University of Central Florida

STARS

Retrospective Theses and Dissertations

1983

Incremental Analysis of Programs

Vida Ghodssi
University of Central Florida

Part of the Computer Sciences Commons
Find similar works at: https://stars.library.ucf.edu/rtd
University of Central Florida Libraries http://library.ucf.edu

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
Ghodssi, Vida, "Incremental Analysis of Programs" (1983). *Retrospective Theses and Dissertations*. 682.
https://stars.library.ucf.edu/rtd/682

Incremental Analysis of Programs

by

Vida Ghodssi

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science at
the University of Central Florida
Orlando, Florida

December 1983

Major Professor: Dr. Charles E. Hughes

ABSTRACT

Algorithms used to determine the control and data flow properties of computer programs are generally designed for one-time analysis of an entire new input. Application of such algorithms when the input is only slightly modified results in an inefficient system.

In this thesis a set of incremental update algorithms are presented for data flow analysis. These algorithms update the solution from a previous analysis to reflect changes in the program. Thus, extensive reanalysis of programs after each program modification can be avoided.

The incremental update algorithms presented for global flow analysis are based on Hecht/Ullman iterative algorithms. Banning's interprocedural data flow analysis algorithms form the basis for the incremental interprocedural algorithms.

ACKNOWLEDGEMENTS

I would like to thank Dr. Charles Hughes, my advisor, for his guidance, technical advice and his patience which made completion of this work possible.

I would also like to thank the members of my committee, Drs. David Workman, Ratan Guha, Ronald Dutton and Christian Bauer for their reading and helpful suggestions.

TABLE OF CONTENTS

LIST OF FIGURES vi

Chapter

1. INTRODUCTION 1

 1.1 Motivation 2
 1.2 Thesis Overview 7
 1.3 Basic Concepts and Applications 8
  1.3.1 Global Flow Analysis 10
  1.3.2 Interprocedural Analysis 13

2. SURVEY 15

 2.1 Data Flow Analysis 15
  2.1.1 Global Flow Analysis 15
  2.1.2 Interprocedural Analysis 18
 2.2 Incremental Algorithms 22
 2.3 Metrics 24
 2.4 Testing 27
 2.5 Static Analysers 29

3. INCREMENTAL DATA FLOW ANALYSIS 32

 3.1 Global Flow Analysis 33
 3.2 Exhaustive / Incremental
   Reaching Definition 33
 3.3 Exhaustive / Incremental Live
   Variable Algorithms 38
 3.4 Analysis Of The Update Algorithms 40
  3.4.1 Time Complexity 43
  3.4.2 Space Complexity 52
 3.5 Validity of the Incremental
   Algorithms 59

4. INTERPROCEDURAL ANALYSIS 70

 4.1 Aliases 70
 4.2 Exhaustive Alias Calculation 71
  4.2.1 Basic Terminology 72

4.2.2 Alias Algorithm 75
4.2.3 Analysis of ALIAS 79
4.3 Incremental Alias Computation 81
4.3.1 Candidates for Possible Aliases 81
4.3.2 Incremental Alias Addition 82
4.3.3 Time Analysis of ADD-ALIAS 83
4.3.4 Validity of ADD-ALIAS 86
4.3.5 Incremental Alias Deletion 87
4.3.6 Time Analysis of REMOVE-ALIAS 97
4.3.7 Validity of REMOVE-ALIAS Algorithm 98
4.3.8 Space Complexity for Incremental Alias Computation 102
4.4 Necessary Aliases 105

5. SIDE EFFECT CALCULATION 107

5.1 Flow Insensitive side-effects 110
5.1.1 Exhaustive Algorithm 110
5.1.2 Incremental Algorithm 112
5.1.3 Time Complexity 114
5.1.4 Space Complexity 120
5.1.5 Validity of the Side-Effect Algorithm 122
5.2 Flow Sensitive side Effects 124
5.2.1 Exhaustive Algorithm 124
5.2.2 Incremental Algorithm 127

6. CONCLUSIONS 129

6.1 Summary 129
6.2 Future Directions 133

APPENDIX: APPLICATION TO SOFTWARE DEVELOPMENT SYSTEMS 135

GLOSSARY 148

LIST OF REFERENCES 155

LIST OF FIGURES

1. Phases of the life cycle 3

2. Cost, error sources and error discovery per phase of the life cycle 4

3. Reaching definition algorithm 35

4. Incremental update algorithm for reaching definitions 39

5. Live variable analysis algorithm 41

6. Incremental live variable analysis algorithm 42

7. Example of a flow graph 48

8. Solution of the example in Figure 7 using Hecht's algorithm 49

9. Solution of the example in Figure 7 using incremental reaching definition algorithm 50

10. Occurrence of a local change in the example of Figure 7, and the updated solution 51

11. Control flow change in the example of Figure 7, and the updated solution 53

12. Reformulation of the incremental reaching definition algorithm of Figure 4 60

13. Banning's algorithm to compute alias information 76

14. A function to test for aliases 77

15. A procedure to record aliases 78

16. Incremental update algorithm to compute alias information after addition of a new call site 84

17. Incremental update algorithm to compute alias information after deletion of a call site 89

18. Function to check the impossibility of alias 90

19. Example for case (a) 93

20. An example for case (b) 94

21. An example for cases (c) and (d) 96

22. A procedure for calculation of incremental flow insensitive side effects 115

23. A procedure for converting DMOD(s) into MOD(s) 116

CHAPTER 1

INTRODUCTION

Program flow analysis is a technique that gathers information about properties of computer programs without the actual execution of them. These properties include the determination of all possible sequences of control and data flow, and other such information which is impossible to find by individual runs of the programs. The algorithms used for program flow analysis are of the exhaustive type. That is, they are designed for the one time analysis of an entire new input. Determination of flow properties of computer programs is a costly and time consuming process. This overhead is even more noticeable when the program is only slightly modified. In such cases, there is a need for algorithms that update the results of a previous analysis, rather than exhaustively analysing the entire program.

Incremental flow analysis is a technique that avoids extensive reanalysis of programs after each modification. Incremental algorithms update the solution from a previous analysis to reflect changes in the program. That is, they propagate the program changes without application of the exhaustive algorithms. The development of incremental update algorithms for data flow analysis is the topic of this thesis.

1.1. Motivation

Information obtained from flow analysis of computer programs has traditionally been used by the optimization phase of compilers and as an aid in the debugging process. More recent applications involve the use of flow analysis in the development of new techniques for software reliability.

The most obvious use of incremental flow analysis algorithms is in interactive program construction environments as an aid in the debugging process. In such environments immediate response is essential and thus analysis has to be done incrementally.

A more important application of incremental flow analysis is in the software development process in order to enhance the reliability of produced software. In such an environment, extensive reanalysis of programs after each modification is unreasonable due to the cost and effort involved.

The most pressing problem facing software developers is the escalating cost of software which is in complete contrast to the trend in hardware cost. To determine the source of software costs, several studies have been conducted on the actual cost of the various phases of a program's life cycle (Glass 1979, Boehm 1978). A brief overview of these phases of the life cycle is given in Figure 1 and some of the more interesting results of the studies mentioned above are displayed in Figure 2. These studies have shown that the majority of errors are generated in the design phase. Yet these errors are not detected until very late in the life cycle. This results in a high maintenance cost,

(1) Functional Specification
This is the problem analysis phase. In general it results in a partial problem solution in document form.

(2) Design
The Functional specification document is analysed and a plan for the complete solution that meets the requirement is given in the software design document.

(3) Implementation
The software design document is translated into a program. The result of this phase is a software system that has yet to be debugged.

(4) Testing
This stage involves the examination of the software system to ensure that it meets the standards, requirements and design. It seeks to find as many programming and design errors as possible.

(5) Maintenance
The purpose of this phase is to keep the working software operational. It is the process of being responsive to user needs - fixing errors, making user specified modifications and in general making the program more useful.

Figure 1. Phases of the life cycle.

PHASE	COST	ERRORS GENERATED	ERRORS DETECTED
Specification	10%		
Design	10%	61-64%	
Implementation	10%	36-39%	
Testing	20%		46%
Maintenance	50%		54%

Figure 2. Cost, error sources and error discovery per phase of the life cycle.

since the cost of fixing such errors rises rapidly during the late phases of the life cycle.

To overcome these problems, there is a need for continuous analysis and comparisons of both the design and source codes. Such analysis permits the collection of essential flow informations which can be subsequently used to analyse the status of the software projects. Obviously, incremental program analysis algorithms are essential in performing the desired continuous analysis in a reasonable time frame, with minimum cost.

Incremental flow analysis algorithms can be used to analyse the design if it is coded in a suitable design specification language. The

results of such analysis combined with metrics during the design phase can help in detecting problem spots at this early stage. Bad module design, poorly designed data structures, and inadequate refinements are examples of errors which can be detected at the design level. Such analysis may result in redesign or modification of the original design.

The second major source of errors is the implementation phase of the life cycle. Incremental data flow analysis can aid in detection of the majority of errors generated during program construction. The algorithms used for such analysis can detect data flow anomalies which are generally the source of deeper errors. When such analysis is done incrementally, it is possible to alert the programmers of anomalies early in the program construction process while the intention is still fresh in their minds.

Other problems in software development include the complexity and general unreliability of software. These can be remedied to a certain extent by the analysis at the design phase. The quantitative measurements at the design level result in a better design which is a prerequisite for reliable software. Information flow measurements at the implementation phase can help in evaluating the complexity of the software. Correction of problem spots during the program construction period results in less complex and easier to test and maintain software. Obviously in the implementation phase, where programs are constantly modified, incremental algorithms are essential.

Incremental flow analysis can also assist programmers during the testing and maintenance phases of the life cycle. More reliable test data can be selected, if information about a program's data and control flow is available. Moreover, incremental flow analysis facilitates software testing throughout the program construction period. During the maintenance phase, incremental flow analysis can be used to assist programmers in determining the global effects of a localized modification. Such information can be obtained by a demand driven flow analyser which must clearly be based on incremental update algorithms.

We believe another problem that is receiving very little attention in present software development environments is supervision of the program development process. Presently, management control is based upon close interaction between project leaders and programmers. This is clearly impractical in a large scale program development environment where a project leader is possibly in charge of several projects and many programmers at the same time. Incremental program flow analysis can alleviate this problem, by being the basis for a set of automated tools to assist the managers. That is, the results of analysis of the source program provide the means of construction of such tools. A few examples of such tools can be found in the Appendix where we overview how our research can lead to the development of a new software development environment based on incremental program analysis.

1.2. Thesis Overview

Beyond this introductory chapter, the thesis is organized as follows. In Chapter 2, a survey of the literature pertinent to this research is presented. The major contribution of our research is described in Chapters 3-5. In Chapter 3, the exhaustive and incremental algorithms for global flow analysis are presented. The problems considered are reaching definitions and live variable analysis. The incremental update algorithms are based on Hecht/Ullman iterative algorithms.

The algorithms for computing aliases are described in Chapter 4. The exhaustive algorithm for computing aliases designed by Banning is described. Two incremental update algorithms for computing possible aliases are presented to deal with insertion and deletion changes separately. The incremental computation of necessary aliases is also discussed in this Chapter.

In Chapter 5, the possible side-effects of a procedure call are described and exhaustive and incremental algorithms for computing them are presented. Banning's algorithms for computing side-effects form the basis for our incremental algorithms.

Chapter 6 discusses the conclusions drawn from this research and indicates possible extensions of our work.

The rest of this current Chapter is devoted to explanation of the basic terminology and a discussion of various applications of program

flow analysis. The Glossary at the end of this report summarizes all notational conventions that we have adopted.

1.3. Basic Concepts and Applications

The term **flow analysis** refers to pre-execution analysis of computer programs. This process involves **control flow analysis** and **data flow analysis**, where control flow analysis is in general, but not necessarily, a prerequisite for data flow analysis.

Control flow analysis involves the construction and representation of the program's control flow structure. The calling relationships among the procedures of a program are generally represented by a directed graph named a **call graph**. Each node of a call graph corresponds to a procedure and each arc(p, q) represents a call from procedure p to procedure q. In contrast a **reverse call graph** is a directed graph in which each node corresponds to a procedure and each arc(p, q) represents a call from procedure q to p.

The possible flow of control within each procedure is usually represented by a directed graph called a **control flow graph** or simply a **flow graph**. The statements in a procedure are partitioned into maximal groups such that no transfer occurs into a group except to the first statement in the group and once the first statement is executed, all statements in the group are executed sequentially. Each of these groups is referred to as a **basic block** or simply a **block**. Each node of a flow graph corresponds to a block of the procedure and each

arc(x, y) represents a potential transfer of control from block x to block y.

A **reducible flow graph** is one that can be decomposed uniquely into a graph with no cycles and backward arcs.

The nodes of the flow graph are usually numbered according to some order on them. One such ordering technique is called the **depth first ordering**. The depth first ordering of the nodes of the flow graph is created by starting at the initial node and searching the entire graph, trying to visit nodes as far away from the initial node as fast as possible (depth first). The reverse of the order in which we visit the nodes by this search results in their depth first ordering.

Data flow analysis is the process of gathering information about the modification, preservation and uses of variables in a program. This information gathering process can be performed on any of the high, intermediate or low-level representations of a program.

For some applications such as source level optimization, anomaly detection and automatic documentation, data flow analysis is performed at the source level. For other applications such as code improvement, data flow analysis is typically performed on an intermediate representation of programs such as quadruples. This process, which is machine independent, is incorporated into optimizing compilers. Machine dependent optimization is performed on the low-level representation of programs.

Data flow analysis consists of two problems **global flow analysis** and **interprocedural analysis** which will be described in the next two sections after a discussion of some general applications for flow analysis.

The information obtained from flow analysis can provide the programmer with knowledge about unreachable code, unused variables and variables that are used before being defined. Information concerning all uses of each definition and all definitions affecting each use can be used in interactive debuggers. For each procedure, information about variables that are used or modified can be described. The information concerning the transitive effects of each procedure can facilitate program modifications and maintenance.

Another application of flow analysis is in program improvement. The information obtained from flow analysis of programs is used to improve the efficiency of program execution. Data flow analysis is performed as a preliminary step in the determination of useless code, common subexpression analysis for elimination of redundant computations, constant propagation to replace run-time computations, code motion for removal of invariant computations from loops and providing register allocation information.

1.3.1. Global Flow Analysis

In analysing programs a class of problems can be distinguished, each of which can be solved in essentially the same manner. These

problems are generally referred to as global flow analysis problems. We describe two problems in this category and indicate some of the applications of such information.

The two problems are called **reaching definitions** and **live variable analysis**. To give a preliminary definition of these problems, we assume all relevant information is available for a particular procedure and we have a control flow graph for this procedure.

In the reaching definitions problem, we wish to determine the set of definitions that can reach the top of each node x, (**IN[x]**) and the set of definitions that can reach the bottom of node x, (**OUT[x]**). The equations used to compute the reaching definitions are

$$IN[x] = \bigcup_{y \in \text{predecessors of } x} OUT[y]$$

$$OUT[x] = (IN[x] - KILL[x]) \cup GEN[x]$$

By **GEN[x]**, we mean the set of definitions that are generated in each node x and can reach the end of node x. **KILL[x]** is the set of definitions outside of x that define variables which also have definitions within x.

For live variable analysis, we wish to determine the set of variables that are live at the top of each node x, (**IN[x]**), and the set of variables that are live at the bottom of each node x, (**OUT[x]**). The equations used to compute live variables are

$$OUT[x] = \bigcup_{s \in \text{successors of } x} IN[s]$$

$$IN[x] = (OUT[x] - DEF[x]) \cup USE[x]$$

The set **DEF[x]** refers to a set of variables that are defined in node x. The set **USE[x]** represents the set of variables that are used in x, prior to any definitions of that variable in x.

From the solutions of reaching definitions and live variables, the solutions of two other problems called "live definitions" and "definitions-use chaining" can be obtained.

A definition is live at the top of a node x if it reaches and defines a variable that is live at the top of that node. The set of live definitions can be used when assigning registers: registers holding dead definitions can be reused immediately.

Definition-use chaining refers to a linking process between the set of definitions that reach the top of node x and the set of variables that are used in x and between the set of variables live at the bottom of node x and the set of definitions that are generated in x. This double linking combined with other local data flow information can be used to determine for a given definition, what uses can be affected by it and for each use, what definitions can affect it. Such information is useful for dead code elimination, constant propagation and anomaly detection. For example, if a given definition affects no uses, that definition can be removed. If all definitions reaching a particular use are the same constant, we can use this fact to perform constant propagation.

