call graph components. Thus at time t there exists no event for $S_t(v)$. Thus by definition $S_t(v)$ is complete.

Property 10 *Let v be a vertex in the call graph and $t1$ and $t2$ be two time instances such that $t2 > t1$. Then $S_{t2}(v) = S_{t1}(v)$ if $S_{t1}(v)$ is complete.*

Proof: Note that an incoming edge is added to a component when a consumer in the component consumes an answer from a producer which belongs to some other component. Following Lemma 8, at time $t1$ there exists no event E in the event queues such that $E.scc \leq_{top} S_{t1}(v)$. Thus according to Lemma 7 at time $t2$ there cannot be an event related to $S_{t1}(v)$. Thus no incoming edges can be added to the component. Thus a completed component does not change after completion.

Property 11 *Let $E = mark(A)$ be an event and A' be an unmarked answer such that $A'.scc = E.scc$ and $A'.ord < A.ord$. Then after E is processed A' can never be marked in the same incremental phase.*

The proof is based on the following facts: (i) the only event which marks an answer is *mark*; (ii) *mark* events are generated only from other *mark* events; (iii) when a $mark(A1)$ event is processed it generates $mark(A2)$ event such that $A2.ord > A1.ord$; (iv) *mark* and *consume_answer* events do not decrease ordinal numbers of answers; (v) the ordinal number of a *mark* event is same as the ordinal number of the answer associated with the *mark* event; and (vi) *mark* events are never pending.

Note that derivation of an answer A' in a call graph component $S(v)$ ($A'.scc = S(v)$) is dependent on the answers in $S(v)$ and/or answers in a component $S(v')$ such that $S(v') <_{top} S(v)$. Using Lemma 9 component $S(v')$ is complete. Thus using the above facts and the fact that we process all *mark* events in increasing order of ordinals for $S(v)$ we derive that *mark* event for any unmarked answer with ordinal $< A.ord$ is never generated.

Property 12 *If $E = consume\_answer(A, C)$ is scheduled when $existed\_answer(A)$ is true which means E was generated from function resolve_goal then A is never marked in the same incremental phase.*

Proof: Consider the following two cases

I. $A.scc = C.scc$

   From definition of ordinal of a *consume_answer* event we derive that $A.ord \leq E.ord$. Also from processing of *resolve_goal* function (line 8) we note that A is unmarked at the time of generation of E. Since we process all the marks events before all the *consume_answer* events of the same ordinal, *mark* events with ordinal $\leq E$ has already been processed before processing the *consume_answer* event. Thus following Property 11 we know that all unmarked answers with ordinals $\leq E.ord$ can never be marked in the incremental phase. Thus A can never be marked in the incremental phase.

II. $A.scc < C.scc$

In this case, A belongs to a component which is completed (following Lemma 9). Thus there can never be any $mark(A)$ event in the event queue. Thus A can never be marked.

Property 13 *If A is a new answer generated while executing the $consume\_answer(A_n, C_n)$ event then A is never marked in the same incremental phase.*

Proof: We prove it by induction on ordinal of the answers generated by *consume_answer* event. Let i be the ordinal of a new answer A ($A.ord = i$) generated by the *consume_answer* event.

Induction hypothesis: The property holds for all the new answers with ordinal less than i.

Base Case: Trivial base case for facts.

Induction Step: A was generated by the function $consume\_answer(A_n, C_n)$ corresponding to the last subgoal in a clause with consumers $C_1, C_2, \ldots, C_n$ for its respective subgoal positions. The primary support of A is $\{A_1, A_2, \ldots, A_n\}$. Note that any answer in $\{A_1, A_2, \ldots, A_n\}$ either belongs to some call SCC which is lower than or equal to $C_n.scc$. If the former case according to Lemma 9 the answer can never be marked. Clearly C_n is not marked. Therefore the answers consumed by $C_1, \ldots, C_{n-1}$ i.e. $\{A_1, \ldots, A_{n-1}\}$ are not marked. *consume_answer* can be called from an event E with $existed\_answer(A_n)$ to be true or false. In either case $E.ord \geq C_n.ord$ (from the definitions of ordinals of *consume_answer*). As the *mark* events of ordinal $\leq E.ord$ have already been processed before event E, it follows from property 11 that the unmarked answers with ordinal $\leq E.ord$ can never be marked. From the definition of ordinal number of consumers, it follows that $\forall j \in [1, n-1], A_j.ord \leq C_n.ord \leq E.ord$. Thus all answers in the set $\{A_1, \ldots, A_{n-1}\}$ cannot be marked. If the event E was generated with $existed\_answer(A_n) = true$ then following Property 12 A_n can never be marked. Note that $A_n.ord < A.ord$ follows from the definition of ordinal of an answer. If the event E is generated for $existsed\_answer(A_n) = false$ (which means it was inserted) then following induction assumption we derive that A_n can never be marked. Thus the primary support $\{A_1, \ldots, A_n\}$ can never be marked and consequently A can never be marked.

Property 14 *If $\{A_1, A_2, \ldots, A_n\}$ is a support of an unmarked answer generated by consume_answer(A_n, C_n) event then $\{A_1, A_2, \ldots, A_n\}$ is never marked in the same incremental phase.*

Proof: *Similar to the proof of Property 13.*

6.8.2 Relation between Incremental and Non-Incremental Evaluation- Proof of Correctness.

A definite logic program P is a tuple $\langle \mathcal{R}, F \rangle$ where F is a set of facts and $\mathcal{R}$ is a set of rules. Let $EVAL_{slg}(P', Q')$ returns a tuple $\langle A', \Gamma' \rangle$ where A' is the union of set of answers generated by SLG evaluation while evaluating the queries in Q' and the set of facts in P', and Γ' is the set of calls generated by SLG evaluation.

Let we are given a definite logic program $P = \langle \mathcal{R}, F \rangle$ and a set of queries Q and $EVAL_{slg}(P, Q) = \langle ans(P), \Gamma \rangle$. Let δ^- and δ^+ be the set of facts deleted from F and inserted to F respectively. We assume that $\delta^- \cap \delta^+ = \phi$. The new/changed program $P^\nu = \langle \mathcal{R}, F - \delta^- \cup \delta^+ \rangle$. Let $EVAL_{slg}(P^\nu, Q \cup \Gamma) = \langle ans(P^\nu), \Gamma' \rangle$. Thus $ans(P^\nu)$ is the set of answers and facts generated on the new program by SLG evaluation of the set of queries $Q \cup \Gamma$ and Γ' is the set of calls thus generated. Note that after the change to the program P, non-incremental tabled evaluation is carried out using a different set of query $Q \cup \Gamma$. Unless stated we consider now onwards that the changed program P^ν is evaluated for the query $Q \cup \Gamma$.

Given P, δ^+, δ^-, and $ans(P)$ as inputs to an incremental algorithm, the set of answers (corresponding to the set of calls Γ') marked, rederived and inserted by the incremental algorithm by M, R, and I respectively.

Also for the following proofs we consider a support S is represented by the set of answers/facts that affects S i.e. $A \in S$ if $S \in A.affected$.

We can say that the set of deleted answers is $ans(P) - ans(P^\nu)$. The soundness of the incremental marking depends on that fact that every deleted answers is marked and is expressed by the following lemma.

Lemma 15 *Let $P = \langle \mathcal{R}, F \rangle$ be a definite logic program, δ^- and δ^+ be the set of deleted and inserted facts respectively, $ans(P)$ and $ans(P^\nu)$ be the set of answers* *generated by non-incremental evaluation on programs P and $P^\nu = \langle \mathcal{R}, F - \delta^- \cup \delta^+ \rangle$ respectively, and M be the set of answers marked by the incremental algorithm. Then every deleted answers are marked i.e. $ans(P) - ans(P^\nu) \subseteq M$.*

Proof: The proof of this lemma is based on the fact that an unmarked answer is never going to be deleted in the same incremental phase. Note that we do not mark an answer if there exists an acyclic support of the answer or an acyclic support is generated by insertion. Using Property 14 we know that an acyclic support generated by insertion is never going to be deleted. Also it is easy to construct a proof for an answer using an acyclic support, by recursively building proofs of answers in the support. Since the support is acyclic, this recursive process will terminate, yielding a proof. Hence it follows that any answer with an unmarked acyclic support has a derivation and hence is not deleted. Thus an unmarked answer is not deleted. Therefore an answer in the set of unmarked answer ($ans(P) - M$) belongs to the final set of answers $ans(P^\nu)$ i.e. i.e. $ans(P) - M \subseteq ans(P^\nu)$. By rearranging this equation we prove the Lemma.

Our rederivation algorithm rederives the answers and supports that are initially marked but belongs to the final set of answers generated, and its soundness is expressed by the following Lemma.

Lemma 16 *Let $P = \langle \mathcal{R}, F \rangle$ be a definite logic program, δ^- and δ^+ be the set of deleted and inserted facts respectively, $ans(P^\nu)$ be the set of answers generated by non-incremental evaluation on program $P^\nu = \langle \mathcal{R}, F - \delta^- \cup \delta^+ \rangle$, and M and R be the set of answers marked and rederived by the incremental algorithm respectively. Then $R \subseteq M \cap ans(P^\nu)$.*

Proof: We rederive an answer only if it is marked. Therefore $R \subseteq M$. Now we need to prove that all rederived answers have atleast one unmarked support at the time of rederivation which is never going to be deleted. From the precondition of generation of rederive event we note that every answer to be rederived has atleast one unmarked support (precondition of generation of rederive events; see Figure 38(a): Line 1). Such a support, say S, consists of unmarked answers which can be unmarked due to three possible reasons: (i) answers that are not marked by *mark*; (ii) answers that are inserted; (iii) answers that are rederived. Since at any call-graph component we start processing *rederive* events after all the *mark* events have been processed, answers due to reasons (i) are never marked. Also following property 13 we know that an unmarked answer due to reason (ii) are never marked. Note that the support S of a rederived answer A can use a rederived answer A' only if $A'.ord < A.ord$ for $same\_scc(A, A')$ or $A'.scc < A.scc$. Thus using induction on ordinals (proof strategy used in proof of Property 13) we prove that A' can never be marked. Thus the support S responsible for rederivation of the answer A can never be marked. Thus a rederived answer in set R can never be marked and consequently belongs to the final set of answer i.e $R \subseteq ans(P^\nu)$. Thus we derive that $R \subseteq M \cap ans(P^\nu)$.

The completeness of the rederivation phase is given by the following lemma.

Lemma 17 *Let $P = \langle \mathcal{R}, F \rangle$ be a definite logic program, δ^- and δ^+ be the set of deleted and inserted facts respectively, $ans(P^\nu)$ be the set of answers generated by non-incremental evaluation on program $P^\nu = \langle \mathcal{R}, F - \delta^- \cup \delta^+ \rangle$, and M and R be the set of answers marked and rederived by the incremental algorithm respectively. Then $M \cap ans(P^\nu) \subseteq R$.*

Proof: We prove the lemma by induction on height of the shortest proof (hereafter refer to as *derivation height* of an answer) of the answers in $M \cap ans(P^\nu)$. Let $A \in M \cap ans(P^\nu)$ be an answer and derivation height of A be i.

