Incremental Analysis of Logic Programs

Manuel Hermenegildo, Germán Puebla
Facultad de Informática, Universidad Politécnica de Madrid
28660-Boadilla del Monte, Madrid, Spain
{herme,german}@fi.upm.es

Kimbal Marriott
Dept. of Computer Science, Monash University
Clayton 3168, Australia
marriott@cs.monash.oz.au

Peter J. Stuckey
Dept. of Computer Science, The University of Melbourne
Parkville 3052, Australia
pjs@cs.mu.oz.au

Abstract

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

1 Introduction

Global program analysis is becoming a practical tool in logic program compilation in which information about calls, answers, and substitutions at different program points is computed statically [11, 17, 14, 18, 3]. The underlying theory, formalized in terms of abstract interpretation [6], and the related implementation techniques are well understood for several general types of analysis and, in particular, for top-down analysis of Prolog [8, 2, 14, 7, 12, 4]. Several generic analysis engines, such as PLAI [14, 13] and GAIA [4], facilitate construction of such top-down analyzers. These generic engines have the description domain and functions on this domain as parameters. Different domains give analyzers which provide different types of information and degrees of accuracy. The core of each generic engine is an algorithm...
for efficient fixpoint computation [13, 14]. Efficiency is obtained by keeping track of which parts of a program must be reexamined when a success pattern is updated. Current generic analysis engines are non-incremental – the entire program is read, analyzed and the analysis results written out.

Despite the obvious progress made in global program analysis, most LP and CLP compilers still perform only local analysis (the &-Prolog [9], Aquarius [17], and Andorra-I [18] systems are notable exceptions). We believe that an important contributing factor to this is the simple, non-incremental model supported by global analysis systems, which is unsatisfactory for at least four reasons. The first reason is that optimizations are often source-to-source transformations, and so optimization consists of an analyze, perform optimization then reanalyze cycle. This is inefficient if the analysis starts from scratch each time. Such analyze-optimize cycles may occur for example when program optimization and multivariant specialization are combined [20, 15]. This is used, for instance, in program parallelization, where an initial analysis is used to introduce specialized predicate definitions with run-time parallelization tests, and then these new definitions are analyzed and redundant tests removed. This is also the case in optimization of CLP(R) in which specialized predicate definitions are reordered and then reanalyzed. The second reason is that incremental analysis supports incremental runtime compilation during the test-debug cycle. Again, for efficiency only those parts of the program which are affected by the changes should be reanalyzed. Incremental compilation is important in the context of logic programs as traditional environments have been interpretive, allowing the rapid generation of prototypes. The third reason is to handle correctly and accurately the optimization of programs in which clauses are asserted or retracted at runtime. The fourth reason is to support incremental compilation of programs broken into modules.

In this paper we describe how the fixpoint algorithm in the generic analysis engines can be extended to support incremental analysis. Surprisingly, there has been little research into incremental analysis for logic programs. Several researchers have looked at compositional analysis of modules in logic programs [5]. There has been much research into incremental analysis for other programming paradigms (see for example the bibliography of Ramalingam and Reps [16]). However, to our knowledge this is the first paper to identify the different types of incremental change which are useful in logic program analysis and to give practical algorithms which handle these types of incremental change. Another contribution of the paper is a simple formalization of the non-incremental fixpoint algorithms used in generic analysis engines.

2 Non-Incremental Analysis Algorithm

We start by providing a stylized formalization of the non-incremental fixpoint algorithms used in a good number of the existing generic analysis engines. The purpose of our presentation is not so much to present a practical algorithm for performing program analysis but rather to capture the core behavior of the standard algorithms and then use this stylized algorithm to present our proposals regarding how to make them incremental.

The aim of the kind of (goal oriented) program analysis performed by the above mentioned engines is, for a particular description domain, to take a program and a set of initial calling patterns and to annotate the program with information about the current environment at each program point whenever that point is reached when executing calls described by the calling patterns.
To illustrate this, we now develop an example. The description domain, as in all of our examples, will be the definite Boolean functions [1]. The key idea in this description is to use implication to capture groundness dependencies. The reading of the function \( x \rightarrow y \) is: “if the program variable \( x \) is (becomes) ground, so is (does) program variable \( y \).” For example, the best description of the constraint \( f(X, Y) = f(a, g(U, V)) \) is \( X \land Y \). Now consider the program for appending lists:

\[
\text{app}(X, Y, Z) \leftarrow X = [], Y = Z.
\text{app}(X, Y, Z) \leftarrow X = [U|V], Z = [U|W], \text{app}(V, Y, W).
\]