For a particular use, we can detect the anomalous situation where the variable used is undefined at that point.

1.3.2. Interprocedural Analysis

Global flow analysis presupposes that local information is immediately available. Unfortunately this is not true in the presence of procedures and procedure calls. The aim of interprocedural analysis is to determine the effects of a procedure call on the variables of a program and to associate this information with the call statement. The effects which we will consider fall into two categories: **variable side-effect** and **aliases**.

Variable side-effects can be classified according to various patterns of referencing and modifying variables. For each call site s, we determine the set of variables whose value may be modified by an execution of s, (**MOD(s)**), the set of variables whose value may be referenced by an execution of s, **(REF(s)**), the set of variables whose value may be referenced by an execution of s before being defined by an execution of s, (**USE(s)**) and the set of variables whose value must be defined by every execution of s, (**DEF(s)**).

MOD is the most useful variable side-effect and can be employed in many of the optimization processes. The process involved in computing REF is easier than that for USE. Moreover, With the exception of live variable analysis, REF and USE are interchangeable in most

contexts. DEF and USE side-effects are mainly used for live variable analysis.

All methods for computing side-effects of a procedure call finds approximations to the solution due to the fact that the perfect determination of side-effects is an undecidable problem. An approximation to a side-effect for a call site is said to be **precise up to symbolic execution** under the assumption that any path through a procedure can be taken when the procedure is called and that the path taken is independent of the call which invoked the procedure.

The other problem in interprocedural analysis is the determination of aliases of variables. It is possible for two variables to refer to the same memory locations at the same time. When this occurs the two variables are aliases of one another and accesses to one variable have the same effect as accesses to the other.

Information obtained from interprocedural analysis can aid in automatic documentation of the source code and facilitate program modifications and maintenance. In addition, any application for global flow analysis is also an application for interprocedural analysis.

CHAPTER 2

SURVEY

In this Chapter we will survey the more important literature related to this research. The background presented here includes some material that is essential to this study and other that is relevant to the potential applications of our research. In particular, this dissertation does not directly address the topics of metrics, testing and static program analysis. Rather, these are areas that can directly benefit from the application of the algorithms developed here.

2.1. Data Flow Analysis

The literature in this area falls into two general categories: the global flow analysis techniques and the interprocedural analysis methods.

2.1.1. Global Flow Analysis

Several forms of algorithms for solving global flow problems can be found in the literature.

One approach is the **interval analysis method** which was developed by Allen and Cocke (Allen 1976, Hecht 1977, Kennedy 1981). Interval analysis collects the relevant information by partitioning the

flow graph of the program into subgraphs called intervals. Given a node h, an interval I(h) is the maximal, single entry subgraph in which h is the only entry node and in which all closed paths contain h. This method replaces each interval by a single node containing the local information within that interval. It continues to find such interval partitions until the graph becomes a single node. At this time global information is propagated locally by reversing the partitioning process. The interval analysis approach works only on reducible flow graphs.

In a similar technique, Hecht and Ullman (1972, 1974) introduced two transformations on program graphs. Transformation T1 is used for removal of loops and transformation T2 is used for merging a node having a unique predecessor with that predecessor. It is demonstrated that when T1 and T2 are repeatedly applied to a graph, the graph is often reduced to a single node. This is in turn used to show that interval analysis is a special case of this more general **reduction method**. Also the result of this analysis has lead to a number of characterizations for reducible flow graphs. A similar algorithm which is based on three transformations is presented in (Graham 1976). This algorithm requires time at worst proportional to (e log e) for a flow graph with e edges. Both the transformation algorithms work only on reducible flow graphs.

Another approach and perhaps the simplest one is the **iterative method** which works on all types of graphs. This technique propagates information in an iterative manner until all required information is

collected ; that is until the process converges. The worst case time bound of iterative algorithms is $O(n^2)$, for a program having n nodes. Several variations of the iterative algorithms can be found in the literature.

The **worklist approach** to iterative algorithms due to Kildall (Hecht 1977) maintains a worklist of nodes to be visited. The worklist is initialized, updated as the algorithm executes and is eventually exhausted. In this version nodes are visited in an arbitrary order. The worklist approach has been further studied by Kam and Ullman (1976, 1977), who present a generalization of Kildall's algorithm.

The **node listing** version (Kennedy 1975) first obtains a list of nodes whose visitation suffices to propagate information. It then propagates information by visiting nodes in the order in which they occur on the list. The node listing, in which nodes are possibly repeated, is calculated such that every simple path in the graph is a subsequence of the list. Aho and Ullman in (1975) show that for reducible flow graphs an $O(n \log n)$ length node listing can be found in $O(n \log n)$ time.

The **round-robin** version of the iterative algorithms (Hecht 1977, Hecht 1975, Aho 1977) propagates information by starting with an initial estimate of the desired information. It then propagates information by repeatedly visiting the nodes in a round-robin fashion until a fixed point is reached. Nodes are visited in the depth first order in this version. Kennedy (1976) has done some detailed comparison of this algorithm and the interval analysis method. This study shows that

interval analysis requires fewer bit vector operations, but is still $O(n^2)$ in the worst case. His study also shows that in practice the simple and easier to implement iterative method may prove faster.

The **method of attributes** developed by Babich (1978a, 1978b) is a high level technique which operates on a parse tree representation of the program. The general approach of the method of attributes is this: at the time the source language is being defined, a set of attribute rules is written for each control structure. These rules summarize the runtime flow of control induced by the structure. The set of rules associated with the grammar production is applied whenever the production appears in the parse tree of a program. **High level data flow analysis** techniques have also been studied by Rosen (1977).

2.1.2. Interprocedural Analysis

There are several approaches to interprocedural analysis. Hecht (1977) presents a number of traditional methods for such analysis. In general these methods can be characterized as pessimistic and inefficient. Worst case, complete expansion and one pass methods are examples of more traditional approaches to interprocedural analysis.

A more recent approach developed by Barth (1977, 1978) takes composition and transitive closure of relations which can be directly constructed from the source program. These relations are found in terms of relationships among procedures and variables excluding any

consideration of subcalls. Such properties are referred to as direct relations.

Two methods for determining MOD, USE and DEF (see Glossary for a definition of these and other notations used here) information, considering aliasing effects and no aliasing effects are presented. MOD and USE are derived precisely up to symbolic execution in the absence of reference parameters. The computation of DEF is not precise up to symbolic execution due to the fact that only one pass is made over the source text and this is not enough to find all the effects associated with procedure calls and the interprocedural flow. The formulas given for calculating MOD and USE in the presence of reference parameters (with aliasing considered) compute this summary information less precisely than the former method. The imprecision arises from the fact that different calls on the same procedure are not handled separately and that aliases are not determined precisely.

For DEF, no new formula is given. Barth assumes that aliasing effects are limited in real programs and there is no obvious way to calculate them for DEF because of the "MUST" characteristic of this relation. He also states that, for achieving a correct formula for DEF, it is unnecessary to consider aliasing effects and that this formula is correct, even though slightly less precise than possible.

Aho and Ullman (1977) present a similar method for computing aliases of variables and calculation of MOD. Aliases are computed by taking the transitive closure of actual-formal correspondences. MOD is computed by taking the union of the set of global and formal param-

eters of a procedure with that of the procedures it calls. Their method does not deal with the nesting of procedures.

Banning (1978, 1980) presents two completely separate algorithms for computing aliases and side effects of procedure calls. His algorithm for computing aliases is presented in Chapter 4 of this paper and will not be dealt with here. Banning characterizes the side effects of concern, MOD, REF, USE and DEF, by considering how the side effect of a collection of statements is derived from the elements of the collection.

MOD and REF are characterized as flow insensitive and are computed precisely up to symbolic execution by making one pass over the source text. The basic method for finding the flow insensitive side effects uses the standard data flow techniques. This method involves solving a flow problem on a program's reverse calls graph (Graham 76). A generalized side effect is assigned to each node as follows. First, an initial approximation to the generalized side effect, GMOD, is assigned to each node. For example, to find GMOD, the initial approximation is IMOD which is the set of variables immediately modified by that procedure. Secondly, a function is assigned to each edge, which describes how the calling procedure's side effect depends on the called procedure's side effect. Then the meet over all paths solution is found which is the Generalized side effect for that procedure. The side effect of a call is then derived from the procedure's side effect.

This method is slightly extended to find DEF and USE which are flow sensitive. However, since flow sensitive side effects depend on the

flow through a procedure as well as the statements in the procedure, this method cannot find them precisely. The imprecision arises from the fact that not enough information about the statements within a procedure is considered in the calculation and that, in the presence of aliases, DEF information can not be calculated precisely. Banning's method for computing side effects is further described in Chapter 5 of this paper.

Rosen (1979) provides the only method to compute both may and must information precise up to symbolic execution. Although an algorithm for finding aliasing information is not given, the effect of aliasing is considered in the calculation of side effects. His method for finding MOD, USE and DEF is complicated, but is precise in the presence of recursion and reference parameters. Like Banning's algorithm, the algorithm provided by Rosen derives information specific to each call. The source of precision is due to the fact that this algorithm considers the local control flow graph of each procedure. The local information is associated with each arc of the graph and is represented as a formula. An initial guess to the values of MOD, USE and DEF is taken to be zero. These initial guesses are then improved by an iterative technique, which uses the direct local flow and parameter passing information. He proves that this guess eventually stabilizes and that the fixed point is the desired information.

Lomet (1977) presents a method for calculating MOD, USE, and DEF which is very similar to the method given by Rosen. His method is less precise, because he computes the side effects for procedures

assuming no aliases. The side effect for the calls are then computed from this information and the aliases created by the call. Lomet does not provide an algorithm for computing aliases.

Myers (1981) presents an algorithm to compute must and flow sensitive summary interprocedural information. He states that the interprocedural live problem is NP-Complete and that avail and must-summary problems are intractable due to the presence of aliasing. But the degree of exponentiality is small. The program model used is a super graph in which the flow graph for each procedure is linked by calls. All alias sets are found by initially taking a local variable which is the alias of itself as an alias set and then repeatedly applying an incarnation propagation function to this basis and all its offsprings until no new sets can be generated. In this way, he is able to find the alias sets for each separate incarnation of a procedure. An iterative technique is used to find the LIVE, AVAIL and MUST summary informations. To determine this information, his technique involves the propagation of alias sets rather than variables. The process of iteration converges when the meet over all paths solution of the super graph is found.

2.2. Incremental Algorithms

There has been remarkably little research on the development of incremental algorithms other than for use in language based program development systems.

Incremental attribute evaluation algorithms are used in the Cornell Program Synthesizer (Demers 1981, Reps 1982 ; 1983). The syntax directed editor of this system is based on an attributed tree representation of the source code. The task of the incremental attribute evaluator is to update the attribute values in the tree after each program modification. The incremental attribute evaluator finds and reevaluates inconsistent attribute instances and then propagates the changes by following attribute dependencies.

Another incremental system is the IPE component of the Gandalf project (Habermann 1980, Medina-Mora 1981). IPE is composed of a syntax directed editor, an incremental program translator and a language oriented debugger. The program is internally presented in two forms : syntax tree and machine representations. The syntax tree is built by the editor and is the common program representation for all the tools in IPE. As the programmer is incrementally changing the tree representation of the program, the IPE system incrementally updates and maintains an executable version by automatically applying the translation phase to program pieces and incorporating them on the target machine. The debugging facility of IPE is implemented using the incremental modification mechanism, i.e., incremental update, translate and load. The code generator provides the mapping from the tree representation to the machine representation.

Incremental algorithms for global flow analysis have been designed by Ryder (1982). The original algorithms considered are based on reduction methods and the only program modifications

allowed are those which result in local changes within a node. Changes in the control flow structure are not considered.

The only other incremental program analysis algorithm that we know of is that for parsing of deterministic context free languages (Ghezzi 1979).

2.3. Metrics

Several recent studies in software engineering have focused attention on the development and validation of a set of quantitative metrics to measure the complexity of software structures. These metrics are useful management aids and important design tools.

One type of metric is based on the lexical content of a program. Studies here include Halstead's work (1977), which counts the number of operators and operands, the McCabe's cyclomatic complexity measure (McCabe 1976), which counts the number of predicates in the code, and the logical complexity measure reported by Gilb (1977), which counts the number of if statements in the program.

Another type of metric is based on the flow of information or control among system components. The work of Oviedo (1980) determines the program complexity in terms of the control flow and data flow complexities. The control flow complexity is the number of edges in the flow graph. The data flow complexity of the program is the sum of the data flow complexities of each node. To determine the data flow complexity of each node, two sets are computed : the set of definitions

which can reach the node and the set of locally exposed variables within the node. The data flow complexity of node n is then the number of prior definitions of locally exposed variables in n that can reach n. The total program complexity is then defined as the sum of the control flow and data flow complexities of the program.

The research of Henry (1979, 1981) is another example of metrics based on information flow techniques. In this work the procedure, module and interface complexities are computed. To compute the procedure complexity, the complexity of procedure code and the complexity of the procedure's connections to its environment are determined. The code complexity is defined as the number of lines of code. The complexity of the procedure's connection to its environment is calculated as

$$(\text{fan-in} * \text{fan-out})^2$$

where fan-in of a procedure is the number of local flows into that procedure and fan-out of a procedure is the number of local flows from that procedure. The formula defining the procedure complexity measure is

$$\text{length} * (\text{fan-in} * \text{fan-out})^2.$$

The procedure complexities are used in turn, to establish module complexities. A module is defined with respect to a data structure D to consist of those procedures which either directly update D or directly retrieve information from D. The module complexity is then

calculated as the sum of the complexities of the procedures within the module. The interface measurements focus on the interfaces which connect system components. The formula given to measure the strength of the connections from module A to module B is

> (the number of procedures exporting information from module A + the number of procedures importing into module B) * the number of information paths .

The coupling measurements show the strength of the connections between two modules and are derived by applying the above formula to the following factors :

(1) The direct flow of information from module A to module B,

(2) The flow of information from module A to the transfer procedures (these procedures are not in any module and their only purpose is to transfer information from A to B), and

(3) The flow of information from the transfer procedures to module A.

2.4. Testing

The technical literature on software testing falls into two main categories : those that deal with the study of theoretical foundation of program testing and those that deal with development of new techniques for producing reliable test data.

The theory of reliable testing addresses the conditions under which a test can be considered equivalent to a program's formal proof of correctness. Goodenough and Gerhart (1977) define an ideal or a reliable test as one that satisfies a valid and reliable test data selection criteria. The successful execution of a reliable test would then demonstrates program correctness. In other words, a successfully executed reliable test is said to be equivalent to a direct proof of correctness. These ideas are further studied by Weyuker (1980). Howden (1976) states that an effective testing strategy which is reliable for all programs can not be constructed.

The techniques for test case design are of two kinds : "black-box" or functional testing and "white-box" or structural analysis techniques (Myers 1976, Miller 1981, Adrion 1982). In black-box testing, test data is derived completely from the external specification of the software, whereas in the white box testing, it is derived completely from the internal specification of the program. In the following we will deal with the literature on white-box testing which is of more importance to this research.

Myers (1976) states that the ultimate testing technique is one that facilitates the execution of every path in the program. Due to the

infeasibility of such a test, he proposes another criterion called multiple condition coverage. This criterion requires one to write sufficient test cases such that all possible combinations of condition outcomes in each decision are produced, and all points of entry are invoked at least once.

Another criterion, introduced by Huang (1977), is known as branch coverage. This criterion states that one must write enough test cases such that each decision has a true and false outcome at least once.

Howden (1976) presents the path testing method. Path testing involves the grouping of the set of all paths through a program into a finite set of classes. It then requires the testing of one path from each class.

Rapps (1982) suggests the use of data flow analysis techniques as a means for path selection criteria. The analysis focuses on the occurrences of variables within the program. The actual functions and predicates play no role. Each variable occurrence is classified as being a definitional occurrence (def), computational-use occurrence (c-use), or predicate-use occurrence (p-use). Def and c-use occurrences are associated with the nodes in a data flow graph, whereas p-use occurrences are associated with the edges. The criterion suggested is called all-du-paths. A path P satisfies this criterion if for every node i and every $x \in$ def(i), P includes every loop-free definition-clear path with respect to x from i to all elements of dpu(x,i) and to all elements of dcu(x,i). Where P is the set of com-

plete paths of the graph, dcu(x,i) is the set of all nodes j such that $x \in$ c-use(j) and for which there is a definition-clear path with respect to x from i to j, and dpu(x,i) is the set of all edges (j,k) such that $x \in$ p-use(j,k) and for which there is a definition-clear path with respect to x from i to (j,k).

