Incremental Data-Flow Analysis Algorithms

BARBARA G. RYDER and MARVIN C. PAULL
Rutgers University

An incremental update algorithm modifies the solution of a problem that has been changed, rather than re-solving the entire problem. ACINCF and ACINCB are incremental update algorithms for forward and backward data-flow analysis, respectively, based on our equations model of Allen–Cocke interval analysis. In addition, we have studied their performance on a "nontoy" structured programming language L. Given a set of localized program changes in a program written in L, we identify a priori the nodes in its flow graph whose corresponding data-flow equations may be affected by the changes. We characterize these possibly affected nodes by their corresponding program structures and their relation to the original change sites, and do so *without actually performing the incremental updates*. Our results can be refined to characterize the reduced equations possibly affected if structured loop exit mechanisms are used, either singly or together, thereby relating richness of programming-language usage to the ease of incremental updating.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Language Constructs—*control structures*; D.3.4 [**Programming Languages**]: Processors—*optimization*; *run-time environments*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Data-flow analysis, elimination methods

1. INTRODUCTION

A global data-flow algorithm gathers information about the definition and use of data in a procedure or a set of programs. The algorithm is usually applied to some intermediate representation of a program. For instance, it may be the control flow graph of a procedure [14]. Alternatively, it may be a call graph describing the calling relations between procedures in a program [3]. In each of these, we have a single-entry digraph representation of control flow, called a *flow graph*. Each node has associated data-flow constants that describe how the code at that node affects data in the program. Data-flow analysis algorithms gather this local information and infer global data flow from it. This global information then is specialized to provide data-flow information with respect to any node in the flow graph.

Portions of this paper were included in a presentation at the "10th Annual ACM Symposium on Principles of Programming Languages," Austin, Texas, January 24–26, 1983.
Authors' address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

An incremental update algorithm for data-flow analysis modifies a known data-flow solution to reflect changes in a program; it obtains the new solution without application of the original algorithm, by recalculating only that part of the solution affected by program changes. We have designed and analyzed incremental update data-flow algorithms based on elimination methods. Given a program and a known data-flow problem solution, the effects of a set of "localized" program changes usually can be determined without requiring full reanalysis of the program. The ease of incremental updating depends on the choice of data-flow algorithm, the direction of data flow (i.e., forward versus backward), the program changes allowed (i.e., control flow changes that affect the structure of the flow graph or changes that affect the local data-flow characteristics of a node), and the structure of the flow graph (e.g., the nesting level of loops). The depth of loop nesting directly affects the complexity of incremental updating; it also is a key element in calculation of the worst-case complexity of Allen–Cocke interval analysis. We have shown that for a reducible flow graph [14] with loop nesting depth bounded by a constant, Allen–Cocke interval analysis has an $O(n)$ worst-case complexity bound [40].

Our models of elimination algorithms show how these algorithms solve the systems of equations defining a data-flow problem [40]. Using them we can ask "If we allow a small change in a structured system of equations whose solution is already known, can we find the effects of that change without totally re-solving the system?" Our model of Allen–Cocke interval analysis views that algorithm as solving a sequence of progressively smaller systems of equations, in order to obtain the solution of the original system. By incrementalizing this model, we have developed the incremental update algorithms presented here, ACINCF and ACINCB, for forward and backward data-flow problems, respectively. We also modeled Hecht–Ullman T1-T2 analysis [47] and Tarjan interval analysis [45, 46]; we designed HUINC, an incremental algorithm based on the Hecht–Ullman algorithm [34], which is discussed here as well.

We have studied the performance of ACINCF and ACINCB on a "nontoy," ALGOL-like structured programming language L, with loop exit structures similar to those of Sail [25]. We have identified those program structures that affect the complexity of incremental updating, and have established the extent of updating required by combinations of such structures. We have shown that, under certain realistic conditions, ACINCF and ACINCB are very efficient.

We have analyzed our incremental algorithms under the assumption of *localized program changes*, nonstructural changes occurring within one interval. *Nonstructural changes* are those that leave the flow graph and its interval structure unchanged.[1] *Structural changes* modify the flow graph itself and may require finding new intervals; they are not accommodated by these incremental algorithms. Recent work on handling structural changes in an interval-based method is reported in [36–38]. We have analyzed localized program changes on a deeply nested loop in an L program and have characterized the set of all variables whose equations may be affected. We have identified the variables and their corresponding program structures; we describe these in terms of their relation to the original

[1] Our nomenclature for categorizing changes is similar, but not identical, to Burke's [8].

change site. Our results enable us to analyze a flow graph of an L program with possible changes and to identify a priori nodes whose equations may be affected by these changes. Thus, we ascertain all data-flow solutions potentially affected by the changes without ever performing the changes themselves.

At the outset of this research, we found no previous published work in incremental data-flow analysis;[2] communication with F. E. Allen confirmed this fact. In [31] Rosen outlined how some global flow algorithms could be adapted for efficient incremental use. We concur in his opinion of the inappropriateness of the conventional worst-case error bounds for these algorithms. Our theoretical studies of algorithm performance on L provide better insight into incremental algorithm complexity. Reps et al., working in the context of syntax-directed editing, have developed an incremental algorithm for updating an attributed tree when one of its subtrees is replaced [26–28]. Their complexity analysis expresses the work of the algorithm as a function of the size of the area affected by that replacement, O(affected area). A significant difference from our work is that the interdependences of their attributes are expressible as an acyclic digraph; in contrast, our data-flow information usually exhibits cyclic interdependences.

Incremental data-flow analysis can be applied to many problem domains, including interprocedural data-flow analysis (private communication from F. E. Allen and J. T. Schwartz (1975), "Determining the Data Relationships in a Collection of Procedures," and [3, 7, 10, 38, 43]). Debugging of large software systems necessitates checks on the use of parameters and global variables. Often in system development, the procedure set remains fixed while the data-flow characteristics of the procedures change. In fact, it is during the writing and debugging of a system that the greatest need exists for data-flow information. Elimination-based interprocedural data-flow algorithms exist (private communication from M. Sharir (1977), "Interprocedural Analysis of Global Variable Usage," and [8, 41]); ACINCF and ACINCB correspond to incremental versions of these algorithms.

Source-to-source transformation systems depend on data-flow information that triggers certain transformations that subsequently change the source code and may change its data-flow characteristics [9, 18, 23]. Currently, ad hoc methods are used to accommodate changes in data flow caused by program changes with localized impact; a more systematic approach would be preferable. Incrementalizing the data-flow algorithms based on attribute grammars or high-level analysis algorithms suggests itself naturally, because source-to-source systems often work with a parse tree representation of a program [5, 6, 16, 26, 30].

Interactive programming environments are popular, useful tools for software development. Data-flow analysis information can be used as a debugging and documentation aid [2, 17, 21, 22, 32, 33, 48, 49]. Data-flow information substantially eases the pain of debugging, if made available as programs are written, usually in an incremental modular fashion. Incremental analysis can also be utilized in an optimizing compiler to keep data-flow information current after transformations such as code motion.

[2] The idea of demand analysis [5] is related to, but not the same as, our notion of incremental analysis.

In the remaining sections of this paper, first we discuss our equations model of Allen–Cocke interval analysis, illustrating it with an example. In Section 3 we describe ACINCF and ACINCB, the incremental update algorithms based on Allen–Cocke interval analysis; we state them formally and demonstrate their behavior on an example. Then we discuss our theoretical results on the performance of these algorithms; the proofs of these results are in the Appendix. In Section 5 we briefly present our experiences with HUINC, our incremental update algorithm based on Hecht–Ullman T1-T2 analysis, and we contrast it with ACINCF. Finally, we summarize our work and indicate interesting application areas.

2. MODEL OF ALLEN–COCKE INTERVAL ANALYSIS

To model Allen–Cocke interval analysis, we present the equations that define data-flow problems and a Gaussian elimination-like solution procedure for them, generally applicable to a system of equations satisfying certain properties [24]. We apply this procedure in the specific context of equations involving union and intersection operators. We then show Allen–Cocke interval analysis is an optimization of this technique for "well-behaved" systems. (Throughout these discussions we assume familiarity with compilation and data-flow analysis [1, 14].)

2.1 Data-Flow Equations

A data-flow problem is defined on a flow graph by a system of equations involving the operators of union and intersection. The four classical data-flow problems, reaching definitions, live uses of variables, available expressions, and very busy expressions, all can be formulated in this manner; they are sufficient for most compiler optimizations (e.g., dead code elimination, constant propagation, common subexpressions elimination) [1, 14].

Each variable Z_m in the system of equations is identified with a unique flow-graph node; its value is the data-flow solution on entry to the node, for forward problems, or alternatively on exit from the node, for backward problems. Given this one-to-one relationship between nodes and variables, we use these terms interchangeably; interpretation will be clear from the context. The following is the general form of the equations $\{Q_m\}_{m=1}^{n}$:

$$Q_m\colon Z_m = \mathop{\ominus}_{k \in S_m} \{a_{m,k} \cap Z_k \cup b_{m,k}\} \cup c_m \qquad \text{for} \quad 1 \le m \le n, \tag{1}$$

where Z_m is the data-flow solution either on entry to or on exit from node m; $\ominus$ is intersection or union; $a_{m,j}$, $b_{m,j}$, and c_m are constants derived from local data-flow information (possibly null); and $S_m \subseteq \{i \mid 1 \le i \le n\}$.

For a forward data-flow problem S_m is the set of immediate predecessors of m in the flow graph, $\{\text{pred}(m)\}$; for a backward problem, S_m is the set of immediate successors of m, $\{\text{succ}(m)\}$. The coefficients and constants in the equations are defined using the local data-flow characteristics associated with the code at each node. Using a flow graph annotated with this information, we can describe an associated data-flow problem by a system of equations of the form of eq. (1).

2.2 Gaussian Elimination-like Solution Procedure

Consider those data-flow problems that can be defined by a system of equations $Q = \{Q_m\}_{m=1}^{n}$, where Q_m is an equation of the form of eq. (1). We assume that the set of possible solutions, each an n-tuple $\langle Z_1, \ldots, Z_n \rangle$ that satisfies the system Q, admits a partial ordering ($\leq$).[3]

In Gaussian elimination, variables are successively eliminated from a system of equations by repeated substitution of the right-hand side of a variable's equation for occurrences of that variable in other equations [15]. We use an analogous substitution process. A *substitution transformation* of a system of equations Q, $s(Q, m, j)$, for $1 \leq m, j \leq n$, is the result of substituting the right-hand side of Q_m for an occurrence of Z_m on the right of equation Q_j, $m \neq j$, and simplifying the resultant right-hand side of Q_j. Then $s(Q, m, j)$ differs from Q by having at most a different Q_j equation; all other equations are the same. It is clear that a solution of $s(Q, m, j)$ is also a solution to Q, and vice versa.

During the substitution transformation, it is possible to introduce a self-reference in an equation if there are cyclic dependences among the variables in the system. A loop-breaking rule handles the possible self-references introduced. *An equation Q_m has a loop-breaking rule* if there is another equation for Z_m called q_m such that

(i) Z_m does not appear on the right-hand side of q_m,
(ii) every solution of q_m is also a solution of Q_m,
(iii) for every solution S of Q_m there is a solution s of q_m such that $s \leq S$, and
(iv) the set of variables on the right-hand side of q_m is a subset of the variables on the right-hand side of Q_m [24].[4]

A set of equations Q is said to have a loop-breaking rule if, for each equation in Q initially, there is a loop-breaking rule, and, for any equation in any set that can result from Q by a sequence of substitution transformations of Q, there is also a loop-breaking rule. A *loop-breaking transformation* of Q, $b(Q, m)$, for $1 \leq m \leq n$, is the result of replacing Q_m by q_m.

The Gaussian elimination-like solution procedure for the system of equations consists of applying a sequence of the substitution and loop-breaking transformations; the procedure is shown in Figure 1. The complexity of this algorithm is $O(n^3)$, assuming (as usually holds) that each application of b is $O(1)$ and s is $O(n)$. It can be shown that if a sequence of these transformations is applied to a system of equations Q, producing the system R, and $\{S_m \mid 1 \leq m \leq n\}$ is a solution to R, then it is also a solution to Q. Further, if $\{L_m \mid 1 \leq m \leq n\}$ is a solution to Q, then there is a solution of R, $\{K_m \mid 1 \leq m \leq n\}$, such that $K_m \leq L_m$ for $1 \leq m \leq n$. If Q has a loop-breaking rule, then the procedure in Figure 1 terminates and obtains the unique minimal solution (in terms of the partial ordering) [24].

For the classical data-flow problems, the implementation of this method can involve bit vector or set operations. The partial ordering on the n-tuples is one

[3] All the classical data-flow problems have this property.

[4] In general, q_m contains all but the self-dependent term from Q_m. This property guarantees the process eventually leads to a solution.

```
/* Elimination */
for i = 1 to n − 1 do
  begin
    Q ← b(Q, i)
    for j = i + 1 to n do Q ← s(Q, i, j)
  end
/* Back Substitution */
for i = n to 2 do
  begin
    for j = i − 1 to 1 do Q ← s(Q, i, j)
  end
```

Fig. 1. Gaussian elimination-like solution procedure.

of component-wise set inclusion for a set implementation and component-wise comparison for a bit vector implementation. The loop-breaking rules for these problems are very simple.[5] In eq. (1), if $\ominus$ is $\cup$ as in reaching definitions, then we have

$$Q_m\colon Z_m = a \cap Z_m \cup \beta, \tag{2}$$

where a is a constant and β can contain variables other than Z_m as well as constants. The corresponding loop-breaking rule substitutes equation q_m for Q_m:

$$q_m\colon Z_m = \beta. \tag{3}$$

In this case we say the loop-breaking rule is to drop the self-referential term (i.e., $a \cap Z_m$). In eq. (1), if $\ominus$ is $\cap$ as in available expressions, then we have

$$Q_m\colon Z_m = (a \cap Z_m \cup c) \cap \beta, \tag{4}$$

where a and c are constants and β can contain variables other than Z_m as well as constants. The corresponding loop-breaking rule substitutes equation q_m for Q_m:

$$q_m\colon Z_m = c \cap \beta. \tag{5}$$

To validate a loop-breaking rule for an equation, we must satisfy the four conditions (i)–(iv) mentioned previously. Clearly, Z_m does not appear on the right-hand side of q_m in eq. (3) (i.e., (i) is satisfied). Next, the solution of the loop-breaking rule q_m must be shown to satisfy the original equation, Q_m. Letting $Z_m = \beta$ in eq. (2), we have

$$\beta =? (a \cap \beta) \cup \beta,$$

which is clearly true (i.e., (ii) is satisfied). For every solution S of Q_m, there must be a solution s of q_m such that $s \leq S$. Here, if S is a solution to Q_m then

$$S = (a \cap S) \cup \beta \Rightarrow \beta \subseteq S.$$

Therefore, $Z_m = \beta \leq S$ for any solution S of Q_m (i.e., (iii) is satisfied). Finally, by examination of eqs. (2) and (3) we see that (iv) is satisfied. By replacing eq. (2) by eq. (3), we are selecting the minimal solution for Z_m from the set of possible solutions satisfying Q_m.

[5] In general, loop-breaking rules are determined by the operators in the equations [24].

Similar arguments validate the loop-breaking rule in eqs. (4) and (5). We use examples of reaching definitions and live uses of variables to illustrate these ideas in Section 2.4.

2.3 Allen–Cocke Interval Analysis Model

The key observation of the Allen–Cocke algorithm is that certain systems of equations exhibit variable interdependences that can be utilized to aid in their solution. Specifically, data-flow equations exhibit highly structured systems of equations due to the "shape" of flow graphs of programs; Allen–Cocke interval analysis optimizes the general solution procedure for these systems, lowering its worst-case complexity to $O(n^2)$.

Given any set of equations of the form of eq. (1), we can define a *dependency graph* as a digraph corresponding to the interdependences of variables given by the equations in that system. Each node represents a variable; each directed edge (m, n) represents the dependence of Z_n on Z_m (i.e., the occurrence of Z_m on the right-hand side of the equation for Z_n). For forward data-flow problems, the dependency graph is the flow graph. Because flow graphs are usually *reducible*, that is, they contain no multiple entry loops [14], Allen–Cocke interval analysis is able to convert the solution of a system of n data-flow equations into the solution of a smaller system of r equations.[6] This is accomplished by partitioning the variables into r subgroups called *intervals*, single-entry regions corresponding approximately to loops in the dependency graph.

For backward data-flow problems, the dependency graph is the converse of the flow graph, that is, the flow graph with the direction of all its edges reversed. We cannot guarantee a "nice" structure of this dependency graph, since multiple exit loops in the flow graph may be multiple entry loops in its converse. Therefore, we use the *reverse dependency graph* of the backward data-flow equations to partition the variables in a backward problem; this reverse dependency graph is the flow graph.[7] We will speak of finding intervals in the context of a forward data-flow problem; the conversion for backward data flow is to use the reverse dependency graph. We indicate this conversion by a parenthesized reference (i.e., "(reverse)" dependency graph) in the following discussions.