Induction hypothesis: All answers in $M \cap ans(P^\nu)$ having derivation height less than i belong to R i.e. they are rederived.

Induction step: As $A \in ans(P^\nu)$ following property of least fix-point derivation there exists a support S of A such that all answers in S have derivation height less than i. For this proof we assume the support S of answer A was generated by resolution of a binary clause. This assumption can be easily extended to the general case. Let $H :- B_1, B_2$ be such a binary clause such that A is an answer to a call γ (γ and A are unifiable), $A = H\theta$, $S = \{A_1, A_2\}$, $A_1 = B_1\theta_1\theta_2$, $A_2 = B_2\theta_1\theta_2\theta_3$, and $\theta = \theta_1\theta_2\theta_3$. As $A \in ans(P^\nu)$, $\{A_1, A_2\} \subseteq ans(P^\nu)$. We partition the set $ans(P^\nu)$ into three disjoint sets U, V, and W where $U = ans(P) - M$ represents the set of answers and facts corresponding to the program P that are not marked, $V = M \cap ans(P^\nu)$ represents the set of answers that are marked but belong to the answer set corresponding to the changed program, and $W = ans(P^\nu) - ans(P)$ represents the inserted answers.

Now we consider the following six cases:

I. $\{A_1, A_2\} \subseteq U$ represents the case where both A_1 and A_2 are answers or facts in the old program that are not marked in the incremental phase.

   In this case the support S exists before the incremental phase and S is not marked in the incremental phase. Now as A is marked there was $mark(A)$ event which marked A. When $mark(A)$ was generated there was a $may\_rederive(A)$ event generated as well. (Figure 36(a) Line 11). Processing $may\_rederive(A)$ event (Lines 1-4, Figure 38(a)) generates $rederive(A)$ event due to the presence of unmarked support S. Processing $rederive(A)$ event removes the delete mark from A (Line 1 Figure 38(b)). Following Lemma 16, once the answer A is rederived it is never going to be deleted in the same incremental phase. Thus $A \in R$.

II. $\{A_1, A_2\} \subseteq V$ represents the case where both A_1 and A_2 are answers that are marked but belong to the answer set corresponding to the changed program.

   By induction hypothesis $\{A_1, A_2\} \subseteq R$. Since $\{A_1, A_2\} \subseteq ans(P)$, S already exists before the incremental phase. Since A_1 and A_2 are rederived there exist two events $rederive(A_1)$ and $rederive(A_2)$. Processing of these events makes S unmarked and generates $rederive(A)$ (Line 12, Figure 38(b)).

III. $A_1 \in U$ and $A_2 \in W$ represent the case where A_1 is an answer or fact that remains unmarked after incremental processing and A_2 is an answer or fact inserted.

   Since $A_1 = B_1\theta_1\theta_2 \in U$ there exists a consumer $C_2 = \langle \gamma, H\theta_1\theta_2, B_2\theta_1\theta_2, [] \rangle$ before the incremental phase has started. Insertion of answer $A_2 = B_2\theta_1\theta_2\theta_3$ generates the event $E = consume\_answer(A_2, C_2)$. Processing event E generates support S. As A is marked before processing of event E thus $mark(A).ord \leq E.ord$. By definition of ordinal of $mark$ event,

$mark(A).ord = A.ord$. Therefore $E.ord \geq A.ord$. Since $E.ord = A_1.ord$ (by definition of ordinal of *consume_answer* event), $A_1.ord \geq A.ord$. As support S contains answer A_1, $S.ord \geq A_1.ord$. Thus execution of Lines 1-11, 18, 19, 20, 22, 23 of Figure 34 generates $may\_rederive(A)$ event. As S is unmarked and non-acyclic therefore processing of $may\_rederive(A)$ event generates the $rederive(A)$ event and subsequently A is rederived.

IV. $\{A_1, A_2\} \in W$ denotes the case where both A_1 and A_2 are inserted by the incremental algorithm.

Insertion of the answer A_1 generates $E = consume\_answer(A_1, C_1)$ event where $C_1 = \langle \gamma, H\theta_1, B_1\theta_1, [B_2\theta_1] \rangle$. Processing of the event $consume\_answer(A_1, C_1)$ generates the consumer C_2.

Two cases can arise:

- A_2 was generated before generation of event E i.e. $existed\_answer(A_2) = true$, in which case, an event $E = consume\_answer(A_2, C_2)$ (Lines 1, 7-9 of Figure 35 will be generated execution of which generates the support S. As A is marked, $A.ord \leq E.ord$. Therefore the execution of this event (same as described in the proof of case (III)) generates S and $may\_rederive(A)$ event which subsequently rederives A.

- A_2 was generated after generation of event E.

  Insertion of answer A_2 generates $consume\_answer(A_2, C_2)$ event. The execution of this event (same as described in the proof of case (III)) generates S and $may\_rederive(A)$ event which subsequently rederives A.

V. $A_1 \in U$ and $A_2 \in V$. In this case A_1 is an unmarked answer or fact of the old program and A_2 is an answer that is marked in the incremental phase but belongs to the set of answers corresponding to the changed program.

By induction hypothesis $A_2 \in R$. Thus $rederive(A_2)$ event is generated. Processing event $rederive(A_2)$ (Lines 1-12 of function in Figure 38(b)) generates $rederive(A)$ event which subsequently rederives A.

VI. $A_1 \in V$ and $A_2 \in W$

Since $A_1 \in V$ we derive that $A_1 \in ans(P)$ and subsequently C_2 is generated. As A_1 is marked there exists an event $E_m = mark(A_1)$.

Depending on time of generation of event $E_i = consume\_answer(A_2, C_2)$ two cases can arise:

- E_i exists in the ready queue while processing E_m.

  As $E_i.ord = C_2.ord = A_1.ord = E_m.ord$, E_i is not processed before E_m event. Processing E_m moves E_i from ready queue to delay queue (Lines 4-6 *mark(b), Figure 36*(a)). By induction hypothesis $A_1 \in R$ and hence there exists an event $E_r = rederive(A_1)$. Processing E_r moves E_i from delay queue to ready queue (Lines 3-4, Figure 38(b)). Processing of E_i generates support S and creates $may\_rederive(A)$ event function $consume\_answer(A_2, C_2)$ Figure 34. Execution of event $may\_rederive(A)$ generates $rederive(A)$ event.

- E_i does not exist in the ready queue while processing E_m.

  By induction hypothesis $A_1 \in R$ and hence there exists an event $E_r = rederive(A_1)$. Processing E_r makes C_2 unmarked. Insertion of A_2 generates E_i and as in previous case execution of E_i subsequently rederives A.

Following Lemma 16, in all the above cases once the answer A is rederived it is never deleted in the same incremental phase.

Base Case: Any answer with derivation height=1 is a fact. Note that no fact can belong to $M \cap ans(P^\nu)$. Thus the base case is vacuously proved.

The completeness of insertion algorithm is given by the following lemma.

Lemma 18 *Let $P = \langle \mathcal{R}, F \rangle$ be a definite logic program, δ^- and δ^+ be the set of deleted and inserted facts respectively, $ans(P^\nu)$ and $ans(P)$ be the set of answers generated by non-incremental evaluation on program $P^\nu = \langle \mathcal{R}, F - \delta^- \cup \delta^+ \rangle$ and P respectively, and I be the set of answers inserted by the incremental algorithm. Then $I \supseteq ans(P^\nu) - ans(P)$.*

Proof: We prove the lemma by induction on derivation height (see Lemma 17) of the answers in $ans(P - \delta^- \cup \delta^+) - ans(P)$. Let $A \in ans(P^\nu) - ans(P)$ be an answer and derivation height of A be i.

Induction hypothesis: Induction hypothesis: All answers in $ans(P^\nu) - ans(P)$ with derivation height less than i are inserted by incremental algorithm i.e. they belong to I.

Base Case. Answers with derivation height=1 are inserted facts and thus belong to I.

Induction step. As $A \in ans(P^\nu) - ans(P)$ following property of least fix-point derivation there exists a support S of A such that all answers in S have derivation height less than i. For this proof we assume the support S of answer A was generated by resolution of a binary clause. This assumption can be easily extended to the general case. Let $H :- B_1, B_2$ be such a binary clause such that A is an answer to a call γ (γ and A are unifiable), $A = H\theta$, $S = \{A_1, A_2\}$, $A_1 = B_1\theta_1\theta_2$, $A_2 = B_2\theta_1\theta_2\theta_3$, and $\theta = \theta_1\theta_2\theta_3$. As $A \in ans(P^\nu)$, $\{A_1, A_2\} \subseteq ans(P^\nu)$.

We partition the set $ans(P^\nu)$ into two disjoint sets U and V where $U = ans(P^\nu) - ans(P)$ represents the set of new answers generated by the SLG evaluation of the changed program and $V = ans(P^\nu) \cap ans(P)$ represents common set of answers in old and changed programs.

I. $A_1 \in V$, $A_2 \in U$ represents the case when A_1 is an answer which already existed in the set of answer corresponding to program P and is not deleted and A_2 is an answer for the changed program which does not exist for the old program.

   Since $A_1 \in V$ there already exists a consumer $C_2 = \langle \gamma, H\theta_1\theta_2, B_2\theta_1\theta_2, [] \rangle$. By induction hypothesis $A_2 \in I$. Insertion of the answer A_2 into $B_2\theta_1\theta_2$

table generates an event *consume_answer*(A_2, C_2). Execution of event *consume_answer*(A_2, C_2) generates A.

II. $A_1 \in U$, $A_2 \in V$

Note that execution of query γ generates the consumer $C_1 = \langle \gamma, H\theta_1, B\theta_1, [C\theta_1] \rangle$. Also by induction hypothesis $A_1 \in I$. Thus insertion of answer $A_1 = B_1\theta_1\theta_2$ into $B_1\theta_1$ table generates an event *consume_answer*(A_1, C_1). Execution of event
consume_answer(A_1, C_1) generates the consumer C_2 and
$E =$ *consume_answer*(A_2, C_2), Figure 34 and Figure 35 . As $A_2 \in V$ i.e A_2 is already in $B_2\theta_1\theta_2$'s answer table. Now two cases can arise:

- A_2 is not marked.

  In this case E is generated in the ready queue and subsequent execution of generates A.

- A_2 is marked deleted.

  In this case *consume_answer*(A_2, C_2) is in the delay queue (Line 11, Figure 35. As $A_2 \in V$ it will be eventually rederived. Rederivation of A_2 will move the *consume_answer* event from delay queue to ready queue (Line 2, Figure 38(b)). Execution of event *consume_answer*(A_2, C_2) generates A.

III. $\{A_1, A_2\} \subseteq U$

By induction hypothesis, $\{A_1, A_2\} \subseteq I$. Insertion of answer A_1 generates $E =$ *consume_answer*(A_1, C_1) event. Execution of this event generates the consumer C_2.

Two cases can arise:

- A_2 was inserted after E was executed.