DeMillo (DeMillo 1978, Budd 1978) presents a method for determining the test data adequacy known as program mutation. In this method, a program P which is correct on a test data T is subjected to a series of mutant operators to produce mutant programs which differ from P in very simple ways. The mutants are then executed on T. If all mutants give incorrect results then it is very likely that P is correct. On the other hand, if some mutants are correct on T then either these mutants are equivalent to P or the test data is inadequate.

Symbolic execution is another testing strategy, (Osterweil 1981, Howden 1977, Clarke 1976), which computes the values of a program's variables as functions. These functions represent the sequence of operations carried out as execution is traced along a specific path through the program.

2.5. Static Analysers

Static analysis techniques involve the examination of the software design and source code for consistency, completeness and structural well-formation (Ramamoorthy 1975). The underlying objective of such

analysis is the detection of various structural and semantic anomalies and the identification of questionable features which should be the target of further dynamic analysis. The main characteristic of a static analysis method is that it does not necessitate the actual execution of the software.

The FACES system (Ramamoorthy 1975) is an example of a static analysis tool. This system is designed for assisting the development, testing, modification and maintenance of Fortran programs. FACES consists of two parts : the Fortran Front End and the Automatic Interrogation Routine (AIR). The Fortran Front End is essentially a language processor to transform the program source code to the appropriate tabular representation which is stored in a data base. The generated data base then consists of three main tables : symbol table, use table and the node table.

The AIR interprets queries and automatically searches the data base for specified language constructs. Identification of syntactically correct but logically suspicious constructs and identification of redundant and unreachable code are some examples of facilities provided by AIR.

DAVE (Osterweil 1976, Fosdick 1976) is a more sophisticated static analysis tool. This system uses data flow analysis techniques to detect suspicious or erroneous use of data in Fortran programs. The data flow anomalies detected by DAVE are : references to uninitialized variables and dead variable definitions. The system examines all paths from the program start node and is capable of determining that no

path, when executed, will cause a reference to an uninitialized variable. It also examines all paths from a variable definition and is capable of determining whether or not there is a subsequent reference to the variable.

DAVE carries out its analysis by performing a flow graph search for each variable in a given unit. It analyses subprograms in a leafs-up order and assumes that no subprogram invocation will be considered until the invoked subprogram has been completely analyzed. The use of data flow analysis for anomaly detection in concurrent software is further investigated by Taylor (1980).

CHAPTER 3

INCREMENTAL DATA FLOW ANALYSIS

An incremental data flow analysis algorithm is one which, by determining and propagating program changes, avoids complete reanalysis after each modification. The complexity of such algorithms depends on the program changes made and the size of the affected area. The possible program modifications are insertion and deletion of one or more source-level statements. Source code replacement is considered to be deletion followed by insertion.

The changes made in the source code may be minor and result in changes in local data flow information within a node. In such cases, changes in the local data flow information can easily be propagated and usually affect only a small portion of the solution from the previous analysis. On the other hand, some program modifications may result in changes in the control flow structure. Such changes in general add to the amount of work of the incremental algorithms.

We have designed incremental analysis versions of the iterative global flow analysis algorithms due to Hecht (Aho 1977), and the interprocedural algorithms due to Banning (1979). These algorithms and their incremental versions are described in the following sections.

3.1. Global Flow Analysis

Global flow analysis involves two types of problems :

(1) The forward flow problems are those which, given a point in the program, ask what can happen before control reaches that point.

(2) The backward flow problems are those which, given a point in the program, ask what can happen after control leaves that point.

The problem type is of importance to our incremental algorithms, since it helps isolate the area of the digraph which is affected by a modification. In the following sections, the reaching definition and live variable analysis algorithms are presented as examples of forward and backward flow problems, respectively. Both these algorithms work on the digraph representation of code, where each node is indexed by its depth first number (DFN).

3.2. Exhaustive / Incremental Reaching Definition

A definition d of a variable V reaches a point P, if there is a path in the flow graph from d to P, such that no other definition of V appears on the path.

Hecht's algorithm begins by computing two sets for each node B in the digraph. These sets are actually represented as bit vectors and are :

GEN [B] - The set of generated definitions, those definitions within B that reach the end of block B.

KILL [B] - The set of definitions outside of B that define identifiers which also have definitions within B.

The next step is to apply the algorithm shown in Figure 3 in order to calculate the sets IN[B] and OUT[B]. IN[B] consists of all definitions reaching the point just before the first statement of block B. OUT[B] is the set of definitions reaching the point just after the last statement of block B.

The algorithm in effect solves the following set of 2N simultaneous equations for a flow graph of N nodes.

$$OUT[n] = (IN[n] - KILL[n]) \cup GEN[n]$$

$$IN[n] = \bigcup_{\substack{p \text{ a pred-} \\ \text{ecessor of } n}} OUT[p]$$

The solution to this set of equations is not unique in the presence of back-edges in the flow graph. We actually look for the smallest possible solution. Hence, the algorithm starts with the assumption that IN[n] for all nodes is empty (i.e. nothing reaches n) and OUT[n] for all nodes is GEN[n]. The algorithm then repeatedly gets better approxi-

```
BEGIN
 FOR I := 1 TO N DO
  BEGIN  (* initialization *)
   IN[I] := φ
   OUT[I] := GEN[I]
  END
 CHANGE := TRUE
 WHILE CHANGE DO
  BEGIN
   CHANGE := FALSE
   FOR I := 1 TO N DO
     BEGIN
      NEWIN :=  U  OUT[p]
           p a prede-
           cessor of node I
      IF IN[I] <> NEWIN THEN
       BEGIN
        IN[I] := NEWIN
        OUT[I] := (IN[I] - KILL[I]) U GEN[I]
        CHANGE := TRUE
       END
     END
  END
END
```

Figure 3. Reaching definition algorithm.

mations by recomputing IN[n] and OUT[n] for all n, using the above relations.

Our incremental update algorithm for reaching definitions is actually a two step process. When a modification occurs, the first step is to calculate or update the data flow information for the affected node. This involves updating the GEN and KILL sets first, followed by one of

the following actions.

The following cases exist.

(a) Insertion of a new node P.

- Recompute the depth first order.
- Compute the GEN and KILL sets

  This may affect the KILL sets of other nodes; if so, recompute the KILL set for each affected node.
- Compute first approximations of IN and OUT for the new node.
- Place all immediate successors of P in a worklist called W.
- Determine the source nodes for all definitions killed by P. Remove all references to the killed definitions from IN and OUT of all nodes.

  Compute the IN and OUT of the source nodes and place all their immediate successors in W.

(b) Deletion of an existing node.

- Recompute the depth first order.
- Remove any references in the KILL, IN, OUT sets of all nodes to the generated definitions within this node.

- Place all immediate successors of P in the worklist W.

(c) Insertion of an arc.

- Place the node directed towards into the worklist W.

(d) Removal of an arc.

- Determine the source nodes for all definitions which were previously propagated and are now blocked by removal of the arc.
- Remove every reference to these definitions from IN and OUT of all nodes.
- Compute the IN and OUT of the source nodes and place all their immediate successors in the worklist W.

If the algorithm is applied to a new procedure, the initial approximations to the sets IN and OUT are :

$$IN[n] = \phi \quad \text{for all n.}$$

$$OUT[n] = GEN[n] \text{ for all n.}$$

The worklist, W, in this case consists of the immediate successors of each node, n, such that $GEN[n] \neq \phi$.

In all cases, a worklist W is determined, which consists of all the immediate successors of the affected node. The worklist is constructed in this manner because in a forward flow problem we are concerned with the portion of the digraph below the affected node.

The second step in our updating algorithm is to propagate the changes using the algorithm given in Figure 4. The procedure PROP actually propagates the changes. It calculates a new IN value for the node; the old value and the new value are compared; if they differ, then the node is considered an affected node. When it is determined that a node is affected, its IN value is updated to reflect the changes; a new OUT value is calculated; the old and new OUT values are compared; if they differ then OUT value is updated and W is expanded to cover the successors of this node.

3.3. Exhaustive / Incremental Live Variable Algorithms

In live variable analysis, we wish to know for name V and point P whether the value of V at P could be used along some path in the flow graph starting at P. If so, we say V is live at P; otherwise V is dead at P.

The exhaustive bit propagation algorithm is shown in Figure 5. This algorithm uses the following sets :

IN[n] - Set of names live at the point immediately before block n.

```
PROCEDURE INC-REACH-DEF;

 PROCEDURE PROP(n,P);
 BEGIN
  NEWIN :=  U   OUT[p]
          p ∈ P
  IF IN[n] <> NEWIN THEN
  BEGIN
   IN[n]  := NEWIN
   NEWOUT := (IN[n] - KILL[n]) U GEN[n]
   IF OUT[n] <> NEWOUT THEN
    BEGIN
     OUT[n] := NEWOUT
     W := W U {x | x ∈  successors of n }
    END
  END

  (* else no more updating is required *)
  (* for this path.                    *)
 END    (* PROP *)

BEGIN
 WHILE W <> ϕ DO
  BEGIN

   (* select and remove node n from W *)
   (* let n be the node with least   *)
   (* index contained in W.          *)

   W := W - [ n ]
   P := {x | x ∈ predecessors of n }
   PROP(n, P)
  END
END
```

Figure 4. Incremental update algorithm for reaching definitions.

OUT[n] - Set of names live at the point immediately
after block n.

DEF[n] - Set of names assigned values in n, prior
to any use of that name in n.

USE[n] - Set of names used in n, prior to any defin-
ition of that name in n.

Our incremental update algorithm for live variable analysis is similar to that for reaching definitions. The only difference is the determination of the members of the worklist W. Here, W contains all the immediate predecessors of the affected node. The reason being that in a backward flow problem, one is concerned with the portion of the digraph just above the affected node. The data flow solution below the affected node will not change.

The first step in our incremental algorithm is to determine the new values for the affected node and the members of W, the worklist. The second step is to propagate the changes using the algorithm in Figure 6.

3.4. Analysis Of The Update Algorithms

In this section, we discuss the time and space complexities of the incremental update algorithms for global flow analysis. The complexity analyses for these algorithms are very much data structure dependent. Therefore, we first discuss our implementation for an incremental PASCAL source level analyser.

```
BEGIN
 FOR I := 1 TO N DO  IN[I] := φ
 WHILE changes occur DO
  FOR I := N TO 1 BY -1 DO

   (* in reverse depth first order *)

    BEGIN
     OUT[I] := U    IN[s]
              s a succ-
              essor of node I
     IN[I] := (OUT[I] - DEF[I]) U USE[I]
    END
END
```

Figure 5. Live variable analysis algorithm.

```
PROCEDURE INC-LIV-VAR;

 PROCEDURE PROP(n,S);
 BEGIN
  NEWOUT :=  U    IN[s]
            s ∈ S
  IF OUT[n] <> NEWOUT THEN
  BEGIN
   OUT[n] := NEWOUT
   NEWIN  := (OUT[n] - DEF[n]) U USE[n]
   IF IN[n] <> NEWIN THEN
    BEGIN
     IN[n] := NEWIN
     W := W U { x | x ∈ predecessors of n }
    END
  END

  (* else no more updating is required  *)
  (* for this path.                     *)

 END    (* PROP *)

BEGIN
 WHILE W <> φ DO
 BEGIN

  (* select and remove node n from W  *)
  (* let n be a node with the highest *)
  (* index in W.                      *)

  W := W - [n]
  S := { x | x ∈ successors of n }
  PROP(n, S)
 END
END.
```

Figure 6. Incremental live variable analysis algorithm.

When the analysis is done at the source level each procedure is decomposed into the language primitives which then make up the nodes of the directed graph. The sets GEN, USE and DEF are computed by scanning the procedure and are assigned to each node. These sets are implemented as bit vectors. That is, using the SET construct of the programming language PASCAL. GEN is defined in terms of the number of statements contained in a procedure. The sets DEF and USE are defined in terms of the average number of exposed variables for a given procedure. That is, USE and DEF are bit vectors having one bit per exposed variable.

We associate with each variable exposed to the procedure, a set of statements that define the variable. After the procedure has been completely scanned, the KILL set for each node is computed using GEN and definition sets associated with each variable. The set KILL is implemented as a bit vector and is defined in terms of the number of statements contained in a procedure.

3.4.1. Time Complexity

To establish the time complexity of the incremental update algorithms, we will briefly discuss the complexity of the Hecht/Ullman's algorithms and define the parameters used in their analysis.

Definition: A reducible flow graph (RFG) is one that can be decomposed uniquely into a DAG (Directed Acyclic Graph) and backward arcs.

Definition: The loop-connectedness of an RFG G, which we shall denote by d, is the largest number of backward (or back) arcs found in any cycle-free path in G.

An important restriction on flow graphs follows from the nature of branches in programs.

Definition: A flow graph in which $r = O(n)$ is called a sparse flow graph, where

- n is the total number of nodes, and
- r is the total number of arcs in a flow graph.

In practice, all flow graphs resulting from programs are sparse because binary branching is generally used for control flow. Also, programmers use disciplined and sparse control flow structures for conceptual simplicity. When no branching more complex than binary is used, $r \leq 2n$. Even flow graphs of programs containing case statements are (almost always) sparse. If an algorithm is $O(r)$, then it is $O(n^2)$ in the worst case, since r is $O(n^2)$ in the worst case. If sparseness is assumed then r is $O(n)$. Thus the algorithm would be $O(n)$.

In the worst case, Hecht's algorithms are bounded by $O(n^2)$ bit vector operations for both reducible and non-reducible graphs. We will discuss this briefly for the reaching definition algorithm of Figure 3. The next theorem and a complete analysis of the iterative algorithms can be found in (Hecht 1975).

Theorem 1: If the numbering of nodes in G (where G is an RFG) is chosen suitably (depth first order), the body of the while-loop in the algorithm will be executed at most d+2 times.

The for-loop in the algorithm will be executed n times. Thus, ignoring initialization the algorithm requires at most (d+2) * (n) steps. In the worst case, d is of O(r). If we assume sparseness r = O(n), then the algorithm requires $O(n^2)$ bit vector steps in the worst case.

The complexity of the incremental update algorithm displayed in Figure 4 is similarly bounded by $O(n^2)$ in the worst case. This is due to the fact that for each back arc in the flow graph, n nodes may ultimately be introduced in the worklist W. By definition, there are d back arcs in an RFG. Thus, the incremental algorithm requires O(dn) steps.

Computation of NEWIN for each node with p predecessors requires (p-1) bit vector steps. Then for n nodes with a total of r predecessors (r-n) bit vector steps are required. Computation of NEWOUT for each node requires two bit vector steps. Then for n nodes 2n bit vector steps are required. Thus the algorithm requires O (d * (r + n)) bit vector steps. In the worst case d is of O(r) and assuming sparseness r = O(n).

If the total number of statements in a procedure is reasonably small so that a bit vector fits in one word, then each of the bit vector steps mentioned above can be performed by a single logical operation. In this case, the update algorithm displayed in Figure 4 is of $O(n^2)$

complexity in the worst case. Otherwise, the bit vector steps can be performed in time proportional to the number of statements in the procedure which is $\frac{n}{B}$ (for an average procedure size n and a word size of B bits). In this case, the incremental reaching definitions algorithm is of $O(n^3)$ complexity in the worst case.

By the same arguments, the incremental live variables algorithm of Figure 6 requires O(dn) steps for a flow graph with n nodes. The computation of NEWOUT for each node with s successors requires (s-1) bit vector steps. Then for n nodes with a total of r successors (r-n) bit vector steps are required. Computation of NEWIN for n nodes require 2n bit vector steps. Then the complexity of the incremental live variables algorithm is exactly the same as that for incremental reaching definitions. That is, both algorithms require $O(n^2)$ bit vector operations.