The variable partitioning algorithm that finds the interval is shown in Figure 2 [4, 14]. If h is the entry node of an interval, we call it the *interval head node* of interval I_h; the corresponding variable is called an *interval head variable.* The order in which nodes are added to an interval is called an *interval order*; it preserves an ancestor-first ordering within each interval with respect to the nonlooping edges of the (reverse) dependency graph. We can form a linear order of all the nodes in the graph that embeds the interval order of every interval. By

[6] Although interval analysis can be applied to irreducible graphs using node splitting or iteration in the irreducible areas, we are omitting these from consideration since they are rarely needed in practice [14, 42].

[7] Given a set of data-flow equations and their corresponding dependency graph, we can test that graph for reducibility. If it is reducible, we find Allen–Cocke intervals on it; if its converse is reducible, we can use the Allen–Cocke intervals on the converse instead. Thus, to use interval analysis as described here, the dependency digraph or its converse needs to be reducible, although irreducible digraphs can be accommodated as previously mentioned.

```
INT := null;  /* list of intervals */
I := null;  /* each interval */
H := {s};  /* header list initialized to flow-graph entry node */

while(H ≠ null) do
  Remove h from H;
  I := {h};  /* form I_h */
  while(There is a node m not s, whose immediate predecessors are all in I but m is not yet in I) do
    Add m to I;
  endwhile;
  Add I to INT;
  while(There is node n not in H and not in INT with at least one predecessor in I) do
    Add n to H;
  endwhile;
endwhile;
```

Fig. 2. Interval-finding algorithm.

writing the equations according to this linear order, we obtain a highly structured coefficient matrix, amenable to simplification by a sequence of substitution transformations. For a forward data-flow problem, this coefficient matrix has a block lower triangular structure, except for possibly full rows corresponding to interval head variables. This structure ensures that the equation for each variable in an interval can be parameterized in terms of its interval head variable. For a backward data-flow problem, the matrix has a block upper triangular structure, except for possibly full columns corresponding to interval head variables. This structure ensures that the equation for each variable in an interval can be parameterized as a linear function of a set of interval head variables. Therefore, equations of interval head nodes in forward and backward problems can also be parameterized in terms of interval head variables. The result of a parameterization is a *reduced equation.*

The Allen–Cocke algorithm consists of two phases: *elimination* and *propagation.* During elimination we perform successive substitution and loop-breaking transformations on the systems of equations, which gather and summarize the local data-flow side effects of code in the program. During propagation we perform back substitutions of solutions for terms in equations, which propagate global data-flow side effects to the local regions in which they apply.

For forward data-flow problems, the elimination phase consists of iterating three steps: finding intervals in the dependency graph associated with the system of equations, reducing the equations to form a new system of the reduced interval-head-variable equations, and forming the dependency graph of the reduced system. Within each interval in the system, a sequence of substitution transformations reduces each equation to a linear function of the interval head variable. A derived system of equations is formed, consisting of the r reduced interval-head-variable equations, depending only on interval head variables from the former system; that is, the reduced interval-head-variable equations become the equations of the derived system. The derived system is then, in turn, partitioned into intervals, each with an interval order, and the *coefficient matrix structure of the original problem is preserved* in its equations. When the original flow graph

is reducible, the three-step process can be continued, until the graph is reduced to a single node. This yields a sequence of K systems of equations with a final system of one equation.

The propagation phase consists of iterating two steps: establishing variable correspondences and substituting interval-head-variable solutions into reduced equations, thus obtaining solutions for non-interval head nodes. To begin, we solve the system of one equation. The final variable is associated with the corresponding interval head variable in the previous system; they share the same solution. Focusing on the previous system, the interval-head-variable solution is substituted into the reduced equations for variables in its interval. Then, each of these newly solved variables is associated with its corresponding interval head variable in the system previous to the one just solved. The solutions for all variables in this system are similarly obtained. This variable correspondence/substitution process is iterated through the derived systems of equations in reverse derivation order until all solutions are obtained.

For backward data-flow problems, the elimination phase is similar to that described previously, except the intervals are found on the (reverse) dependency graph of each system of equations. Also, the reduced equation for a variable is parameterized in terms of a set of interval head variables, not just its own interval head variable.[8] During propagation the variable correspondence steps are the same, but we may have to substitute a set of interval head solutions into the reduced equation of a variable rather than just one interval head solution.

In both forward and backward problems, the sequence of (reverse) dependency graphs $\{G^i\}_{i=1}^{K}$ corresponding to the sequence of systems of equations is called the *derived sequence of graphs*. We call G^{i+1} the *derived graph of* G^i. When we say that y in G^{i+1} *represents* I_h in G^i, we are asserting that all variables in I_h are represented by variable Z_y in G^{i+1}, whose corresponding node in G^{i+1} is y; the reduced equation of Z_h in G^i is the equation of Z_y in G^{i+1}. The definition of *represents* is extendible over finite subsequences of the derived sequence. Therefore, if

$$m_1 \in I_{m_2} \subseteq G^i, \ldots, m_k \in I_{m_{k+1}} \subseteq G^{i+k-1}$$

we say that m_k *represents* m_1 in G^{i+k-1}.

In a forward data-flow problem during elimination, we remove all dependence in the system of equations on variables in I_h, replacing them by a dependence on X_h. The graphical interpretation of this action is that h in G^2 represents I_h in G^1. In Figure 3, node 2 in G^2 represents I_2 in G^1. This signifies that non-interval head nodes in I_2 on G^1 (i.e., {3}) do not appear in the derived system corresponding to G^2. Of course, since we partition the nodes of both G^1 and G^2 into intervals, node 2 belongs to two different intervals on the two graphs; node 2 is in I_2 on G^1 and in I_1 on G^2. That is, there are two different systems of equations depicted in these graphs; X_2 is in a different partition element in each system. Similar arguments hold for a backward data-flow problem.

[8] Recall an interval is single entry, but possibly multiexited; thus, backward data flow may require a dependence within an interval on the targets of more than one interval exit.

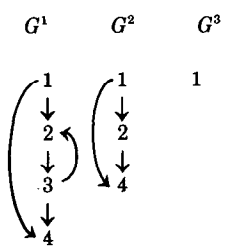


Fig. 3. Derived sequence.

$I_1 = \{1\}$ $\quad I_1 = \{1\ 2\ 4\}$ $\quad I_1 = \{1\}$
$I_2 = \{2\ 3\}$
$I_4 = \{4\}$

Equations on G^1:

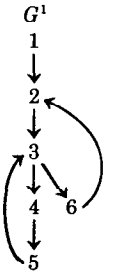
$X_1 = \varnothing$

$X_2 = (p_1 \cap X_1 \cup d_1) \cup (p_6 \cap X_6 \cup d_6)$

$X_3 = (p_2 \cap X_2 \cup d_2) \cup (p_5 \cap X_5 \cup d_5)$

$X_4 = X_6 = p_3 \cap X_3 \cup d_3$

$X_5 = p_4 \cap X_4 \cup d_4$



Intervals on G^1: $I_1 = \{1\}$, $I_2 = \{2\}$, $I_3 = \{3\ 4\ 5\ 6\}$

Let $d_{i_1,\ldots,i_k} = (p_{i_1} \cap \cdots \cap p_{i_{k-1}} \cap d_{i_k}) \cup (p_{i_1} \cap \cdots \cap p_{i_{k-2}} \cap d_{i_{k-1}}) \cup \cdots \cup d_{i_1}$
Let $p_{i_1,\ldots,i_k} = p_{i_1} \cap p_{i_2} \cap \cdots \cap p_{i_k}$

Reduced equations on G^1:

On I_3: X_4 and X_6 are already in reduced form

$$X_5 = (p_{43} \cap X_3) \cup d_{43}$$
$$X_3 = (p_2 \cap X_2 \cup d_2) \cup (p_{543} \cap X_3) \cup d_{543}$$

After loop-breaking: $X_3 = (p_2 \cap X_2 \cup d_2) \cup d_{543}$

On I_2:

$$X_2 = (p_1 \cap X_1 \cup d_1) \cup (p_{63} \cap X_3 \cup d_{63})$$

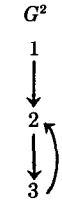
on I_1: X_1 is already in reduced form

Equations on G^2

$X_1 = \varnothing$

$X_2 = (p_1 \cap X_1 \cup d_1) \cup (p_{63} \cap X_3 \cup d_{63})$

$X_3 = (p_2 \cap X_2 \cup d_2) \cup d_{543}$



Intervals on G^2: $I_1 = \{1\}$, $I_2 = \{2\ 3\}$

Reduced equations on G^2:

On I_2: X_3 is already in reduced form

$$X_2 = (p_1 \cap X_1 \cup d_1) \cup (p_{632} \cap X_2 \cup d_{632}) \cup (p_{63} \cap d_{543})$$

Fig. 4. Solution of the reaching-definitions problem.

After loop-breaking: $X_2 = (p_1 \cap X_1 \cup d_1) \cup d_{632} \cup (p_{63} \cap d_{543})$

On I_1: X_1 is already in reduced form

Equations on G^3: G^3

$X_1 = \emptyset$ 1

↓

$X_2 = (p_1 \cap X_1 \cup d_1) \cup d_{632} \cup (p_{63} \cap d_{543})$ 2

Intervals on G^3: $I_1 = \{1\ 2\}$

Reduced equations on G^3:

On I_1: X_1 and X_2 are already in reduced form

Equations on G^4: G^4

$X_1 = \emptyset$ 1

Intervals on G^4: $I_1 = \{1\}$

Solution on G^4:

$$X_1 = \emptyset$$

Solutions on G^3:

$$X_1 = \emptyset$$
$$X_2 = d_1 \cup d_{632} \cup (p_{63} \cap d_{543})$$

Solutions on G^2:

$$X_1 = \emptyset$$
$$X_2 = d_1 \cup d_{632} \cup (p_{63} \cap d_{543})$$
$$X_3 = d_{21} \cup d_{263} \cup d_{543}$$

Solutions on G^1:

$$X_1 = \emptyset$$
$$X_2 = d_1 \cup d_{632} \cup (p_{63} \cap d_{543})$$
$$X_3 = d_{21} \cup d_{263} \cup d_{543}$$
$$X_4 = X_6 = d_{321} \cup d_{354} \cup d_{326}$$
$$X_5 = d_{4321} \cup d_{4326} \cup d_{435}$$

Fig. 4. (*continued*)

2.4 Examples

The following examples illustrate how the Allen–Cocke algorithm solves the reaching-definitions (forward) and live-uses-of-variables (backward) problems. In the former we calculate which variable definitions can determine the value of a variable at a particular program point. In the latter we calculate uses of a variable that occur before a redefinition of that variable from some program point, along possible execution paths in a program. For both problems we show the equations, reduced equations, and simplified solutions for each system in the derived sequence.

In Figure 4 we are given the flow graph, its corresponding derived sequence, the equations defining reaching definitions, and their solution using the Allen–

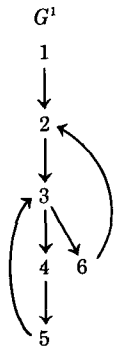
Equations on G^1:

$Y_1 = Y_6 = r_2 \cap Y_2 \cup u_2$

$Y_2 = Y_5 = r_3 \cap Y_3 \cup u_3$

$Y_3 = (r_4 \cap Y_4 \cup u_4) \cup (r_6 \cap Y_6 \cup u_6)$

$Y_4 = r_5 \cap Y_5 \cup u_5$



Let $u_{i_1, \ldots, i_k} = (p_{i_1} \cap \cdots \cap p_{i_{k-1}} \cap u_{i_k}) \cup (p_{i_1} \cap \cdots \cap p_{i_{k-2}} \cap u_{i_{k-1}}) \cup \cdots \cup u_{i_1}$
Let $r_{i_1, \ldots, i_k} = r_{i_1} \cap r_{i_2} \cap \cdots \cap r_{i_k}$

Reduced equations on G^1:

On I_3: Y_6 and Y_5 are already in reduced form

$$Y_4 = r_{53} \cap Y_3 \cup u_{53}$$
$$Y_3 = (r_{453} \cap Y_3 \cup u_{453}) \cup (r_{62} \cap Y_2 \cup u_{62})$$

After loop-breaking: $Y_3 = u_{453} \cup (r_{62} \cap Y_2 \cup u_{62})$

On I_1: Y_1 is already in reduced form

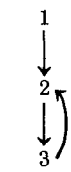
On I_2: Y_2 is already in reduced form

Equations on G^2:

$Y_1 = r_2 \cap Y_2 \cup u_2$

$Y_2 = r_3 \cap Y_3 \cup u_3$

$Y_3 = u_{453} \cup (r_{62} \cap Y_2 \cup u_{62})$



Reduced equations on G^2:

On I_2: Y_3 is already in reduced form

$$Y_2 = u_{345} \cup (r_{362} \cap Y_2) \cup u_{362}$$

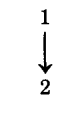
After loop-breaking: $Y_2 = u_{345} \cup u_{362}$

On I_1: Y_1 is already in reduced form

Equations on G^3:

$Y_1 = r_2 \cap Y_2 \cup u_2$

$Y_2 = u_{345} \cup u_{362}$



Reduced equations on G^3:

On I_1: Y_2 is already in reduced form

$$Y_1 = u_{2345} \cup u_{236}$$

Fig. 5. Solution of the live-uses-of-variables problems.

Equations on G^4: G^4

$Y_1 = u_{2345} \cup u_{236}$ 1

Solution on G^4:

$$Y_1 = u_{2345} \cup u_{236}$$

Solution on G^3:

$$Y_1 = u_{2345} \cup u_{236}$$
$$Y_2 = u_{345} \cup u_{362}$$

Solution on G^2:

$$Y_1 = u_{2345} \cup u_{236}$$
$$Y_2 = u_{345} \cup u_{362}$$
$$Y_3 = u_{453} \cup u_{623}$$

Solution on G^1:

$$Y_1 = Y_6 = u_{2345} \cup u_{236}$$
$$Y_2 = Y_5 = u_{345} \cup u_{362}$$
$$Y_3 = u_{453} \cup u_{623}$$
$$Y_4 = u_{534} \cup u_{5362}$$

Fig. 5. (*continued*)

Cocke algorithm. The application of a loop-breaking rule is shown where needed; algebraic simplification is performed on the equations and their solutions. Here p_j is the set of all definitions of variables that are preserved through node j (i.e., definitions corresponding to those variables not redefined at j), and d_j is the set of all new definitions of variables at j. We use X_i as the variable in this problem to emphasize that we are solving for data flow on entry to node i. In terms of eq. (1), $a_{m,j} = p_j$, $b_{m,j} = d_j$, and $c_m = \emptyset$.

In Figure 5 we show the solution of a live-uses-of-variables problem on the same derived sequence as in Figure 4. Here r_j is the set of all uses of variables that are preserved through node j (i.e., uses corresponding to variables not defined at node j), and u_j is the set of all uses of a variable at node j that occur before any definition at j. We use Y_i as the variable in this problem to emphasize that we are solving for data flow on exit from node i. In terms of eq. (1), $a_{m,j} = r_j$, $b_{m,j} = u_j$, and $c_m = \emptyset$. Loop-breaking and algebraic simplification are performed here as well.

3. ACINCF AND ACINCB: INCREMENTAL INTERVAL ANALYSIS

Our incremental update algorithms ACINCF and ACINCB consist of two phases corresponding to the two phases of our model of Allen–Cocke interval analysis described in Section 2.3. They are designed to update a data-flow solution after a set of *localized program changes*, that is, a set of changes within one interval. In the *elimination* phase, coefficients and constants in all data-flow equations affected by the changes are recalculated. In the *propagation* phase, all affected solutions are recalculated. In this section we informally describe ACINCF and

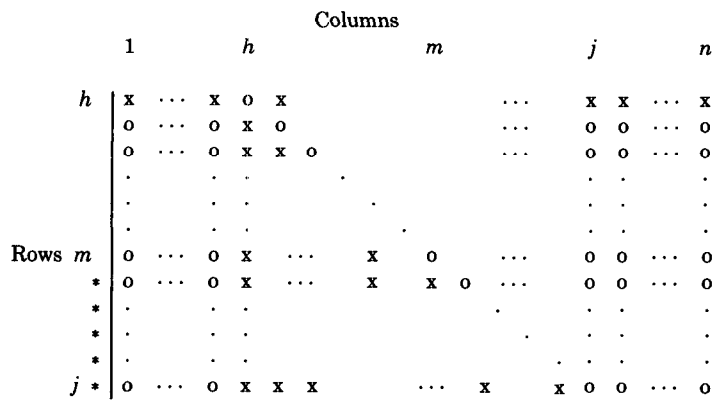


Fig. 6. Affected coefficients for the forward data-flow problem.

ACINCB, followed by formal algorithm statements and examples of their application.