  Insertion of answer A_2 generates *consume_answer*(A_2, C_2) event which when processed generates answer A.

- Execution of E generates *E'=consume_answer(A_2,C_2)* in the ready_queue. Execution of E' generates A.

The soundness of our insertion algorithm is given by the following lemma.

Lemma 19 *Let $P = \langle \mathcal{R}, F \rangle$ be a definite logic program, δ^- and δ^+ be the set of deleted and inserted facts respectively, $ans(P^\nu)$ and $ans(P)$ be the set of answers generated by non-incremental evaluation on program $P^\nu = \langle \mathcal{R}, F - \delta^- \cup \delta^+ \rangle$ and P respectively, and I be the set of answers inserted by the incremental algorithm. Then $I \subseteq ans(P^\nu) - ans(P)$.*

Proof Sketch: The proof of the lemma is based on the following facts: (i) all new answers generated by incremental algorithm are logical consequences of the existing facts and rules (ii) all new answer generated are not marked and hence not deleted (Property 13).

Theorem 20 *Let $P = \langle \mathcal{R}, F \rangle$ be a definite logic program, δ^- and δ^+ be the set of deleted and inserted facts respectively, $ans(P^\nu)$ and $ans(P)$ be the set of answers generated by non-incremental evaluation on program $P^\nu = \langle \mathcal{R}, F - \delta^- \cup \delta^+ \rangle$ and P respectively, and M, R, I are the set of answers marked, rederived and inserted respectively. Then the set of answers generated by non-incremental evaluation is same as the set of answers generated by the incremental algorithm i.e. $ans(P) - (M - R) \cup I = ans(P^\nu)$.*

Proof: The proof of this theorem follows from all of the above lemmas.

$$
\begin{aligned}
& ans(P) - (M - R) \cup I. \\
&= ans(P) - (M - R) \cup (ans(P^\nu) - ans(P)) \\
&= ans(P) - (M - (M \cap P^\nu))) \cup (ans(P^\nu) - ans(P)) \\
&= ans(P) - (M - P^\nu)) \cup (ans(P^\nu) - ans(P)) \\
&\qquad [M \supseteq ans(P) - P^\nu) \Rightarrow \\
&\qquad M - ans(P^\nu) = ans(P) - ans(P^\nu)] \\
&= ans(P) - (ans(P) - ans(P^\nu)) \cup (ans(P^\nu) - ans(P)) \\
&= (ans(P) \cap ans(P^\nu)) \cup (ans(P^\nu) - ans(P)) \\
&= ans(P^\nu).
\end{aligned}
$$

6.9 Related Work

Our primary-support-based algorithm (Chapter 4) improved on the DRed strategy by significantly reducing the need to propagate deletions. 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 local algorithm presented in this chapter further optimizes and extends the deletion mark propagation: (i) using the effect of addition of new facts and answers which is very useful in updates where addition 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 Chapter 4 incremental addition was done by evaluating difference rules (obtained by program transformation) which are evaluated top-down. In contrast, in this chapter we presented a combined bottom-up algorithm to handle both additions and deletions.

The idea of using SCC-reduced dependency graphs to optimize propagation of changes has been seen in various past works [Jon90, WJ88, PH96, HPMS00, CNDE05]. Among these, Hermenegildo et. al.'s works [PH96, HPMS00] on reanalyzing (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 [PH96] addition 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 [HPMS00]. However, we use event ordering to control propagation of changes *within* an SCC, leading to finer-grained interleaving between addition and deletion operations.

6.10 Discussion

In this chapter we presented an efficient algorithm for incrementally evaluating definite logic programs with the rules/facts of the program are changed: added, deleted, or updated. The key to the algorithm is the interleaving of addition and deletion operations based on an order. The algorithm naturally generalizes to techniques that were developed in the 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 additions 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.

The focus of this work 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, and to establish how this generalizes previous special-case algorithms. The algorithm maintains extensive dependency information. We believe that techniques such as those used in symbolic support graphs (Chapter 5) can be used to compactly store the dependencies.

Chapter 7

Extending Incremental Tabled Evaluation Beyond Pure Logic Programs

In the previous chapters of this thesis we have developed time and space efficient techniques for incremental evaluation of tabled logic programs. However, these techniques cannot be readily applied to arbitrary tabled logic programs, especially those that use aggregation and other Prolog built-ins, or have non-stratified negation. In the presence of non-monotonic operators, it is often difficult to determine whether the addition of an answer to a table results in addition or deletion of an answer to another table.

In this chapter, we present an incremental evaluation algorithm that is based on *call* dependencies instead of answer dependencies, and process additions as well as deletions using a single method. At a high level, the technique works as follows. When facts or rules of a program change, we first mark all calls in tables whose answers may be affected by this change. In the next step we re-evaluate the marked calls. Naive re-evaluation is often inefficient since the call dependencies are too coarse compared to answer dependencies. Our algorithm chooses calls to be re-evaluated optimally, and sequences the re-evaluations judiciously to minimize the number of wasteful computations (see Section 7.1).

The salient advantages of this technique are:

- The technique can be used on any tabled program, regardless of the use of intermediate non-tabled predicates and Prolog built-ins.
- The technique is agnostic to the sign of a dependency— i.e. whether a call depends negatively or positively on another— and hence can be used without change on general logic programs: *even those with non-stratified negation.*
- The re-evaluation phase issues calls in an optimal order, re-evaluating calls only when needed, and resulting in good performance in practice.
- Call graphs are generally small, and hence the technique scales to large examples.

7.1 Incremental Evaluation based on Call Dependencies

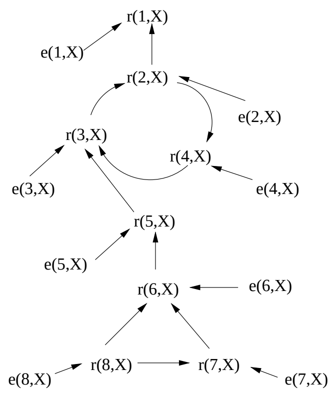
Our technical development is based on the SLG resolution [CW96]; however the definitions as well as the results of this chapter can be ported to other tabled evaluation schemes as well [ZSYY01, GG01, e.g.]. Although SLG resolution is usually described using pure logic programs, it has been integrated into Prolog-based systems such as [XSB, RSC00] seamlessly enough to permit programs to mix tabled and non-tabled predicates, use aggregate and other Prolog builtins, and even use cuts over non-tabled predicates. Analogously, the concepts formally developed below based on SLG resolution can be extended to the more general class of tabled Prolog programs.

Given a program P and an initial query q, the set of call tables constructed by SLG resolution is denoted by $\mathit{calls}(q, P)$. The set of answers computed for a subgoal q over program P is denoted by $\mathit{ans}(q, P)$. The set of all answer tables constructed during evaluation of a query q, denoted by $\mathit{answer\_tables}(q, P)$ is given by the collection $\{\mathit{ans}(q', P) \mid q' \in \mathit{calls}(q, P)\}$. In SLG resolution derivations are captured as an SLG *forest*, where each tree corresponds to a single call and its associated answer table. Our incremental algorithm makes a non-trivial change to only one of the operations used to build the SLG forest: namely, the *completion check* operation, which determines whether any more answers can be added to an answer table. The other operations are either unchanged or are changed trivially to record call dependency information.

```
:- 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(3,5).
e(4,2).
e(5,6).
e(6,7).
e(6,8).
e(7,8).
```

(a)



(b)

Figure 42: Example program (a); and called-by graph (b) for evaluating `r(1,X)`

We consider incremental evaluation of tabled programs, where facts or rules may be added or deleted after query evaluation is completed. Each complete query evaluation is called a *run*. Between each run, a set of rules in the program may change. We denote this set by C and partition C into two sets C^+ and C^- that contain the added and deleted rules respectively. Given a program P, the changed program P' obtained by applying the changes in C is given by $P' = P \cup C^+ - C^-$. Note that our technical development is general and considers changes to a program's *rules*. Facts, which are rules with empty bodies, naturally become a special case.

Our algorithm is based on tracking dependencies between the calls during query evaluation. The smallest set of calls that need to be re-examined after a change, defined formally below, are those whose answer tables are modified by the change.

Definition 22 (Changed Calls) *Let P be a program, $C = C^+ \cup C^-$ be the set of rules that are changed, and $P' = P \cup C^+ - C^-$ be the changed program. Let Q be the set of calls due to evaluation of some query over P. The set of changed calls, denoted by changed(P, C) is the set of all calls in Q such that ans$(q, P) \not\equiv$ ans(q, P').*

We assume that all predicates whose definitions are subject to change between runs are marked as *volatile*. For instance, in the program in Figure 42(a), `edge/2` is a volatile predicate. In general a volatile predicate may be defined by rules, and may even be tabled.

Our call-dependency-based incremental evaluation technique is based on a data structure known as *called-by* graph, defined below.

Definition 23 (Called-By Graph) *The called-by graph due to the evaluation of query q over program P is a directed graph (V, E) such that (i) $V = V_t \cup V_f$ where V_t is the set of tabled subgoals that occur as roots of trees in the SLG forest, and V_f is the set of selected literals in the SLG forest that unify with the head of some volatile rule; and (ii) $(c_1, c_2) \in E$ if and only if c_1 is a selected subgoal in a tree with c_2 as the root (i.e. c_1 is called by c_2).*

The called-by graph after evaluation of query `r(1,X)` over the program in Figure 42(a) is given in Figure 42(b). The graph captures the dependencies between tabled calls and calls to volatile predicates. It is first generated in the initial (non-incremental) run, and maintained over subsequent incremental runs. Note that the called-by graph is the transpose of the subgoal dependency graph [CSW95] extended with edges from calls to volatile predicates.

The incremental algorithm has two phases. The first is the *invalidation* phase, where calls that may be affected by the change are marked as *affected*.

Definition 24 (Initially Changed Calls) *Given a called-by graph $G = (V, E)$ and a non-empty set $C = C^+ \cup C^-$ of rules that were changed (added or deleted) since the last run, the set of initially changed calls, denoted by $init(G, C)$ are those $v \in V$ such that v unifies with the head of some rule in C.*

Definition 25 (Affected Calls) *Given a called-by graph $G = (V, E)$ and a non-empty set $C = C^+ \cup C^-$ of rules that were changed (added or deleted) since the last run, the set of affected calls, denoted by affected(G, C), is the smallest set such that $v \in affected(C, G)$ if*

- *$v \in init(G, C)$, or*
- *$\exists v' \in affected(G, C)$ such that $(v', v) \in E$.*

The set of affected calls (based on the above definition) can be found by simply traversing the called-by graph starting from the vertices that unify with changed rule heads (case (i) above). Note that the direction of edges in the called-by graph is from callee to caller which enables us to compute the affected calls by traversing the called-by graph.