The only difference arises due to the fact that the bit vectors used in the live variables problem are defined in terms of the number of exposed variables of a procedure. If the total number of exposed variables for a procedure is reasonably small so that the a bit vector fits in one word, then each of the bit vector steps can be performed by a single logical operation and thus the algorithm of Figure 6 is of $O(n^2)$ complexity in the worst case. Otherwise, each bit vector step require time proportional to its size which is $\frac{|v|}{B}$ (where |v| denotes the total

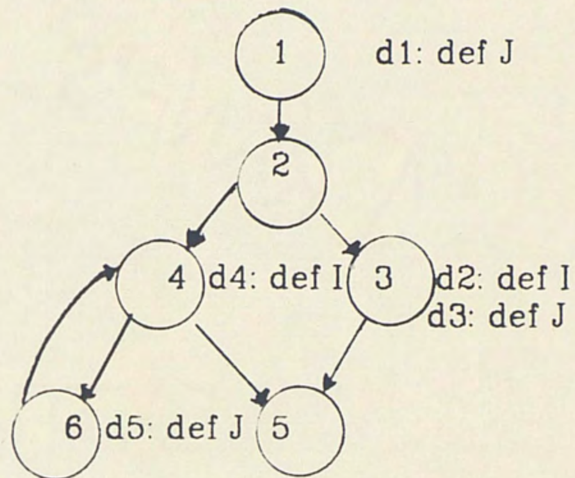
number of exposed variables and B is the word size). In this case, the incremental live variables algorithm is of $O(|v| * n^2)$ in the worst case.

Empirical surveys (Knuth 1971), show that in programs written with a disciplined control flow structure d is rarely more than 3, (d is essentially the maximum nesting of while-loops). In practice then, both the exhaustive and the incremental algorithms require O(n) bit vector operations.

We should point out that in the incremental update algorithms all nodes are visited only in extreme cases, whereas in the exhaustive algorithm all the nodes must be visited for each iteration of the while loop. This situation can be seen in the examples shown in Figures 7 to 9.

A problem flow graph is shown in Figure 7. In Figure 9, the solution to the example is found using the incremental reaching definition algorithm. This is an extreme case, since no previous solution exists. But even in this case the update algorithm has a better performance. It visits only 6 nodes, whereas the exhaustive algorithm applied in Figure 8 visits 18 nodes.

Moreover, a minor modification in local information within a node in general would affect only a small number of nodes. An example of this situation is shown in Figure 10. The local change in node 6 affects the data flow solution of node 4 only. One node is visited for convergence. The exhaustive algorithm must visit 18 nodes to find the solution.



node n	GEN[n]	bit vector	KILL[n]	bit vector
1	{d1}	10000	{d3,d5}	00101
2	{}	00000	{}	00000
3	{d2,d3}	01100	{d1,d4,d5}	10011
4	{d4}	00010	{d2}	01000
5	{}	00000	{}	00000
6	{d5}	00001	{d1,d3}	10100

Figure 7. Example of a flow graph. All nodes are numbered in reverse post order.

node n	initial IN[n]	initial OUT[n]	pass 1 IN[n]	pass 1 OUT[n]
1	00000	10000	00000	10000
2	00000	00000	10000	10000
3	00000	01100	10000	01100
4	00000	00010	10001	10011
5	00000	00000	11111	11111
6	00000	00001	10011	00011

node n	initial IN[n]	initial OUT[n]	pass 1 IN[n]	pass 1 OUT[n]
1	00000	10000	00000	10000
2	10000	10000	10000	10000
3	10000	01100	10000	01100
4	10011	10011	10011	10011
5	11111	11111	11111	11111
6	10011	00011	10011	00011

Figure 8. Solution of the example in Figure 7 using Hecht's algorithm. If we ignore initialization, a total of 18 nodes are visited before convergence.

The initial approximation is

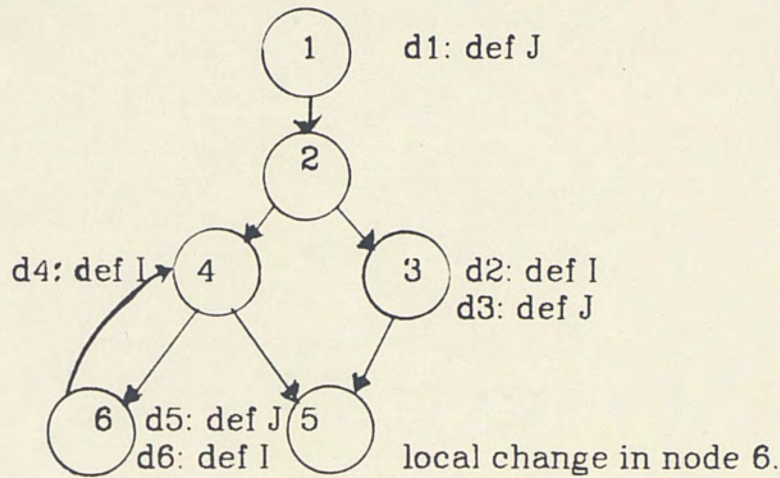
$$IN[n] = \phi \quad \text{for all } n$$

$$OUT[n] = GEN[n] \quad \text{for all } n$$

The worklist W = {2, 4, 5, 6}

visit n	IN[n]	OUT[n]	W
2	10000	10000	{3,4,5,6}
3	10000	01100	{4,5,6}
4	10001	10011	{5,6}
5	11111	11111	{6}
6	10011	00011	{4}
4	10011	10011	{}

Figure 9. Solution of the example in Figure 7, using incremental reaching definition algorithm. A total of 6 nodes are visited before convergence.



node n	GEN[n]	bit vector	KILL[n]	bit vector
1	{d1}	100000	{d3,d5}	001010
2	{}	000000	{}	000000
3	{d2,d3}	011000	{d1,d4,d5,d6}	100111
4	{d4}	000100	{d2,d6}	010001
5	{}	000000	{}	000000
6	{d5,d6}	000011	{d1,d2,d3,d4}	111100

1st phase : OUT[6]=(100110-111100)+000011
=000011 W = {4}

2nd phase :

visit n	IN[n]	OUT[n]	W
4	100011	100110	{}

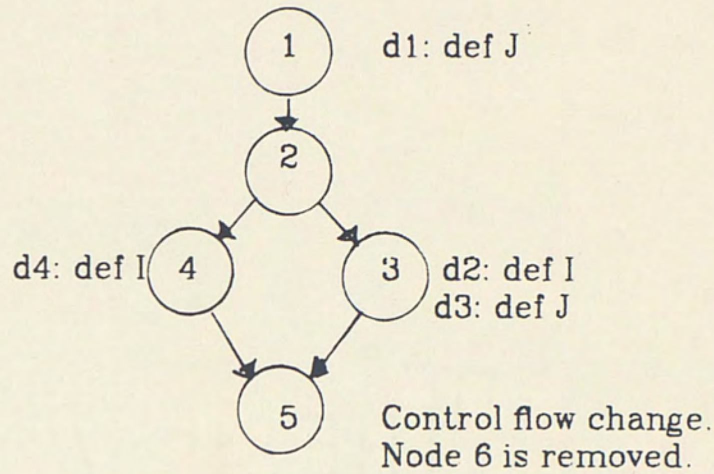
Figure 10. Occurrence of a local change in the example of Figure 7, and the updated solution. Only one node is visited for convergence. Application of the exhaustive algorithm, requires the visitation of 18 nodes.

Even with control flow changes the incremental algorithms are usually better since they visit fewer nodes. An example is shown in Figure 11, where the example of Figure 7 is changed by removal of node 6. This change in control flow structure affects the data flow solution of node 4. One node is visited before convergence. The exhaustive algorithm must visit 5 nodes.

3.4.2. Space Complexity

To analyse the space complexity for the incremental update algorithm, we need to consider both the storage space to save information from one analysis to the next and the actual storage required by the algorithm. In our discussion m is the average number of statements in a procedure, $|v|$ denotes the total number of variables exposed to a procedure and B is the word size.

We first deal with the storage space required to save information from one analysis to the next for incremental reaching definitions. The following information needs to be saved for every procedure.



node n	GEN[n]	bit vector	KILL[n]	bit vector
1	{d1}	1000	{d3}	0010
2	{}	0000	{}	0000
3	{d2,d3}	0110	{d1,d4}	1001
4	{d4}	0001	{d2}	0100
5	{}	0000	{}	0000

1st phase : W = {4}

2nd phase :

visit n	IN[n]	OUT[n]	W
4	1000	1001	{}

Figure 11. Control flow change in the example of Figure 7, and the updated solution.

Information	Storage
name	1 word
number	1 word
start (line#)	1 word
end (line#)	1 word

For each variable exposed to this procedure:

variable name	1 word
variable number	1 word
definitions	$\frac{m}{B}$

For each node, k in this procedure:

Information	Storage
Position of k	1 word
DFN[k]	1 word
KILL[k]-bit vector	$\frac{m}{B}$
GEN[k]-bit vector	$\frac{m}{B}$
IN[k]-bit vector	$\frac{m}{B}$
OUT[k]-bit vector	$\frac{m}{B}$
SUCC[k]-bit vector	$\frac{m}{B}$
PRED[k]-bit vector	$\frac{m}{B}$

Thus for incremental reaching definition algorithm, for each procedure we need a total of

$$4 + |v| * (2 + \frac{m}{B}) + m * (2 + \frac{6m}{B}) =$$

$$4 + 2 * (|v| + m) + \frac{m}{B} * (|v| + 6m)$$

words of storage space to save information. Then for a program consisting of N statements, the external storage space requirement is

$$N_p * (4 + 2 * (|v| + m) + \frac{m}{B} * (|v| + 6m))$$

words, where N_p is the total number of procedures in a program. N_p is equivalent to $\frac{N}{m}$. Hence, the storage needs are

$$\frac{N}{m} * (4 + 2 * (|v| + m) + \frac{m}{B} * (|v| + 6m)) =$$

$$\frac{4N}{m} + 2N + (|v|*N)*(\frac{2}{m} + \frac{1}{B}) + \frac{6Nm}{B} = O (N * (m+|v|))$$

In practice the average procedure size is very small in comparison to the actual program size. In a survey of 89 PASCAL programs, Carter (1982) reports N to be on average 1749 and m to be approximately 53. The other parameter, namely |v| the number of exposed variables, needs to be discussed. Although, |v| is large in older languages such as FORTRAN, in newer languages this tend to be much smaller in comparison to the total size of the program. In a language such as PASCAL, due to the scope rules, |v| is small. |v| tends to be very small in

languages such as ADA or Path PASCAL due to the package and object constructs. Hence, we can assume the external storage complexity to be of O(N). That is, the external storage complexity increases in the number of statements in the program.

The actual update algorithm of Figure 4 require three bit vectors for the worklist W and the sets NEWIN, NEWOUT. Each of these bit vectors requires $\frac{m}{B}$ space. Also, the previous data flow information for the procedure under analysis must be made available. This requires

$$4 + 2 * (|v| + m) + \frac{m}{B} * (|v| + 6m) = O(m^2)$$

Thus the space complexity for the incremental update algorithm for reaching definitions increases in the number of statements in a procedure and is as follows

$$O(m^2) + \frac{3m}{B} = O(m^2)$$

The space complexity for the incremental live variables algorithm follows the same arguments. The following information needs to be saved for every procedure from one analysis to the next.

Information	Storage
name	1 word
number	1 word
start (line#)	1 word
end (line#)	1 word

For each variable exposed to this procedure:

variable name	1 word
variable number	1 word

For each node, k in this procedure:

Information	Storage
Position of k	1 word
DFN[k]	1 word
USE[k]-bit vector	$\frac{\lvert v \rvert}{B}$
DEF[k]-bit vector	$\frac{\lvert v \rvert}{B}$
IN[k]-bit vector	$\frac{\lvert v \rvert}{B}$
OUT[k]-bit vector	$\frac{\lvert v \rvert}{B}$
SUCC[k]-bit vector	$\frac{m}{B}$
PRED[k]-bit vector	$\frac{m}{B}$

Thus, for incremental live variables algorithm, for each procedure we need a total of

$$4 + (2 * |v|) + 2m + \frac{m}{B} * (2m + (4 * |v|))$$

words of storage space to save information. Then for a program consisting of N statements, the external storage space requirement is

$$\frac{N}{m} * (4 + (2 * |v|) + 2m + \frac{m}{B} * (2m + (4 * |v|)))$$

which results in the external space complexity of

$$O(N * (m + |v|))$$

By the same arguments discussed for reaching definitions, the size of m and |v| are small and the external storage complexity can be represented by O(N).

The actual update algorithm of Figure 6 require 3 bit vectors and the solutions of a previous analysis of the procedure under consideration. This requires

$$(3 * |v|) + 4 + (2 * |v|) + 2m + \frac{m}{B} * (2m + (4 * |v|))$$

That is, the space complexity for the incremental live variable algorithm is $O(m^2)$.

The results of our analysis depicts that for both algorithms the external storage complexity increases with the number of statements

in the program and the space complexity for the actual update algorithms increases with the number of statements in the procedure.

3.5. Validity of the Incremental Update Algorithms

In this section, we discuss the validity of the incremental reaching definition algorithm. Similar arguments apply for the incremental live variable analysis.

For purposes of analysis, we define the terms IN and OUT for a procedure, R, and redefine the algorithm shown in Figure 4.

For a procedure, R with n blocks, the ith approximation to IN is defined as

$$IN_i[R] = [\ IN_i[1]_R, \ldots ,\ IN_i[n]_R\ ]$$

where $IN_i[j]_R$ is the ith approximation for block j. Similarly the ith approximation to OUT is defined as

$$OUT_i[R] = [\ OUT_i[1]_R, \ldots ,\ OUT_i[n]_R\ ]$$

In each of the above, R is omitted whenever this omission leads to no confusion. We define inclusion of sets of the above form by

$$\begin{aligned} IN_i[R] \subseteq IN_j[R] \ \text{iff} \\ IN_i[k] \subseteq IN_j[k] \quad \text{for all } k = 1..n \end{aligned}$$

In Figure 12, we present a reformulation of the incremental reaching definition algorithm of Figure 4. This reformulation better enables us to show convergence of the algorithm. The main difference is that

```
1  PROCEDURE INC-REACH-DEF;

2    PROCEDURE PROP(n,P);
3     BEGIN
4      NEWIN :=   U   OUT_i[p]
                 p ∈ P
5      IF IN_i[n] <> NEWIN THEN
6       BEGIN
7        IN_i[n]  := NEWIN
8        NEWOUT := (IN_i[n] - KILL[n]) U GEN[n]
9        IF OUT_i[n] <> NEWOUT THEN
10        BEGIN
11         OUT_i[n] := NEWOUT
12         W_i+1 := W_i+1 U ({x | x ∈  successors of n} ∩ W̄_i )
13        END
14       END
      (* else no more updating is required *)
      (* for this path.                    *)
15    END    (* PROP *)

16   BEGIN
17    i := 1
18    W_1 := W
19    WHILE W_i <> φ DO
20     BEGIN
21      IN_i[k] := IN_i-1[k]  for all k = 1..n
22      OUT_i[k] := OUT_i-1[k]  for all k = 1..n
23      W_i+1 := φ
24      WHILE W_i <> φ DO
25       BEGIN
        (* select and remove node n from W *)
        (* let n be the node with least    *)
        (* index contained in W.           *)
26        W_i := W_i - [ n ]
27        P := {x | x ∈ predecessors of n }
27        PROP(n, P)
28       END
29      i := i + 1
30     END
31   END
```

Figure 12. Reformulation of the incremental reaching definition algorithm of Figure 4.

now there is an order on processing the elements of W, the worklist. We will first process all the elements of a current worklist. If during this process the IN value of a node is affected, we will record its immediate successors in a separate worklist only if they are not members of the current worklist. The elements of this new worklist will be dealt with in the next approximation.

For implementation purposes, the algorithm of Figure 4 should be used, since the processing of nodes in depth first order will ensure faster convergence.

Theorem 2: The incremental update algorithm for reaching definitions (shown in Figure 12) terminates and is correct.

Proof: We deal with termination and correctness separately.

Termination: We prove termination by a series of Lemmas. For the next two Lemmas, assume the algorithm of Figure 12 is applied to a new procedure. That is, no previous solution exists. Under this circumstance,

$$IN_0[k] := \phi \quad \text{for all } k=1 \,..\, n$$

$$OUT_0[k] := GEN[k] \quad \text{for all } k=1 \,..\, n$$

and, therefore,

$$IN_0[R] = [\ \phi, \ldots\ldots, \phi\ ]$$

$$OUT_0[R] = [\ GEN[1], \ldots\ldots, GEN[n]\ ]$$

Lemma 1: For each $i \geq 1$, $IN_{i-1}[k] \subseteq IN_i[k]$ for all k=1 .. n. Similarly for OUT.