Assume that we are interested in analyzing the program for the call \( \text{app}(X, Y, Z) \) with initial calling pattern \( Y \) indicating that we wish to analyze it for any call to \( \text{app} \) with the second argument definitely ground. In essence the analyzer must produce the program analysis graph given in Figure 1, which can be viewed as a finite representation (through a “widening”) of the set of AND-OR trees explored by the concrete execution [2]. The graph has two sorts of nodes: those belonging to rules (also called “AND-nodes”) and those belonging to atoms (also called “OR-nodes”). For example, the atom node \( (\text{app}(X, Y, Z) : Y \leftrightarrow Y \land X \leftrightarrow Z) \) indicates that the calling pattern \( Y \) for the atom \( \text{append}(X, Y, Z) \) has answer pattern \( Y \land X \leftrightarrow (U \land V) \). This answer pattern depends on the two rules defining \( \text{app} \) which are attached by arcs to the node. These rules are annotated by descriptions at each program point of the constraint store when the rule is executed from the calling pattern of the node connected to the rules. The program points are entry to the rule, the point between each two literals, and return from the call. Atoms in the rule body have arcs to OR-nodes with the corresponding calling pattern. If such a node is already in the tree it becomes a recursive call. Thus, the analysis graph in Figure 1 has a recursive call to the calling pattern \( \text{app}(X, Y, Z) : Y \).

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

- \( A\text{project}(CP, L) \) which performs the abstract restriction of a calling pattern \( CP \) to the variables in the literal \( L \);
- \( A\text{add}(C, CP) \) which performs the abstract operation of conjoining the actual constraint \( C \) with the description \( CP \);

Figure 1: Example Program Analysis Graph
• Acombine\((CP_1, CP_2)\) which performs the abstract conjunction of two descriptions;
• Alub\((CP_1, CP_2)\) which performs the abstract disjunction of two descriptions.

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

We now give an algorithm which computes the least analysis graph. We first introduce some notation. \(CP_i\), possibly subscripted, stands for a calling pattern (in the abstract domain). \(AP_i\), possibly subscripted, stands for an answer pattern (in the abstract domain). Each literal in the program is subscripted with an identifier or pair of identifiers. \(A : CP\) stands for an atom (unsubscripted) together with a calling pattern. \(A_k : CP\) or \(A_{k,i} : CP\) stands for subscripted literals together with a calling pattern. Rules are assumed to be normalized and each rule for a predicate \(p\) has identical sets of variables \(p(x_1, \ldots, x_n)\) in the head atom. Call this the base form of \(p\). Rules in the program are written with a unique subscript attached to the head atom (the rule number), and dual subscript (rule number, body position) attached to each body atom (and constraint redundantly) e.g. \(H_k \leftarrow B_{k,1}, \ldots, B_{k,n_b}\) where \(B_{k,i}\) is a subscripted atom or constraint. The rule may also be referred to as rule \(k\), the subscript of the head atom. E.g. the append program is written:

\[
\begin{align*}
\text{app}(X, Y, Z)_1 & : = X=[\_1, Y=Z_2. \\
\text{app}(X, Y, Z)_2 & : = X=[[U|V]_{2,1}, Z=[U|W]_{2,2}, \text{app}(V, Y, W)_{2,3}.
\end{align*}
\]

The program analysis graph is implicitly represented in the algorithm by means of two data structures, the answer table and the dependency arc table. The answer table contains entries of the form \(A : CP \rightarrow AP\). \(A\) is always a base form. This represents a node in the analysis graph of the form \((A : CP \rightarrow AP)\). It is interpreted as the answer pattern for calls of the form \(CP\) to \(A\) is \(AP\). A dependency arc is of the form \(H_k : CP_0 \rightarrow AP_0 \rightarrow B_{k,i} : CP_i\). This is interpreted as that if the rule with \(H_k\) as head is called with calling pattern \(CP_0\) then this causes literal \(B_{k,i}\) to be called with calling pattern \(CP_i\). The remaining part \(CP_i\) is the program annotation just before \(B_{k,i}\) is reached and contains information about all variables in rule \(k\). \(CP_i\) is not really necessary, but is included for efficiency. Dependency arcs represent the arcs in the program analysis graph from atoms in a rule body to an atom node. E.g. the program analysis graph in Figure 1 is represented by

answer table:

\[
\begin{align*}
\text{app}(X, Y, Z) : Y & \rightarrow Y \land (X \leftrightarrow Z)
\end{align*}
\]

dependency arc table:

\[
\begin{align*}
\text{app}_{2,3}(X, Y, Z) : Y & \rightarrow [Y \land X \leftrightarrow (U \land V) \land Z \leftrightarrow (U \land W)] \\
\text{app}_{2,3}(V, Y, W) : Y & \rightarrow (V \land W)
\end{align*}
\]