The idea behind the invalidation phase is that the calls that are not deemed affected are unchanged by the modification, as formally stated below:

Theorem 21 *Let P be an initial program, $C = C^+ \cup C^-$ be the set of changed rules, and $P' = P \cup C^+ - C^-$ be the changed program. Let $G = (V, E)$ be the called-by graph for some query over P. Then, every changed call is affected; i.e. $changed(P, C) \subseteq affected(G, C)$.*

Naive Re-Evaluation: Theorem 21 means that when some program rules change, it is sufficient to re-evaluate the set of affected calls. Our naive strategy is to remove all table entries corresponding to the affected calls (i.e. their entries in the call table, as well as their answer tables) in the invalidation phase. In the second phase, called re-evaluation phase, we re-do program clause resolution of all the affected calls. Note that *all* affected calls and their answer tables are deleted to ensure that any answer derived for an affected call is based only on valid information: either rederived answers of another affected call, or existing answers of an unaffected call. While deleting the table entries for an affected call, we also remove the corresponding vertex and the edges incident on it from the called-by graph. Note that the re-evaluation may generate new vertices and edges in the called-by graph. Thus the called-by graph itself is (incrementally) modified when processing incremental changes.

For example, consider the deletion of the fact `e(3,5)` from the program in Figure 42(a). The invalidation phase identifies the calls `e(3,X)`, `r(3,X)`, `r(2,X)`, `r(4,X)` and `r(1,X)` as affected. Since these calls will be re-evaluated, the edges incident on these vertices, i.e. $e(3,X) \rightarrow r(3,X)$, $r(5,X) \rightarrow r(3,X)$, $e(4,X) \rightarrow r(4,X)$, $r(4,X) \rightarrow r(3,X)$, $e(2,X) \rightarrow r(2,X)$, $r(2,X) \rightarrow r(4,X)$, $e(1,X) \rightarrow r(1,X)$, and $r(2,X) \rightarrow r(1,X)$, are deleted from the called-by graph. In the re-evaluation phase, the call `r(1,X)` gives rise to calls `r(2,X)`, `r(3,X)`, and `r(4,X)`, and their answers are subsequently computed. These calls and the corresponding edges are added (back) to the called-by graph. Note that, answers to unaffected calls can be found directly from the tables. For example, the call `r(3,X)` uses already existing answers for `e(3,X)` and `r(5,X)`; calls such as `r(5,X)` are unaffected by the deletion and are not re-evaluated, thereby saving expensive program clause resolution steps.

Optimized Re-evaluation: The set of affected calls overapproximates the set of changed calls. In many cases, the approximation may be severe and the naive re-evaluation strategy wastefully re-evaluates unchanged calls. Consider the deletion of fact `e(7,8)` from the program Figure 42(a). The invalidation phase identifies the calls `e(7,X)`, `r(7,X)`, `r(6,X)`, `r(5,X)`, `r(3,X)`, `r(2,X)`, `r(4,X)`, and `r(1,X)` as affected. However, the set of changed calls is only `e(7,X)` and `r(7,X)`, but the naive strategy also re-evaluates all other affected calls.

We obtain a better approximation of the changed set, called the recomputed set defined as follows.

Definition 26 (Recomputed Set) *Let P be a program, $C = C^+ \cup C^-$ be the set of changed rules, and $P' = P \cup C^+ - C^-$ be the changed program. Let $G = (V, E)$ be the called-by graph for some query q over P. Then, the set of recomputed calls, denoted by recomputed(G, C), is the smallest set such that $c \in recomputed(G, C)$ if*

I. $c \in init(G, C)$, or

II. there is some c' such that $(c', c) \in E$ and $c' \in changed(P, C)$, or

III. there is some c' such that c and c' are in the same strongly connected component of G, and $c' \in recomputed(G, C)$.

The recomputed set represents the smallest set of calls that need to be re-evaluated. The intuition behind this definition follows from the following observations:

I. Every changed call needs to be re-evaluated.

II. Every call that immediately depends on a changed call needs to be re-evaluated (even if it itself is not changed). Note that the called-by graph contains no qualitative information on *how* the change of a call affects another. Only the program has this information embedded in it, and hence the only way to determine whether or not such a call changes is to re-evaluate it.

III. If a re-evaluated call is in a SCC, then all calls in that SCC need to be re-evaluated. For instance, when `e(3,5)` is deleted from the program in Figure 42(a), `e(3,X)` is changed, and hence `r(3,X)` is recomputed. Note that we cannot simply delete `r(3,X)`'s tables and re-evaluate it: since `r(4,X)` currently contains the answer `X=5`, and `e(3,4)` holds, we will then (incorrectly) conclude that `r(3,5)` still holds. Hence, we have to re-evaluate all mutually dependent calls simultaneously (`r(3,X)`, `r(4,X)` and `r(2,X)`, in this case).

It follows from the definition that every changed call is also in the recomputed set. It can also be readily shown that every call in the recomputed set is affected. Formally,

Proposition 22 *Let P be a program, $C = C^+ \cup C^-$ be the set of changed rules, and $P' = P \cup C^+ - C^-$ be the changed program. Let $G = (V, E)$ be the called-by graph for some query q over P. Then changed$(P, C) \subseteq$ recomputed$(G, C) \subseteq$ affected(G, C).*

In optimized re-evaluation we redo the calls in the *recomputed set*. We need two basic mechanisms to accomplish this: (a) determine whether a re-evaluated call is changed or not, and (b) determine SCCs in the called-by graph.

a. Marking Changed Calls: First of all, instead of deleting all the affected tables in the invalidation phase, we only mark the answers of a recomputed call as (currently) invalid just before the call is re-evaluated. We do not mark the answers of the affected calls which are not scheduled for re-evaluation. Invalid answers are ignored when doing answer clause resolution. With each such recomputed call, we also keep the number of invalid answers (in a counter called *invalid_count*), initialized to the total number of answers at the beginning of the re-evaluation phase. Finally, we keep a flag with each recomputed call (called *addl_answer*) to indicate whether a new answer was added to this call's answer table in the re-evaluation phase. During re-evaluation, whenever an answer is added to a table, if the answer already exists but is invalid, we remove the invalid mark and decrement *invalid_count* for the table. If the answer did not exist before, we add the answer and set *addl_answer* of the call to true. When a call is completely re-evaluated (at the Completion operation of SLG), we can determine that the call is *changed* iff *addl_answer* is true or *invalid_count* is non-zero.

b. Evaluating SCCs: Finding SCCs in the called-by graph is fundamental to evaluating the *recomputed* set. Apart from the explicit use of SCC information in its definition, note that we determine whether or not a call is *changed* only after completion. Consequently we need to evaluate the calls "bottom-up" through the called-by graph, and triggering re-evaluations at higher levels only after confirming that the lower-level calls have changed. This strategy has been shown to be optimal for acyclic graphs.

Algorithms for finding SCCs typically need an additional pass over the graph. We now describe a technique to find SCCs without making this additional pass, by slightly modifying the traversal used in the invalidation phase. This technique is based on Kosaraju and Sharir's SCC computation algorithm [Sha81], which works as follows. To find SCCs in a graph G, we first traverse G and give post-order numbers to the vertices in G. We then traverse G^T, the transpose of G, starting from the vertex with the highest post-order number; this traversal builds a spanning tree for one SCC of G. Whenever the traversal ends, we begin a new traversal from the unvisited vertex with the highest post-order number, thereby building a spanning tree for another SCC. This process continues until all vertices have been visited, enumerating all SCCs of G. The order in which SCCs are found by the Kosaraju-Sharir algorithm is a topological order in the SCC-reduced graph of G: if (v_1, v_2) is an edge in E, then the SCC containing v_1 is found at least as early as the one containing v_2.

The Re-Evaluation Algorithm: We now describe a re-evaluation algorithm that implicitly finds SCCs. In the invalidation phase, we traverse the called-by graph and assign a post-order number to each affected call. With each affected call we keep a flag *processed* which is initialized to false. In the re-evaluation phase, shown in Figure 43, we maintain a sequence of calls to be re-evaluated in a global data structure known as the *working sequence* (variable *ws* in the algorithm). This sequence is maintained using a heap data structure, keeping the calls in the descending order of their post-order numbers. During re-evaluation, we pick the call with the highest post-order number from *ws* and invoke the call. Re-evaluation continues until the working sequence becomes empty. When the re-evaluation of a call c is complete, and c has changed, we add all its immediate successors in the called-by graph to the working sequence.

```
    re_eval(G, C)
1.  ws := init(G,C);
2.  while (ws is not empty)
3.      remove c, the call with the
          highest PO number from ws;
4.      call(c);

    In SLG's Completion Op. for call c:
1.  if (c.addl_answer) or
       (c.invalid_count > 0)
2.      foreach $c'$ such that $(c, c') \in E$
3.          if not $c'$.processed
4.              add $c'$ to ws
5.              $c'$.processed := true
```

Figure 43: Optimized Re-Evaluation Algorithm

Note that, during re-evaluation, if c_2 calls c_1 then corresponding edge in the called-by graph is (c_1, c_2). *Thus re-evaluation implicitly traverses the transpose of the called-by graph.* If c_1's table is either unaffected or has been recomputed completely, then c_2 can use the answers from that table. Otherwise, c_1 will also be re-evaluated. This ensures that all calls in an SCC of the called-by graph will be evaluated simultaneously.

The correctness of the algorithm, stated in the following theorem, can be established following the properties of the Kosaraju-Sharir algorithm and the definition of *recomputed* set.

Theorem 23 *The set of calls picked by the re-evaluation algorithm (line 3 of re_eval in Figure 43) is the same as the recomputed set.*

In the example, when `e(7,8)` is deleted, the reverse postorder of affected calls is given by the sequence `e(7,X)`, `r(7,X)`, `r(6,X)`, `r(5,X)`, `r(3,X)`, `r(4,X)`, `r(2,X)`, `r(1,X)`. The set of initially changed calls is {`e(7,X)`}. When `e(7,X)` is re-evaluated, its answer `e(7,8)` is removed, and hence we deem the call to have changed. This causes `r(7,X)` to be added to the working sequence. When this call is re-evaluated, it too is deemed to have changed (answer `r(7,8)` is no longer derivable). Hence we add `r(6,X)` to the working sequence. Re-evaluating `r(6,X)`, we find that it has not changed. The working sequence is now empty and the re-evaluation is complete. Thus, among the 8 affected calls, we re-evaluated only 3.

7.2 Experimental Results

Below we present experimental results on the performance of the naive and optimized algorithms on various classes of tabled logic programs. The algorithms were implemented by extending XSB logic programming system [XSB] (v2.7.1). All measurements were taken on a PC with 3GHz Pentium 4 processor with 2GB of physical memory running Linux (RedHat) version 2.6.9. Our implementation, benchmarks, additional experimental results on simple reachability analysis and push down model checking are available in [SR06].

Dynamic Programming: We measured the performance of our algorithms on a set of familiar dynamic programming problems. Support graph based incremental techniques discussed in Chapter 5 cannot be directly used to capture the answer dependencies in these problems due to the use of aggregation operations (min, max etc.). Figure 44 summarizes the relative time performance of incremental evaluation (w.r.t. from-scratch evaluation time) averaged over several possible changes for different dynamic programming problems: longest common subsequence (LCS), minimum edit distance (EDD), and matrix chain multiplication (MM).