Proof: We prove Lemma 1, by induction on i, the number of an approximation.

Basis: (i = 1)

Since $IN_0[k] = \phi$ for all k then $IN_0[k] \subseteq IN_1[k]$, no matter what we set $IN_1[k]$ to be. $OUT_1[k]$ will contain GEN[k] no matter what, since it can change only by execution of statement 8.
Thus $OUT_0[k] \subseteq OUT_1[k]$.

Inductive step: (i = t+1)

Assume $IN_{t-1}[k] \subseteq IN_t[k]$ and $OUT_{t-1}[k] \subseteq OUT_t[k]$ on the tth approximation to the solution for node k and note that the sets GEN and KILL remain unchanged from one approximation to the next.

On the (t+1)st approximation, $OUT_{t+1}[p]$ starts as $OUT_t[p]$ due to statement 22. Thus $OUT_{t+1}[p]$ (in statement 4) can not have decreased. Hence, NEWIN would be at least as large as it was during the tth stage and therefore,

$$IN_t[k] \subseteq IN_{t+1}[k]$$

Since $IN_{t+1}[k]$ can only grow at each stage, the same property holds for NEWOUT and thus,

$$OUT_t[k] \subseteq OUT_{t+1}[k]$$

Lemma 2: $IN_{i-1}[R] \subseteq IN_i[R]$ and $OUT_{i-1}[R] \subseteq OUT_i[R]$ for all i.

Proof: The proof of this Lemma is a trivial consequence of Lemma 1 and the definition of $IN_i[R]$ in terms of the $IN_i[k]_R$'s and $OUT_i[R]$ in terms of the $OUT_i[k]_R$'s.

Lemma 3: There exists a j such that $IN_j[R] = IN_{j+1}[R]$, $OUT_j[R] = OUT_{j+1}[R]$ and such that the worklist $W_{j+1}=\phi$. that is, the algorithm terminates.

Proof: Each $IN_i[R]$ and $OUT_i[R]$ is an approximation to a finite set. By Lemma 2, $IN_i[R]$'s and $OUT_i[R]$'s form a non-decreasing sequence of approximations. But since each $IN_i[R]$ and $OUT_i[R]$ is a subset of a finite set, then they can not increase in size indefinitely. Thus, there must be an approximation j, such that $IN_j[R] = IN_{j+1}[R]$, $OUT_j[R] = OUT_{j+1}[R]$.

Moreover, if $IN_j[R] = IN_{j+1}[R]$, $OUT_j[R] = OUT_{j+1}[R]$ then the block of code from 10-13 is never executed and thus the worklist $W_{j+1} = \phi$, at all times.

$W_{j+1} = \phi$ results in the termination of the algorithm, since, for the previous worklist W_j, the main control removes an element from W_j during each iteration. When $W_j = \phi$, since W_{j+1} is also empty, the while-loop of line 19 is not executed. Thus, the algorithm terminates.

Correctness: We must show that at termination, the algorithm results in the final values of $IN_i[R]$ and $OUT_i[R]$ that equal the smallest correct values for IN[R] and OUT[R], respectively.

Recall from section 3.2 that in solving a reaching definition problem, we look for the smallest possible solution to a set of simultaneous equations. That is, the smallest possible solution to IN contains the correct set of definitions which reach any node k, for all k.

Let $IN'[R]$ and $OUT'[R]$ be the final solutions to the reaching definitions problem that is produced by the algorithm of Figure 12.

In the following discussions, consider first that the incremental algorithm is being applied to a new procedure. Then the first phase records in W the set of immediate successors of any node k such that $GEN[k] \neq \phi$.

Before presenting the next Lemma, we will state some basic facts.

Fact: Both the exhaustive and the incremental algorithms require all nodes to be reachable from the initial node. This requirement is easily met by the depth first search algorithm (DFS). That is, the set of nodes that are not reachable from the root are identified by the DFS algorithm and can be removed.

Fact: In a forward flow problem, OUT is a transfer function for IN. That is, if OUT' is incorrect, then IN' is also incorrect, but not necessarily vice versa. Thus, we need to consider only the case where IN' is incorrect.

Fact: A definition d reaches a point P if and only if there is a path from d to P along which d is not killed. Obviously if there is such a path, then there is a cycle free one.

Fact: The proper ordering of nodes in the flow graph insures faster convergence for the algorithms.

Definition: We refer to the shortest cycle-free path from d to P as the minimal path for a definition. It should be noted that for fast convergence, we visit nodes along this minimal path. This phenomenon, referred to as the minimality property of visitation, is achieved when we process nodes in depth first order.

Fact: Prior to the first step of the ith approximation, if $IN_i[k]$ is incorrect, then at least one of the immediate predecessors of node k must have an incorrect IN.

Fact: If $IN_i[k]$ is incorrect, then there must be some definition d such that $d \in IN[k]$ and a definition clear path exists from the point at which d is generated to k. We refer to the node in which the definition d is generated as the source node k' and note that k' may be k.

Definition: The set $K_d(k)$ consists of all nodes in a minimal length, d-definition clear path of nodes from k', the source of definition d, to some node k.

$$K_d(k) = \{k_0, k_1, \ldots\ldots, k_s, k_{s+1}\}$$

where $k_0 = k'$, $k_{s+1} = k$ and k_i is an immediate predecessor of k_{i+1}, for i= 0 to s.

Lemma 4: If $IN_0[k]$ is incorrect and missing definition d then, letting $K_d(k) = \{k_0, k_1, \ldots, k_s, k_{s+1}\}$, there is some node $j \ni k_j \in K_d(k)$, $d \in OUT_0[k_j]$ and $k_{j+1} \in W_1$.

Proof: In our incremental analysis d may be missing from $IN_0[k]$ if either

(a) d is a new definition resulting from an insertion, or

(b) d can reach k along some new path that was created by a deletion or a control flow change.

In case (a), the immediate successors of the source of definition d are placed in the worklist. Thus the Lemma is satisfied by letting j=0.

In case (b), there will be a node m that has d in its OUT set from the previous approximation and at least one successor of m that should, but does not have d in its IN set. The set up for our incremental algorithm places all successors of m in the worklist W_1. Thus, m will be one of the k_j's and one of its successors will be the element k_{j+1} that belongs to W_1.

Lemma 5: If $IN_0[k]$ is incorrect and missing definition d then, if $K_d(k) = \{k_0, k_1, \ldots\ldots, k_s, k_{s+1}\}$, $IN_t[k]$ contains d, for some $t \leq s+1$.

Proof: By Lemma 4, there is a $j \ni k_j \in K_d(k)$, $d \in OUT_0[k_j]$ and $k_{j+1} \in W_1$. Let m be the largest such j. Then $d \notin OUT_0[k_{j+1}]$ and hence $d \notin IN_0[k_{j+1}]$. The algorithm of Figure 12 insures us that $d \in IN_1[k_{j+1}]$. If d is not in $KILL[k_{j+1}]$ then $d \in OUT_1[k_{j+1}]$ and $k_{j+2} \in W_2$. But d cannot be in $KILL[k_{j+1}]$ if $j+1 \leq s$. Hence in one iteration of our algorithm d passes one node farther in the IN sets of the sequence of nodes $\{k_0, k_1, \ldots\ldots, k_s, k_{s+1}\}$. We need carry this process out at most s+1 times to insure that d has been propagated to each of the elements of $k_d(k)$. Hence $d \in IN_t[k]$, for some $t \leq s+1$.

If we treat each program modification as a replacement, then the correctness of the algorithm when applied to an existing procedure follows directly from the above discussion.

Program modifications can be treated as replacements in the following way. If a new node is added, it will replace an empty node. This may then block the path for a definition, in which case a new definition must have been introduced and must be propagated forward.

When an existing node is removed, it will be replaced by an empty node. This may then unblock the path for a definition which must be propagated forward.

By Lemma 5, these changes are propagated correctly by the algorithm.

Lemma 6: IN[R] and OUT[R] are minimal and correct solutions.

Proof: If the algorithm of Figure 12 is applied to a new procedure, then the initial values are set to $IN_0[k] = \phi$, $OUT_0[k] = GEN[k]$ for all k . If the algorithm is applied to an existing procedure for update purposes, then the initial values are $IN_0[k] = IN[k]$ and $OUT_0[k] = OUT[k]$ for all k, where IN[k] and OUT[k] are the smallest possible solutions found in the previous analysis, adjusted to reflect the worst case posed by the program modification being considered.

These initial values and the fact that $IN_i[R]$'s and $OUT_i[R]$'s form a non-decreasing sequence of approximations (by Lemma 2) and the fact that new values are added to IN and OUT only when absolutely necessary insures us that we will arrive at a minimal solution upon convergence. By Lemma 5, any definition that belongs in IN or OUT will be propagated correctly. Thus the solution is both minimal and correct.

The termination and correctness of the incremental live variable analysis can be argued in a similar manner. Since this is a backward flow problem, IN is a transfer function for OUT. Hence, in Lemmas 4 and 5 IN and OUT must be interchanged. In proving correctness for backward flow problem, it should be noted that if IN is incorrect then OUT is also incorrect but not necessarily vice versa. Moreover, since the flow is in the opposite direction then we need to talk about a sequence of backwards flowing nodes (predecessors) in the definition

$K_d(k)$ used in Lemmas 4 and 5. With these facts in mind, the proof follows directly from the proof for the forward flow problem.

CHAPTER 4

INTERPROCEDURAL ANALYSIS

The aim of interprocedural analysis is to determine the side effects of procedure calls. This determination involves the calculation of aliases and the side effects due to the execution of a procedure on variables at the point from which the procedure is called.

In this Chapter, we present the incremental and exhaustive algorithms for the alias computation. The discussion here and in Chapter 5 is limited to a language with PASCAL-like scope rules, simple variables, reference parameters and recursion.

4.1. Aliases

Two variables are aliases when both refer to the same location at the same time. The mechanism which maps variables to storage locations during the execution of a program has a strong effect on creation of aliases. This mapping depends on the language in which the program is written and results in different forms of aliasing.

Static aliasing occurs by using a programming language in which mappings are mainly static. FORTRAN is an example of such a language. In FORTRAN, with the exception of parameters, all variables are mapped to locations when the execution of the program begins.

This mapping remains in existence until the execution of the program terminates.

In many languages, particularly block structured languages, pieces of code may execute in different environments at different times during a program's execution. The programs written in such languages result in a dynamic form of aliasing.

In a block structured language, a new environment is created whenever a procedure is called. This environment disappears when the procedure returns. The mappings of variables to locations are made in accordance with the scope rules of such languages.

Local variables are mapped to new locations and global variables are mapped to the same locations as in the calling procedure's environment.

In PASCAL, which is the main language considered in this thesis, aliases are created due to the following features of the language:

(1) The parameter passing mechanism of procedure calls,

(2) The free variant mechanism, and

(3) The use of pointer variables.

We will only deal with the first source of aliasing in this paper.

4.2. Exhaustive Alias Calculation

In this section, we describe Banning's approach to computation of aliases (Banning 1979). The term alias has been traditionally defined

with respect to the older languages such as FORTRAN where the program executes in one environment. Banning gives the only definition of the term alias for block-structured languages. Two variables are said to be aliases of one another, if they both map to the same location in the same environment. He also distinguishes between may and must aliases.

Two variables are called **necessary aliases** if they are aliases in every environment in which they are both mapped. This is strictly a must information. The term possible aliases refers to may information. Two variables are **possible aliases** if they are aliases in some environment which could occur during the execution of the program containing them.

In the remainder of this section, after presenting the notation used in the algorithms, we will discuss Banning's method for finding possible aliases.

4.2.1. Basic Terminology

In this section, we present the basic definitions of the terms used in the rest of this Chapter and Chapter 5. These definitions are adopted from Banning (1978).

A program is a tuple PG = (P, V, IMOD, IREF, V_r, S, FROM, TO, BIND). The elements of PG are as follows.

P is a set of procedures. The elements of P are the procedures and functions of a block-structured program. *p* is

an element of P, called the main procedure. *p* is the most global procedure in P.

V — is a set of variables. These are the variables and parameters in a block-structured program. We insist that *variable and procedure names be unique* , a condition which any program can easily meet by means of straightforward renaming or by means of qualifying a simple name by the name of its containing blocks (that is, using path names).

Before defining IMOD and IREF, the definition of the next two terms are needed.

Definition: GLOBAL(p) is the set of objects global to procedure p according to the rules of the block-structured language.

Definition: VISIBLE(p) is the set of objects accessible to procedure p.

IMOD(p) — This is a mapping from P into subsets of V. That is, IMOD : P -> 2^V. We have the requirement that IMOD(p) $\subseteq$ VISIBLE(p) $\cap$ V for all p in P. This mapping specifies the variables which may be assigned by the execution of statements in procedure p. It excludes consideration of the effect of procedures called by p.

IREF(p) — the definition of IREF(p) is analogous as that for IMOD(p). However, this mapping specifies the variables which may be referenced by the execution of statements in procedure p.

V_r is a subset of V. It is the set of reference parameters of the program PG.

S is a set of call sites with distinguished element *s*. We can think of each element of S - { *s* } as corresponding to a call statement in some procedure with s being a call to *p* from outside the program. The mappings FROM, TO and BIND define the attributes of each call site.

FROM(s) This is a mapping from elements of S - { *s* } into P. That is FROM: S - { s } -> P. FROM(s) is the procedure from which the call associated with s will be made. We assume that call site s lies on some execution path through procedure FROM(s).

TO(s) TO: S -> P. TO(s) is the procedure which is called by call site s. i.e. the target of s.

BIND(s,X) BIND is a partial mapping. (S - { *s* }) x V_r -> V. BIND(s, X) gives the actual parameter which is bound to formal parameter X by call s. For BIND(s, X) to be defined, X must be a reference parameter of the procedure called by s.

BINDLIST(Y) is a set associated with each actual parameter, Y. We associate with Y a set of pairs of variables and call sites (s, X) for which BIND(s, X) = Y. Thus X is a formal parameter to which Y is bound by call s. This set can be built as a linked list which is built as call sites are scanned.

AS(X) A set AS(X) is associated with every reference parameter X in V_r. Let ALS(X) be the set of all aliases associated with any variable X. The set ALS(X) is not necessarily a subset of some VISIBLE(p). The set AS(X) contain every variable in ALS(X) * GLOBAL(p), where P is the procedure in which X is declared. We use AS(X) and not ALS(X) since in this way every non-trivial alias pair is recorded in exactly one place and each of the sets holding information is a subset of VISIBLE(p).

NUM(X) denotes the number associated with variable X. We number the variables by keeping the dictionary of variables in a stack which is kept as an array. As a procedure's local variables are scanned (before scanning local procedures), they are put on the stack. After the scanning of a procedure is finished, the variables are removed from the stack. The index of the array element into which the variable is put is the number associated with that variable.

4.2.2. Alias Algorithm

Banning's algorithm for finding pairs of possible aliases is shown in Figures 13 to 15. This algorithm deals with aliases created by parameter passing mechanisms of procedure calls. It is important to note that if two variables X and Y (X <> Y) are aliases, then each must be

```
PROCEDURE ALIAS

PROCEDURE VISIT(X, Y ∈ V)
 BEGIN
  IF (X = Y) or (not TEST(X, Y))
  THEN BEGIN
   SET(X, Y)

   FOR every (X', S) ∈ BINDLIST(X) DO
    FOR every Y'∋(Y', S) ∈ BINDLIST(Y) DO
     IF (X' <> Y') THEN VISIT (X', Y')

   FOR every (X' ,S) ∈ BINDLIST(X) DO
    IF Y ∈ GLOBAL(TO(S)) THEN
     VISIT (X', Y)

   IF (X <> Y) THEN (* avoids duplicate calls *)
    FOR every (Y', S) ∈ BINDLIST(Y) DO
     IF X ∈ GLOBAL(TO(S)) THEN
      VISIT(X, Y')
  END
 END

BEGIN
 FOR every X ∈ V_r DO AS(X) := {}

 FOR every X ∈ V∋BIND(S, Y) = X for
  some S and Y DO VISIT(X, X)
END.
```

Figure 13. Banning's algorithm to compute alias information.

```
FUNCTION TEST(X, Y ∈ VISIBLE(p), p ∈ P) : BOOLEAN
BEGIN
 IF X = Y
  THEN TEST := true
  ELSE IF NUM(X) > NUM(Y)
       THEN IF X ∈ V_r
            THEN TEST := Y ∈ AS(X)
            ELSE TEST := false
       ELSE IF Y ∈ V_r
            THEN TEST := X ∈ AS(Y)
            ELSE TEST := false
END
```

Figure 14. A function to test for aliases (Banning 1978).

```
Procedure SET(X, Y ∈ VISIBLE(p), p ∈ P)
BEGIN
 IF X <> Y
  THEN IF NUM(X) > NUM(Y)
       THEN BEGIN
         IF X ∈ V_r THEN
           AS(X) := AS(X) + {Y}
       END
       ELSE BEGIN
         IF Y ∈ V_r THEN
           AS(Y) := AS(Y) + {X}
       END
END
```

Figure 15. A procedure to record aliases (Banning 1978).

either a global variable or a local reference parameter. The algorithm's construction takes advantage of the fact that pairs of aliases are either trivial (i.e. X is an alias of X) or they derive from another pair of aliases through the actions of a call.