Intuitively, the analysis algorithm is just a graph traversal algorithm which places entries in the answer table and dependency arc table as new nodes and arcs in the program analysis graph are encountered. To capture the different graph traversal strategies used in different fixpoint algorithms, we use a priority queue. Thus, the third, and final structure used in our algorithms is a prioritized event queue. Events are of three forms. The first, updated\((A : CP)\), indicates that the answer pattern to atom \(A\) with calling pattern \(CP\) has been changed. The second, arc\((R)\), indicates that the rule referred to in \(R\) needs to be (re)computed from the position indicated. The third, newcall\((A : CP)\), indicates that a new call has been encountered. The priority mechanism for the queue is left as a parameter of the algorithm.
analyze(S)
  foreach A : CP ∈ S
    add_event(newcall(A : CP))
  main_loop()

main_loop()
  while E := next_event()
    if (E == newcall(A : CP))
      new_calling_pattern(A : CP)
    elseif (E == updated(A : CP))
      add_dependent_rules(^4 : CP)
    elseif (E == arc(R))
      process_arc(P)
    endwhile
  remove_useless_calls(S)

new_calling_pattern(A : CP)
  foreach rule A_k ← B_{k,1}, ..., B_{k,n_k}
    CP_k := Aproject(CP, B_{k,1})
    add_event(arc(A_k : CP => [CP] B_{k,1} : CP_1))
    AP_k := initial_guess(A : CP)
    if (AP_k ⊥)
      add_event(updated(A : CP))
    add A : CP → AP to answer_table
  endforeach

add_dependent_rules(A : CP)
  foreach arc of the form H_k : CP_0 ⇒ [CP] B_{k,1} : CP_2
    in graph
    where there exists renaming σ
    s.t. A : CP = (B_{k,1} : CP_2)σ
    add_event(arc(H_k : CP_0 => [CP] B_{k,1} : CP_2))

process.arc(H_k : CP_0 ⇒ [CP] B_{k,1} : CP_2)
  if (B_{k,1} is not a constraint)
    add H_k : CP_0 ⇒ [CP] B_{k,1} : CP_2 to dependency.arc_table
    AP_0 := get_answer(B_{k,1} : CP_2)
    CP_1 := Acombine(CP_1, AP_0)
  if (CP_3 ⊥ and i ⊥ n_k)
    CP_4 := Aproject(CP_3, B_{k,1} : CP_2)
    add.event( arc(H_k : CP_0 ⇒ [CP_3] B_{k,1} : CP_4))
  elseif (CP_3 ⊥ and i == n_k)
    AP_1 := Aproject(CP_3, H_k)
    insert_answer_info(H : CP_0 ⇒ AP_1)

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

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

insert_answer_info(H : CP ⇒ AP)
  if (L is a constraint)
    return Add(L, CP)
  else return lookup_answer(L, CP)

Figure 2: Non-incremental analysis algorithm

The non-incremental analysis algorithm is given in Figure 2. Apart from the parametric description domain dependent functions, the algorithm has several other undefined functions. The functions add_event and next_event respectively add an event to the priority queue and return (and delete) the event of highest priority. When an event being added to the priority queue is already in the priority queue, a single event with the maximum of the priorities is kept in the queue. When an arc is added to the dependency arc table, it overwrites any other arcs of the form H_k : CP ⇒ [CP] B_{k,1} : CP_2 in the table and in the priority queue. The function initial_guess returns an initial guess for the answer to a new calling pattern. The default value is ⊥ but if the calling pattern is more general than an already computed call then its current value may be returned. The procedure remove_useless_calls traverses the dependency graph given by the dependency arcs from the initial calls S and marks those entries in the dependency arc and answer table which are reachable. The remainder are removed.