3.1 Overview of ACINCF

In Figure 6 we show the coefficient structure of the equations of a forward data-flow problem for the variables in I_h (i.e., X_h, X_{h+1}, ..., X_j), assuming the variables are ordered in an interval order. Possibly nonzero coefficients are indicated by x. By performing our substitution transformations for variables in interval order (i.e., X_{h+1}, X_{h+2}, etc.), we calculate reduced equations on I_h. Given a change in the code at a non-interval head node $m \in I_h$, the *s indicate the rows corresponding to variables whose reduced equations may change because of a change in equation Q_m; we must recalculate these reduced equations. Then we must check to see if there has been an interval-head-variable reduced equation affected by these changes; if none has been affected, we are finished. Otherwise, the reduced equation for some X_s is affected; this reduced equation is the equation for X_y, where y represents I_s in the first derived system, G^2. In that system we find the interval containing y, I_r. Then we can find the reduced equations in I_r that are affected by this change in the equation for X_y as previously. We must iterate this process through the sequence of systems of equations until all changes have been revealed.

If there is a change in the code at node h, an interval head node, then within I_h only the reduced equation for X_h can be affected. Since X_h remains as a variable in the derived system, no other interval-head-variable reduced equations are affected because we never substitute for their X_h terms. We must recalculate the reduced equation for X_h, and if it has changed, then we proceed as above, when there is a change to an interval-head-variable reduced equation.

Eventually, either we will find a system where no interval-head-variable reduced equation is affected, or we will reach the last system of equations. In the former case, we can reperform the back substitutions in the changed reduced equations in this system, obtaining new solutions. In the latter case, we can solve the equation for the last variable in the system, obtaining a new solution. In either case, we identify each changed solution in the final system with an interval-

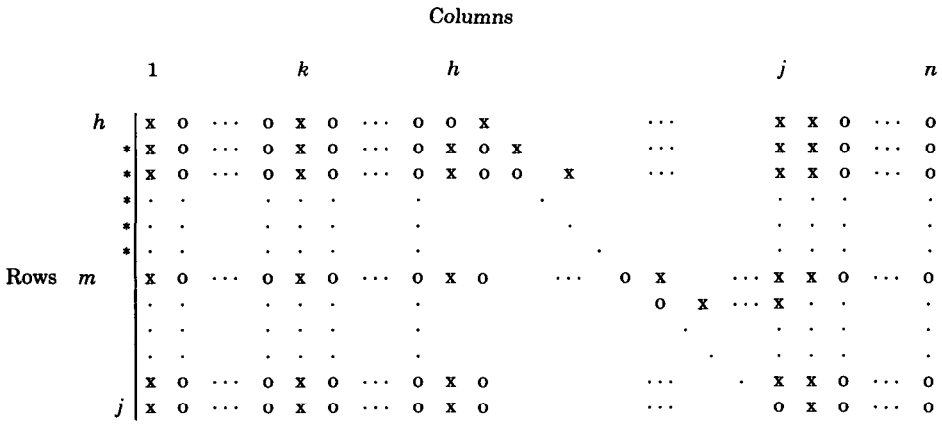


Fig. 7. Affected coefficients for the backward data-flow problem.

head-variable solution in the previous system. We substitute each changed interval-head-variable solution in the reduced equations for all variables within its interval, obtaining all solutions in that interval. We also reevaluate any reduced equations on this system that have been changed, whose solutions have not yet been recalculated. This back substitution process continues through the sequence of systems in reverse derivation order, updating all solutions corresponding to changed reduced equations and/or changed interval-head-variable solutions.

3.2 Overview of ACINCB

Our incremental update algorithm ACINCB is similar to ACINCF; the differences arise from the dissimilarities in the coefficient matrix structure and the reduced equation form. Figure 7, analogous to Figure 6, shows the coefficient structure of the equations of a backward data-flow problem for the variables in I_h, assuming the variables are ordered in an interval order.[9]

Given a change in the code at non-interval head node $m \in I_h$, the *s indicate rows whose reduced equations may need recalculation because of the effects of this change at m. We must recalculate reduced equations in this region when necessary. Then we must check to see if an interval-head-variable reduced equation has been affected by these changes; however, because an interval is a single-entry connected subgraph, Y_h is the only interval head variable possibly affected. The reduced equation for Y_h is the equation of Y_s, where s represents I_h in the derived system. If $s \in I_r$ in the derived system, then we can find the reduced equations in I_r possibly affected by this change in the equation for Y_s as previously. We iterate this process through the sequence of systems of equations until all coefficient/constant changes have been propagated as far as possible.

If the code at node $h \in I_h$ is changed, by the same reasoning as in Section 3.1, this can only affect the reduced equation for Y_h. If this reduced equation is changed, we proceed as above.

[9] Note the possibly full columns 1, k, h, and $(j + 1)$ corresponding to interval head variables Y_1, Y_k, Y_h, and Y_{j+1}.

Finally, either we will find a system where no interval-head-variable equation is affected, or we will reach the last system of equations. In the former case, we can reperform the back substitutions in the changed reduced equations in this system, obtaining new solutions. In the latter case, we can solve for the final variable. We identify each changed solution with an interval-head-variable solution in the previous system. Then we must recalculate all non-interval-head-variable solutions in the previous system corresponding to reduced equations containing that interval head variable. We also recalculate all non-interval-head-variable solutions in the previous system corresponding to a reduced equation whose coefficients/constants have been changed by the elimination phase, being careful to avoid duplicate recalculation. This back-substitution process continues through the sequence of systems in reverse derivation order, updating all solutions corresponding to changed reduced equations and/or changed interval-head-variable solutions.

3.3 Algorithm Statements

We assume much of the intermediate data-flow information has been saved from the initial Allen–Cocke interval analysis of the program. Specifically, we require as input

(i) the sequence of systems of equations and the reduced equations for non-interval head variables in each system;
(ii) the sequence of dependency graphs $\{G^i\}_{i=1}^{K}$ corresponding to the systems of equations, and the intervals of each graph with their interval orders; and
(iii) the data-flow solutions for all the variables in all the systems.

As output, our algorithm produces

(i) an updated sequence of systems of equations that reflects the effects of a set of program changes within one interval, and updated reduced equations corresponding to each system; and
(ii) a set of new data-flow solutions that exhibit the effects of the program changes.

When there is a program change to the code represented by node v, the coefficients and/or constants "associated" with v may change. This change affects the initialization of our incremental algorithms; however, this change at v may also change the underlying solution space for the data-flow problem. For example, in Figure 9 (shown later) the addition of a variable definition at node 5 expands the set of potential reaching definitions, and thus the solution space, affecting coefficients at every node. Similarly, a variable use deleted from node 4 decreases the set of potential live uses at every node, as in Figure 11 (shown later), contracting the solution space. The coefficient changes that result from the expansion or contraction of the solution space do not affect our initialization.

Given a program change at node v, we need to discover which equations contain the coefficients and constants associated with node v, in order to know where to

restart elimination. This will depend on the form of eq. (1) used and the type of data-flow problem (i.e., forward or backward). When a forward data-flow problem is solved on entry to the nodes, we use

$$Z_m = \underset{j \in \text{pred}(m)}{\ominus} \{a_{m,j} \cap Z_j \cup b_{m,j}\} \qquad \text{for} \quad 1 \leq m \leq n.$$

Here, coefficients $a_{*,v}$ and constants $b_{*,v}$ are *associated* with node v; that is, they are derived from the data-flow characteristics of the code represented by v. Thus, the set of variables whose equations contain these coefficients and constants is the set of immediate successors of node v in the dependency graph (i.e., the immediate successor set in the flow graph). This is the set T on G^1, referred to in our formulation of ACINCF.

There is a similar equation form for forward problems solved on exit from the nodes:

$$Z_m = \underset{j \in \text{pred}(m)}{\ominus} \{a_{m,j} \cap Z_j\} \cup c_m \qquad \text{for} \quad 1 \leq m \leq n.$$

Here, coefficients $a_{v,*}$ and constant c_v are associated with node v. In this case set T consists only of node v itself.

Similarly, for backward problems using the solved-on-exit formulation, T in ACINCB will be the dependency-graph immediate successors of node v (i.e., the immediate predecessor set in the flow graph), and for the solved-on-entry formulation, T will be node v itself.

The algorithms are formulated using sets S and T. S is used for record keeping during the elimination phase of the algorithm. As explained above, the initial value of T depends on the equation form used and the type of data-flow problem. On G^i, $i > 1$, T is the set of variables whose equations are changed because they correspond to changed reduced equations in the immediately previous system.

3.3.1 *ACINCF: Incremental Update Algorithm for Allen–Cocke Interval Analysis for Forward Data-Flow Problems*

Elimination phase

(i) Assume a program change occurs that changes a data-flow coefficient or constant associated with a node m. Initialize T to be m itself or the set of m's dependency-graph immediate successors, depending on the form of eq. (1) used and the type of data-flow problem being solved. Recalculate the reduced equations for every non-interval head node in T. Let $S = T$ and $k = 1$.

(ii) If G^k is one node, then solve for the final variable, and go to (vii).

(iii) Iterate this step on G^k, at each iteration removing an m from T, until T is empty. Then go to (iv).

Assume $m \in I_h$.

If $m = h$, go to the next iteration of (iii).

If $m \neq h$, assuming the nodes of I_h are numbered from h to j in an interval order, examine the equations for X_{m+1} through X_j in order. If the equation for X_l

contains a dependence on X_r for $r \in S$ and $m < l \leq j$, then recalculate the reduced equation for X_l; that is, substitute the right-hand side of the reduced equation for each X_r, $r \in S$, for X_r in the right-hand side of the equation for X_l, and simplify. If the resulting reduced equation for X_l differs from the previous reduced equation, add l to S.

(iv) Find all interval head variables that depend on any X_r for $r \in S$ and all interval head nodes themselves in S. For each interval head variable found, X_t, find the corresponding variable in G^{k+1}, X_y (i.e., y represents I_t in G^{k+1}). Then recalculate the reduced equation for X_t by substitution transformations using the current reduced equation for each variable and applying a loop-breaking rule where necessary. Compare the old and new reduced equations for X_t; if they differ, replace the equation for X_y in G^{k+1} with the new reduced equation for X_t in G^k, and mark it replaced.

(v) If there are no marked equations in G^{k+1}, go to (vi). Otherwise, form T from the subscripts of variables whose equations are marked in G^{k+1}, increment k by 1, let $S = T$, and repeat steps (ii)–(iv).

Propagation phase

(vi) Recalculate by back substitution the solutions corresponding to each reduced equation in G^k that has been changed. That is, if the reduced equation for X_j has been changed and j is in I_q, substitute the value of X_q into the newly changed reduced equation for X_j to find the updated value of X_j.

(vii) If $k = 1$ then stop. Otherwise, iterate this step, setting $k = k - 1$, and updating the affected data-flow solutions in G^k as follows:

(a) Use the value of each updated data-flow solution in G^{k+1}, X_r, to set the value of the corresponding interval head variable in G^k, X_h (i.e., set $X_h = X_r$, where r represents I_h in G^{k+1}).
(b) Recalculate the solutions for all variables in G^k in intervals whose interval head variables have had their values changed.
(c) Recalculate the solutions for all non-interval head variables in G^k whose reduced equations have been changed and whose solutions were not recalculated by (vii(b)).

The changes necessary to transform ACINCF into ACINCB are indicated below.[10]

3.3.2 *ACINCB: Incremental Update Algorithm for Data-Flow Analysis for Backward Data-Flow Problems*

(iii) Iterate this step on G^k, at each iteration removing an m from T, until T is empty. Then go to (iv).

[10] For this algorithm assume the X_j in the statement of ACINCF are Y_j, the data-flow solution on exit from node j. The algorithms can be formulated for the alternative equation forms trivially.

Assume $m \in I_h$.

If $m = h$ go to the next iteration of (iii).

If $m \neq h$, assuming the nodes of I_h are numbered from h to j in an interval order, examine the equations for Y_h through Y_{m-1} in reverse interval order (i.e., equations for Y_l, $l = m - 1, m - 2, \ldots, h$). If the equation for Y_l contains a dependence on Y_r for $r \in S$ and $h \leq l < m$, then recalculate the reduced equation for Y_l; that is, substitute the right-hand side of the reduced equation for each Y_r, $r \in S$, for Y_r in the right-hand side of the equation for Y_l, and simplify. If the resulting reduced equation for Y_l differs from the previous reduced equation, add l to S.

(iv) Check the reduced equation for Y_h to see if it has been affected by the changes within I_h. If so, find the corresponding variable Y_y in G^{k+1} (i.e., y represents I_h in G^{k+1}). Replace the equation for Y_y in G^{k+1} by the new reduced equation for Y_h, and mark it replaced.

(vi) Recalculate by substitution the solutions corresponding to each reduced equation in G^k that has been changed. That is, if the reduced equation for Y_j has been changed and is $Y_j = f(Y_h, \ldots, Y_r)$ for f linear, $\{h, \ldots, r\}$ interval head nodes, then substitute the values of $Y_h, \ldots, Y_r$ into this equation, and update the value of Y_j.

(vii(b)) Recalculate the solutions for all non-interval head variables in G^k whose reduced equations contain a dependence on a changed interval-head-variable solution.

3.4 Examples of ACINCF and ACINCB

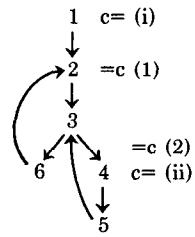
The following two examples use the flow graph in Figure 4 annotated with definitions ($c=$) and uses ($=c$) of variable c, which we assume are the only appearances of c in the program (see Figure 8). Using this program we trace an application of ACINCF to update a reaching-definitions problem and an application of ACINCB to update a live-uses-of-variable problem. Each of these problems is solved solely with respect to variable c.

Given the flow graph in Figure 8, we can easily calculate the p_j, d_j for each node. We see that $p_1 = p_4 = \emptyset$ and $p_2 = p_3 = p_5 = p_6 = \{\text{i ii}\}$. Also, $d_1 = \{\text{i}\}$, $d_4 = \{\text{ii}\}$, and $d_2 = d_3 = d_5 = d_6 = \emptyset$. Substituting those in the equations given in Figure 4, we obtain those in Figure 8. Since {i ii} are all the definitions of c in the program, $\{\text{i ii}\} \cap X = X$.

Suppose we add a definition of c to node 5 (iii). Then, $p_2 = p_3 = p_6 = \{\text{i ii iii}\}$, $p_5 = \emptyset$, $d_5 = \{\text{iii}\}$, and all other p_j, d_j are unchanged. Figure 9 traces ACINCF in updating the solution.

Given the flow graph in Figure 8, we can calculate r_j and u_j for each node. We see that $r_1 = r_4 = \emptyset$, $r_2 = r_3 = r_5 = r_6 = \{1\ 2\}$, $u_2 = \{1\}$, $u_4 = \{2\}$, and $u_1 = u_3 = u_5 = u_6 = \emptyset$. Substituting those in the equations given in Figure 5, we obtain those in Figure 10.

Suppose we delete the use of c at node 4 in Figure 8. Then, all the r_j are the same, u_4 changes from $\{2\}$ to $\emptyset$, and all other u_j are the same. Figure 11 traces ACINCB in updating the solution.



Flow graph

Equations on G^1:

$$X_1 = \emptyset$$
$$X_2 = \{i\} \cup X_6$$
$$X_3 = X_2 \cup X_5$$
$$X_4 = X_6 = X_3$$
$$X_5 = \{ii\}$$

Reduced equations on G^1:

X_1, X_4, X_5,and X_6
are already in reduced form
$X_2 = \{i\} \cup X_3$
$X_3 = X_2 \cup \{ii\}$

Equations on G^2:

$$X_1 = \emptyset$$
$$X_2 = \{i\} \cup X_3$$
$$X_3 = X_2 \cup \{ii\}$$

Equations on G^3:

$$X_1 = \emptyset$$
$$X_2 = \{i\ ii\}$$

Equations on G^4:

$$X_1 = \emptyset$$

Solution on G^4:

$$X_1 = \emptyset$$

Solutions on G^3:

X_1 is the same as on G^4
$X_2 = \{i\ ii\}$

Solutions on G^2:

X_1 and X_2 are the same as on G^3
$X_3 = \{i\ ii\}$

Solutions on G^1:

X_1, X_2, and X_3 are the same as on G^2
$X_4 = X_6 = \{i\ ii\}$
$X_5 = \{ii\}$

Fig. 8. Reaching-definitions example.

4. COMPLEXITY OF ACINCF AND ACINCB

In this section we discuss the complexity of ACINCF and ACINCB. First, we illustrate the inappropriateness of worst-case analysis for incremental update algorithms. Second, we explain our results, proved in the Appendix, bounding the equation update work of our algorithms on reducible flow graphs. Third, we describe the structured programming language L, which features Sail-like loop exits, and our analytic results on the performance of ACINCF and ACINCB applied to programs in L.[11] We are able to specify the effects of localized program

[11] Of course, our incremental algorithms can be applied to more general programs than those in L (e.g., program with **gotos**); only our complexity results are restricted to programs in L.