LCS: We evaluated the performance of incremental evaluation on LCS by changing the character at some position in one of the strings. On average, 50% calls are affected, and 11% are changed and 15% are recomputed. Although only 15% of the calls are re-evaluated by our optimized incremental algorithm, the time taken for re-evaluation is close 30%. This is due to the overhead of answer clause resolution that our current implementation performs (from the top-level) even for calls that are not recomputed. Incremental evaluation of LCS is sensitive to positions of characters in the string that were changed, as shown by Figure 45.

EDD: The solution to EDD is very similar to that of LCS. The two problems differ in the number of dependent calls for each call. Every call in EDD evaluation is connected to 3 calls in the call-by graph whereas in LCS each call is connected to at

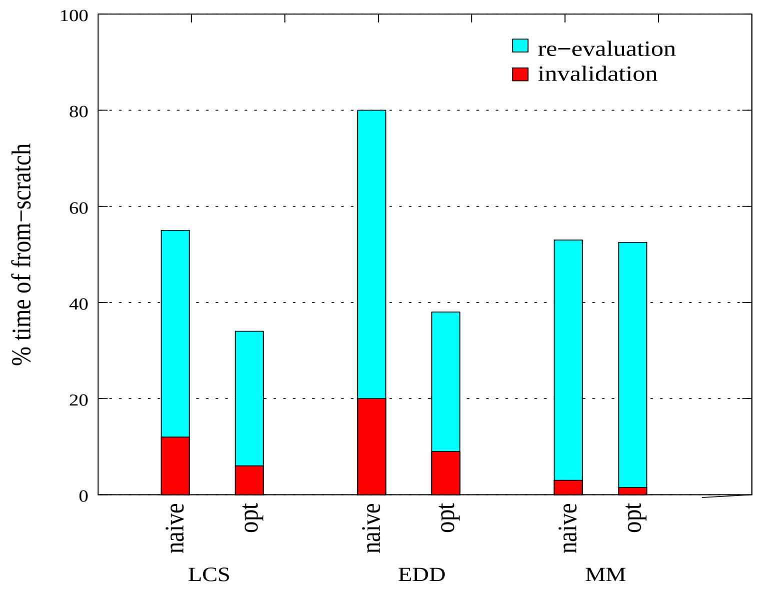


Figure 44: Performance on Dynamic Programming problems

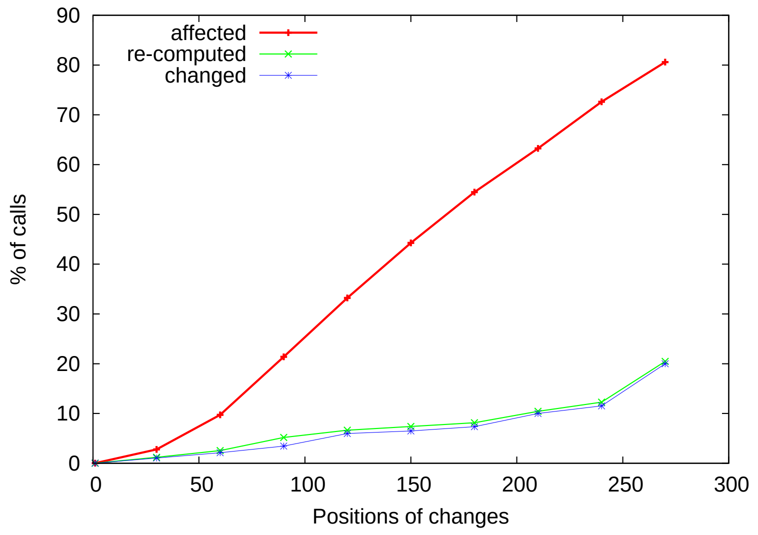


Figure 45: The effectiveness of the optimized algorithm on LCS.

most 2 calls. Hence the number of affected calls in higher in EDD, resulting in higher invalidation time.

MM: For matrix chain multiplication, we deleted one matrix from the chain and measured the incremental and from-scratch time. For such a change, all affected calls are recomputed. Hence the optimized algorithm performs no better than the naive one.

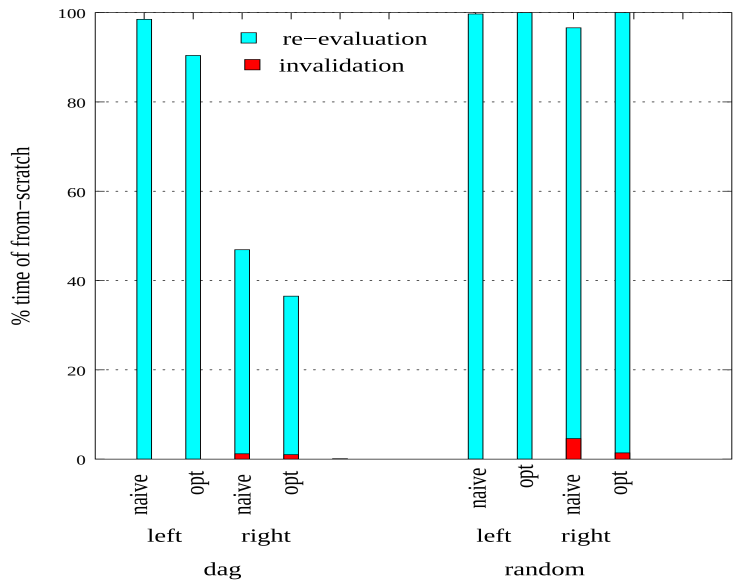


Figure 46: Performance on All-Pair Shortest Path

All-Pair Shortest Path: We experimented with encodings of the all-pair shortest path problem on a directed acyclic graph having 50K nodes and randomly generated graph having 50K edges and 250 nodes (close to complete graph). We performed separate experiments with two different logic program encodings (with left and right recursion, resp.). For the almost-complete graph, incremental evaluation algorithms are not effective since almost all calls are recomputed. For DAGs, the right-recursive version shows better incremental performance due to the availability of non-trivial call dependency information.

Data Flow Analysis Reaching definition analysis for imperative programs is a well-known data flow analysis which determines, for each program point, the set of variable definitions (assignments) that may reach that point [ASU86]. We extended the intra-procedural analysis to an inter-procedural setting using the classical approach of replacing procedure calls with jumps: from the call site to the entry point of the callee, and from the exit point of the callee to the statement following the call site. The experiments were performed on various large C programs and for each benchmark 100 random statements (one per incremental run) were chosen for replacement with a skip statement. The logic programming formulation of data-flow analysis uses stratified negation, hence the techniques based on answer dependency

Benchmark	Non	Non-opt. Incr.		Opt. Incr.		% calls	% of aff. calls	
	Incr.	Re-eval	%	Re-eval	%	affected	recomp.	changed
assembler	5.95	3.60	60.6	3.64	61.2	23.5	85	1
diff	4.55	2.23	49.0	2.24	49.2	30.9	97	1
dixie	1.73	0.96	55.6	0.94	54.4	26.8	95	7
gnugo	4.41	2.38	53.9	2.42	54.8	30.6	99	1
learn	1.29	0.53	40.7	0.54	41.4	26.6	93	9
smail	5.50	2.89	52.4	2.85	51.7	25.4	98	2

Table 12: Data flow analysis; One statement replaced with skip; Time is seconds

Benchmark	Naive	Optimized	% of calls	% of affected calls	
	Incr.(%)	Incr. (%)	affected	recomputed	changed
m88ksim	10.1	6.8	1.1	56.4	25.2
vpr	30.8	27.8	4.0	57.9	6.1
smail	72.2	72.3	6.0	90.3	25.8
twmc	42.5	41.7	2.9	85.7	6.0
nethack	82.8	82.2	5.6	67.2	12.8
vortex	97.5	91.3	5.5	68.3	6.6

Table 13: Performance of naive and optimized algorithms on pointer analysis

(Chapters 4, 5) cannot be readily used in this case.

Table 12 shows that incremental algorithms takes on average 50% of from-scratch time although number of affected calls is close to 30%. In all the experiments the invalidation times were negligible. Closer inspection reveal that for these examples 90% of the call nodes belong to a few non-trivial SCCs in the called-by graph. The formation of such large SCCs is due to the inter-procedural jumps which introduce cycles even when the original program had no recursion. Due to the large SCCs, most affected calls are also recomputed. For example in benchmark `learn` 93% of the affected calls are recomputed but only 9% of the affected calls are changed.

Pointer Analysis We used the call-graph based techniques for the incremental evaluation of Anderson's Points-to analysis [And94] encoded as a tabled logic program in Chapter 3.

Table 13 shows the relative performance of naive and optimized incremental algorithms after removal of one (source-level) statement from the benchmark programs, as a percentage of from-scratch time. Deleting one source level assignment statement l

No. of	Non-Incr	Naive Re-eval.	
Elements		Re-eval	%
12K	0.18	0.00	1.25
120K	1.89	0.03	1.55
240K	3.79	0.06	1.59
360K	5.67	0.09	1.64
480K	7.63	0.12	1.62
600K	9.60	0.16	1.64

Table 14: XML Validation; deletion of one element; Time is seconds

may delete multiple primitive assignments statements and hence multiple facts. The results were averaged over 100 randomly chosen deletion of source statements.

Observe that the incremental times for large benchmarks are close to the non-incremental times. We investigated the vortex program to explain its behavior. Pointer analysis of vortex makes 68K calls in total of which on average 4K calls are affected. Close inspection of affected calls revealed the existence of large SCC (consisting 2.7K nodes) in the call graph. Also about 90% of the time taken by pointer analysis is attributed to the calls in the large SCC. Since the nodes in the SCC are part of the affected set, re-evaluation takes almost same time as from-scratch analysis. The calls in the SCC are also in the recomputed set and hence we do not observe any appreciable difference in the performance of the optimized algorithm relative to its naive counterpart.

The presence of large SCCs limits the performance of call-graph-based algorithms. In contrast, techniques based on the finer-grained answer dependencies perform very well for this program. For instance, the times for incremental evaluation after deletion of one source statement from the vortex benchmark are 0.1%, 15%, and 0.2% using complete support graph, partial support graph, and symbolic support graph respectively. Hence it would be useful to incorporate these specialized techniques into the more general call dependency based algorithm.

XML Validation We investigated incremental validation of XML documents with respect to Document Type Definitions (DTD) [BPV04]. The basic validation problem checks whether a string belongs to a regular language or not.