Space limitations prevent us from providing examples for this algorithm or for the incremental versions proposed – they can be found in [10]. It is also important to remember the purpose of this algorithm. It is not intended as a practical algorithm for computing a program analysis graph, as the overhead of event handling is too
incremental_addition(R)
    foreach rule A_k ← B_{k,1},...,B_{k,q_k} ∈ R
        foreach entry A : CP → AP in the answer_table
            CP_i := A.project(CP, B_{k,1})
            add.event(arc(A_k : CP ⇒ [CP] B_{k,1} : CP_1))
    main_loop()

Figure 3: Incremental Addition Algorithm

high. Rather it is intended to capture the behavior of several algorithms which are
used for computing the program analysis graph. Different algorithms correspond
to different event processing strategies. In addition, practical algorithms incorporate
a series of optimizations. The strategy used in PLAI is presented in Section 6.

Theorem 1 For a program P and call patterns S, the non-incremental analysis
algorithm returns an answer table and dependency arc table which represents the
least program analysis graph of P and S.

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

Corollary 1 The result of the non-incremental analysis algorithm does not depend
on the strategy used to prioritize events.

3 Incremental Addition

Since the answer and dependency arc tables are incrementally extended in the non-
incremental analysis of a program, incremental addition of new rules and new calling
patterns does not place extra demands on the analysis algorithm. If the analysis is
required for new calling patterns, then the routine analyze(S), where S is the set
of new calling patterns may be repeatedly called.

The new routine for analysis of programs in which rules are added incrementally
is given in Figure 3. The routine takes as input the set of new rules R. If these
define a calling pattern of interest, then requests to process the rule are placed
on the priority queue. Subsequent processing is exactly as for the non-incremental
case.

Correctness of the incremental addition algorithm follows from correctness of
the original non-incremental algorithm. Execution of the incremental addition al­
gorithm corresponds to executing the non-incremental algorithm with all rules but
with the new rules having the lowest priority for processing.

Theorem 2 If the rules in a program are analyzed incrementally with the incremen­
tal addition algorithm, the same answer and dependency arc tables will be obtained
as when all rules are analyzed at once by the non-incremental algorithm.

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

In this section we consider deletion of clauses from an already analyzed program and how to incrementally update the analysis information. The first thing to note is that we need not change the analysis results at all. The current approximation is trivially guaranteed to be correct. This approach is obviously inaccurate but simple. More accuracy can be obtained by applying a narrowing like strategy. The current approximation in the answer table is greater than the least fixpoint of the semantic equations. Thus, applying the analysis engine as usual except taking the greatest lower bound of (the lub of) the new answers with the old rather than the least upper bound is guaranteed to produce a safe result. The disadvantage of this method is its inaccuracy. Starting the analysis from scratch will often give a more accurate result. We now give two algorithms which are incremental yet are as accurate as the non-incremental analysis.

“Top-Down” Deletion Algorithm

The first method for incremental analysis of programs after deletion is to remove all information in the answer and dependency arc tables which depends on the rules which have been deleted and then to restart the analysis. Not only will removal of rules change the answers in the answer table, it will also mean that subsequent calling patterns may change. Thus we must also remove entries for calling patterns which may no longer exist.

Information in the dependency arc table allows us to find these no longer valid call and answer patterns. Consider the dependency arcs of the analyzed program. Let $D$ be the set of deleted rules. Let $H$ be the set of atoms which occur as the head of the deleted rules.

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

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

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

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


bottom_up_delete($S$, $D$)

$H := \emptyset$

**foreach** rule $A_k \leftarrow B_k, \ldots, B_{k,n_k} \in D$

**foreach** $A : CP \mapsto AP$ in table

$H := H \cup (A : CP)$

**while** $H$ is not empty

**let** $B : CP \in H$ be such that $B$ is of minimum predicate SCC level

$T :=$ calling patterns in dependency graph for predicates in same predicate SCC as $B$

**foreach** $A : CP \in T$

delete each arc $A_k : CP \Rightarrow \{CP_i\} B_{k,i} : CP_2$ from dependency_arc_table

**foreach** $A : CP \in external_calls(T, S)$

move entry $A : CP \Rightarrow AP$ from answer_table to old_table

**foreach** arc $B_k : CP_0 \Rightarrow \{CP_i\} B_{k,i} : CP_2$ in old_table

**where** there exists renaming $\sigma$ s.t. $(A : CP) = (B_{k,i} : CP_2)\sigma$

move $B_k : CP_0 \Rightarrow \{CP_i\} B_{k,i} : CP_2$ to dependency_arc_table

add_event(newcall($A : CP$))

main_loop()