Add definition (iii) for c at node 5 in Figure 8.
p_5 changes from {i ii} to $\varnothing$, and d_5 changes from $\varnothing$ to {iii}.
Also the new value for p_2, p_3, p_6 is {i ii iii}. $T = \{3\}$.

On G^1, check the interval head reduced equation for X_3 obtaining

$$X_3 = X_2 \cup \{\text{iii}\},$$

which is different from its old value

$$X_3 = X_2 \cup \{\text{ii}\},$$

so we mark it in G^2.

On G^2, $3 \in I_2$, check the reduced equations in I_2, but find that X_3 is the last variable in the interval.
Check the interval head reduced equation for X_2 obtaining

$$X_2 = \{\text{i iii}\},$$

which is different from its old value

$$X_2 = \{\text{i ii}\},$$

so we mark it in G^3.

On G^3, $2 \in I_1$, check the reduced equations in I_1, but find that X_2 is the last variable in the interval.
No interval head variables are affected.
Resubstitute into changed reduced equation for X_2.
(Since $X_2 = \{\text{i ii}\}$, this has no effect.)

On G^2, set $X_2 = \{\text{i ii}\}$, the value of X_2 on G^2.
Resubstitute into reduced equations on I_2 obtaining

$$X_3 = \{\text{i iii}\} \cup \{\text{iii}\} = \{\text{i iii}\}.$$

On G^1, set $X_3 = \{\text{i iii}\}$, the value of X_3 in G^2.
Resubstitute into reduced equations on I_3 obtaining

$$X_4 = X_6 = \{\text{i iii}\}$$
$$X_5 = \{\text{ii}\}.$$

Updated solution on G^1:

$$X_1 = \varnothing$$
$$X_2 = X_3 = X_4 = X_6 = \{\text{i iii}\}$$
$$X_5 = \{\text{ii}\}$$

Fig. 9. Trace of ACINCF on the example in Figure 8.

changes, characterizing the equations possibly affected, both by the program structures corresponding to those nodes representing changed code and by their relation to the original program change sites.

Consider applying ACINCF to update the solution of the reaching-definitions problem on the digraph in Figure 12, where definitions of a are a=. Deletion of the definition of variable a at node 3 requires recalculation of all reduced equations in all the systems associated with this example; it also requires that all solutions to all equations be recalculated. Thus, applying ACINCF here is tantamount to reperforming the entire Allen–Cocke algorithm, $O(n^2)$ work on this graph. But these heavily nested-loop structures, an $O(n)$ nested loop on

Equations on G^1:

$$Y_1 = Y_2 \cup \{1\}$$
$$Y_2 = Y_3$$
$$Y_3 = Y_6 \cup \{2\}$$
$$Y_4 = Y_5$$
$$Y_5 = Y_3$$
$$Y_6 = Y_2 \cup \{1\}$$

Reduced Equations on G^1:

$$Y_1 = Y_2 \cup \{1\}$$
$$Y_2 = Y_3$$
$$Y_3 = Y_2 \cup \{1\ 2\}$$
$$Y_4 = Y_5 = Y_3$$
$$Y_6 = Y_2 \cup \{1\}$$

Equations on G^2:

$$Y_1 = Y_2 \cup \{1\}$$
$$Y_2 = Y_3$$
$$Y_3 = Y_2 \cup \{1\ 2\}$$

Equations on G^3:

$$Y_1 = Y_2 \cup \{1\}$$
$$Y_2 = \{1\ 2\}$$

Equations on G^4:

$$Y_1 = \{1\ 2\}$$

Solutions on G^4:

$$Y_1 = \{1\ 2\}$$

Solutions on G^3:

$$Y_1 = Y_2 = \{1\ 2\}$$

Solutions on G^2:

$$Y_1 = Y_2 = Y_3 = \{1\ 2\}$$

Solutions on G^1:

$$Y_1 = Y_2 = Y_3 = Y_4 = Y_5 = Y_6 = \{1\ 2\}$$

Fig. 10. The live-uses-of-variables problem on the flow graph in Figure 8.

n nodes, are uncommon in modern programming-language usage (private communication from F. E. Allen (1975) and [11, 19, 29]). Empirical surveys of Fortran, PL/1, and Pascal usage reported that shallow loops are common and that the maximum depth of a loop usually is bounded by a constant rather than by a function of the number of nodes in the program. Thus, worst-case analysis does not yield a realistic estimate of the complexity of our incremental algorithms on actual programs.[12] In this section we explore alternative analyses of these algorithms.

Since a loop is a strongly connected component of a flow graph, it is reasonable that a program change in a loop may affect every data-flow solution in that loop. The back-substitution work necessary to obtain those new solutions will be proportional to the number of nodes in the loop. This upper bound may occur in practice; however, the equation update work of the elimination phase on any of the derived systems of equations will be limited.

Intuitively, in a forward data-flow problem if there is an edge in the dependency graph between m, $m \in I_h$, and interval head node j, and the reduced equation for X_m is affected by a program change, then this change may affect the reduced equation for X_j. If I_h is a loop, then the change at X_m may affect the reduced equation for X_h as well. In the dependency graph of the derived system of equations, assuming h and j represent I_h and I_j, respectively, there is an edge (h, j). There are four alternatives:

(i) Both h and j are still interval head nodes.

(ii) h is an interval head node; j is not; $j \in I_h$ in the derived system.

(iii) j is interval head node; h is not.

[12] This argument holds for exhaustive analysis algorithms as well.

Assume we delete the use of c at node 4 (2) in Figure 8.
u_4 changes from $\{2\}$ to $\emptyset$ while the value of $r_4 = \emptyset$ remains the same. $T = \{3\}$.

On G^1, this affects the equation for interval head variable Y_3, which becomes

$$Y_3 = Y_6.$$

Then the reduced equation for Y_3 becomes

$$Y_3 = Y_2 \cup \{1\}$$

which is different from its old value

$$Y_3 = Y_2 \cup \{1\ 2\},$$

so we mark it in G^2.

On G^2, $3 \in I_2$, recalculate the reduced equation for Y_2:

$$Y_2 = \{1\},$$

which is different from its old value

$$Y_2 = \{1\ 2\},$$

so we mark it in G^3.

On G^3, $2 \in I_1$, recalculate the reduced equation for Y_1:

$$Y_1 = \{1\},$$

which is different from its old value

$$Y_1 = \{1\ 2\},$$

so we mark it in G^4.

On G^4, solve for Y_1:

$$Y_1 = \{1\}.$$

On G^3, set $Y_1 = \emptyset$, the solution for Y_1 on G^4.
(Since $Y_2 = \{1\}$, this has no effect in I_1.)

On G^2, set $Y_2 = \{1\}$, the solution for Y_2 on G^3.
Resubstitute the value of Y_2 into the reduced equation for Y_3 obtaining

$$Y_3 = \{1\}.$$

On G^1, set $Y_2 = \{1\}$, the solution for Y_2 on G^2, and set $Y_3 = \{1\}$, the solution for Y_3 on G^2.
Resubstitute the values of Y_2 and Y_3 into the reduced equations in I_3 obtaining

$$Y_6 = Y_5 = Y_4 = \{1\}.$$

Updated solution on G^1:

$$Y_1 = Y_2 = Y_3 = Y_4 = Y_5 = Y_6 = \{1\}$$

Fig. 11. Trace of ACINCB on the example in Figure 10.

(iv) Both h and j are not interval head nodes; existence of (h, j) implies h and j are in the same interval.

For example, alternative (iv) occurs when both h and j are entry nodes of simple **while** statements (i.e., **while** statements containing no other **while** statements);

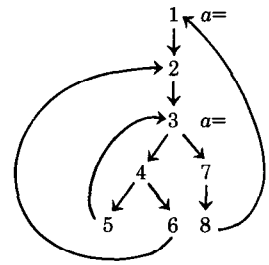


Fig. 12. Pathological digraph for the Allen–Cocke algorithm [47].

(iii) occurs when h is an entry node of a simple **while** statement and j is the entry node of a **while** statement containing another **while** statement.

Consider which reduced equations in the derived system can be affected by the change in the reduced equation of X_m in the previous system. In alternative (i), interval head nodes h and j are possibly affected. In (ii) and (iv), the changes are within one interval in the derived system, either I_h (ii) or I_r, where $h, j \in I_r$ (iv). In (iii), there are changes in I_r, $h \in I_r$, and to an interval head node j. These cases illustrate the general result proved in the Appendix in Theorem 1. For a forward data-flow problem, in any derived system, the equations possibly affected during the elimination phase will consist *only* of a set of interval head equations and, *at most*, the equations in *one* interval in the system.

In a backward data-flow problem, the reverse direction of the data flow and the single-entry property of intervals guarantees that only the reduced equation of one interval head node in each derived system can ever be affected by changes within an interval. That is, in any derived system the equations affected will consist *only* of the equations in *one* interval in the system. Theorem 2 is proved in the Appendix.

To refine our understanding of the equation updating process, we defined a "nontoy" structured programming language L that consists of straight-line code (e.g., **assignment, I/O** statements), **while** statements, compound **if** statements, **done** statements, and **continue** statements. We assume semantics for these statements similar to those in Sail [25]. The **done** statement causes control to pass to the statement following the syntactically innermost **while** loop containing that **done** statement. If a label appears in the **done** statement, then control passes to the statement following the **while** loop so labeled. The **continue** statement causes control to pass to the test of the syntactically innermost **while** loop containing that **continue** statement. If a label appears in the **continue** statement, then control passes to the test of the **while** statement so labeled.

Figure 13 illustrates the use of these statements. If the **done** C statement within loop C is executed, it causes control to pass to the statement following loop C, namely, the entry test of loop E. If the **continue** A statement within loop E is executed, it causes control to pass to the evaluation and test of the expression governing the execution of loop A. It is clear from this example that **done** and **continue** offer powerful structured loop-exit mechanisms, leading to programs considerably more complex than those constructed using single-entry/single-exit loops.

```
A: while exp do
  B: while exp do S1 endwhile B
  if exp then
    { C: while exp do
         (**)
         if exp then D: while exp do S2 endwhile D
           else done C
         endwhile C }
  E: while exp do
    (*)
    if exp then S3
      else continue A
  endwhile E
endwhile A
```

Fig. 13. Nested **while** statements.

For ease of description, in the remainder of this section we present our results under the assumption that all the intervals in an L program correspond to **while** loops.[13] Given a program written in L with a forward or backward data-flow problem solution and a set of localized program changes, we characterize the reduced equations that may need recalculation in terms of the program structures corresponding to these variables and their relation to the sites of the original changes, using the following definitions for nested loops and three binary relations defined on them. If the code for a **while** loop h is syntactically nested within the code for **while** loop w in a program in L, then loop h is a *descendant* of loop w, and loop w is a *parent* of loop h. If loop h is a descendant of loop w and loop h is not nested within any other descendant of loop w, then loop h is an *immediate descendant* loop of loop w, its *immediate parent.* In Figure 13 loop D is a descendant of loops A and C; loop C is an immediate descendant of loop A.

The *sibling* relation is a symmetric relation describing the nesting of two loops within the same immediate parent. In Figure 13 loop B is a sibling of loops C and E, but *not* a sibling of loop D, since loops B and D do not share the same immediate parent.

We can also describe the location of the code corresponding to one loop in the program, in relation to the code corresponding to another loop. Loop h is a *right sibling* (*rsib*) of loop w if the loops are siblings and the code for loop w precedes the code for loop h in the program. If loop h is an rsib of loop w, then loop w is a *left sibling* (*lsib*) of loop h. In Figure 13 loops C and E are rsibs of loop B.

We can distinguish further among the right (left) siblings of a loop by the point in the sequence of derived systems at which all the variables in an interval have been eliminated from the equations. Loop h is *right greater sibling* (*rgsib*) of loop w if loop h is an rsib of loop w and all the variables in loop w are eliminated from the derived sequence before all the variables in loop h. For example, if loop h is an rsib of loop w, node w is first eliminated from the derived sequence in G^k, node h is first eliminated in G^m, and $k < m$, then loop h is an rgsib of loop w. Loop h is a *right equal sibling* (*resib*) of loop w if loop h is an rsib

[13] This is true of virtually all intervals.

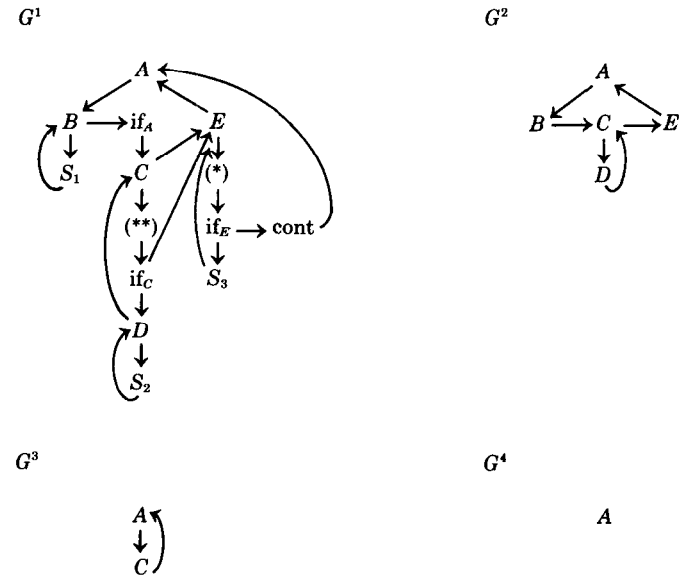


Fig. 14. Derived sequence for the example in Figure 13.

of loop w and all the variables in loop w are eliminated from the derived sequence at the same point as all the variables in loop h; that is, nodes w and h are first eliminated from the derived sequence in the same system G^k. In Figure 14 the variables in loop C still exist on G^3, whereas the variables in loop B have been eliminated. Therefore, loop C is an rgsib of loop B. Also, the variables in both loops B and E are eliminated first from the derived sequence in G^3; therefore, loop E is an resib of loop B. Corresponding definitions exist for *lgsib* and *lesib*; for example, loop B is an lesib of loop E.

Our results on the complexity of the elimination phase of ACINCF and ACINCB are proved in Theorems 3 and 4, respectively, in the Appendix; we illustrate them here (see Figures 15–19). These analytic results enable us to perform a priori analysis of data-flow effects of program changes; they limit the elimination work to a prescribed set of equations related to the loop structure of the program near the change sites. Since the program changes of widest impact occur in deeply nested loops, we consider the complexity of ACINCF and ACINCB on these structures. Also, we indicate the potential increase in complexity introduced by the various loop-exit statements.

To illustrate the effects of **continue** and **done** statements in increasing the equation update work of our incremental algorithm, consider the program in Figures 13 and 14. A program change at (*) for a forward data-flow problem can directly affect the reduced equation for X_A on G^1 because of the **continue** A statement. Without that **continue** statement, the effect would be propagated through loop E and would change the reduced equation for X_E. On G^2 this effect would be propagated to the reduced equation for X_A. Therefore, in general, the presence of a **continue** statement does not increase the number of affected

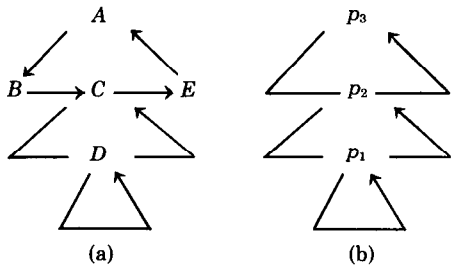


Fig. 15. Diagrams of nested loops. (a) Loops in Figure 13. (b) Nested loops.

variables, but can increase the number of equations to be recalculated for those variables.

Consider a program change at (**) for a forward data-flow problem. This change can directly affect the reduced equation of X_E because of the **done** C statement; without it, this equation cannot be affected. In the latter case, the program change would be propagated to the reduced equation for X_{if_C} and interval head variable X_D on G^1 (see Figure 14). On G^2 the effect would be transmitted to X_C since C and D are both in I_C there. On G^3, since all variables in loop E have been eliminated, no change to the equation for X_E would be possible.[14] This argument does not imply that data-flow information for loop E is unaffected by the program change if the **done** statement is not present. Rather, any data-flow effects in loop E will be calculated solely during the propagation phase of ACINCF. Thus, the presence of a **done** statement can increase the number of variables whose reduced equations will need recalculation in the elimination phase of ACINCF.

Figure 15 shows the diagrams that depict nested loops; each triangle represents an interval and a set of paths through that interval. Figure 15(a) depicts the doubly nested structure of loops A, C, and D from Figure 13. Loops B and E are represented by their interval head nodes. We only show relevant sibling loops. Figure 15(b) shows a k-nested loop for $k = 3$. The entry nodes of the nested loops are $p_k, \ldots, p_2, p_1$, ordered with p_k, the outermost loop entry node; loop p_i is an immediate descendant of loop p_{i+1}, $1 \leq i \leq k - 1$.