At the heart of the algorithm is a recursive routine which, given a pair of variables that are possible aliases, finds all the other pairs of possible aliases which are created by the original pair (and calls itself with these new pairs). The routine is started by calling it with all the trivial pairs of aliases.

The three loops in VISIT are designed to take care of the three possible cases for the relationship between (X, Y) and (X' ,Y') and the call site s. The three possible cases are :

(1) X is bound as an actual to X' by s and Y is bound as an actual to Y' by s.

(2) Y and Y' are a single variable which is global to the procedure called by s and s binds X as an actual to reference parameter X'.

(3) X and X' are a single variable which is global to the procedure called by s and s binds Y as an actual to reference parameter Y'.

4.2.3. Analysis of ALIAS

To establish the worst case time complexity of alias computation, we consider the algorithm in Figure 13.

The first loop in ALIAS, initializes the alias set associated with each reference parameter of the program. This then is $O(N_r)$, where N_r is the total number of reference parameters in a program. The second loop in ALIAS calls VISIT for every actual parameter in the program. But this can be better dealt with by considering VISIT.

The body of VISIT is executed at most once for each pair of possible aliases. The first loop in VISIT is executed at most the maximum number of times any variable is bound plus the number of elements with identical call sites in the two BINDLISTs.

The comparison of the two lists for elements with identical call sites can be done in linear time, if the BINDLISTs are built according to some order on the call sites which caused the bindings. Recall that BINDLIST is kept as a linked list and is built as call sites are scanned. If the ordering on the call sites is the same as the order in which they appeared in the program, then this is the order in which elements are added to every BINDLIST. Thus the comparison can be done in linear time.

In our implementation of the PASCAL source level analyser, BINDLISTs are implemented as described above. The data structure used to represent the call graph is a linked list. Each call record

contains the bindings and TO and FROM informations. TO and FROM are each a procedure number (to and from which the call is made).

The second and third loop in VISIT are each executed at most the maximum number of times any variable is bound. Then for each possible alias pair VISIT (ignoring the time requirement for FUNCTION TEST and PROCEDURE SET of Figures 14 and 15) has a time requirement proportional to

$$3 * |\text{bindings}| + |\text{elements with identical call sites}|$$

This is O(|bindings|) for each possible alias pair. Procedure VISIT is

$$O(|\text{possible aliases}| * |\text{bindings}|)$$

Then the time requirement for ALIAS is

$$O(N_r + |\text{possible aliases}| * |\text{bindings}|)$$

The total number of reference parameters, N_r is smaller in size compared to |possible aliases|. Therefore, the time complexity for ALIAS is

$$O(|\text{possible aliases}| * |\text{bindings}|)$$

In the worst case, there are an exponential number of possible aliases $O(2^{N_v})$ (where N_v is the total number of variables in a program). Thus the alias algorithm in the worst case is exponential and increases in the number of bindings.

Banning reports on a survey of 20 PASCAL programs in (Banning 1978). In the 20 programs, a total of 3523 pairs of possible aliases

were created. This was approximately 2.9 possible alias pairs for each of the 1196 reference parameters found.

The first loop in VISIT had an average of 2.7 iterations for each of the 3523 alias pairs. The second loop was executed an average of 2.3 times for each alias pair and the third loop was executed an average of 3 times.

Thus, in practice it appears that the ALIAS algorithm is linear in the number of reference parameters in the program.

4.3. Incremental Alias Computation

The program modifications which result in recomputation of aliases are addition of a new call site and deletion of an existing call site. In this section, we present two incremental update alias algorithms to deal with addition and deletion changes separately.

4.3.1. Candidates for Possible Aliases

For two variables X and Y to be possible aliases, one of the following conditions must hold :

(1) X = Y

(2) X and Y are distinct elements of the set of reference parameters declared in the same procedure and for which there is a call that binds two aliases to X and Y.

(3) X is a reference parameter for procedure p and Y is global to p.

(4) Y is a reference parameter for procedure p and X is global to p.

4.3.2. Incremental Alias Addition

When a new call site is added, it binds a set of actual parameters of the calling procedures to some formal parameters of the called procedure. The actual, formal pairs may be candidates for possible aliases (that is if one of the conditions in section 4.3.1 holds). If none of the conditions hold and the actual parameter is itself a formal parameter, then aliases for the actual parameter must be considered. The incremental algorithm to deal with addition changes consists of two steps.

The first step in incremental alias addition is to find and record the possible alias pairs (generated by the new call site) in the worklist, W. The worklist is created as follows.

(1) Examine each actual, formal pair; if it satisfies either of the conditions 3 or 4 of section 4.3.1 then add the pair to W. If the pair is not a candidate for possible alias, then examine the actual parameter passed by the call site. The only possible case is that the actual parameter is a formal parameter of the calling procedure. Then for each element a_i of the alias list for the actual parameter, X, add the pair

(a_i, Y) to W. It should be noted that a_i must be an element of VISIBLE(TO(s)), where s is the new call site.

(2) Update the BINDLIST information associated with the actual parameter referenced in the new call site.

At this stage the worklist, W, consists of all the pairs of possible aliases directly generated by the new call site. The second step is to calculate alias information for the elements of W and all variables related to them, using the algorithm given in Figure 16. The procedure VISIT in the incremental algorithm is exactly like its exhaustive counterpart. The only difference is that it is called from ADD-ALIAS with the elements of W. The procedure ADD-ALIAS assumes the existence of alias information from the previous analysis.

4.3.3. Time Analysis of ADD-ALIAS

The worst case time complexity of ADD-ALIAS follows the same argument for its exhaustive counterpart. Recall from section 4.2.3 that procedure VISIT is

$$O(|\text{possible aliases}| * |\text{bindings}|).$$

In the worst case, there is an exponential number of possible alias pairs. VISIT is called from ADD-ALIAS at most once for each member of W, the worklist. The size of W is bounded by s_r * |VISIBLE(TO(s))|, where s_r denotes the number of reference parameters associated with a call site s. The members of W are pairs of variables that are flagged

```
PROCEDURE ADD-ALIAS

PROCEDURE VISIT(X, Y ∈ possible aliases)
 BEGIN
  IF (X = Y) or (not TEST(X, Y))
  THEN BEGIN
   SET(X, Y)

   FOR every (X', S) ∈ BINDLIST(X) DO
    FOR every Y' ∋ (Y', S) ∈ BINDLIST(Y) DO
     IF (X' <> Y') THEN VISIT (X', Y')

   FOR every (X' ,S) ∈ BINDLIST(X) DO
    IF Y ∈ GLOBAL(TO(S)) THEN
     VISIT (X', Y)

   IF (X <> Y) THEN (* avoids duplicate calls *)
    FOR every (Y', S) ∈ BINDLIST(Y) DO
     IF X ∈ GLOBAL(TO(S)) THEN
      VISIT(X, Y')
  END
 END

BEGIN
 WHILE W <> ∅ DO
 BEGIN
  W := W - [(X, Y)]
  VISIT(X, Y)
 END
END.
```

Figure 16. Incremental update algorithm to compute alias information after addition of a new call site.

as possible aliases. Then Visit is called with s_r * |VISIBLE(TO(s))| pairs of possible aliases which may in the worst case leads to generation of all pairs of possible aliases of a program. The time complexity for ADD-ALIAS is then

$$O(|\text{possible aliases}| * |\text{bindings}|).$$

In the worst case, there is an exponential number of possible aliases. Thus the ADD-ALIAS algorithm is of exponential time complexity in the worst case.

For each of the reference parameters in a program, empirical evidence shows there are generally about 2.9 pairs of possible aliases. The total number of possible aliases generated by addition of a new call site is s_r * 2.9. In practice, the number of reference parameters associated with each call site is small.

Thus, the body of VISIT is executed only a small number of times (at most once for each pair of possible aliases generated by addition of the new call site). The survey made by Banning indicate that the first loop in VISIT is executed at most 2.7 times; the second loop is executed at most 2.3 times and the third loop is executed at most 3 times for each pair of possible aliases. Furthermore, some of the possible aliases related to the elements of W may have already been established as possible aliases in a previous analysis which will shortcircuit the execution of the body of procedure VISIT.

Thus, in reality the incremental ADD-ALIAS algorithm is linear in the |W| which is bounded by s_r * |VISIBLE(TO(s))|.

4.3.4. Validity of ADD-ALIAS

ADD-ALIAS assumes the validity of Banning's ALIAS algorithm.

Theorem 3: ADD-ALIAS is correct and it terminates.

Proof:

Termination: The body of VISIT is executed a finite number of times. In fact at most once for each pair of possible aliases. Each For-loop in VISIT is executed and consequently invokes VISIT a finite number of times. Thus, any invocation of VISIT must terminate in a finite amount of time.

The body of ADD-ALIAS calls VISIT a finite number of times. It calls VISIT for each member of W. Moreover, it removes the element from W each time it invokes VISIT. Then W eventually becomes empty. This result combined with the fact that the body of VISIT is not executed if a pair of variables have already been established as possible aliases leads to the termination of the ADD-ALIAS algorithm.

Correctness: Correctness of ADD-ALIAS follows that of Banning's ALIAS algorithm. We refer the reader to pages 93 to 97 in (Banning 1978) for a complete proof.

VISIT is exactly the same and is invoked with pairs of possible aliases in both algorithms. The only difference between the two algorithms is in the body of the main control. ALIAS initializes all the AS sets to empty and invokes VISIT for each trivial possible alias pair.

ADD-ALIAS assumes the correctness of the previous solution and invokes VISIT by exactly those pairs of variables which were

established as candidates for possible aliases. Thus both algorithm invoke visit with pairs of possible aliases.

4.3.5. Incremental Alias Deletion

When an existing call site is removed, the possible aliases induced by the call site may have to be removed. The call site binds a set of actual parameters of the calling procedure to some formal parameters of the called procedure.

In the rest of this discussion, we will use the following additional notations:

$A = \{A_1, A_2, \ldots\}$ is the set of actual parameters.

$F = \{F_1, F_2, \ldots\}$ is the set of formal parameters.

(x, y) is a possible alias pair.

(x, y) is <x, y> if NUM(x) <= NUM(y)

<y, x> otherwise.

To update alias information after deletion of a call site, we use a two step process. In the first step, the worklist, W is constructed using the following set of rules.

(1) The removal of a call site, s, has no effect on the alias sets, if there is another call site which is exactly the same or which contains s. If this situation occurs then we need to update only the BINDLIST information associated with s.

(2) Examine the actual, formal pair (A_i, F_i)

(a) If the pair satisfies one of the conditions for possible aliases, add the pair to W.

(b) If the pair does not satisfy any of the conditions for possible aliases, then A_i must be a formal parameter of the calling procedure. If this is the case, for each a' $\in$ AS(A_i) if a' $\in$ VISIBLE(TO(s)), add the pair (a', F_i) to W.

(3) If more than one parameter is passed by the call site, then examine the actual parameters. If the actual parameters are the same or aliases of each other, then their corresponding formals are possible aliases. Thus add the pair (F_1, F_2) to W.

(4) Remove the BINDLIST information associated with the call site.

The second step of the algorithm is to propagate the effect of the removal by applying the REMOVE-ALIAS algorithm to the elements of W. This algorithm is shown in Figures 17 and 18.

The REMOVE-ALIAS algorithm is constructed in this manner since it needs to determine the impossibility of alias before any removal can be made. Recall that a pair of possible aliases (x, y) can result in one or both of the following cases.

```
PROCEDURE REMOVE-ALIAS

PROCEDURE VISIT(X, Y ∈ possible aliases)
 BEGIN
  mark (X, Y) visited

  IF (X ∈ AS(Y)) and (CHECK(X, Y)) THEN
   AS(Y) := AS(Y) - {X}

  FOR every (X', S) ∈ BINDLIST(X) DO
   FOR every Y'∋(Y', S) ∈ BINDLIST(Y) DO
    IF (X' <> Y') and ((X', Y') is not visited)
     THEN VISIT (X', Y')

  FOR every (X' ,S) ∈ BINDLIST(X) DO
   IF (Y ∈ GLOBAL(TO(S))) and ((X' Y) not visited)
    THEN VISIT (X', Y)

  IF (X <> Y) THEN (* avoids duplicate calls *)
   FOR every (Y', S) ∈ BINDLIST(Y) DO
    IF (X ∈ GLOBAL(TO(S))) and ((X, Y') not visited)
     THEN VISIT(X, Y')
 END

BEGIN
 WHILE W <> φ DO
  BEGIN
   W := W - [(X, Y)]
   VISIT(X, Y)
  END
END.
```

Figure 17. Incremental update algorithm to compute alias information after deletion of a call site.

```
FUNCTION CHECK(X,Y ∈ possible aliases) : Boolean
BEGIN
 CHECK := true

 IF BINDLIST(X) <> {} (* X is an actual parameter *)
 THEN BEGIN
  IF (X, Y) ∈ V_r for some procedure P THEN BEGIN

   FOR every s ∈ S∋TO(s) = P DO
    IF BIND(s, X) ∈ V_r THEN
     IF BIND(s, Y) ∈ AS(BIND(s, X)) THEN CHECK := false
    ELSE IF BIND(s, Y) ∈ V_r THEN
     IF BIND(s, X) ∈ AS(BIND(s, Y)) THEN CHECK := false
  END

  ELSE IF (X ∈ V_r for some procedure P) and
      (Y ∈ GLOBAL(P)) THEN BEGIN

      FOR every s ∈ S∋TO(s) = P DO
       IF BIND(s, X) ∈ V_r THEN
        IF Y ∈ AS(BIND(s, X)) THEN CHECK := false
       ELSE IF BIND(s, X) = Y THEN CHECK := false
      END

      ELSE IF (Y ∈ V_r for some procedure P) and
          (X ∈ GLOBAL(P)) THEN BEGIN

          FOR every s ∈ S∋TO(s) = P DO
           IF BIND(s, Y) ∈ V_r THEN
            IF X ∈ AS(BIND(s, Y)) THEN
             CHECK := false
           ELSE IF BIND(s, Y) = X THEN
             CHECK := false
          END
 END
END
```

Figure 18. Function to check the impossibility of alias.

(1) appearance of x in AS(y) if NUM(y) > NUM(x), and/or

(2) Establishment of a chain of pairs $(x_1, y_1) \ldots (x_n, y_n)$ where $x = x_1$, $y = y_1$ which leads the variables x_1, y_1 to the variables x_n, y_n and makes (x_n, y_n) possible aliases.

The members of W are pairs of variables that have been flagged for not being possible aliases. While processing elements of W, if case 1 holds then REMOVE-ALIAS must establish that there is no other call site which binds x to y or results in x, y becoming aliases. This is exactly the purpose of FUNCTION CHECK. If CHECK establishes the impossibility of alias then REMOVE-ALIAS removes x from the alias set for y.

When the impossibility of an alias has been established or when case 2 holds, REMOVE-ALIAS must find all the chain of pairs related to alias pair (x, y) and establish the impossibility of each of them. This is the function of the three loops in VISIT.

To better demonstrate the working of our incremental remove alias algorithm, we present the reader with a few simple examples. Assume we have computed the aliases for a given program using Banning's algorithm. A call site is removed and we use the REMOVE-ALIAS algorithm to update the original solution. Given a pair of possible aliases (X, Y) marked for examination after removal of call site s, the following cases can occur :

(a) That (X, Y) may have been directly created by another call site s'. The call site s' can be exactly the same as s or it can contain s.

(b) That (X, Y) may have been indirectly created by another call site.