**foreach** $A : CP \in external_calls(T, S)$

if answer pattern in old_table and answer_table agree

**foreach** arc $B_k : CP_0 \Rightarrow \{CP_i\} B_{k,i} : CP_2$ in old_table

**where** there exists renaming $\sigma$ s.t. $(A : CP) = (B_{k,i} : CP_2)\sigma$

move $B_k : CP_0 \Rightarrow \{CP_i\} B_{k,i} : CP_2$ to dependency_arc_table

else

$H := H \cup (B : CP_0)$

$H := H \cup T$

delete old_table

**external_calls(T, S)**

$U := \emptyset$

**foreach** $A : CP \in T$

**where** exists arc $B_k : CP_0 \Rightarrow \{CP_i\} B_{k,i} : CP_2$

and $B : CP_0 \notin T$

and there exists renaming $\sigma$ s.t. $(A : CP) = (B_{k,i} : CP_2)\sigma$

%\% this means there is an external call

$U := U \cup (A : CP)$

return $U \cup (T \cap S)$

Figure 5: Bottom-up Incremental Deletion Algorithm

of a deleted rule. Let $up(H)$ be the set of atom/calling pattern pairs whose answers depended on an atom in $H$. That is, all of the atom/calling pattern pairs that can reach a pair $p : CP$ in the dependency graph where $p \in H$. After entries concerning these now invalid atom/calling pattern pairs are deleted, the usual non-incremental analysis is performed. The routine for top-down rule deletion is given in Figure 4. It is called with the set of deleted rules $D$ and a set of initial calls $S$.

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

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

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

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

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

**“Bottom-up” Deletion Algorithm**

The last theorem shows that the top-down deletion algorithm is never worse than starting the analysis from scratch. However, in practice it is unlikely to be that much better, as on average deleting a single rule will mean that half of the dependency arcs and answers are deleted in the first phase of the algorithm. The reason is that the top-down algorithm is very pessimistic – deleting everything unless it is sure that it will be useful. For this reason we now consider a more optimistic algorithm. The algorithm assumes that calling patterns to changed predicate definitions are still likely to be useful. In the worst case it may spend a large amount of time reanalyzing calling patterns that end up being useless. But in the best case we do not need to reexamine large parts of the program above changes when no actual effect is felt. The algorithm proceeds by computing new answers for calls to the lowest strongly connected component (SCC) in the program call graph which is affected by the rule deletion, and then moving upwards to higher SCCs. At each stage the algorithm recomputes or verifies the current answers to the calls to the SCC without considering dependency arcs from SCC in higher levels. This is possible because if the answer changes, the arc events they would generate are computed anyway. If the answers are unchanged then the algorithm stops, otherwise it examines the SCCs which depend on the changed answers (using the dependency arcs). For obvious reasons we call the algorithm Bottom-Up Deletion. It is shown in Figure 5.

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

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

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

### 5 Arbitrary Change

Given the above algorithms for addition and deletion of clauses we can handle any possible change of a set of clauses by first deleting the original and then adding the revised version. This is inefficient since the revision may not involve very far
Local Change

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

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

Correctness of the local change algorithm essentially follows from correctness of the bottom-up deletion algorithm.

Let \( A \leftarrow B \) and \( A \leftarrow B' \) be two rules. They are local variants with respect to the calls \( S \) and program \( P \) if for each call pattern in \( S \) \( P \cup \{ A \leftarrow B \} \) has the same answer patterns as \( P \cup \{ A \leftarrow B' \} \).

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

The cost of performing the actual analysis incrementally can be no worse than performing the analysis all at once. Let \( C_{local}(F, P, R, R', S) \) be the worst case number of calls to the parametric functions \( F \) when analyzing the program \( P \) with rule \( R \) changed to \( R' \) for all possible priority strategies with the local change algorithm after already analyzing the program \( P \) for call patterns \( S \).
**Theorem 8** Let $P$ be a program analyzed for the initial call patterns $S$. Let $R$ be a rule in $P$ which in the analysis is called with call patterns $S'$ and let $R'$ be a local variant of $R$ with respect to $S'$ and $P \setminus \{R\}$. For any parametric functions $F$, 