Table 14 shows the result of applying the naive algorithm for incremental validation of XML documents for different number of elements (first column). The example

Application	Table Space	Called-by Graph Space
Pointer Analysis (vortex)	51.0	13.6
Pointer Analysis (twmc)	18.3	3.5
Matrix Multiplication (chain 200)	4.0	75.0
Longest Common Subsequence (strlen 1000)	168.7	50.1
Minimum Edit Distance (strlen 600)	63.3	21.6
Reaching Definition (diff)	211.0	39.3
XML validation (60K elements)	107.0	13.0

Table 15: Space usage (in MB) of the incremental algorithm

XML documents and DTD describe a library catalog which contains zero or more number of books. Each book contains zero or more number of authors followed by title. Each author has a name, zero or more emails and an address. We generated XML documents having 1K–50K books, with up to 3 authors per book and up to 3 email addresses per author. Each update consists of deletion of one book element from the chain of book elements of the library. The number of affected calls is less than 0.01% of total number of calls. The savings due to incremental evaluation arise from reusing the prior validation of each book element. Since the number of books is large, it results in considerable savings due to incremental evaluation. We encoded the validator using left recursion. Since this results in only one call, we do not see any additional benefits due to the optimized algorithm.

Space Overhead We measured the space needed for keeping the called-by graph for sample applications. Note that although the number of nodes in the called-by graph is bounded by the number of tabled calls, the number of edges can be large. Observe from Table 15 that space needed for the called-by graph is about 30% of the table space for most of the applications. For matrix chain multiplication with chain length n, the number of calls is $O(n^2)$ but the number of called-by graph edges is $O(n^3)$. This contributes to the large size of the called-by graph compared to its table space. For such applications, it will be better not to materialize the graph, as described in Section 7.5.

7.3 Related Work

Techniques discussed in Chapter 2 and 4 cannot be readily applied to arbitrary tabled logic programs, especially those that use aggregation and other Prolog built-ins, or have non-stratified negation. However, when applicable, the fine-grained dependency information (i.e. between answers) used by these algorithms will enable them to outperform the call-graph based algorithms.

Our algorithm is very similar to these [HPMS95, PH96, HPMS00] in terms of using call graphs for change propagation. Notable differences are as follows. Firstly, through the use of post-order numbers, we perform re-evaluation without explicitly computing the SCCs whereas they use a separate SCC maintenance phase. Secondly, we use full-fledged tabled resolution to recompute answers and hence can handle prolog builtins, aggregates and non-stratified negation. In contrast, the other algorithms keep track of the direction of a change (i.e. add or delete) and hence are difficult to generalize for arbitrary programs (e.g. those with findall). Processing of addition of rules was improved in [PH96] by making the non-incremental algorithm SCC-preserving without explicitly computing the SCCs by using a specialized event scheduling strategy. We obtain the same effect by using XSB's local scheduling [FSW96].

7.4 Integration to XSB Prolog Engine

In the previous chapters of this thesis we have described various time and space efficient algorithms for incremental evaluation of definite logic programs. The algorithm presented in this chapter extends the scope of incremental evaluation from definite logic program to arbitrary tabled logic programs by handling negation and Prolog builtins. The generality of the call graph based algorithm makes it the best candidate for incorporating into XSB logic programming system [XSB]. The integration of the call graph based algorithm to XSB and its application to Deductive Spreadsheet [RRW06] raised a number of interesting issues. In this section we discuss some of those issues and their solutions.

7.4.1 Selective Incrementally Maintained Tables

The algorithm described earlier in this chapter maintains a called-by graph that keeps the dependencies among tabled calls and volatile rules. Note that volatile rules correspond to the facts and rules that are dynamic and update to which should update any tabled predicate that are dependent on it.

We expose to the programmers a builtin operator called `incrdynamic/1` using which a programmer can declare certain dynamic predicates to be volatile. This gives the programmer the flexibility to choose certain dynamic extensional predicates to be volatile. Our experience shows that in a typical incremental application the programmer is aware of the volatile predicates. In this case the call dependency structure includes volatile facts and excludes the non-volatile facts. This in turn reduces the space requirement of called-by graph. The call-graph based algorithm creates the calling patterns of subgoals of incrdynamic predicates and stores them in a call trie. In contrast to the tabled calls, the calls for incrdynamic predicates are not associated to any answer tries.

The definition of Called-by Graph in the Definition 23 requires that call dependencies should exist among all tabled subgoals. However, not all tabled predicates are dependent (directly or indirectly) on volatile facts and thus keeping call dependencies between the calls of such tabled predicates is clearly redundant. We thus give the user the flexibility to define tabled predicates whose calls needs to be incrementally maintained. We expose an operator `incr` using which the user can declare certain predicates to be incrementally maintained. We refer to such tabled predicates as *incremental*. Thus called-by graph only contains the dependencies among calls of incremental and volatile predicates.

An interesting question arises when a subgoal of an incremental predicate (say $C1$) calls a subgoal of a non-incremental tabled predicate (say $C2$). If $C2$ does not directly or indirectly depend on a volatile predicate then it is not necessary to be declared as incremental. However, at runtime this information is not available as $C2$ may be a new call. We conservatively assume this to be an error on programmers' part and flag a runtime error with appropriate error message. We make an exception to the above action only if programmer describes the predicate of call $C2$ to be non-incremental. We allow such an option to the user exposing the operator `opaque`.

The predicates defined by opaque (hereafter called opaque predicates) also possesses an interesting property - a subgoal (say $C3$) that is directly or indirectly called by an opaque predicate is also considered to be opaque. This property holds irrespective of whether $C3$ is declared incremental, non-incremental, or opaque. However, $C3$ can be called directly or indirectly from an incremental subgoal ($C1$) without any opaque call in between. In this case, $C3$'s behavior depends on its original incremental property (incremental or non-incremental or opaque). Thus, the incremental behavior of a tabled subgoal depends on the *context* in which it is called.

7.4.2 Deletion of Incrementally Maintained Tables

The algorithms presented in this thesis incrementally maintains tables in response to changes to volatile predicates. The data structures and tables are maintained as long as the session is running. However, in practise a demand can be completely lost on an incrementally maintained table and maintaining such tables is space inefficient. We therefore expose the functionality of deletion and space reclamation of incremental tables.

The design of the functionality of abolishing incremental tables is based on the following two observations: (i) it is difficult for the application programmer to know all the tables that are no longer needed to be incrementally maintained; (ii) the programmer is usually aware of the top-level calls which defines the interface between the application program and tabled engine, and the top-level calls that are no longer required. Based on these observations we provide a builtin `abolish_call(C)` which takes as the argument an incremental call C which is intended to be abolished and tries to abolish C and all calls that are called directly or indirectly by C. Below we define the set of calls that are deleted when a particular incremental call is called for deletion.

Definition 27 *Given a called-by graph $G = (V, E)$, the set $not\_deleted(C)$ defines the set of calls that should not be deleted when $abolish\_call(C)$ is called. The set $not\_deleted(C)$ is the least set satisfying the relation below: $C' \in not\_deleted(C)$*

- *if C is not reachable (reflexive and transitive) from C' in called-by graph*
- *$\exists C'' \in not\_deleted(C)$ and $(C', C'') \in E$.*

```
abolish(C,CallGraph=(V,E))
  init(marked_set)
  init(assumption_set)
  mark(C)
  checkassumptionset()
  deletecalls()

mark(C)
  C.marked = true
  add(marked_set,C)
  if (C.outcount>0)
    add(assumption_set,C)
  $\forall$ C' $\in$ (C',C)$\in$ E
     C'.outcount−−;
     if(!C'.marked)
       mark(C')

checkassumption()
  $\forall$ C $\in$ assumption_set
     if (C.outcount>0)
       delete(marked_set,C)
       if (C.marked)
         unmark(C)

unmark(C)
  C.marked = false
  $\forall$ C' $\in$ (C',C)$\in$ E
     C'.outcount++;
     if(C'.marked)
       unmark(C')

deletecalls()
  $\forall$ C $\in$ marked_set
     if (C.marked)
       abolish(C)
```

Figure 47: Algorithm for abolishing incremental calls

The set of deleted calls (denoted by deleted(C)) due to abolishing incremental call C is the complement of the set not_deleted(C) over all incrementally maintained calls present in called-by graph.

We present a called-by graph based algorithm for determining the set *deleted*(C) in Figure 47. The algorithm has three phases marking, checking assumption, and deletion. The marking phase overapproximates the calls that need to be deleted and subsequent phases prune the overapproximation. A call is marked if its attribute *marked* is true. The attribute *outcount* associated with each call denotes the number of un-marked successor calls in the called-by graph. The *marked* attribute for each call is initialized to false, and *outcount* attribute of a call is initialized to number of its successor calls in the called-by graph.

7.4.3 Integration with Deductive Spreadsheet

The idea of Deductive Spreadsheet [RRW06] is to bring the power of rule-based computing within the familiar paradigm of spreadsheets. The underlying idea is to treat sets as the fundamental data type and rules as specifying relationship among sets and use the spreadsheet metaphor to view the materialized sets. The paradigm extends the functionality of treating a cell for a single value to a set of values. The spreadsheet shows the extensional and intensional values and hence needs to maintain relationship between cell values whenever the user changes the values in some cells. As values of the intensional cells are represented as tables, this problem boils down to incremental maintenance of tables.

In DSS the set of the values present in a cell typically represents answers to a tabled call or facts which unify with the calls to facts. As mentioned earlier, calls to facts can be stored by defining corresponding predicate incrdynamic. In DSS environment when a cell value is changed by the user, it is difficult to determine which element of the set represented by the cell has been changed. This requires us to expose the functionality by which it is possible to deem a particular call of an incrdynamic predicate as affected. In other words, user can specify the initially changed calls.

In spreadsheet environment, changes to the content of one cell can trigger changes to the contents of many other cells. It is hard for the user to determine which cell content has changed in response to the changes made by the user. As incremental tabled evaluation computes the set of changed calls due to changes in the factbase, we expose builtins by which the set of changed calls can be determined and the corresponding cells can be highlighted. It is also possible for the user to view the dependencies among the cells by using the builtins to navigate through the called-by graph.

7.5 Discussion

In this section we discuss possible extensions to the algorithms presented in Section 7.1.

Lazy re-evaluation. The algorithms presented in Section 7.1 refreshes all answer tables such that after each incremental phase the set of answers is sound and complete with respect to the changed program. Certain applications (e.g. ontology management systems), access tables through a graphical user interface, and access some or all of the answers only when required. In such cases, it will be better to re-evaluate a call only on demand. This can be done by keeping a subgoal dependency graph to propagate demand top-down, while keeping the called-by graph to perform re-evaluations bottom-up. Since the invalidation phase takes very little time, it can still be done eagerly. That will ensure that the optimized algorithm can still be used in order to re-evaluate only those calls in the *recomputed* set that are also demanded.

Addition for Definite Logic Programs. The algorithm presented here re-evaluates a call by generating all its answers using program clause resolution. When the direction of the change (i.e. whether it is an addition, deletion or both) is known, we can do better. If the change made is an addition and the program has no negation, we can derive a new program that computes these changes efficiently. The rules of the new program are called "delta rules" and are derived by finite-differencing the original definite program [GMS93, SR03]. This has a potential to significantly improve incremental evaluation times. For example, a single statement addition using delta rules takes on average 8% of from-scratch time for pointer analysis in vortex benchmark whereas it takes 90% of from-scratch time when the affected calls are completely re-evaluated. While it is relatively straightforward to use the "delta rules" program for incrementally processing additions for predicates without negation, light-weight re-evaluation techniques for other kinds of changes and for general logic programs remains an open problem.