In Figure 16 we picture the behavior of ACINCF during a step of the elimination phase on a k-nested loop. Nodes corresponding to variables in loop p_k (i.e., the entire k-nested loop) with possibly affected reduced equations are indicated by dashed circles or lie on dashed paths. Assume that the reduced equation of X_q was changed in the previous system and that loop $q = p_n$ is represented by one node q in the current system. Then, there will have been a change to the equation for X_q in the current system. Consider which reduced equations in the current system may be affected by the changes in the equation for X_q.[15] There are four cases:

(i) If there are no **done** or **continue** statements in loop p_k, then possibly affected reduced equations correspond to entry nodes of the immediate

[14] Note that I_E is an rlsib of I_C and is the nearest rsib of loop C.

[15] The code in the k-nested loop p_k determines which of these possibly affected variables are actually affected by this change.

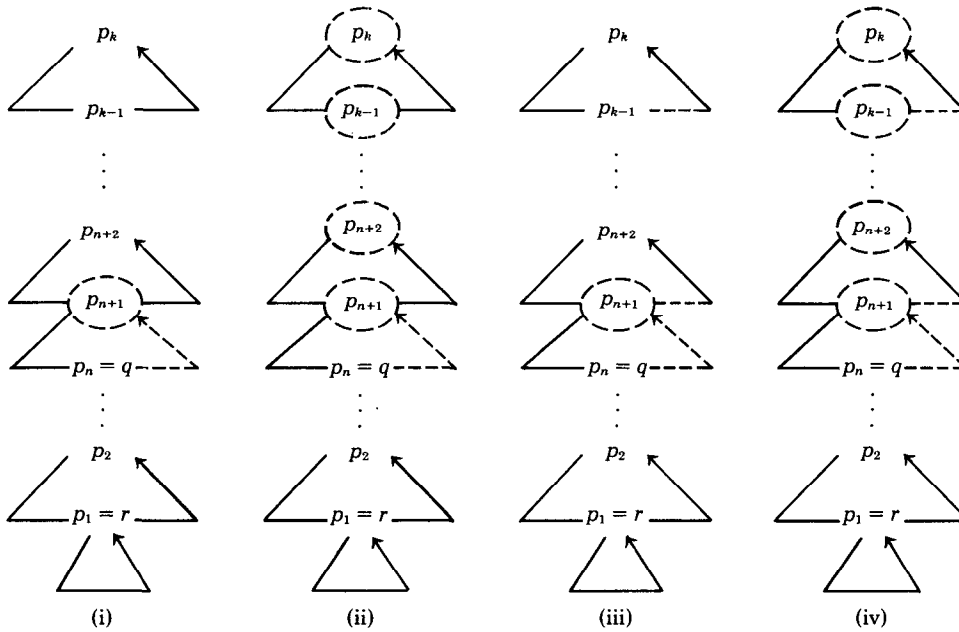


Fig. 16. A snapshot of ACINCF.

parent loop of loop q (i.e., loop p_{n+1}), *or* resibs/rgsibs (i.e., resibs or rgsibs) of q.[16]

(ii) If there are no **done** statements, but there are **continue** statements, then possibly affected reduced equations correspond to entry nodes of resibs/rgsibs of loop q *or* a subset of $\{p_{n+1}, p_{n+2}, \ldots, p_k\}$.

(iii) If there are no **continue** statements, but there are **done** statements, then possibly affected reduced equations correspond to the entry nodes of resibs/rgsibs of loop q, the immediate parent loop of q (i.e, p_{n+1}), *or* nearest rsibs of a subset of $\{p_{n+1}, p_{n+2}, \ldots, p_k\}$.[17]

(iv) If there are **done** and **continue** statements in loop p_k, then possibly affected reduced equations correspond to entry nodes of resibs/rgsibs of loop q, a subset of $\{p_{n+1}, p_{n+2}, \ldots, p_k\}$, *or* nearest rsibs of a subset of $\{p_{n+1}, p_{n+2}, \ldots, p_k\}$.

Figure 17 summarizes these results for ACINCF on a k-nested loop in a program in L. If program changes occur in I_r, then all the variables whose reduced equations may be affected during elimination correspond to nodes along the dashed paths in Figure 17. If no **done** statements occur within the loop p_k, the affected variables (besides those in loop r) correspond to the entry nodes of rgsibs/resibs of loop r, parent loops of loop r, *or* rgsibs/resibs of parent loops of

[16] Note that (i) ⊆ (ii), (i) ⊆ (iii), (ii) ⊆ (iv), and (iii) ⊆ (iv).

[17] In cases (iii) and (iv), if a loop has no rsib, the presence of a **done** statement has the same effect as a **continue** statement.

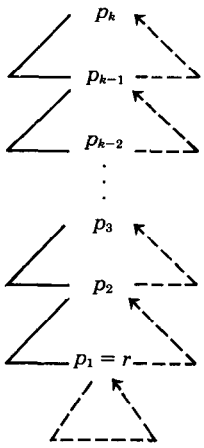


Fig. 17. ACINCF on a k-nested loop.

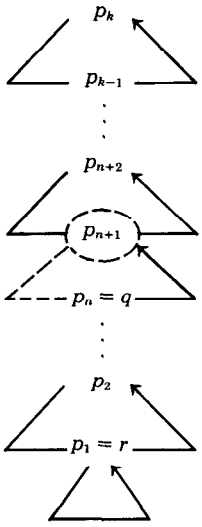


Fig. 18. A snapshot of ACINCB.

loop r. If **done** statements do occur, these variables correspond to the entry nodes of rsibs of loop r, parent loops of loop r, *or* rsibs of parent loops of loop r.

Figure 18 shows the same k-nested loop structure as Figure 16. It illustrates the behavior of ACINCB during a step of the elimination phase. Assume that the reduced equation of X_q is changed in the previous system and that loop q is represented by one node q in the current system. Irrespective of the type of loop-exit statements in loop p_k, the same variables can possibly be affected as a result of this change because the data-flow information travels in a direction opposite to control flow. The nodes corresponding to these affected variables are indicated by dashed circles or lie on dashed paths in Figure 18. They are the entry node of p_{n+1}, the immediate parent loop of loop q, *or* entry nodes of lesibs/lgsibs of loop q.

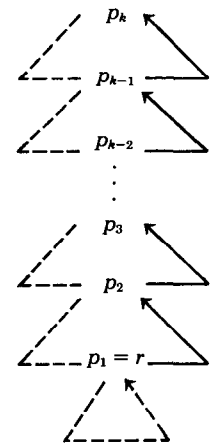


Fig. 19. ACINCB on a k-nested loop.

Figure 19 is analogous to Figure 17 and summarizes our results for ACINCB on a k-nested loop in a program in L. If program changes occur in I_r, then the variables whose equations may be affected correspond to nodes along the dashed paths in Figure 19. Any variables affected correspond to the entry nodes of a parent loop of loop r, an lgsib/lesib of a parent loop, an lgsib/lesib of loop r, or nodes in loop r itself. In this case the presence of **done** and **continue** statements does not affect our result; here, the effect of program changes is determined by the single-entry property of loops.

Empirical surveys that characterize high-level programming-language usage support our claim that L is a "nontoy," powerful structured programming language. Indeed, L seems a viable model of the "structured programming" style of the PL/1 programs reported in [12]. In this 1977 survey, Elshoff reported on the changes in programming style of PL/I programmers, after structured programming was taught at several General Motors commercial computation centers. He studied programs from before (NSP) and after (SP) instruction was given. The number of "**goto**-less" programs increased dramatically from 0 (NSP) to 32 (SP) percent. In a total of 67 percent of all the programs in the latter sample, **goto** statements constituted no more than 2 percent of all program statements. A 1980 survey of Pascal compilers written in Pascal corroborated this lack of **goto** statements in structured programs [44]. Therefore, our Sail-like **while** loop exits in L are not excessively restrictive; L is a reasonable model of a modern structured programming language.

Although our analytic results deal with L, a structured programming language, ACINCF and ACINCB work on a digraph representation of a program. They are applicable to any reducible digraph and therefore can handle programs containing **goto** statements as long as all loops are single-entry. Empirical evidence shows that irreducible flow graphs are rare. By suggesting interprocedural applications for our algorithms, we are hypothesizing that irreducible call graphs are also rare. Empirical studies are necessary to corroborate this hypothesis. Given the empirical results available to date, for reasonable programs ACINCF and ACINCB are clearly useful and workable.

5. EXPERIENCE WITH HUINC

Our model of Hecht–Ullman T1-T2 analysis describes how the variables in the data-flow equations are grouped in *regions*; the equation of each variable can be parameterized as a function of its region head variable. Mergers of regions eliminate variables from the system; these mergers are repeated until all the variables are within one region.[18] The solution for the final variable is obtained. Back substitution of region-head-variable solutions into the reduced equations for their region variables yields solutions for all the variables [40].

This process seems similar to interval analysis, but there is an important difference. Recall that, for a forward data-flow problem, all dependence on a variable in the system of equations is transformed by substitution transformations into a dependence on its interval head variable. When a variable is added to a region, all dependence on that variable *within* the region is transformed to a dependence on the region head variable; however, substitutions to eliminate dependence on that variable outside its region are *delayed* until that variable and the variable dependent on it are in the same region. This delay enables the Hecht–Ullman algorithm to avoid performing recalculation of common coefficient/constant factors in some of the reduced equations, which occur because of common substitution sequences. These delayed substitutions enable the Hecht–Ullman algorithm to solve data-flow problems defined on the pathological flow graph shown in Figure 12 in $O(n \log n)$ calculations, whereas the Allen–Cocke algorithm requires $O(n^2)$ [40].[19]

Some auxiliary data structure is necessary to contain the common factor information. The Hecht–Ullman algorithm uses a 2-3 tree, which complicates the incrementalization of the algorithm. Additional information has to be saved as the tree is constructed, so that we can re-create calculations performed on the partially constructed tree; that is, we must re-create its intermediate states. A related algorithm, which is an improvement over Allen–Cocke interval analysis, also uses auxiliary data structures [46].[20] In order to save this intermediate state information, additional storage costs are incurred. In HUINC the amount of storage needed is bounded by $O(T) + O(n\ e)$, where T is the size of the final 2-3 tree, n is the number of nodes in the original digraph, and e is the number of edges in the graph.

In order that sets of program changes may be accommodated before new data-flow solutions are obtained, any auxiliary data structure must be validated during updating; that is, all constants and coefficients needing recalculation must be so marked. This introduces a data-structure-dependent, time-complexity bound. For HUINC an additive factor of $O(t)$ appears in the time complexity, where t is the number of tree edges that must be marked invalid because of changed coefficient/constant calculations. A gross upper bound on $O(t)$ is $O(n\ e)$; however, from the examples we investigated this bound is not achieved on reasonable flow graphs.

[18] For irreducible digraphs a subgraph remains that is a double-entry loop; iterative analysis can be used to quickly solve the equations for classical problems in this case.

[19] These delayed substitutions are also utilized by Tarjan in his interval analysis algorithm [46].

[20] Another related technique [13] does not use an auxiliary data structure.

Our design methodology of developing incremental update algorithms by incrementalizing a model of the global data-flow algorithm worked better in ACINCF than in HUINC. Our aim in incrementalizing the Hecht–Ullman algorithm was to improve on ACINCF by experimenting with the less constrained variable substitution order, while maintaining the savings from the common substitution factors. Although we gained insight into the problems of incrementalizing an algorithm that uses an auxiliary data structure, HUINC was a disappointment [34]. Our performance analyses of HUINC indicate that the additional storage and execution costs introduced to maintain and validate the 2-3 tree during updating may dominate the complexity savings of the underlying data-flow analysis algorithm.

6. APPLICATIONS

ACINCF and ACINCB are particularly useful in software systems development. Here, interprocedural data-flow analysis algorithms provide information about global data and parameters. The call graph, the digraph statically modeling the procedure calls of a set of procedures, rarely undergoes structural changes; however, the component procedures and their use of global variables and parameters change often during system development. Thus, this is a suitable application for our incremental update algorithms. Applied to existing interprocedural elimination algorithms, our incremental update algorithms can provide powerful debugging and documentation information for the software system designer.

An important application of our incremental update algorithms is in the area of software maintenance. Large applications often maintain histories of source-code changes during system development and maintenance. Studies of these histories would yield information about the kinds of changes large systems are likely to undergo. This information alone would be valuable to software designers as the transformation of a set of algorithms and data structures into a working software system is not well understood in large practical applications, although various software design techniques exist. Our experience with the PFORT Verifier attests to the user's need for even the most rudimentary data-flow information with respect to interprocedural analysis of software systems [32, 33]. Incremental update algorithms for interprocedural data-flow analysis would enable us to delineate the scope of a system change, impossible today where often a "let's try it and see" attitude prevails. This information would be highly useful in system debugging and maintenance.

Currently we are working on a prototype implementation of interval-based incremental update algorithms applied to the modification problem of interprocedural data-flow analysis [35–39], as an example of how our techniques can be applied to flow-insensitive interprocedural summary problems in general. These algorithms are extensions of ACINCF and ACINCB, in that they use Tarjan intervals and handle structural as well as nonstructural changes to the call graph. There are as yet unanswered questions concerning the best representation for data-flow information, given that deletions and additions of variable definitions, procedure calls, procedure definitions, etc., will occur. We also must deal with structural changes that introduce irreducibilities into the call graph; so far, we can identify them, but do not attempt to continue incremental updating in their

presence. We expect to be answering more practical implementation questions for incremental algorithms as that work progresses.

We also plan to gather more current information on programming language usage to augment the empirical studies cited here. This would offer further evidence that our structured programming language L is a reasonable model of the loop structure of modern programming languages. It would enable us to concentrate our attention on program structures that are utilized sufficiently to ensure that our efforts to accommodate them in incremental updating will pay off.

7. SUMMARY

We have presented incremental update algorithms for data-flow analysis based on Allen–Cocke interval analysis; these algorithms handle nonstructural changes on reducible digraphs. Our complexity analyses assume all changes initially occur within one interval. We have shown the inappropriateness of worst-case bounds for these incremental algorithms. We have complexity results for our algorithms on a "nontoy," structured programming language L, which enable us a priori to characterize those variables whose equations are possibly affected by a set of localized program changes with respect to their corresponding program structures and the original site of the changes. These results verify the desirability of incremental update algorithms for data-flow analysis. We have also reported on our work on an incremental update algorithm based on Hecht–Ullman T1-T2 analysis and outlined the difficulties encountered.

To review, our incremental algorithms were designed by patterning an incremental update algorithm after a global data-flow analysis algorithm, reperforming only those steps necessary to ascertain the effects of a set of program changes. This methodology led to a simple incremental update algorithm for Allen–Cocke interval analysis; however, for Hecht–Ullman T1-T2 analysis it committed us to a complicated algorithm that used a height-balanced 2-3 tree to store the common factor information. All elimination algorithms that ascertain common factors need some data structure to store them. Even if we had used a more easily manipulated data structure, HUINC would have been more difficult to understand than ACINCF. Here the "slower" exhaustive algorithm (Allen–Cocke) is the better algorithm to incrementalize, leading to a simpler incremental algorithm; it is proved efficient by our complexity studies on L and is really not slower in practice. This is an example of behavior cited in [31], in which the best exhaustive algorithm in terms of worst-case complexity *does not necessarily* yield the best incremental algorithm. We call the Allen–Cocke algorithm slower with reservations, knowing that reasonable programs exhibit shallow loop-nesting depths.

The results of our complexity study of ACINCF and ACINCB on a nontoy, structured programming language with Sail-like loop-exit statements enable us to determine a priori which reduced equations may need recalculation because of a localized set of program changes (i.e., all changes within one interval in the flow graph). These possibly affected reduced equations are identified in terms of their corresponding program structures near the localized changes. Our results can be refined to characterize the reduced equations possibly affected when the

structured loop-exit mechanisms are used singly or together. Specifically, affected equations are characterized when the program features both **done** and **continue** statements, only **done** statements, only **continue** statements, and neither of these. Therefore, we can relate the richness of programming language usage to the ease of incremental updating, gaining insight into the influence of a variety of program structures on data flow. In addition, since we can determine a priori how much equation update work a set of program changes will entail, we have a powerful tool for software modification. We also obtained a characterization of the elimination phase work of ACINCF and ACINCB on any reducible digraph. Namely, we showed that, for any reducible digraph with a set of localized program changes, each digraph in its derived sequence has the equations of some interval head nodes and *at most* the equations of one interval possibly affected. Thus, if a program is reasonably structured, the update work required by a localized set of changes remains localized and relatively easy.