(c) That (X, Y) may have been created only by s.

(d) That alias pairs created by s result in formal parameters of a procedure as being aliases.

In dealing with case (a), we need to show that the algorithm can identify a call site s' = s. An example of case (a) is shown in Figure 19. The possible alias pair (Y, Y1) is created by the two call sites S2 and S3 and results in addition of Y to the alias set associated with Y1. Assume S2 is removed. The first step in the preprocessing phase of the algorithm identifies call site S3 which has the exact same property as S2. Hence, (Y1, S2) is removed from the BINDLIST(Y) and AS(Y1) remains unchanged.

To show that the algorithm works correctly given case (b), we need to show that alias sets remain unchanged as in (a) above. An example of case b) is shown in Figure 20. The possible alias pair (Y, X2) is created directly by call site S2 and indirectly through call sites S1 and S3. Assume call site S2 is removed. The worklist, W, consists of the pair (Y, X2). The procedure VISIT is called with this pair and establishes the fact that Y $\in$ AS(X2). Function CHECK is then invoked to establish the impossibility of (Y, X2) as an alias pair. The last portion of CHECK (if Y $\in$ V_r) applies to this case. Call site S3 is the only

```
program P;
var X, Y  : integer;
 procedure p1(var  Y1 : integer);
 begin
  .
  p1(Y);       S3
  .
 end;

begin (*p*)
 .
 p1(X);         S1
 p1(Y);        S2
 .
end.
```

```
actuals       BINDLIST
-------      -------------

 X           (Y1, S1)
 Y           (Y1, S2) --> (Y1, S3)
```

AS(Y1) = {X, Y}

Figure 19. Example for case (a).

```
program P;
var Y : integer;
 procedure P1 (var  Y1 : integer);

  procedure P2 (var  X2 : integer);
  begin.....end;

 begin (* P1 *)
  .
  P2(Y);       S2
  P2(Y1);      S3
  .
 end;

begin (* P *)
 .
 P1(Y);          S1
 .
end.

actuals       BINDLIST
-------       ------------

Y             (Y1, S1)--> (X2, S2)
Y1            (X2, S3)

AS(Y1) = {Y}
AS(X2) = {Y, Y1}
```

Figure 20. An example for case (b).

call to P2. BIND(S3, X2) = Y1 which is a reference parameter. Therefore the alias set for Y1 is examined and it is concluded that Y $\in$ AS(Y1). The function CHECK then returns false and Y is not removed from AS(X2). The algorithm then terminates with no change in alias information.

In showing correctness of our algorithm for case (c), we need to show that alias sets will be updated to reflect the impossibility of alias pair (X, Y). An example of case (c) is shown in Figure 21. Assume S1 is removed. BINDLIST(X) becomes empty by the first phase and $W = \{(X, Y1)\}$. (note: (X, Y1) is created by S1). Visit is invoked with (X, Y1) and establishes that $X \in AS(Y1)$. Function CHECK returns the value true since BINDLIST{X} = {}. Hence, X is removed from AS(Y1). The third loop in VISIT creates alias pairs (X, X2) and (X, X3). By the same argument X will be removed from AS(X2) and AS(X3).

To show correctness in case (d), we need to show that the resulting alias pair is added to W and that CHECK can identify other cases for this formal pair as aliases. This case is also shown in Figure 21, where (X3, Y3) are aliases. (X3, Y3) are created by call site S3 and S5. Assume call site S3 is removed. By steps 2(a) and 3 of the first phase, $W = \{(Z, X3), (Z, Y3), (X3, Y3)\}$. By step 4, the pairs (X3, S3) and (Y3, S3) are removed from BINDLIST(Z). Z is removed from AS(X3) and AS(Y3), through similar reasoning given in case (c). In dealing with the pair (X3, Y3), VISIT recognizes that $X3 \in AS(Y3)$. The first part of CHECK applies to this case and results in examination of S5 since TO(S5) = P3. BIND(S5, X3) = Y1 which is an element of V_r; BIND(S5, Y3) = Y and $Y \in AS(Y1)$. Hence, CHECK returns false and AS(Y3) remains unchanged.

```
program P;
var X, Y, Z : integer;
 procedure P3 (var X3, Y3 : integer);
 begin....end;

 procedure P1 (var  Y1 : integer);

  procedure P2(var  X2 : integer);
   var  K : integer;
   begin
    P1(K)             S6
   end;

  begin (* P1 *)
   P2(Y1);            S4
   P3(Y1, Y)          S5
  end;

begin (* P *)
 P1(X);               S1
 P1(Y);               S2
 P3(Z, Z)             S3
end.

actuals        BINDLIST
-------        ------------

  X            (Y1, S1)
  Y            (Y1, S2) --> (Y3, S5)
  Z            (X3, S3) --> (Y3, S3)
  Y1           (X2, S4) --> (X3, S5)
  K            (Y1, S6)

AS(Y1) = {X, Y}
AS(X2) = {X, Y, Y1}
AS(X3) = {X, Y, Z}
AS(Y3) = {Y, X3, Z}
```

Figure 21. An example for cases (c) and (d).

4.3.6. Time Analysis of REMOVE-ALIAS

To establish the time complexity of REMOVE-ALIAS, we consider each loop structure separately.

The WHILE-loop in REMOVE-ALIAS is executed $|W|$ times. As discussed in section 4.3.3, $|W|$ is at most bounded by $s_r * |VISIBLE(TO(s))|$. This is the number of times that VISIT is called from REMOVE-ALIAS.

The three loops of VISIT are exactly the same as those in Figure 13. Thus the time complexity for VISIT follow the arguments given in section 4.2.3 and is $O(|bindings|)$ for each possible alias pair.

Each of the three FOR-loops in FUNCTION CHECK is executed at most the maximum number of calls to a procedure. For each possible alias pair at most one of the FOR-loops will be executed. Then, CHECK is $O(N_s)$ for each possible alias pair.

The time requirement for REMOVE-ALIAS is

$$O(s_r * |VISIBLE(TO(s))|) * O(|bindings| + N_s)$$

The size of N_s is small in comparison to the $|bindings|$. Thus, the REMOVE-ALIAS algorithm is

$$O((s_r * |VISIBLE(TO(s))|) * |bindings|)$$

The members of W are pairs of variables that are flagged as possible non-aliases. In the worst case, every possible alias pair in a program can be related to the members of W. The number of possible alias pairs for a given program is exponential in the worst case.

Therefore, the REMOVE-ALIAS algorithm is exponential in the worst case.

The expected complexity of REMOVE-ALIAS follows the same arguments given in section 4.3.3 for ADD-ALIAS and is linear in $|W|$.

4.3.7. Validity of REMOVE-ALIAS Algorithm

The condition for two variables not to be possible aliases is the exact opposite of that for possible aliases.

By definition, two variables are possible aliases if they are aliases in some environment. Two variables X, Y being aliases implies a sequence of pairs of variables (x_1, y_1) .... (x_n, y_n) such that $X=x_1$, $Y=y_1$, $x_n=y_n$, and for every i, $1 \leq i \leq n$ there is a call site s_i for which

1. $BIND(s_i, x_i)=x_{i+1}$ and $BIND(s_i, y_i)=y_{i+1}$
2. $BIND(s_i, x_i)=x_{i+1}$, $y_i=y_{i+1}$, and y_i is in $GLOBAL(TO(s_i))$, or
3. $BIND(s_i, y_i)=y_{i+1}$, $x_i=x_{i+1}$, and x_i is in $GLOBAL(TO(s_i))$.

In contrast, for two variables X, Y to not be possible aliases, X, Y must not become aliases in any environment. The condition for variables not to be possible aliases is that there does not exist any sequence of pairs of variables as described above.

Assume we have the correct AS sets found in a previous analysis (using Banning's ALIAS algorithm). A call site is removed and we use the REMOVE-ALIAS algorithm to update the AS sets.

Theorem 4: REMOVE-ALIAS terminates and is correct.

Termination: The body of VISIT and subsequently CHECK is executed at most once for each pair of possible non-aliases related to members of W.

Each FOR-loop in CHECK is executed at most N_s times, where N_s is the total number of call sites in the program. Thus, each invocation of CHECK must terminate in a finite amount of time. CHECK is invoked by VISIT a finite number of times. Each FOR-loop in VISIT is executed and consequently calls VISIT a finite number of times. Then any invocation of VISIT terminates in a finite amount of time.

The body of REMOVE-ALIAS calls VISIT a finite number of times. That is for each member of W. Each time VISIT is called at this point, an element is removed from W. The worklist, W, eventually becomes empty and REMOVE-ALIAS terminates.

Correctness: To prove correctness of the REMOVE-ALIAS algorithm, we need to show that it deals with the following cases correctly.

(a) AS sets are correctly updated. That is, the impossibility or possibility of aliases can be determined by the algorithm given a pair of possible non-aliases.

(b) The sequence of pairs of variables related to a possible non-alias pair can be determined by the algorithm.

To show (a), we note that after deletion of a call site, all pairs of variables marked as possible non-aliases are recorded in the worklist, W. Thus, a pair $(X, Y) \in W$ might map to different locations in the environment under consideration.

To determine the impossibility of alias pair (X, Y), REMOVE-ALIAS must establish that (X, Y) are not aliases in any environment. This can be done by showing that no call site s_i with the properties stated at the start of this section (in the definition for possible aliases) exists for (X, Y).

VISIT is called with (X, Y) which subsequently calls CHECK with this pair of variables. For each pair (X, Y), one of the FOR-loops in CHECK is executed. The first loop in CHECK determines whether there is a call site s_i with the property that

$$BIND(s_i, x_i)=x_{i+1} \text{ and } BIND(s_i, y_i)=y_{i+1}$$

The second loop in CHECK deals with the second condition for a call site which is

$$BIND(s_i, x_i)=x_{i+1},\ y_i=y_{i+1}, \text{ and}$$
$$y_i \text{ is in } GLOBAL(TO(s_i)).$$

The third loop in visit determines the existence or non-existence of a call site with the following property

$$BIND(s_j, y_j)=y_{i+1}, x_i=x_{i+1}, \text{ and } x_i \text{ is in } GLOBAL(TO(s_j)) .$$

In each case if no such call site exists then CHECK returns true and AS(Y) is updated. If a call site which satisfies one of the above properties exists then check returns false. This implies that another sequence of pairs of variables exists which results in (X, Y) becoming possible aliases. Thus alias sets are not updated.

Since, alias sets are not updated until the impossibility of possible alias is established, REMOVE-ALIAS updates the AS sets correctly.

Proof of (b) follows directly from correctness proof for the ALIAS algorithm. The three loops in VISIT determine the sequence of pairs of variables related to members of the worklist. These correspond directly to procedure VISIT in ALIAS algorithm.

The correctness of the solution from a previous analysis and the correctness of the algorithm in dealing with cases (a) and (b) result in the correctness of the REMOVE-ALIAS algorithm.

4.3.8. Space Complexity for Incremental Alias Computation

In this section, the space requirement for both alias addition and deletion is considered.

The storage space required to save information from one analysis to the next:

1) For each procedure, p:

- name
- GLOBAL(p)
- reference parameters.

One word is required for the name and $\frac{|VISIBLE(p)|}{B}$ words for each of the two bit vectors.

2) For each call site, s:

- call number
- TO(s)
- FROM(s)
- bindings (actual, formal)

One word is required for each of the three first elements. 2 * |bindings| words are required for bindings.

3) For each reference parameter, F:

- AS(F)

This is a bit vector of size VISIBLE(p)- which requires $\frac{|VISIBLE(p)|}{B}$ space.

4) For each actual parameter, A:

- BINDLIST(A)

This is a linked list of pairs of variable and call site. Three words are required for each pair plus the pointer. An extra word is required to store the header.

5) Dictionary of variables:

- name
- number

Two words for each variable.

The space requirement for 2 and 4 increases by ADD-ALIAS and decreases by REMOVE-ALIAS algorithms. Both algorithms require a worklist, W. This worklist is implemented as a linked list where each element contains a pair of variables and a pointer. Thus, each element requires 3 words and the worklist requires 3 * |W| words. |W| is s_r * |VISIBLE(TO(s))|. The space requirement for incremental alias is

$$N_p + 2N_v + 3 * (s_r * |\, VISIBLE(TO(s))\, |) + \frac{|\, VISIBLE(p)\, |}{B} * (2N_p + N_r) + N_s(3+2b_s) + N_a(1+3b_a)$$

where,

- N_p total number of procedures,
- N_r total number of reference parameters,
- N_v total number of variables,
- N_s total number of call sites,

N_a total number of actual parameters,

B word size,

b_s total number of bindings for a call,

b_a total number of bindings for each actual parameter,

s_r total number of reference parameters associated with call site s.

In any reasonably structured program, the terms N_r, s_r, N_a and VISIBLE(p) are all smaller than N_v (VISIBLE(p) and VISIBLE(TO(s)) are equal). That is, N_v is an upper bound on each of these terms. Moreover, the terms b_s and b_a tend to be small in most programs. Then, the space requirement formula can be reduced to $N_v + N_p + N_s$.

For any correct program, N_p is smaller than N_s (since otherwise, there may exist a procedure which is never called). In addition, an upperbound on N_s can be N, where N is the total number of statements in a program. Then, the incremental alias computation require N_v + N words of storage which is in reality the program size. Thus, the space complexity of the incremental alias computation increases in the size of the program.

4.4. Necessary Aliases

In this section, we briefly discuss the exhaustive and incremental necessary alias computation. Two variables are necessary aliases, if

they are aliases in every environment in which they are both mapped. This is strictly a must information and is essential in determination of must side-effects of call statements.

Banning's method for computing the necessary alias (IDENT) informations involves the following steps:

(1) The algorithm begins by initializing the IDENT set for each reference parameter to its AS set. That is, possible aliases are the first approximation to the necessary aliases.

(2) It then finds every pair of non-aliases and all the pairs of aliases which are directly related to each of them.

(3) For each such alias pair, the algorithm updates the appropriate IDENT set to reflect the fact that this pair of possible aliases is not a necessary alias pair. The IDENT sets for the pairs of variables related to this non-necessary alias pair are then updated by means of a longer chain.

To incrementally update the necessary alias solutions of a previous analysis, we make use of the solutions found by incremental possible alias computation.

Suppose a call site is removed and it has been established (by the REMOVE-ALIAS algorithm) that the pair of variables (x, y) are no longer possible aliases. That is, x is removed from AS(y). Thus, there is now at least one environment in which both x and y map and the pair(x, y) are non-aliases. If x is a member of IDENT(y), then it must be removed from IDENT(y)

and all pairs of variables related to pair (x, y) should be examined and their IDENT sets must be accordingly updated. On the other hand, if x is not a member of IDENT(y), then the information obtained from incremental possible alias computation has no effect on the necessary aliases.

In contrast, suppose a new call site is added and it has been established (by the ADD-ALIAS algorithm) that the pair (x, y) are possible aliases. Addition of x to AS(y), can introduce new necessary aliases. If x was not previously a member of IDENT(y), then it may now belong in the set. To be able to add x to IDENT(y), we must ensure that pair (x, y) are aliases in every environment in which they are both mapped. If this is the case, after updating IDENT(y), every pair of variables related to (x, y) must be examined and their IDENT sets should be accordingly updated.

CHAPTER 5

SIDE EFFECT CALCULATION

Another aspect of interprocedural analysis is the determination of the side effects of procedure calls. To find the summary information of a call statement, we must find the effects of the called procedure and its descendants on the environment of the calling procedure.

The side effects of concern for a call statement s are:

MOD(s) - The set of variables whose values may be modified by an execution of s.

REF(s) - The set of variables whose values may be inspected or referenced by an execution of s.

USE(s) - The set of variables whose values may be inspected by an execution of s before being defined.

DEF(s) - The set of variables whose values must be defined by every execution of s.

There are a number of characteristics of these side effects which influence the method for finding them. We will discuss some of the more important characteristics.

May side effects - The side effects MOD, REF and USE are in this category. Each of these side effects consists of variables about which a weak claim is made. The weakness is that these variables are affected by some, but not necessarily all calls.

Must side effects - DEF is an example of a must side effect. It consists of variables about which a strong claim is made. That is, those variables which are defined on every execution of a call statement. Must side effects are considerably harder to determine than may side effects.

Flow sensitive side effects - DEF and USE fall in this category. The determination of these values depends on the flow through a piece of code as well as upon its constituents.