Mixed Strategy. In Chapter 5 we described a space efficient technique for storing answer dependencies in the form of symbolic support graph. Symbolic support graph based deletion algorithm is extremely efficient in practice— taking less than 5% of from-scratch time in all the applications we have tested. We can combine these two techniques, keeping call dependencies in general but keeping symbolic support graphs whenever possible to efficiently process deletions.

Non-materialized called-by graph. Although the call dependencies are typically smaller than answer dependencies, and the number of calls is bounded by table space, the called-by graph itself may take more space than the tables (e.g. the matrix chain multiplication example in Section 7.2). It is hence worth exploring whether we can avoid storing the edges of the called-by graph, and instead compute them on the fly. It is relatively easy to derive the called-by relation for a given definite logic program. For instance, from every rule of the form $p :- q_1, q_2, \ldots, q_n$ we can derive "called-by" rules such as `called_by`(q_i, p) :- $q_1, q_2, \ldots, q_{i-1}$. While the computed called-by relation is a space-efficient alternative to storing large called-by graphs, it is not clear whether such rules can be derived for arbitrary logic programs (especially those employing impure constructs such as cuts).

Summary. We presented an incremental evaluation algorithm based on call dependencies that can handle tabled logic programs with negation, aggregation and Prolog builtins. Experiments show that the general algorithm is useful although not as effective as the (more restricted) answer-dependency-based techniques. The algorithm identifies a small set of calls to be re-evaluated and invokes them in a particular order to ensure optimality. The actual re-evaluation itself is performed rather naively, by (effectively) removing all answers from a table to be re-evaluated and using program clause resolution to restore the answer table. More sophisticated techniques that optimize the re-evaluation itself are of significant interest. Our experience with this algorithm shows that programs written for efficient tabled evaluation may not be most suited for efficient incremental evaluation too. Developing a methodology to write efficient incremental programs (analogous to recursion transformations and supplementary tabling for tabled programs) is an important avenue of future research.

Chapter 8

Conclusion

In this thesis we addressed the problem of efficient incremental evaluation of tabled logic programs and its applications. Below we present summary of our major results followed by a brief description of our experience with incremental tabled evaluation, and the description of some of the future work related to incremental computation.

8.1 Summary of Major Results

- **Handling Changes.** In contrast to the algorithms present in various fields of research viz. materialized view maintenance, incremental functional programming we presented efficient algorithms which handle various kinds of incremental changes — batched updates and addition and deletion to rules.

- **Efficiency.** We presented incremental algorithms for recursive programs which record dependency between answers and calls in tabled resolution and confine the propagation of changes locally. This prevents the over-eager propagation of changes shown in existing algorithms for recursive programs. Our local algorithm naturally specializes to optimal algorithms for non-recursive programs.

- **Space-time Tradeoff.** Although the fine-grained dependency graph based incremental algorithms show appreciable time-efficiency for small changes, it requires large amount of space to store dependency graphs in memory. We provide solutions to this problem by either storing partial support graph, or symbolic support graph (whenever applicable) which exploits the common substructures of the dependency graph.

- **Applications.** The effectiveness of incremental tabled resolution has been demonstrated on a wide variety of analysis: pointer analysis, dynamic programming, data-flow analysis, push-down model checking, and XML validation.

8.2 Discussion

We presented various incremental algorithms which showed varied degree of efficiency on different analyses. For example, our symbolic support graph based algorithm (Section 5.2) demonstrated great time and space efficiency when applied to flow-insensitive program analysis, even in presence of updates. The main reason for time efficiency is that the deletion propagation can be confined by identifying acyclic supports. However, for flow-sensitive analysis where the number of supports on average is generally less, our local algorithm performs better than support graph based algorithms. Note that for cases where support graph based algorithm is almost optimal, the local algorithm may be inefficient as it incurs time overhead due to event scheduling. In contrast to the support graph based algorithms which handles pure (no cuts or aggregation builtins) definite logic program, the call graph based algorithm (discussed in Chapter 7) extends the scope of incremental evaluation to general logic programs involving cuts, negation, and aggregation operators, even though it is not as efficient as support graph based algorithms in cases where both of them are applicable.

Another important observation is that efficient from-scratch evaluation of an analysis does not always guarantee efficient incremental evaluation. For example, consider left- and right- transitive closure with a single-source reachability query. Left-recursive transitive closure will produce only one call with a self-loop SCC in the call graph. In this case the dependencies among reachable nodes are captured using answer dependencies (support graph). Whereas right-recursive transitive will produce nontrivial call as well as answer dependencies. Note that the local algorithm restricts propagation of changes using precise SCC decomposition of call dependencies followed by approximate cycle detection (using ordinals) of answer dependencies. Because of this, right-recursive transitive closure is more suitable for incremental evaluation although for single-source query left-recursive transitive closure is known to be efficient for non-incremental evaluation.

In Section 6.6 we showed that the local algorithm specializes to optimal algorithms existed for attribute evaluation and incremental functional programming. However we came across examples in other fields for which straightforward tabled logic encoding of the problem does not guarantee optimal incremental algorithm. One such example is XML validation algorithm where worst-case quadratic time non-incremental algorithm and linear-logarithmic time incremental algorithm exist [BPV04]. Straightforward encoding of XML validation in tabled logic program can only achieve quadratic time non-incremental and incremental algorithms. This is because simple encoding of the problem generates a linear dependency graph whereas the optimal algorithm can generate a logarithmic height dependency graph. It is possible to generate such dependency graph by careful encoding, but it is cumbersome and voids the purpose of declarative encoding.

Our incremental table evaluation algorithms were based on XSB tabled prolog engine. We have several versions of XSB with different incremental algorithms implemented in each of them. These can be downloaded from [IXS06] and available for all sorts of experiments in all sorts of domains.

8.3 Future Work

In this section we discuss some of the avenues of future work in the area of incremental evaluation. We categorize these ideas into two areas: one which further extends the algorithms of incremental tabled evaluation presented in this thesis; and the other which visualizes possible application areas of incremental computation.

8.3.1 Algorithms for Incremental Computation

- **A Combined Answer and Call Dependency Based Algorithm.** In Chapter 7 we described an algorithm for incremental evaluation for arbitrary tabled Prolog programs including those that use Prolog builtins, cuts, aggregation and non-stratified negation. That algorithm maintained a much coarser dependency structure based on calls compared to the algorithms presented in Chapters 4, 5, and 6, where finer grained dependency structures are built based on answer dependencies. The results of Section 7.2 showed that answer-dependency based approaches perform significantly better, but cannot be easily extended beyond pure programs. Moreover, Chapter 6 showed that fine-grained dependencies are needed to achieve better performance by scheduling insertion and deletion operations. In future we would like to devise an algorithm which uses fine-grained local algorithm wherever applicable within the more general setting of arbitrary logic programs.

- **Goal Directed Incremental Computation.** The current techniques for incremental table/view maintenance propagate changes bottom up, thereby updating all tables that are affected by the changes. However, all tables may not require to be updated at all times as the user may be interested only in a subset of all tables. This is common in Deductive Spreadsheet [RRW06] environment, where only a small portion of the view may be seen by the user at any given time. Hence it is important to develop algorithms to incrementally maintain only that part of the view that is demanded by the user.

- **Persistence Data Structures.** In XSB, the tables are currently maintained in memory, and so are the support graphs and call graphs. The incremental algorithms therefore work only within a single XSB session. We hence need to devise ways to efficiently store XSB's tables and support graphs at the end of a session and restore them at a later time. It would also be interesting to investigate whether this information can be restored only to the extent needed by the subsequent session.

- **Incremental Evaluation in Presence of Incomplete Tables.** The incremental algorithms discussed in this thesis assume that before incremental evaluation is started all the tables are completed, i.e. there cannot be further generation of answers due to incomplete operations. In future, we would like to devise an incremental algorithm to remove the above restriction. However, before designing such algorithm we need to carefully define the semantics of incomplete operations in presence of incremental evaluation.

8.3.2 Applications of Incremental Computation

- **Static Analysis.** In this thesis we have shown effectiveness of incremental tabled evaluation to several program analysis problems. This motivates us to incorporate incremental tabled evaluation into static analysis system for popular programming languages like C, C++, and Java. In this effort, currently we are incorporating incremental tabled evaluation into Magellan [Mag06], a static analysis framework for Java. Magellan is integrated into the build process of ECLIPSE, an integrated development environment. We would like to investigate characteristics of incremental computation for static analysis of Java programs. Currently code query systems (e.g. [HVdMdV05]) which use Datalog to query patterns in the code, use non-incremental Datalog systems for query evaluation. It would also be useful to extend such systems with incremental tabled evaluation.

- **Model Checking.**

  - **Incremental CTL/LTL Model Checking.** Traditional model checkers like SPIN, NuSMV do not save any information about state space search during a model checking run. As a result if a transition system is changed after a model checking run, similar state space can be explored again for checking the new system. This scenario is common when model checking is employed in the design process of a system where small updates occur after the initial design is created. We plan to incorporate efficient incremental verification capabilities to these widely used model checkers.

  - **Incremental Relational Coarsest Partition Refinement.** Relational coarsest partition refinement (RCPP) algorithm determines the coarsest *partition* of a set, *stable* with respect to a given relation. The RCPP problem is equivalent to bisimulation equivalence problem which is widely employed in many areas: verification, XML indexing, and so forth. We plan to develop an efficient algorithm which incrementally re-partitions the set in response to small changes to the given relation. The result will have widespread impact on incremental algorithms in different fields.

  - **Incremental Abstraction-Refinement based Software Model Checking.** Traditionally model checking is used for verifying communication protocol and hardware designs. In recent past, model checking combined with program analysis and automated deduction, is used for verification of software systems. For finite as well as infinite state verification, *iterative abstraction refinement* is considered to be the key technique for complex program verification. Incremental algorithms can be very useful in this iterative paradigm where in each iteration the algorithm can re-use the information gathered in the previous iteration. The key questions here are, what information needs to be stored for maximal re-use and what is the relation between the configurations of states of different iterations.

- **Security Analysis**. Recently logic based systems have been used for security and vulnerability analysis [OGA05]. In this framework logical rules describe security policies, vulnerabilities, and interaction between different vulnerabilities that can be exploited by the attacker. The host and network configuration are expressed as facts. Application of the rules on the facts derives whether single- or multi- stage attacks are possible in the network. In this scenario it is very useful to have a model describing all possible ways of performing such attacks. A model that is most commonly known used is *attack graph*. In a logical framework attack graphs can be easily obtained by selecting dependencies between answers of user selected predicates in support and call graphs which are built during query evaluation. As incremental tabled evaluation maintains the support and call graphs, we can easily maintain attack graphs incrementally. This will leverage the functionality of changing the configurations of the network system and quickly view the effect of such changes.