APPENDIX

This section presents proofs of the results described in Section 4; concepts used are described in that section. Theorems 1 and 2 describe the behavior of ACINCF and ACINCB on reducible flow graphs. They state that, given any reducible flow graph with a localized set of changes (i.e., all changes within one interval), the elimination effects generated on the derived systems of equations are limited; that is, in a particular derived system, the equations possibly affected consist *only* of a set of interval head equations and, at most, the equations in *one* interval in the system. Lemmas 1–3 concern the properties of Allen–Cocke interval analysis and our incremental update algorithms. Lemma 3 is stated for ACINCF with the changes necessary for ACINCB noted in roman type.

Lemmas 4–8 characterize the properties of programs written in L when interval analysis is applied to their flow graphs. Theorems 3 and 4 characterize the equations that may be affected by a set of localized program changes during elimination in terms of the relation of those nodes to the site of the program changes and to their corresponding control structures in the program.

LEMMA 1. *There exists a path in reducible digraph G from interval head node h to node t, free from any other interval head nodes if and only if noninterval head node $t \in I_h$ in G.*

PROOF. If there exists a path in G from h to t, free from interval head nodes except for h, the proof that $t \in I_h$ is by contradiction. If $t \notin I_h$ in G, then there is another interval I_q in G such that all immediate predecessors of t are in I_q, by the properties of the interval-finding algorithm in Figure 2. By finite induction, we can see that this implies q lies on the path from h to t. Therefore, our assumption was false, and $t \in I_h$, by contradiction.

If $t \in I_h$, the proof that the specified path from h to t exists is by the properties of the interval-finding algorithm of Figure 2. □

LEMMA 2. *Given a reducible digraph G, assume u represents I_q and v represents I_h in the derived graph of G. Then, the edge (v, u) exists in the derived graph if and only if there exists edge (m, q) in G for q, an interval head node, and $m \in I_h$.*

PROOF. The proof is by induction, using the properties of the interval analysis algorithm [34]. □

LEMMA 3. *Assume there is a sequence of systems of equations solved by Allen–Cocke interval analysis with its associated derived sequence. If m is a node in I_q in G^k such that the reduced equation for X_m is marked as changed initially or by step (iii) of ACINCF (ACINCB), then reduced equations updated in G^k due to this change correspond* either *to*

(*i*) *a node in I_q on a path from m*

  (*a*) *to an interval head node h,*
  (*b*) *back to q, or*
  (*c*) *to a terminal node in I_q*

  (a node in I_q on a path from q to m); *or*

(*ii*) *an interval head node*

  (*a*) *h, which is some interval exit target on a path in* (*i*(*a*)), or
  (*b*) *q itself*

  (*q* itself).

PROOF. The proof is by induction on the number of substitution transformations of equations within I_q and uses our incremental update algorithms from Sections 3.3.1 and 3.3.2. □

Theorems 1 and 2 limit the possible "fanout" of reduced equation updates performed by ACINCF and ACINCB during elimination. To prove Theorem 1, we make use of the diagram in Figure 20, which shows that, in G, ACINCF changes the reduced equations of interval head nodes $\{i_1, \ldots, i_n\}$ and may change the reduced equations of nodes in interval I_h, including h. Each diagram edge (h, i_j) indicates that in G in the equation for X_{i_j} there is a dependence on a variable X_m for some $m \in I_h$ (i.e., there is an edge (m, i_j) in G). In the proofs of Theorems 1 and 2, we assume I_r in G^i is represented by r in G^{i+1} for any interval head node r, unless otherwise specified.

THEOREM 1. *Given that*

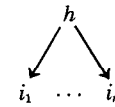
(*i*) *Allen–Cocke interval analysis is applied to a reducible flow graph G to solve a forward data-flow problem,*
(*ii*) *program changes occur all within one interval in G (i.e., localized), and*
(*iii*) *ACINCF is applied to determine the effects of the changes on the data-flow solution,*

then, in each derived system of equations and its corresponding G^i, $i \geq 1$, the elimination phase of ACINCF will update the reduced equations of

(*i*) *at most one interval I_h in that system, and*
(*ii*) *a set of interval head nodes H related to I_h as in Figure* 20.

If reduced equations in I_h are updated and v represents I_h in G^{i+1}, then, for every $j \in H$, edges (v, j) exist in G^{i+1}.

Fig. 20. Reduced equation changes in G.



PROOF. The proof will be by induction on the length of the derived sequence $\{G\}_{i=1}^{K}$.

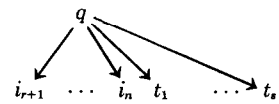
Basis. Assume all the initial equations changed in G^1 correspond to nodes within one interval I_h. By Lemma 3, the possibly recalculated reduced equations in I_h correspond to nodes on paths in I_h. If at least one path leads to an interval head node that is not h (i.e., i_j), then the diagram of Figure 20 shows the possible reduced equation changes in G^1; n is equal to the number of such interval head nodes. By Lemma 2 and the interpretation of the diagram in Figure 20, in G^2 there are edges (v, i_j), $1 \leq j \leq n$, where v represents I_h in G^2. If no such path exists, then the only interval head node possibly updated in G^2 is h; the diagram of Figure 20 with $n = 0$ still reflects the changes in G^1.

The remaining cases occur when there is only one program change at node h itself or when the reduced equation changes in G^1 do not propagate to any interval head nodes. The latter case is a trivial one in which the only reduced equation changes occur in I_h in G^1. The former case results in at most the reduced equation for X_h changed on G^1. Therefore, both cases are represented by Figure 20 with $n = 0$.

Thus, the theorem is satisfied on G^1, and Figure 20 correctly represents all reduced equation changes in G^1.

Induction Hypothesis. Assume the theorem is true on G^{k-1}. Assume the diagram in Figure 20 encapsulates the possible reduced equation changes in a system of equations corresponding to G^{k-1}. Our induction argument will show that a similar diagram describes the reduced equation changes in G^k.

Assume that $\{i_1, \ldots, i_r\}$ are no longer interval head nodes on G^k and that $\{i_{r+1}, \ldots, i_n\}$ are interval head nodes for some r, $0 \leq r \leq n$. The induction hypothesis and Lemma 2 imply that, if v represents I_h in G^k, then there are edges (v, i_j) in G^k for $1 \leq j \leq n$. Then, by the properties of the interval-finding algorithm in Figure 2, each i_j will be in the same interval in G^k as v for $1 \leq j \leq r$. Therefore, there is an interval I_q in G^k containing $\{v, i_1, \ldots, i_r\}$. According to Lemma 3(i), reduced equations of nodes in I_q may be changed due to the effects of the changes at $\{i_1, \ldots, i_r\}$. There also may be interval head nodes affected by these changes according to Lemma 3(ii). Call these interval head nodes $\{t_1, \ldots, t_s\}$ for $s \geq 0$. The equation for each t_j will contain a dependence on some variable X_m for $m \in I_q$. By Lemma 2, if w represents I_q in G^{k+1}, edges (w, t_j) exist in G^{k+1}. Therefore, the diagram for G^k is



where there are reduced equation changes in I_q, the reduced equation for X_q may change, and the reduced equations of interval head nodes $\{i_{r+1}, \ldots, i_n, t_1, \ldots, t_s\}$

are changed. By the induction hypothesis and the above arguments, diagram edges represent graph edges (w, i_j) and (w, t_i) in G^{k+1}. This satisfies our induction hypothesis on G^k. □

THEOREM 2. *Given that*

(*i*) *Allen–Cocke interval analysis is applied to a reducible flow graph G to solve a backward data-flow problem,*

(*ii*) *program changes occur all within one interval in G, and*

(*iii*) *ACINCB is applied to determine the effects of the changes on the data-flow solution,*

then, on each system of equations and its corresponding G^i, $i \geq 1$, *the elimination phase of ACINCB will update the reduced equations of*

(*i*) *an interval head node q; and*

(*ii*) *at most, nodes in one interval* I_q.

PROOF. The proof will be by induction on the length of the derived sequence, $\{G^i\}_{i=1}^{K}$.

Basis. By Lemma 3, if all the program changes in G^1 affect nodes within one interval I_q, then the reduced equations of nodes on paths from q to some changed node may need recalculation. Also, the reduced equation for X_q may need recalculation. If the only program change occurs within the code at node q, an interval head node in G^1, then the reduced equation for X_q is recalculated. Thus, the theorem is satisfied on G^1 by $\{q\ I_q\}$ or $\{q\}$.

Induction Hypothesis. Assume the theorem is true on G^{k-1}.

Assume the reduced equation for X_q is changed in G^{k-1}. If q is still an interval head node in G^k, steps (iii) and (iv) of ACINCB will update the equation for X_q in G^k. If q is not an interval head node in G^k, then there exists an interval I_y such that $q \in I_y$ in G^k. The reduced equations recalculated by steps (iii) and (iv) of ACINCB on G^k correspond to nodes on paths from y to q, by Lemma 3. The reduced equation of X_y is also recalculated.

In either case, the theorem is satisfied in G^k by $\{y\ I_y\}$ or by $\{q\}$. □

In these results for programs written in L, we use the term *g-loop* as a synonym for interval; not all g-loops are actually loops in the program, although they are always single-entry connected subgraphs.

LEMMA 4. *Nested g-loops in L are collapsed in innermost to outermost order (using the interval-finding algorithm in Figure 2). Each internal head node in* G^1 *is an entry node of a g-loop in* G^1, *and each node in* G^i, $i \geq 2$, *represents such an entry node.*

PROOF. The proof is by induction using the properties of the interval-finding algorithm in Figure 2, the interval analysis method, and the semantics of L. □

LEMMA 5. *All programs in L have reducible flow graphs.*

PROOF. All loops in L are single entry, and therefore, flow graphs containing them are reducible [14]. □

LEMMA 6. *A **done** statement in a program in L, syntactically nested within innermost **while** loop q, appears in the flow graph as an edge whose target is the entry node of the nearest*[21] *rsib of loop q or the nearest rsib of a parent loop of loop q in G^1. A **continue** statement in a program in L, syntactically nested within innermost **while** loop q appears in the flow graph as an edge whose target is the entry node of loop q or a parent loop of loop q. The targets of a **done** or **continue** statement are interval head nodes in G^1.*

PROOF. The proof uses the definition of rsib, the semantics of L, and Lemma 4. □

The *collapse index* of a g-loop is a numeric value that indicates when all the variables in that interval are eliminated from the derived sequence. If the equation for at least one variable in interval I_h exists in the systems $G^1, G^2, \ldots, G^j$ and if there is no equation for any variable in I_h in G^{j+1}, then the collapse index of g-loop h is j ($\mathrm{ci}(h) = j$). In Figure 14, $\mathrm{ci}(D) = 2$ and $\mathrm{ci}(C) = 3$.

LEMMA 7. *In G^1, if there exists a path from t, the entry node of g-loop t, to x, the entry node of g-loop x, and $ci(x) < ci(t)$, then for some G^k, $2 \leq k \leq ci(x)$, u represents I_t, v represents I_x, and there exists a path from u to v in G^k.*

PROOF. The proof is by finite induction using Lemma 2 and the definition of represents (see Section 2.3). □

LEMMA 8. *Assume g-loop x with $ci(x) = i$ is syntactically nested within a **while** loop in G^1. If $y \in I_h$ represents I_x in G^i, then g-loop h is either the immediate parent loop of g-loop x or the nearest lgsib of g-loop x.*

PROOF. We prove this by considering the two cases in turn.

(i) Assume g-loop x has a unique nearest lgsib g-loop t, using path length as a metric. By definition of lgsib, there is a path from the entry node of g-loop t to the entry node of g-loop x in G^1. Since $\mathrm{ci}(t) > i$, by Lemma 7 there is a path in G^i from u representing I_t to v representing I_x. Because by assumption g-loop t is the nearest lgsib of g-loop x, by Lemma 4 and the definition of lesibs, all nodes on this path correspond to lesibs of g-loop x and, therefore, correspond to collapsed g-loops in G^i. By the definition of a collapsed g-loop, there are no interval head nodes on this path. By the assumption that g-loop t is an lgsib of g-loop x and since $\mathrm{ci}(x) = i$, u is an interval head node in G^i. Thus, by Lemma 1, $v \in I_u$ in G^k.

Our assumption that g-loop t is the unique nearest lgsib to g-loop x is valid because, if there were more than one lgsib t_1 and t_2, the above arguments would imply the existence of a path to v from both u_1 and u_2 representing t_1 and t_2, free from interval head nodes. But u_1 and u_2 are interval head nodes in G^i by definition of lgsib. Therefore, by the properties of the interval-finding algorithm in Figure 2, v would be an interval head node. Therefore, by contradiction, our assumption of the uniqueness of g-loop t is valid.

[21] Use path length as a metric.

Hypothesis 1: I_h and H exist.
Hypothesis 2: H exists, but I_h does *not* exist.

hypothesis 1 → hypothesis 1 ∨ hypothesis 2
hypothesis 2 → hypothesis 1 ∨ hypothesis 2

Fig. 21. Proof outline for Theorems 3 and 4.

(ii) If g-loop x has no lgsibs, let loop p be the immediate parent loop of g-loop x in G^1. By definition, there exists a path in G^1 from the entry node of loop p to the entry node of g-loop x. By Lemma 4 and the definition of ci(x), q, which represents I_p in G^i, is an interval head node. By Lemma 7, there is a path in G^i from q to v that represents I_x in G^i. By our assumption of no lgsibs of g-loop x, Lemma 4, and the definition of lesib, any node on this path corresponds to an lesib of g-loop x and therefore to a collapsed g-loop. By the definition of a collapsed g-loop, there are no interval head nodes on this path. Thus, by Lemma 1, $v \in I_q$ in G^k. □

We prove Theorems 3 and 4 using strong induction. There are two mutually exclusive induction hypotheses. We assume hypothesis 1 true on G^{k-1} and show that each of a set of cases leads to hypothesis 1 or hypothesis 2 true on G^k. Similarly, we assume hypothesis 2 true on G^{k-1} and show that each of a set of cases leads to hypothesis 1 or hypothesis 2 true on G^k. The sets of cases for each hypothesis reflect the different possibilities for how the reduction process proceeds in ACINCF or ACINCB. For ACINCF the cases are defined by different assumptions on the current set of interval head nodes whose equations are affected. For ACINCB the cases are distinguished by different assumptions on the single interval head node involved. Assuming H is the set of interval head nodes whose equations are changed in G^{k-1}, and I_h is the interval whose equations are changed in G^{k-1}, then Figure 21 outlines the hypotheses and proof technique used in Theorems 3 and 4.

The results of these theorems enable us to perform a priori analysis of the data-flow effects of a program change. The theorems and their proofs, which follow, are lengthy and laborious. Their key results, which clearly state the elimination phase work in ACINCF and ACINCB, have been given in Section 4.

THEOREM 3. *Given a program P in L = {**while**, **done**, **continue**, **if** statements, and straight-line code}, solve a forward data-flow problem for P by Allen–Cocke interval analysis. Assume ACINCF is applied to update the solution with respect to program changes in P corresponding to nodes in interval I_r in G^1. Let loop w be the syntactically innermost loop containing I_r. (Note that $w = r$ is possible.) Then, the elimination phase of ACINCF updates, on a system of equations corresponding to G^i, are characterized as follows:*

(i) *The reduced equations recalculated in any one system of equations G^k, $1 \le k \le K$, are a set of equations of interval head nodes Q, and the equations of, at most, one interval in G^k, I_q.*

(ii) *If the reduced equations of the interval I_q are recalculated in G^k, $k \ge 1$, then the corresponding set of interval head nodes in Q depends on the relation of q to the initial changes in I_r in G^1 in the following ways (see Figure 22 for*

illustrations of these cases):

(*a*) *If q is a parent loop of g-loop r, then a set member corresponds to the entry node of an immediate descendant g-loop of q, which is*

(1) *an rgsib or resib of loop t, a parent loop of g-loop r, or*
(2) *an rgsib or resib of g-loop r (i.e., q = w);*

or

(*b*) *If q is a parent loop of g-loop r, then a set member corresponds to an rsib of g-loop q or to q itself. (Note: if no* **done** ⟨*q*⟩ *statement exists within loop q, the rsibs can only be resibs or rgsibs of loop q.)*

(*c*) *If q is an lgsib of v, which is a parent loop of g-loop r or an lgsib of g-loop r itself, then a set member corresponds to an resib or rgsib of v or r, respectively.*

(*d*) *In addition, whether q is a parent loop of g-loop r or q is an lgsib of v, which is a parent loop of g-loop r or an lgsib of g-loop r itself, a set member corresponds to an entry node of a parent loop of g-loop q or its rsib. (Note: if no* **done** ⟨*x*⟩ *statement exists within loop x, then the rsibs can only be resibs or rgsibs of loop x.)*

(*iii*) *If the interval I_q is recalculated in G^k for $k > 1$, then the nodes in I_q, whose reduced equations are updated, represent the entry nodes of*

(*a*) *parent loops of g-loop r and their rsibs, or*
(*b*) *immediate descendant g-loops of loop w that are resibs or rgsibs of g-loop r.*

(*iv*) *G-loop q is a parent loop of g-loop r, g-loop r itself, an immediate descendant g-loop of w, which is an lgsib of g-loop r, or an lgsib of any of these.*

PROOF

Outline. By Lemma 5 and Theorem 1, (i) is true for any program in L. The proof of (ii)–(iv) uses the principle of strong induction applied to the length of the derived sequence and follows the outline of Figure 21. First, we assume hypothesis 1, that in G^{k-1} ACINCF updates the equations in a nonempty interval I_h and the equations of a set of interval head nodes H. Second, we assume hypothesis 2, that in G^{k-1} our incremental algorithm updates only the equations of a set of interval head nodes H (i.e., I_h is empty). Each subcase of the hypotheses is illustrated in Figure 22. The nodes in Q lie on the dashed paths or are indicated by dashed circles. As in Figure 15, each triangle edge corresponds to at least one path through an interval; the label at the top vertex of the triangle names the loop-entry node corresponding to the interval head node. A vertical dotted line represents a sequence of triangles, each attached by its top vertex to the base of the triangle directly above. This represents a finite sequence of immediate-parent/immediate-descendant relations between the g-loops. The diagrams present examples of each case, not a comprehensive listing of all subcases that can occur.