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

### 6 Experimental Results

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Cl</th>
<th>Strd</th>
<th>Incr</th>
<th>Lcl</th>
<th>NI_cl</th>
<th>I SD</th>
<th>NI SD</th>
<th>I SU</th>
</tr>
</thead>
<tbody>
<tr>
<td>aikl</td>
<td>12</td>
<td>3746</td>
<td>3859</td>
<td>4050</td>
<td>6153</td>
<td>1.05</td>
<td>1.59</td>
<td>1.52</td>
</tr>
<tr>
<td>ann</td>
<td>170</td>
<td>7882</td>
<td>7746</td>
<td>50830</td>
<td>643609</td>
<td>6.56</td>
<td>83.09</td>
<td>12.66</td>
</tr>
<tr>
<td>bid</td>
<td>50</td>
<td>916</td>
<td>962</td>
<td>4436</td>
<td>16937</td>
<td>4.61</td>
<td>17.61</td>
<td>3.82</td>
</tr>
<tr>
<td>boyer</td>
<td>133</td>
<td>3625</td>
<td>2808</td>
<td>20853</td>
<td>271043</td>
<td>7.43</td>
<td>96.53</td>
<td>13.00</td>
</tr>
<tr>
<td>browse</td>
<td>29</td>
<td>495</td>
<td>516</td>
<td>1703</td>
<td>9560</td>
<td>3.30</td>
<td>18.53</td>
<td>5.61</td>
</tr>
<tr>
<td>deriv</td>
<td>10</td>
<td>766</td>
<td>492</td>
<td>3199</td>
<td>1736</td>
<td>6.50</td>
<td>3.53</td>
<td>0.34</td>
</tr>
<tr>
<td>fib</td>
<td>3</td>
<td>46</td>
<td>52</td>
<td>53</td>
<td>89</td>
<td>1.02</td>
<td>1.71</td>
<td>1.68</td>
</tr>
<tr>
<td>grammar</td>
<td>15</td>
<td>155</td>
<td>175</td>
<td>496</td>
<td>1193</td>
<td>2.83</td>
<td>6.82</td>
<td>2.41</td>
</tr>
<tr>
<td>hanoiapp</td>
<td>4</td>
<td>613</td>
<td>629</td>
<td>1036</td>
<td>1419</td>
<td>1.65</td>
<td>2.26</td>
<td>1.37</td>
</tr>
<tr>
<td>mmatrix</td>
<td>6</td>
<td>306</td>
<td>329</td>
<td>733</td>
<td>1003</td>
<td>2.23</td>
<td>3.05</td>
<td>1.37</td>
</tr>
<tr>
<td>occur</td>
<td>8</td>
<td>342</td>
<td>335</td>
<td>396</td>
<td>523</td>
<td>1.18</td>
<td>1.56</td>
<td>1.32</td>
</tr>
<tr>
<td>peephole</td>
<td>134</td>
<td>7256</td>
<td>6605</td>
<td>65546</td>
<td>567572</td>
<td>9.92</td>
<td>85.93</td>
<td>8.66</td>
</tr>
<tr>
<td>progeom</td>
<td>18</td>
<td>240</td>
<td>256</td>
<td>406</td>
<td>1066</td>
<td>1.59</td>
<td>4.16</td>
<td>2.63</td>
</tr>
<tr>
<td>qplan</td>
<td>148</td>
<td>1973</td>
<td>2036</td>
<td>41912</td>
<td>154500</td>
<td>20.59</td>
<td>75.88</td>
<td>3.69</td>
</tr>
<tr>
<td>qsortapp</td>
<td>7</td>
<td>346</td>
<td>372</td>
<td>646</td>
<td>1169</td>
<td>1.74</td>
<td>3.14</td>
<td>1.81</td>
</tr>
<tr>
<td>query</td>
<td>52</td>
<td>176</td>
<td>185</td>
<td>2079</td>
<td>4626</td>
<td>11.24</td>
<td>25.01</td>
<td>2.23</td>
</tr>
<tr>
<td>rdtok</td>
<td>54</td>
<td>2032</td>
<td>2793</td>
<td>23606</td>
<td>39193</td>
<td>8.45</td>
<td>14.03</td>
<td>1.66</td>
</tr>
<tr>
<td>read</td>
<td>88</td>
<td>36416</td>
<td>47899</td>
<td>187512</td>
<td>1044112</td>
<td>3.91</td>
<td>21.80</td>
<td>5.57</td>
</tr>
<tr>
<td>serialize</td>
<td>12</td>
<td>569</td>
<td>733</td>
<td>1596</td>
<td>3556</td>
<td>2.18</td>
<td>4.85</td>
<td>2.23</td>
</tr>
<tr>
<td>tak</td>
<td>2</td>
<td>109</td>
<td>123</td>
<td>127</td>
<td>166</td>
<td>1.03</td>
<td>1.35</td>
<td>1.31</td>
</tr>
<tr>
<td>warplan</td>
<td>101</td>
<td>4852</td>
<td>3966</td>
<td>45502</td>
<td>122562</td>
<td>11.56</td>
<td>30.90</td>
<td>2.69</td>
</tr>
<tr>
<td>witt</td>
<td>160</td>
<td>2543</td>
<td>2526</td>
<td>21143</td>
<td>65109</td>
<td>8.37</td>
<td>25.78</td>
<td>3.08</td>
</tr>
<tr>
<td>zebra</td>
<td>18</td>
<td>4068</td>
<td>4146</td>
<td>9312</td>
<td>45340</td>
<td>2.25</td>
<td>10.94</td>
<td>4.87</td>
</tr>
</tbody>
</table>