8.4 Final Notes

Optimizations such as incremental computation are generally useful in dynamic environments or evolving systems. Our goal to incorporate efficient incremental capability to rule based systems serves multifaceted purpose. Firstly, deployment of these efficient algorithms in programming language framework vouches for its general purpose use. Secondly, by hiding most of the complexities of generating and maintaining space-efficient dependency graphs and efficient change propagation through it, we preserve the declarative capability of rule based languages. Lastly, incorporation of efficient incremental algorithms extends the use of rule based programming language paradigm to dynamic systems.

Bibliography

[ABH02] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *ACM Conference on Principles of Programming Languages*, volume 37, pages 247–259, New York, NY, USA, 2002. ACM Press.

[ABH03a] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive memoization. Technical report, Carnegie Mellon University, November 2003.

[ABH03b] U. A. Acar, G. E. Blelloch, and R. Harper. Selective memoization. In *ACM Conference on Principles of Programming Languages*, pages 14–25, New York, NY, USA, 2003. ACM Press.

[AG98] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Foundations of Software Engineering*, pages 46–55, 1998.

[AHR$^+$90] 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. Society for Industrial and Applied Mathematics, 1990.

[ALS02] G. Agrawal, J. Li, and Q. Su. Evaluating a demand driven technique for call graph construction. In *Computational Complexity*, volume 2916 of *LNCS*, pages 29–45. Springer-Verlag, 2002.

[And94] L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, pages 585–718. Addison-Wesley, 1986.

[BKPR02] S. Basu, K. N. Kumar, L. R. Pokorny, and C. R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 236–250, 2002.

[BPV04] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[BR90] M. G. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transaction of Software Engineering*, 16(7):723–728, 1990.

[Bra95] S. Brass. Magic sets vs. SLD-resolution. In *International Workshop on Advances in Databases and Information Systems*, pages 101–108, Moscow, 27–30 1995. Phasis.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bur90] M. G. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transaction of Programming Languages and Systems*, 12(3):341–395, 1990.

[CNDE05] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *Computer Aided Verification*, volume 3576 of *LNCS*, pages 449–461, Edinburgh, Scotland, July 2005.

[CR88] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *ACM Conference on Principles of Programming Languages*, pages 274–284. ACM Press, 1988.

[CSW95] W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under well founded semantics. *Journal of Logic Programming*, 1995.

[CW96] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of ACM*, 43(1):20–74, 1996.

[DGS97] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transaction of Programming Languages and Systems*, 19(6):992–1030, 1997.

[dMS03] O. de Moor and G. Sittapalam. Combining memoization and change propagation. Technical report, Oxford University, October 2003.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[DRT81] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *ACM Conference on Principles of Programming Languages*, pages 105–116. ACM Press, 1981.

[DRW96] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *ACM Conference on Programming Language Design and Implementation*, pages 117–126, 1996.

[DS87] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *STOC '87: ACM conference on Theory of computing*, pages 365–372, New York, NY, USA, 1987. ACM Press.

[DT92] G. Dong and R. W. Topor. Incremental evaluation of datalog queries. In *International Conference on Database Theory*, volume 646 of *LNCS*, pages 282–296, 1992.

[FFA00] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In *Static Analysis Symposium*, volume 1824 of *LNCS*, pages 175–198, 2000.

[FFSA98] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *ACM Conference* *on Programming Language Design and Implementation*, pages 85–96. ACM Press, 1998.

[FRD00] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *ACM Conference on Programming Language Design and Implementation*, pages 253–263. ACM Press, 2000.

[FSW96] J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *Symposium on Programming Language Implementation and Logic Programming*, pages 243–258, 1996.

[GG01] H. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *International Conference on Logic Programming*, pages 181–196. Springer, 2001.

[GG04] H. Guo and G. Gupta. Simplifying dynamic programming via tabling. In *Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2004.

[GKM92] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992.

[GL03] S. Guyer and C. Lin. Client-driven pointer analysis. In *Static Analysis Symposium*, pages 214–236, 2003.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD*, pages 157–166, 1993.

[HD92] J.V. Harrison and S.W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In *Workshop on Deductive Databases, JICSLP*, pages 56–65, 1992.

[Hin01] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[HPMS95] M. V. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental analysis of logic programs. In *International Conference on Logic Programming*, MIT Press, pages 797–811, 1995.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental analysis of constraint logic programs. *ACM Transaction of Programming Languages and Systems*, 22(2):187–223, 2000.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Foundations of Software Engineering*, pages 104–115, 1995.

[HT01a] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *ACM Conference on Programming Language Design and Implementation*, pages 24–34. ACM Press, 2001.

[HT01b] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *ACM Conference on Programming Language Design and Implementation*, pages 254–263. ACM Press, 2001.

[HVdMdV05] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder. Codequest: querying source code with datalog. In *Companion to the Proceedings of OOPSLA 2005*, pages 102–103. ACM Press, 2005.

[IXS06] Incremental xsb engine, 2006. Available at `http://www.lmc.cs.sunysb.edu/~dsaha/downloads`.

[Jon90] L. G. Jones. Efficient evaluation of circular attribute grammars. *ACM Transaction of Programming Languages and Systems*, 12(3):429–462, 1990.

[K$\ddot{9}$1] V. Küchenhoff. On the efficient computation of the difference between consecutive database states. In *DOOD'91*, pages 478–502, 1991.

[LH01] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symposium*, pages 279–298. Springer-Verlag, 2001.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[LMSS95] J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD*, pages 340–351, 1995.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM Conference on Programming Language Design and Implementation*, volume 27, pages 235–248. ACM Press, 1992.

[LS03] Y. Liu and S. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *ACM Conference on Principles and Practice of Declarative Programming*, 2003.

[LST98] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transaction of Programming Languages and Systems*, 20(3):546–585, 1998.

[Mag06] Magellan. `http://www.st.informatik.tu-darmstadt.de`, 2006.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *ACM Conference on Principles of Programming Languages*, pages 184–196. ACM Press, 1990.

[MT99] E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *ER Workshops*, pages 62–73, 1999.

[NM00] Ulf Nilsson and Jan Maluszynski. *Logic Programming and Prolog*. Online Publication, 2000.

[NMRW02] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[OGA05] X. Ou, S. Govindavajhala, and A. W. Appel. Mulval: A logic-based network security analyzer. In *USENIX Security Symposium*. Society for Industrial and Applied Mathematics, 2005.

[PAF] PAF. Prolangs analysis framework. Available at `http://www.prolangs.rutgers.edu/public.html`.

[PH96] G. Puebla and M. V. Hermenegildo. Optimized algorithms for incremental analysis of logic programs. In *Static Analysis Symposium*, pages 270–284, 1996.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *TOPLAS*, 4(3):402–454, 1982.

[PK03] D. J. Pearce and P. H. J. Kelly. Online algorithms for topological order and strongly connected components. Technical report, Imperial College, London, 2003.

[PS89] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transaction of Software Engineering*, 15(12):1537–1549, 1989.

[PT89] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *ACM Conference on Principles of Programming Languages*, pages 315–328, New York, NY, USA, 1989. ACM Press.

[Ram00] C. R. Ramakrishnan et al. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification*, number 1855 in LNCS, pages 576–580, 2000.

[Rep82] Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *ACM Conference on Principles of Programming Languages*, pages 169–176, New York, NY, USA, 1982. ACM Press.

[Rep84] T. W. Reps. *Generating language-based environments*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1984.

[Rep93] T. W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases, ILPS*, pages 163–196, 1993.

[RP88] B. G. Ryder and M. C. Paull. Incremental data-flow analysis algorithms. *ACM Transaction of Programming Languages and Systems*, 10(1):1–50, 1988.

[RR93] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation. In *ACM Conference on Principles of Programming Languages*, pages 502–510, New York, NY, USA, 1993. ACM Press.

[RRR96] P. Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A thread in time saves tabling time. In *Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996.

[RRR$^+$97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification*. Springer-Verlag, July 1997.

[RRS$^+$95] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, pages 697–711, 1995.

[RRW06] C. R. Ramakrishnan, I. V. Ramakrishnan, and D. S. Warren. Deductive spreadsheets using tabled logic programming. In *International Conference on Logic Programming*. Springer, 2006. To Appear.

[RSC00] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Workshop on Tabling in Parsing and Deduction*, 2000.

[RSS$^+$97] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for efficiently computing WFS. In *Logic Programming and Non-monotonic Reasoning*, pages 431–441, 1997.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transaction of Programming Languages and Systems*, 5(3):449–477, 1983.

[SBS95] G. Swamy, R. K. Brayton, and V. Singhal. Incremental methods for FSM traversal. In *Intl. Conference on Computer Design (ICCD).* IEEE Computer Society, 1995.

[SdS99] R.R. Seljee and H.C.M. de Swart. Three types of redundancy in integrity checking; an optimal solution. *Journal of Data and Knowledge Engineering*, 30:135–151, 1999.

[SFA00] Z. Su, M. Fahndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *ACM Conference on Principles of Programming Languages*, pages 81–95. ACM Press, 2000.

[Sha81] M. Sharir. A strong connectivity algorithm and its application in data flow analysis. *Computer and Mathematics with Applications*, 7(1):67–72, 1981.

[SJ96] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *VLDB - Very Large Databases*, pages 75–86, 1996.

[SR03] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 389–406, 2003.

[SR05] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *ACM Conference on Principles and Practice of Declarative Programming.* ACM Press, 2005.

[SR06] D. Saha and C. R. Ramakrishnan. A local algorithm for incremental evaluation of logic programs. In *International Conference on Logic*

Programming, volume 4079 of *LNCS*, pages 56–71, Seattle, USA, Aug 2006. Springer. http://www.lmc.cs.sunysb.edu/ dsaha/local/.

[SS94] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Computer Aided Verification*, volume 818 of *LNCS*, pages 351–363, 1994.

[SSS04] B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004. Available at http://www.cs.sunysb.edu/˜stoller/WITS2004.html.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Conference on Principles of Programming Languages*, pages 32–41. ACM Press, 1996.

[Swa96] G. Swamy. *Incremental Methods for Formal Verification and Logic Synthesis*. PhD thesis, University of California at Berkeley, 1996.

[Swi99] Terrance Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.

[TS86] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98, 1986.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-base Systems, Volume II*. Computer Science Press, 1989.

[UO92] T. Urpí and A. Olivé. A method for change computation in deductive databases. In *VLDB - Very Large Databases*, pages 225–237, 1992.

[WJ88] J. A. Walz and G. F. Johnson. Incremental evaluation for a general class of circular attribute grammars. In *ACM Conference on Programming Language Design and Implementation*, pages 209–221, New York, NY, USA, 1988. ACM Press.

[WL04] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM Conference on*

Programming Language Design and Implementation, pages 131–144. ACM Press, 2004.

[XSB] XSB. The XSB logic programming system. Available at `http://xsb.sourceforge.net`.

[YK00] 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.

[YRL99] J. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *International Conference on Software Engineering*, pages 442–451, 1999.

[YRLS97] J. Yur, B. G. Ryder, W. Landi, and P. Stocks. Incremental analysis of side effects for C software system. In *International Conference on Software Engineering*, pages 422–432, 1997.

[ZSYY01] N. Zhou, Y. Shen, L. Yuan, and J. You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.