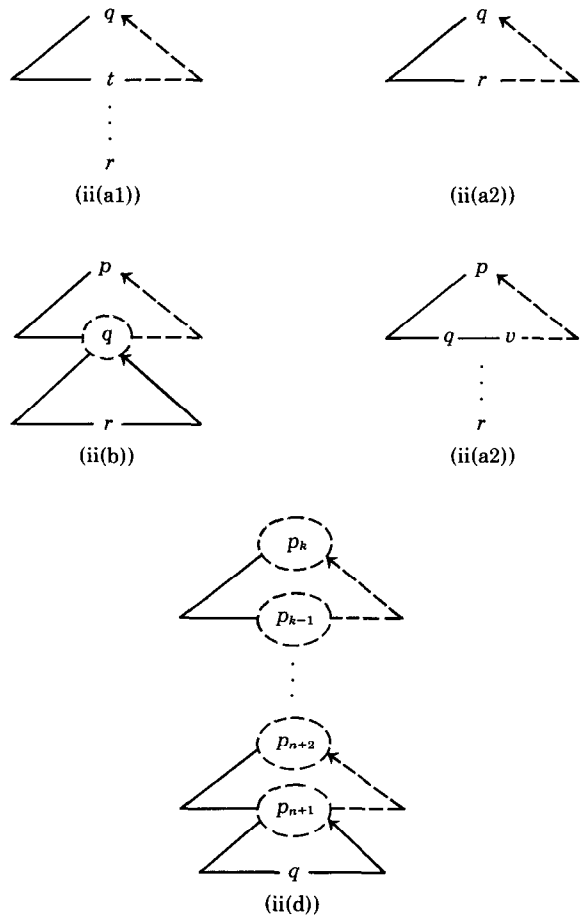


Fig. 22. Examples of Theorem 3(ii).

Basis

(i) If the only program change occurs in code at node r, an interval head node in G^1, then steps (iii) and (iv) of ACINCF recalculate the reduced equation for X_r. Here $Q = \{r\}$, and I_q is empty. This satisfies (iv).

(ii) Assume a program change occurs in the code at node $m \in I_r \in G^1$. Then, *either* loop w has an immediate descendant g-loop r, such that the code represented by its entry node syntactically precedes the code represented by node m in the program and g-loop r is the nearest such g-loop to node m, *or* $r = w$. In either case, $I_q = I_r$; this satisfies (iv).

Nodes can be added to Q in two ways:

(ii(a)) If there is a **done** or **continue** statement on a path from node m to an exit of I_r, then the reduced equation of the target node, p, is recalculated by ACINCF in step (iv). By Lemma 6, an interval head node p is the entry node of

a parent loop of g-loop r or a nearest rsib of a parent loop. Q contains $\{p\}$, which satisfies (ii(b)) or (ii(d)) if $r = w$, or (ii(d)) if g-loop r is an immediate descendant g-loop of loop w.

(ii(b)) If loop w contains an immediate descendant g-loop d and there is an edge from m, a node whose reduced equation has been updated in I_r, to the entry node of g-loop d, then the reduced equation for the entry node of g-loop d is recalculated. Here, Q contains $\{d\}$; this satisfied (ii(a)) if $r = w$, (ii(c)) if g-loop r is an immediate descendant g-loop of loop w.

Therefore, (ii) and (iv) are satisfied.[22]

Induction hypothesis. Assume the theorem holds for G^i, $1 \leq i \leq k - 1$. The following two-part induction argument shows that it holds for G^k.

(i) *Hypothesis* 1. Assume reduced equations in I_h and H are updated in G^{k-1}.

(i(a)) Use Figures 22 and 23 for reference in this proof. Assume H contains nodes satisfying (ii(a)). Let D contain all such nodes that correspond to collapsed g-loops in G^k.

If D is empty, then $Q = H$, and the induction hypothesis guarantees the theorem is satisfied.

If D is not empty, then by Theorem 1 and Lemmas 4 and 5, there exists a g-loop q such that all the nodes in D are in I_q in G^k. By Lemma 8, g-loop q is an immediate parent loop of or an lgsib with respect to each of the g-loops represented by the nodes in D. By Theorem 1 and Lemma 5, the edges (h, d) exist in G^k for all $d \in D$ and node h in G^k representing I_h in G^{k-1}. Since nodes in D represent immediate descendant g-loops of g-loop h, by Lemma 4 g-loop h cannot be collapsed in G^k. Therefore, by Lemma 1, each $d \in I_h$; here, $q = h$. By the induction hypothesis in h, g-loop q satisfies (iv).

By Lemma 3, the reduced equations recalculated in I_h include those of nodes on paths in I_h from the node representing some g-loop d in D to an exit of I_h, to h itself, or to a terminal node in I_h. By Lemma 4 and the definition of resib, these nodes represent resibs of the g-loop d. By the induction hypothesis that D satisfies (ii(a)), these nodes must be resibs and/or rgsibs of a parent loop of g-loop r or of g-loop r itself; therefore, they satisfy (iii).

In addition, there may be nodes in H satisfying (ii(b)) that are collapsed in G^k; call the set of these nodes E. By the same reasoning as above, each $e \in E$ is in I_h in G^k, and the reduced equations of nodes updated in I_h include those of resibs of nodes in E. Since nodes in E satisfy (ii(b)), the nodes with updated reduced equations satisfy (iii(a)).

There may also be nodes in H satisfying (ii(d)) that are collapsed on G^k; call this set F. By the same reasoning as above, each $f \in F$ is in I_h in G^k, and the reduced equations updated include those of resibs of nodes in F. Since nodes in F satisfy (ii(d)), these updated nodes satisfy (iii(a)).

Here,

$$Q = (H - D - E - F) \cup T \cup R,$$

[22] Note (iii) does not hold on G^1.

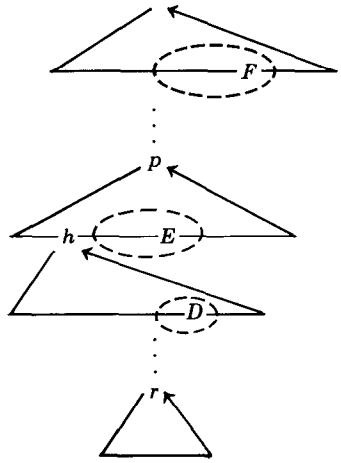


Fig. 23. Case (i(a)).

with the following conditions:

—$(H - D - E - F)$ should satisfy (ii) by the induction hypothesis of (ii) on H, since $q = h$.[23]

—T should contain entry nodes of g-loops that are branched to from **done** and **continue** statements within the nodes whose reduced equations are updated in I_h. By Lemma 6, these branch targets are entry nodes of parent loops or rsibs of parent loops of some $d \in D$, $e \in E$, or $f \in F$. By the arguments above, all nodes with recalculated reduced equations are resibs of a $d \in D$, an $e \in E$, or an $f \in F$, and resibs share the same parent loops. Therefore, since the nodes in D satisfy (ii(a)), the nodes in E satisfy (ii(b)), and the nodes in F satisfy (ii(d)) by assumption, each of these target nodes satisfies (ii(b)) or (ii(d)).

—R should contain the interval head nodes found in step (iii) of ACINCF. By Lemma 3, each of these nodes is an interval exit target of I_h or h itself. By definition of L, each node represents an rgsib of a node in I_h or the entry node of the immediate parent loop of a g-loop $d \in D$ (i.e., h), of g-loop $e \in E$ (i.e., the entry node of the immediate parent loop of g-loop h, loop p), or of g-loop $f \in F$ (i.e., the entry node of the immediate parent node of a parent of g-loop h). Each node in I_h is an resib of a node in D, E, or F. Therefore, each node in R is an rgsib of a node in D, E, or F; $\{h\}$ itself; or the entry node of a parent loop of g-loop h. By the induction hypothesis that D satisfies (ii(a)), E satisfies (ii(b)), and F satisfies (ii(d)), these rgsibs satisfy (ii(a)), (ii(b)), or (ii(d)). If $\{h\}$ is in R as well, it satisfies (ii(b)).

Therefore, (ii)–(iv) are satisfied by I_q and Q.

(i(b)) Assume H contains no nodes satisfying (ii(a)), but some nodes satisfying (ii(b)), (ii(c)), and (ii(d)). Let S contain all such nodes that correspond to collapsed g-loops in G^k.

[23] T contains backward exits from I_h; R contains forward exits from I_h and possibly h itself.

If S is empty, then $Q = H$, and the induction hypothesis guarantees the theorem is satisfied.

If S is not empty, by Theorem 1 and Lemmas 4 and 5, there is a g-loop q such that the nodes in S are in I_q in G^k. Also, edges (h, s) for each $s \in S$ exist in G^k, assuming that h in G^k represents I_h in G^{k-1}. There are three possibilities for g-loop q according to Lemmas 1 and 8:

(A) if g-loop h is not collapsed in G^k, $I_q = I_h$ in G^k;

(B) if g-loop h is collapsed, g-loop q is an lgsib of g-loop h; or

(C) G-loop q is an immediate parent loop of g-loop h.

By the induction hypothesis on h, each possibility for q satisfies (iv). The following four cases prove the induction hypothesis using the sets of assumptions given below:

Subcase	*satisfies*	*q satisfies*
(i(b1))	(ii(c)) V (ii(d))	A V B
(i(b2))	(ii(c)) V (ii(d))	C
(i(b3))	(ii(b)) V (ii(d))	B V C
(i(b4))	(ii(b)) V (ii(d))	A

These four cases are illustrated by examples in Figure 24, with the nodes in set R appearing on paths represented by dashed lines.

(i(b1)) Assume the nodes in S satisfy (ii(c)) or (ii(d)), so that h is an lgsib of g-loop r or an lgsib of a parent loop of g-loop r. Assume that $q = h$ (A) or that g-loop q is an lgsib of g-loop h (B). Then nodes in I_q whose reduced equations are updated are resibs of some g-loop s in S, by Lemmas 3 and 4 and the definition of resib. By transitivity and the induction hypothesis on S, the updated nodes are resibs or rgsibs of some parent loop of g-loop r, or of g-loop r itself. Therefore, these nodes satisfy (iii).

In addition,

$$Q = (H - S) \cup T \cup R \tag{6}$$

with the following conditions:

—$H - S$ should satisfy (ii) by the induction hypothesis on H.

—T should contain entry nodes of g-loops that are branch targets from **done** and **continue** statements within the nodes whose reduced equations are updated in I_q. By Lemma 6 and the fact that resibs share the same parent loops, these branch targets are entry nodes of parent loops or rsibs of parent loops of some $s \in S$. These nodes satisfy (ii(d)).

—R should contain the nodes found in step (iii) of ACINCF. By Lemma 3 and previous definitions, each node represents either an rgsib of a node in I_q, the entry node of loop p, the immediate parent loop of g-loop q, or the immediate parent loop of nodes in S. By assumption, every node in S satisfies (ii(c)) or (ii(d)); therefore, by transitivity, these rgsibs satisfy (ii(c)) or (ii(d)). If $p \in R$, then $\{p\}$ satisfies (ii(d)).

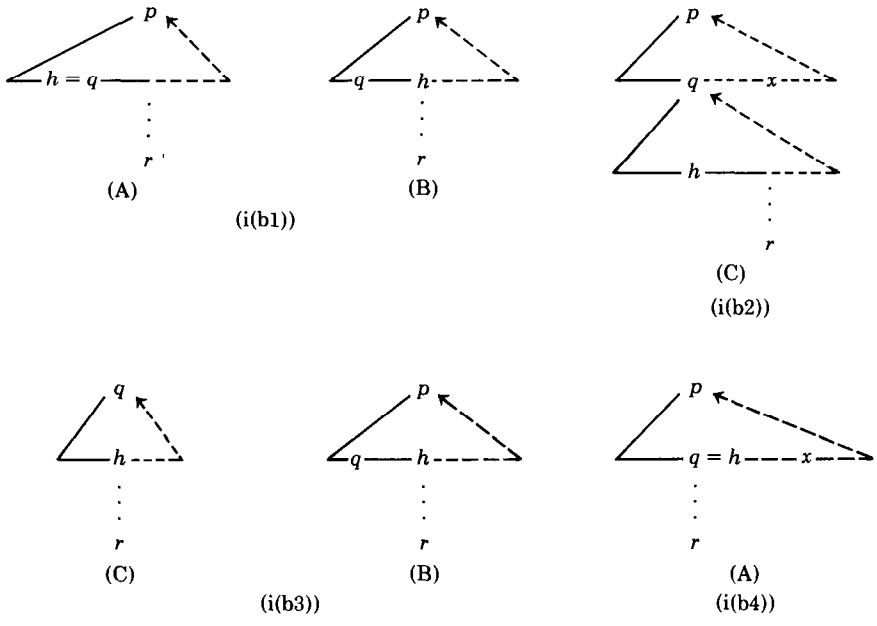


Fig. 24. Case (i(b)).

(i(b2)) Assume the nodes in S satisfy (ii(c)) or (ii(d)), so that h is an lgsib of g-loop r or an lgsib of a parent loop of g-loop r. Assume g-loop q is the immediate parent loop of g-loop h (C). Here, using eq. (6) to define Q, we see that nodes in $H - S$ that satisfied (ii(c)) will satisfy (ii(a)) and that nodes that satisfied (ii(d)) will satisfy (ii(b)) or (ii(d)) with respect to I_q.

T contains entry nodes of g-loops that are branch targets from **done** and **continue** statements within the nodes whose reduced equations are updated on I_q. By Lemma 6 and the fact that resibs share the same parent loops, these branch targets are entry nodes of parent loops or nearest rsibs of parent loops of some $s \in S$. These nodes satisfy (ii(b)) or (ii(d)).

By similar arguments to those in (i(b1)), R can contain rgsibs of nodes in S; q itself; p, the entry node of the immediate parent loop of loop q; and the immediate parent loops of rsibs of parent loops of loop q. These nodes satisfy (ii(a)), (ii(b)), or (ii(d)) with respect to I_q.

(i(b3)) Assume the nodes in S satisfy (ii(b)) or (ii(d)), so that h is a parent loop of g-loop r. Assume loop h is collapsed in G^k. By Lemma 8, $h \in I_q$ in G^k means that q represents the entry node of the immediate parent loop of h or an lgsib of g-loop h. By Lemmas 3 and 4, the definition of resib, the induction hypothesis on S, and transitivity, nodes in I_q whose reduced equations are updated represent resibs of an rsib of a parent loop of g-loop r and satisfy (iii(a)).

Using eq. (6) to define Q, we see that $H - S$ satisfies (ii) by the induction hypothesis, if g-loop q is an lgsib of loop h (B); otherwise (C), nodes in $H - S$ that satisfied (ii(b)) will satisfy (ii(a)), and nodes that satisfied (ii(d)) will satisfy (ii(b)) or (ii(d)) with respect to I_q.

By the same reasoning as in (i(b1)) and (i(b2)), if g-loop q is an lgsib of loop h (B), the nodes in T satisfy (ii(d)); otherwise (C), they satisfy (ii(b)) or (ii(d)). Similarly, if g-loop q is an lgsib of loop h, the nodes in R satisfy (ii(c)) or (ii(d)). If loop q is the immediate parent loop of loop h, these nodes satisfy (ii(a)) or (ii(b)).

(i(b4)) Assume the nodes in S satisfy (ii(b)) or (ii(d)), so that h is a parent loop of g-loop r (A). Assume g-loop h is not collapsed in G^k. By Theorem 1 and Lemmas 1 and 5, $q = h$. By Lemmas 3 and 4 and the definition of resib, nodes updated in I_h are resibs of some g-loop x in S and therefore, by transitivity, rsibs of loop h or rsibs of a parent of loop h. Here, (iii(a)) is satisfied.

In addition, using eq. (6), we see that $H - S$ satisfies (ii) by the induction hypothesis on I_h. By the same arguments as in (i(b1)), T contains nodes satisfying (ii(d)).