**Table 1: Incremental vs. Non-incremental Addition**

We have conducted a number of experiments using the PLAI generic abstract interpretation system in order to assess the practicality of the techniques proposed in the previous sections. The original fixpoint used in PLAI is quite close to the stylized non-incremental algorithm that we have used as a starting point, and uses the concrete strategy of always performing `newcall` events first, processing non-recursive rules before recursive rules, and finishing processing a rule before starting another. Prior to the invocation of the fixpoint algorithm a step is performed in which the set of predicates in the program is split into the SCCs based on the call graph of the program found using Tarjan’s algorithm [19]. This information is used among other things to determine which predicates and clauses of a predicate are recursive.

PLAI also incorporates some additional optimizations such as dealing directly with non-normalized programs and filtering out non eligible clauses using concrete unification (or constraint solving) when possible. Also, instead of explicitly storing the annotation and call-pattern in the dependency arcs, it is recomputed from the head of the rule. In one way, however, PLAI is somewhat simpler than the proposed algorithm: in order to simplify the implementation, the original fixpoint algorithm does not keep track of dependencies at the level of literals, but rather, in a coarser way, at the level of clauses.
respect to analyzing the whole program at once (i.e., with respect to \texttt{Incr}). \texttt{I SU} (NLcl / Lcl) shows the speedup obtained by incremental analysis. The results are quite encouraging: in the worst case studied of compiling clause by clause the slowdown with respect to analyzing the entire file in one block is on the average of only a factor of 5.44. Doing the same thing with the non-incremental analysis implies an average slowdown of 33.53 (i.e., over six times worse than \texttt{Incr}).

Table 3: Local Change

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Cl</th>
<th>1$^{st}$</th>
<th>Inc</th>
<th>Scr</th>
<th>Scr$\backslash$Inc</th>
<th>Scr(1^{st}+\text{Inc})</th>
</tr>
</thead>
<tbody>
<tr>
<td>aail</td>
<td>1</td>
<td>3859</td>
<td>642</td>
<td>4096</td>
<td>6.30</td>
<td>0.90</td>
</tr>
<tr>
<td>ann</td>
<td>11</td>
<td>7746</td>
<td>3526</td>
<td>10256</td>
<td>2.91</td>
<td>0.91</td>
</tr>
<tr>
<td>bid</td>
<td>6</td>
<td>962</td>
<td>533</td>
<td>1395</td>
<td>2.62</td>
<td>0.93</td>
</tr>
<tr>
<td>boyer</td>
<td>2</td>
<td>2808</td>
<td>909</td>
<td>4725</td>
<td>5.20</td>
<td>1.27</td>
</tr>
<tr>
<td>browse</td>
<td>4</td>
<td>516</td>
<td>545</td>
<td>1043</td>
<td>1.91</td>
<td>0.98</td>
</tr>
<tr>
<td>deriv</td>
<td>4</td>
<td>492</td>
<td>1018</td>
<td>1239</td>
<td>1.22</td>
<td>0.82</td>
</tr>
<tr>
<td>hanoiapp</td>
<td>1</td>
<td>629</td>
<td>319</td>
<td>778</td>
<td>2.44</td>
<td>0.82</td>
</tr>
<tr>
<td>mmatrix</td>
<td>2</td>
<td>329</td>
<td>416</td>
<td>669</td>
<td>1.61</td>
<td>0.90</td>
</tr>
<tr>
<td>occur</td>
<td>2</td>
<td>335</td>
<td>473</td>
<td>752</td>
<td>1.59</td>
<td>0.93</td>
</tr>
<tr>
<td>peephole</td>
<td>2</td>
<td>6605</td>
<td>1082</td>
<td>7546</td>
<td>6.92</td>
<td>0.98</td>
</tr>
<tr>
<td>progeom</td>
<td>1</td>
<td>256</td>
<td>52</td>
<td>276</td>
<td>5.31</td>
<td>0.90</td>
</tr>
<tr>
<td>qplan</td>
<td>2</td>
<td>2036</td>
<td>273</td>
<td>2312</td>
<td>8.47</td>
<td>1.00</td>
</tr>
<tr>
<td>query</td>
<td>2</td>
<td>185</td>
<td>109</td>
<td>242</td>
<td>2.22</td>
<td>0.82</td>
</tr>
<tr>
<td>read</td>
<td>1</td>
<td>47899</td>
<td>52</td>
<td>48618</td>
<td>934.96</td>
<td>1.01</td>
</tr>
<tr>
<td>serialize</td>
<td>1</td>
<td>733</td>
<td>58</td>
<td>803</td>
<td>13.84</td>
<td>1.02</td>
</tr>
<tr>
<td>warplan</td>
<td>8</td>
<td>3966</td>
<td>4086</td>
<td>22589</td>
<td>5.53</td>
<td>2.81</td>
</tr>
<tr>
<td>zebra</td>
<td>1</td>
<td>4146</td>
<td>3019</td>
<td>4729</td>
<td>1.57</td>
<td>0.66</td>
</tr>
</tbody>
</table>