R contains nodes that, by Lemmas 3 and 4 and the definition of rgsib, are rgsibs of a g-loop x in S. By transitivity and the definition of rsib, these nodes satisfy (ii(b)) or (ii(d)).

Therefore, in all cases of (i(b)), (ii)–(iv) are satisfied by I_q and Q.

(i(c)) Assume H contains no nodes satisfying (ii(a)), (ii(b)), or (ii(c)). Since reduced equations in I_h were recalculated in G^{k-1} by the induction hypothesis, at best, g-loop h is collapsed in G^k. By Lemma 4, an immediate parent loop of g-loop h cannot be collapsed in G^k. Therefore, $Q = H$, and by the induction hypothesis on H, the theorem is satisfied.

(ii) *Hypothesis* 2. Assume there is no interval in which reduced equations are updated in G^{k-1}.

If H contains no nodes that are collapsed in G^k, then $Q = H$, and the induction hypothesis guarantees the theorem is satisfied.

If some nodes in H are collapsed in G^k, call the subset of such nodes C. By our induction hypothesis, there exists a G^N such that the equations in some interval I_h are updated in G^N, and only interval head equations are updated in G^i for $N < i \leq k - 1$. By Theorem 1 and Lemma 5, in G^{N+1} there are edges (h, c) for each $c \in C \subseteq H$, where h represents I_h in G^{N+1}. In addition, the nodes in C are in one interval in G^k.

Figure 25 presents examples of these cases.

If $\mathrm{ci}(h) > k$, then these edges (h, c) exist in G^k. Here, $q = h$.

If $\mathrm{ci}(h) = k$, then $h \in I_v$ in G^k. Therefore, by Lemma 1, there exists a path free from interval head nodes from v to h, where h in G^k represents I_h in G^{k-1}. By a finite number of applications of Lemma 2, edges (h, c) exist in G^k for all $c \in C$. Therefore, by Lemma 1, the $c \in C$ are also in I_v and $q = v$. By Lemma 8, g-loop v is an lgsib of or an immediate parent of g-loop h.

If $\mathrm{ci}(h) < k$, then g-loop h is represented in G^k by node x corresponding to the entry node of g-loop u where a finite number of applications of Lemma 8 can show that g-loop u is either an lgsib of g-loop h, a parent loop of g-loop h, or an lgsib of a parent loop of g-loop h. Similarly, using Lemma 2, we can show that the edges (h, c) in G^{N+1} for each $c \in C$ correspond to edges (x, c) in G^k. Therefore, $q = x$, and I_q contains all $c \in C$.

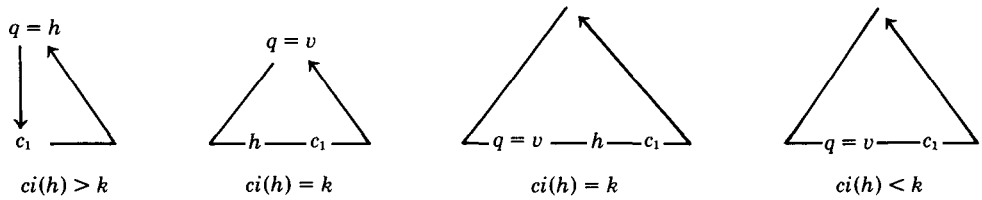


Fig. 25. Case (ii).

If $q = h$, q is a parent loop of g-loop h, or an lgsib of g-loop h or of a parent loop of g-loop h; the induction hypothesis ensures that q satisfies (iv).

If $q = h$ or q represents an immediate parent loop of g-loop h, then the $c \in C$ satisfy (ii(a)), (ii(b)), or (ii(d)) with respect to I_h in G^N, and the arguments of (i(a)) show that (ii)–(iii) are satisfied. If g-loop q is an lgsib of g-loop h, then the $c \in C$ satisfy (ii(c)) or (ii(d)) with respect to I_h in G^N, and the arguments of (i(b)) show that (ii)–(iii) are satisfied.

If g-loop q is a parent loop of g-loop h, which is not an immediate parent loop, or g-loop q is an lgsib of a parent loop of g-loop h, then the nodes in C satisfy (ii(d)) with respect to I_h in G^N. By Lemmas 3 and 4 and the definition of resib, the nodes whose equations are updated in G^k are resibs of some $c \in C$, each of which, by assumption, is a parent loop of g-loop h or an rsib of a parent loop. Thus, they satisfy (iii(a)).

In addition,

$$Q = (H - C) \cup T \cup R.$$

Any nodes in $(H - C)$ that satisfied (ii(d)) in G^N will satisfy (ii) with respect to g-loop q. T contains targets of **done** or **continue** statements in resibs of some $c \in C$. By the induction hypothesis on C, Lemma 6, and the fact that, by definition, rsibs share the same parent loops, these targets are parent loops of loop h or rsibs of such parent loops. Therefore, they satisfy (ii). R contains the nodes found in step (iii) of ACINCF. By Lemmas 3 and 4 and the definition of rgsib, each represents an rgsib of some $c \in C$ or possibly q itself. In each case, the nodes in C satisfy (ii); therefore, these nodes satisfy (ii). Thus, (ii)–(iv) are satisfied by I_q and Q. □

THEOREM 4. *Given a program P in L, solve a backward data-flow problem for P by Allen–Cocke interval analysis. Assume ACINCB is applied to update the solution with respect to program changes in P corresponding to nodes in interval I_r in G^1. Let loop w be the syntactically innermost loop containing I_r. Then, the elimination phase changes of ACINCB on a system of equations corresponding to G^i are characterized as follows:*

(*i*) *The reduced equations recalculated in any one system of equations corresponding to G^k, $1 \le i \le K$, are those of an interval head node q and, possibly, the equations of nodes in its interval I_q.*

(*ii*) *For G^k, $k \ge 1$, q is the entry node of a parent loop of g-loop r, an lgsib of a parent loop, g-loop r itself, or an lgsib of g-loop r.*

(*iii*) *For* G^k, $k > 1$, *the nodes in* I_q *whose equations are updated correspond to entry nodes of parent loops of g-loop r or lesibs or lgsibs of these, and/or lesibs or lgsibs of g-loop r.*

PROOF. The proof will apply the principle of strong induction to the length of the derived sequence. Theorem 2 and Lemma 5 imply that (i) is satisfied always.

Basis

(i) Assume there is only one program change in the code at node r, an interval head node. Then, by steps (iii) and (iv) of ACINCB, the reduced equation for X_r is updated in G^1. Thus, (i) and (ii) are satisfied.

(ii) If there is a program change in the code at node $m \in I_r$, then for g-loop r in loop w, *either* loop w has an immediate descendant g-loop r such that the code at the entry node of g-loop r syntactically precedes the code at m and g-loop r is the nearest such g-loop to m, *or* $r = w$. By Lemma 3 in both the former and latter cases, $I_q = I_r$, and the reduced equation for X_r is recalculated. Thus, (i)–(ii) are satisfied by $\{r\ I_r\}$ or $\{r\}$ on G^1.

Induction hypothesis. Assume the theorem is satisfied for $\{G^i\}_{i=1}^{k-1}$. The following two-part induction argument will show it is satisfied on G^k.

(i) *Hypothesis* 1. Assume the reduced equations of nodes in I_h and the reduced equation for X_h are changed in G^{k-1}.

If g-loop h is not collapsed in G^k, then only the reduced equation for X_h is changed in G^k. By the induction hypothesis on h, (ii) is satisfied with $q = h$.

If g-loop h is collapsed in G^k, then, by Lemma 8, $h \in I_q$ in G^k for g-loop q, an lgsib, or an immediate parent loop of g-loop h. By the induction hypothesis on h, q satisfies (ii). By Lemma 3, the reduced equations of nodes in I_q that lie on paths from q to h are recalculated. These nodes are lesibs of g-loop h by Lemma 4 and the definition of lesib. By transitivity and the induction hypothesis on h, these nodes satisfy (iii).

Thus, (ii)–(iii) are satisfied by $\{q\ I_q\}$ or $\{q\}$.

(ii) *Hypothesis* 2. Assume only the equation for X_h is changed in G^{k-1}. By the same arguments as in (i), (ii)–(iii) are satisfied by $\{q\}$ or $\{q\ I_q\}$. □

ACKNOWLEDGMENTS

We are grateful to the referees for their careful reading of our work and helpful comments; their input substantially improved the presentation. We especially thank Tom Marlowe for his improvements of the presentation.

REFERENCES

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
2. ALBERGA, C. N., BROWN, A. L., LEEMAN, G. B., MIKELSONS, M., AND WEGMAN, M. N. A program development tool. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 1981). ACM, New York, 1981, pp. 92–104.
3. ALLEN, F. E. Interprocedural data flow analysis. In *Proceedings of 1974 IFIP Congress.* IEEE Press, New York, 1974, pp. 398–402.

4. ALLEN, F. E., AND COCKE, J. A program data flow analysis procedure. *Commun. ACM 19*, 3 (Mar. 1976), 137–147.
5. BABICH, W. A., AND JAZAYERI, M. The method of attributes for data flow analysis, Part II: Demand analysis. *Acta Inf. 10* (1978), 265–272.
6. BABICH, W. A., AND JAZAYERI, M. The method of attributes for data flow analysis, Part I: Exhaustive analysis. *Acta Inf. 10* (1978), 245–264.
7. BANNING, J. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Langauges* (San Antonio, Tex., Jan. 1979). ACM, New York, 1979, pp. 29–41.
8. BURKE, M. An interval analysis approach toward interprocedural data flow analysis. Comput. Sci. Tech. Rep. RC 10640, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., July 1984.
9. BURSTALL, R. M., AND DARLINGTON, J. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 44–67.
10. COOPER, K., AND KENNEDY, K. Efficient computation of flow insensitive interprocedural summary information. Proceedings of SIGPLAN 84 Symposium on Compiler Construction. *SIGPLAN Not.* (ACM) *19*, 6 (June 1984), 247–258.
11. ELSHOFF, J. A numerical profile of commercial PL/I programs. *Softw. Pract. Exper. 6*, 4 (1976), 505–525.
12. ELSHOFF, J. The influence of structured programming on PL/I program profiles. *IEEE Trans. Softw. Eng. SE-3*, 5 (Sept. 1977), 364–368.
13. GRAHAM, S. L., AND WEGMAN, M. A fast and usually linear algorithm for global flow analysis. *J. ACM 23*, 1 (Jan. 1976), 172–202.
14. HECHT, M. S. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1977.
15. ISAACSON, E., AND KELLER, H. B. *Analysis of Numerical Methods.* Wiley, New York, 1966.
16. JOHNSON, G., AND FISCHER, C. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Conference Record of the 12th Annual ACM Symposium on the Principles of Programming Languages* (New Orleans, La., Jan. 1985). ACM, New York, 1985, pp. 141–151.
17. JOHNSON, S. C. LINT, A C program checker. In *UNIX System Programmer's Manual Vol. 2.* Holt, Rinehart and Winston, New York, 1979, pp. 278–290.
18. KIBLER, D. F., NEIGHBORS, J. M., AND STANDISH, T. A. Program manipulation via efficient production systems. *SIGPLAN Not.* (ACM) *12*, 8 (Aug. 1977), 163–173.
19. KNUTH, D. E. An empirical study of FORTRAN programs. *Softw. Pract. Exper. 1* (1971), 105–133.
20. LOVEMAN, D. Program improvement by source-to-source transformation. *J. ACM 24*, 1 (Jan. 1977), 121–145.
21. MASINTER, L. M. Global program analysis in an interactive environment. Ph.D. thesis, Dept. of Computer Science, Stanford Univ., Stanford, Calif., 1979.
22. OSTERWEIL, L. J., AND FOSDICK, L. D. DAVE—A validation, error detection, and documentation system for FORTRAN programs. *Softw. Pract. Exper. 6* (Sept. 1976), 473–486.
23. PAIGE, R. *Formal Differentiation—A Program Synthesis Technique.* UMI Research Press, Ann Arbor, Mich., 1981.
24. PAULL, M. C. *Algorithm Design: A Recursion Transformation Framework.* Wiley-Interscience, New York. In press, Spring 1988.
25. REISER, J., ED. SAIL. Memo AIM-289, Artificial Intelligence Laboratory, Stanford Univ., Stanford, Calif., Aug. 1976.
26. REPS, T. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M., Jan. 1982). ACM, New York 1982, pp. 169–176.
27. REPS, T., MARCEAU, C., AND TEITELBAUM, T. Remote attribute updating for language-based editors. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 1986). ACM, New York, 1986, pp. 1–13.
28. REPS, T., TEITELBAUM, T., AND DEMERS, A. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449–477.
29. ROBINSON, S. K., AND TORSUN, I. S. An empirical analysis of FORTRAN programs. *Comput. J. 19*, 1 (1976), 56–62.

30. ROSEN, B. K. High-level data flow analysis. *Commun. ACM 20*, 10 (Oct. 1977), 712–724.
31. ROSEN, B. K. Linear cost is sometimes quadratic. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan. 1981), ACM, New York, 1981, pp. 117–124.
32. RYDER, B. G. The PFORT verifier. *Softw. Pract. Exper. 4* (1974), 359–377.
33. RYDER, B. G. Constructing the call graph of a program. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 216–225.
34. RYDER, B. G. Incremental data flow analysis based on a unified model of elimination algorithms. Ph.D. thesis, Dept. of Computer Science, Rutgers Univ., New Brunswick, N.J., 1982.
35. RYDER, B. G. An application of static program analysis to software maintenance. In *Proceedings of the 12th Hawaii International Conference on System Sciences, Vol. II: Software* (Kona, Hawaii, Jan. 1987). University of Hawaii and University of S.W. Louisiana, in cooperation with the ACM and the IEEE Technical Committee on Computational Medicine. pp. 82–91. (Also available in abridged version as Tech. Rep. CAIP-TR-009, Center for Computer Aids for Industrial Productivity, Rutgers Univ., New Brunswick, N.J., July 1986.)
36. RYDER, B. G., AND CARROLL, M. D. A new structure for performing interval-based incremental data flow analysis. Tech. Rep. CAIP-TR-008, Center for Computer Aids for Industrial Productivity, Rutgers Univ., New Brunswick, N.J., May 1986.
37. RYDER, B. G., AND CARROLL, M. D. Interval-based incremental data flow analysis. Tech. Rep. CAIP-TR-007, Center for Computer Aids for Industrial Productivity, Rutgers Univ., New Brunswick, N.J., May 1986.
38. RYDER, B. G., AND CARROLL, M. D. An incremental algorithm for software analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Palo Alto, Calif., Dec. 1986), ACM, New York, 1986, pp. 171–179. (Also available in *SIGPLAN Not. 22*, 1 (Jan. 1987), and in abridged version as Tech. Rep. CAIP-TR-006, Center for Computer Aids for Industrial Productivity, Rutgers Univ., New Brunswick, N.J.)
39. RYDER, B. G., AND CARROLL, M. D. An incremental analysis algorithm for software systems. Tech. Rep. CAIP-TR-035, Center for Computer Aids for Industrial Productivity, Rutgers Univ., New Brunswick, N.J., May 1987.
40. RYDER, B. G., AND PAULL, M. C. Elimination algorithms for data flow analysis. *ACM Comput. Surv. 18*, 3 (Sept. 1986), 277–316.
41. SCHWARTZ, J. T. Interprocedural optimizations. SETL Newsl. 134, Courant Institute of Mathematical Sciences, New York Univ., New York, July 1, 1974.
42. SCHWARTZ, J. T., AND SHARIR, M. A design for optimizations of the bitvectoring class. Comput. Sci. Tech. Rep. 17, Courant Institute of Mathematical Sciences, New York Univ., New York, Sept. 1979.
43. SHARIR, M., AND PNUELI, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, N.J. 1981, pp. 189–234.
44. SHIMASAKI, M., FUKAYA, S., IKEDA, K., AND KIYONO, T. An analysis of PASCAL programs in compiler writing. *Softw. Pract. Exper. 10*, 2 (Feb. 1980), 149–158.
45. TARJAN, R. E. Testing flow graph reducibility. *J. Comput. Syst. Sci. 9* (1974), 355–365.
46. TARJAN, R. E. Fast algorithms for solving path problems. *J. ACM 28*, 3 (July 1981), 594–614.
47. ULLMAN, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Inf. 2*, 3 (1973), 191–213.
48. ZADECK, F. K. Incremental data flow analysis in a structured program editor. Ph.D. thesis, Dept. of Mathematical Sciences, Rice Univ., Houston, Tex., 1983.
49. ZELLWEGER, P. T. An interactive high-level debugger for control flow optimized programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging* (Pacific Grove, Calif., Mar. 1983), ACM, New York, 1983, pp. 159–172. (Also in *SIGPLAN Not. 18*, 8.)

Received July 1983; revised May 1984, January 1985, October 1986, and July 1987; accepted September 1987