Average: 6.55 1.11

In order to test the relative performance of incremental and non incremental analysis in the context of deletion, we timed the analysis of the same benchmarks but deleting the clauses one by one. Starting from an already analyzed file, the last clause was deleted and the resulting program (re-)analyzed. This process was repeated until the file was empty. The total time involved in this process is given in Table 2 by \texttt{NI}, for the case of restarting the analysis from scratch every time a clause is deleted (this is equivalent to NLcl - Incr in Table 1), by \texttt{Ltd} for the case of incremental analysis using the “top-down” algorithm, and by \texttt{I bu} for the “bottom-up” algorithm. \texttt{L td SU} and \texttt{I bu SU} represent the speedups obtained by the top-down and bottom up algorithms, respectively, with respect to non-incremental analysis (\texttt{NI}). The results are also very encouraging: in the worst case studied of compiling clause by clause the speedup with respect to the non-incremental algorithm is on the average a factor of 2.63 for top-down and 8.21 for bottom-up. The results seem to favour the bottom-up algorithm, as shown by \texttt{td/bu} in Table 2, giving an average speedup of 3.12 with respect to top-down.

Although we have implemented it, we do not report on the performance of arbitrary change because of the difficulty in modeling in a systematic way the types of changes that are likely to occur in the circumstances in which this type of change occurs (as, for example, during an interactive program development session). We have studied however the case of local reanalysis in a realistic environment: within the &-Prolog compiler, in which, after a first pass of analysis, new, specialized clauses are generated containing run-time tests, and a reanalysis is performed in order to propagate the more precise information which can be obtained in the program beyond the points where the new tests have been introduced. This more
precise information is then used for multiple specialization [15]. The results are shown in Table 3 in which benchmarks that do not generate run-time tests have been left out, since no specialization is performed for them and no reanalysis is needed in that case. Only program entry points are given to the analysis, i.e., no input patterns are specified for such entry points. This represents the likely situation where the user provides little information to the analyzer and also produces more run-time tests and thus more specializations and reanalysis, which allows us to study more benchmarks (note that if very precise information is given by the user then many benchmarks are parallelized without any run-time tests). \( CI \) is the number of clauses that have changed. \( 1^{st} \) is the time for analysis of the program in the first pass. \( Inc \) is the time for additional analysis after annotation (using the incremental algorithm). \( Scr \) is the time for additional analysis after annotation but restarting the analysis from scratch, i.e., no incrementality. \( Scr \setminus Inc \) is the speedup in the reanalysis part due to incrementality. \( Scr \setminus (1^{st} + Inc) \) is a measure of the incrementality (close to one or over one is desirable). The results of incremental analysis of local change are even more encouraging than the previous ones. The speedups are quite impressive and the incrementality level is high or very high in all cases. In fact, in \texttt{boyer}, and, specially, in \texttt{warplan} incrementality is indeed very high. This is related to the fact that there is a high degree of specialization in these programs and the first analysis is run over many less clauses than the second pass, which penalizes reanalyzing from scratch.

**Acknowledgments**

This work was supported in part by ESPRIT project 6707 “ParForCE” and CICYT project TIC93-0737-C02-01. The authors would also like to thank M. J. García de la Banda, Harald Sondergaard, and the anonymous referees for useful comments.

**References**