Flow insensitive side effects - The side effects MOD and REF are flow insensitive. That is, their calculation depend only on the contents of the code. Flow insensitive side effects are easier to calculate in comparison to flow sensitive ones.

A perfectly accurate determination of the side effect of a procedure call is an undecidable problem. The accurate calculation depends on the possible states of program variables at the point of each call. For example to accurately determine the effect of a procedure call, we must be able to at least determine whether or not a certain statement that modifies or uses a variable will even be executed. Since a precise solution to flow sensitive side effects can not be calculated, heuristics are used to

compute the closest approximation to the most precise solution.

The may/must distinction is of importance in determining what constitutes a valid approximation to a side effect. Any underestimate or subset of the most precise information is a safe approximation to a must side effect and any overestimate or superset is safe for may side effects.

The basic method for calculating side effects considered in this Chapter is due to Banning (1978). The method involves solving a flow problem on a graph. The graph's nodes correspond to procedures and the edges correspond to calls between procedures. Associated with each edge is a function that describes how the calling procedure's side effects depend on the side effects of the called procedure. By solving this problem, the algorithm assigns to each procedure generalized side effects for the procedure. The side effects of a call on a procedure can easily be derived from the called procedure's generalized side effects.

In the remainder of this Chapter, we deal with flow sensitive and insensitive side effects separately. In each case, we will first describe Banning's approach for finding the side effects and then present our incremental update algorithm.

5.1. Flow Insensitive Side Effects

In the discussions in this section we will deal with only MOD side effects. The determination of REF side effects follows directly.

5.1.1. Exhaustive Algorithm

To find the MOD side effects, Banning performs global flow analysis on the reverse calls graph of a program.

A program's reverse calls graph has the following properties:

(1) A node corresponding to each procedure in the program.

(2) A directed edge from node p to node q for every call in procedure q to p.

The reverse calls graph is used to help in finding for each procedure a generalized modification side effect (GMOD). The method for finding the GMOD side effect involves the construction of a flow problem for the reverse calls graph. The construction is as follows:

(1) Assign to each procedure node the set of variables immediately modified by the procedure IMOD(P). IMOD can be thought of as an initial approximation to the generalized side effect.

(2) Assign to each edge for a call site s a function for that call site which maps sets of variables into sets of variables as follows:

$f_s(X) = \{ PASS(s, x) \mid x \in X * GLOPARM(s) \}$

PASS(s, x) is the variables passed to x by call site s.

GLOPARM(s) is the set of variables global to the procedure called by s plus the set of reference parameters of that procedure.

(3) A path function f_E is defined for any path $E=e_1, \cdots, e_n$ as follows:

$$f_E = f_{e_1} \circ \cdots \circ f_{e_n}$$

Then the meet over all paths solution to this problem is found. This solution assigns to each node p the union of f_e(IMOD(q)) for every path E to node p from any node q. This set is called GMOD(p) and it contains two kinds of variables:

(1) Variables which are global to that procedure and are modified by calling it. These will be visible at any site which calls this procedure and then will be in the

MOD side effect of any such call site.

(2) Reference parameters declared in the called procedure which are modified by executing the procedure. The side effect of any call on this procedure will include the actual parameters which are bound to these formal parameters.

The direct modification side effect for any call s (DMOD(s)), such that TO(s) = p, can easily be calculated from GMOD(p). DMOD(s) = f_s(GMOD(p)), where f_s is the edge function for call s. DMOD(s) contains two kinds of variables: modified variables global to p and the actual parameters which are bound by call s to modified formal parameters of p.

5.1.2. Incremental Algorithm

The program changes that may affect the side effect solution of a previous analysis are as follows:

(1) Addition of a new call site, s

(2) Deletion of an existing call site, s

(3) Changes in the IMOD side effect of a given procedure

(4) Changes in the GLOPARM set of a given procedure.

The first step of our incremental update algorithm is to construct the worklist W and update the side effect of the updated procedure where applicable. The second step is to propagate the changes to other affected procedures.

In describing the first phase of the algorithm, we deal with each type of program change (stated above) separately.

The first type of program change involves addition of a new call site. Assume call site s has been added. The site s calls procedure q from procedure p, i.e., TO(s) = q and FROM(s) = p. To deal with this modification, the following information must be found:

(1) $DMOD(s) = GMOD(q) * GLOBAL(q) + \{ BIND(s, Y) \mid Y \in GMOD(q) \}$

If DMOD(s) is changed then perform the following steps:

(2) Find MOD(s) using DMOD(s) and aliases

(3) $GMOD(p) = EIMOD(p) * GLOPARM(p)$
where EIMOD(p) is extended IMOD(p),
$EIMOD(p) = IMOD(p) + \{MOD(s') \mid FROM(s')=p\}$

(4) If GMOD(p) is changed, then
$W = \{s' \mid TO(s') = p\}$

The second type of program change results in removal of DMOD(s) and MOD(s). GMOD(p) is then updated using step 3 above (procedure p contains the removed call site). The worklist W is constructed using step 4 above.

The other two types of program changes can be dealt with by steps 3 and 4 above.

The second phase of the incremental algorithm is to propagate the changes to all affected procedures or nodes of the reverse calls graph. The algorithm for performing this propagation is presented in Figure 22.

5.1.3. Time Complexity

The complexity analysis for incremental side-effect calculation is data structure dependent. In our implementation, MOD and DMOD are implemented as PASCAL sets. These are bit vectors of size |VISIBLE(p)| which are associated with each call record.

The sets IMOD, GMOD, GLOBAL and GLOPARM are all bit vectors of size |VISIBLE(p)| which are associated with each node of the flow graph, that is with each procedure. The sets NEWDMOD, NEWGMOD and EIMOD are also bit vectors of size |VISIBLE(p)|. The worklist, W, is a bit vector of size N_s. The body of *update* is executed once for each member of W, the worklist. This however, may be more than N_s times since a call site can be added several times due to recursion (cycles in the call graph).

The FOR-loop in procedure *expand* which is called from *update* is executed at most once for each reference parameter of the given procedure. The body of the loop requires at most

```
PROCEDURE side-effect;

 PROCEDURE update (s ∈ call-sites;  p ∈ procedures);
 BEGIN
  expand(DMOD(s),MOD(s))
  EIMOD(p) := IMOD(p) + {MOD(s') | FROM(s')=p}
  NEWGMOD := EIMOD(p) * GLOPARM(p)
  IF GMOD(p) <> NEWGMOD THEN
   BEGIN
    GMOD(p) := NEWGMOD
    W := W ∪ { s_i | TO(s_i) = p}
   END
 END

BEGIN  (* side-effect *)
 WHILE W <> φ DO
  BEGIN
   (* let s be some member of W *)
   W := W - [s]
   NEWDMOD := GMOD(TO(s)) * GLOBAL(TO(s)) +
            { BIND(s, Y) | Y ∈ GMOD(TO(s)) }
   IF DMOD(s) <> NEWDMOD THEN
    BEGIN
     DMOD(s) := NEWDMOD
     update (s, FROM(s))
    END
  END
END
```

Figure 22. A procedure for calculation of incremental flow insensitive side effects.

```
PROCEDURE expand (p ∈ procedures;
              DMOD : subset of VISIBLE(p);
               VAR  MOD : subset of VISIBLE(p));
BEGIN
 MOD := DMOD;
 FOR every v in VISIBLE(p) * V_r DO
  IF v is in DMOD
   THEN MOD := MOD + AS(v)
    ELSE IF DMOD * AS(v) <> {}
        THEN MOD := MOD + {v}
END
```

Figure 23. A procedure for converting DMOD(s) into MOD(s) (Banning 1978).

two bit vector steps. Then *expand* requires 2 * |r| bit vector steps where |r| is the maximum number of reference parameters for each procedure.

In *update* , the computation of EIMOD requires |f| bit vector steps where |f| represents the maximum number of calls contained in a procedure. One bit vector step is required to calculate NEWGMOD. To determine what is added to W, |t| bit vector steps are required where |t| represents the maximum number of calls made to a procedure.

Thus, procedure *update* requires a total of

$$2 * |r| + |f| + |t| + 1$$

bit vector steps.

The computation of NEWDMOD in *side-effect* requires $1 + b_s$ bit vector steps where b_s represents the maximum number of bindings for each call. The set NEWDMOD is found for each member of W, the worklist. Procedure *update* is called from *side-effect* at most once for each member of W. Then the time requirement for each member of W is

$$2 + b_s + 2*|r| + |f| + |t|$$

bit vector operations. The total number of elements introduced in W is different in the presence and absence of recursion. To determine the worst case time complexity of side-effect algorithm, we deal with each of these situations separately.

In the absence of recursion, assuming procedures are in reverse invocation order and the order on call site is the order in which they appear in the program, W in the worst case is of $O(N_s)$. This is due to the fact that there are no cycles in the call graph. Once a bit is set in the side effect of a procedure, it needs to be propagated along every path starting from the modified procedure. Since no such path can cycle, its length in the worst case is N_s. The time requirement of side-effect propagation, in the absence of recursion, is

$$N_s * (2 + b_s + 2 * |r| + |f| + |t|) \text{ bit vector steps.}$$

The parameters b_s and $|r|$ are negligible in size to $|f|$ and $|t|$. The upperbound on $|f|$ and $|t|$ is N_s. Thus, the side-effect algorithm in

the absence of recursion is $O(N_s^2)$ bit vector steps in the worst case.

In the presence of recursion, a call site can be added to W several times due to cycles in the call graph. Suppose a variable is introduced in the side-effect of a procedure by a program modification. To propagate this information, we need to consider the effects of aliases.

If there is no change in the previous alias solution, then the new side-effect information needs to be propagated through every non-cyclic path. The existence of cycles have no effect due to the fact that no new variables will be introduced by propagating through the non-cyclic path. Then, the time complexity of side-effect propagation with recursion and no changes in aliases in the program is the same as that in the absence of recursion.

If there are changes in the previous alias solution, then by propagating the new variable side effect through every non-cyclic path, new variables may be introduced which need to be propagated through the back arcs in the call graph. For each variable introduced along the propagation path, it is possible to introduce all the arcs of the call graph in the worklist. That is, N_s call sites can be introduced in the worklist. In the worst case, In the worst case, for each new variable side effect, N_v variables may be introduced along the propagation path. In addition, there is the possibility of N_v new variable side effect.

Then, in the worst case, $N_v * (N_v * N_s)$ sites may be introduced in the worklist. The time requirement for side-effect propagation is then

$$(N_v^2 * N_s) * (2\, b_s + 2 * |r| + |f| + |t|)$$

bit vector steps. Then in the worst case, side-effect propagation in the presence of recursion and changes in the previous alias solutions is of

$$O(N_v^2 * N_s^2)$$

bit vector operations.

The bit vector steps in this algorithm are union and intersection. These operations can be performed by a single operation if the size of each bit vector is no bigger than the word size. Otherwise, union and intersection can be performed in time proportional to the size of the bit vector (which is at most $\frac{|VISIBLE(p)|}{B}$ for most bit vectors used except W which is at most $\frac{N_s}{B}$).

However, ***update*** is not usually called for each member of W. That is, if there is no change in DMOD then ***update*** is not called. In practice, this is usually the case since changes in the side effect of one procedure affects only a small number of procedures. The report by Banning indicates that 1.54 passes through the call graph was required for convergence of the side effects solutions using an iterative technique. In addition, the

call graph of most programs is not very complex and the presence of mutual recursion (which is the main source of the high worst case complexity bound for this algorithm) is rare. Thus, the side-effect algorithm is expected to be of $O(N_s)$ complexity in practical cases.

5.1.4. Space Complexity

The storage space required to save information from one analysis to the next is as follows:

(1) For each procedure, p:

- name
- GMOD(p)
- IMOD(p)
- GLOBAL(p)
- reference parameters.

One word is required to store the name and $\frac{|VISIBLE(p)|}{B}$ is required for each of the 4 sets.

(2) For each call site, s:

- call number
- TO(s)
- FROM(s)
- DMOD(s)
- MOD(s)

- bindings (actual, formal)

One word is required for each of the 3 first elements. DMOD and MOD are bit vectors and require $\frac{|VISIBLE(p)|}{B}$. |bindings| * 2 words are required to store the binding information.

(3) Dictionary of variables:

- name
- number

Two words are required for each variable.

The actual side-effect algorithm requires extra storage space for the following:

- NEWDMOD
- NEWGMOD
- EIMOD
- W

The first 3 sets are bit vectors of size VISIBLE(p). Thus, each require space of size $\frac{|VISIBLE(p)|}{B}$. W is a bit vector and requires space of size $\frac{N_s}{B}$.

Then, the space requirement for the incremental side-effect algorithm is

$$N_p + 3N_s + 2N_v \frac{+|VISIBLE(p)|}{B} * (4N_p + 2N_s + 3) + 2N_s * b_s + \frac{N_s}{B}$$

where,

N_p total number of procedures,

N_s total number of call sites,

b_s total number of bindings for a call,

B word size.

By the arguments presented in section 4.4.8, the space requirements for incremental side-effect calculation can be stated as N_v + N which is the program size. Then, it can be concluded that the space complexity of the incremental side-effect calculation increases in the size of the program.

5.1.5. Validity of the Side-Effect Algorithm

The incremental side-effect algorithm assumes the correctness of the exhaustive algorithm and consequently the previous solution. In proving correctness of the update side-effect algorithm, we need to show that

(1) The affected area of the reverse calls graph is correctly determined for propagation purposes.

(2) The program changes are correctly reflected in the side effect information of the affected area.

To show correctness for case (1), we note that side effects propagate from called to the calling procedures. Hence, to find the affected area of the call graph, it is sufficient to determine a call chain $C = s_1, \ldots, s_n$ where s_n calls the procedure whose side effects has changed. The change can be then propagated through the chain s_n to s_1.

The construction of the worklist in the update side-effect algorithm insures the determination of the correct call chain. W initially contains all calls s_n to the procedure whose side effect has been updated. In procedure update, whenever there is change in GMOD of a procedure, all calls to that procedure are added to W. So, in effect a change is propagated by following the sequence $s_n, \ldots, s_1$.

In proving (2), assume a program change occurs in procedure p. The first phase of the algorithm recalculates the side-effect information for procedure p. In the second phase, when necessary the side effect of other procedures and calls to those procedures are recalculated. This complete recalculation at each stage insures that all side-effect information is correctly updated after a program change.

The correctness of the update side-effect algorithm follows directly from the correctness of (1) and (2).

5.2. Flow Sensitive Side Effects

The side effects DEF and USE are determined by considering both the flow through a piece of code as well as its elements.

5.2.1. Exhaustive Algorithm

The method begins by finding summary information about flow through each of the procedures in the program. For DEF, the following quantities of information are collected.

IDEF(p)

Set of variables defined by statements directly contained in p along every path through p. The effects of procedures called by p are excluded.

MCALL(p)

Set of procedures which must be called during every execution of p.

MBIND(p,v)

Those variables which will be bound to v (reference parameters called by p) by some call from p during every execution of p.

A slightly different reverse calls graph is then constructed and a set of different functions are assigned to edges. This graph has a single edge from procedure p to procedure q iff p is

in MCALL(q). Thus, the reverse must call graph is used to find the DEF side effect.

Initially IDEF(p) is assigned to each procedure node p. The function assigned to an edge from p to q is

$$f_{pq}(X) = \{MPASS(q, p, x) \mid x \in X * GLOPARM(p)\}$$

where

MPASS(q, p, x) is {x} if $x \in$ GLOBAL(p)

is MBIND(q, x) if $x \in V_r(p)$

The meet over all paths solution of this flow problem is GDEF(p) - the generalized DEF side effect. For any call site s which calls procedure p, we can find the direct definition side effect (DDEF(s)) by applying the edge function for call s to GDEF(p). Then

$$DDEF(s) = f_s(GDEF(p))$$

DDEF(s) is the set of all variables X for which there exists a must call chain

$$C = s_1, \ldots, s_n$$

and variable Y such that $s_1 = s$, C must pass X to Y and Y must be in DDEF(TO(s_n)). Thus, due to the must characteristics of DEF, we look at only what is defined, called, or bound during every execution of a procedure and propagate side effects according to these restrictions.

The side effect USE is dependent on the DEF side effect. Therefore, the summary information for use is collected in two steps and is more complex to determine. In the first step, information about variable usage is collected, without any knowledge of the DEF side effects of calls. The second step is applied once the DEF side effect is known. This step combines the information obtained in the first step with the DEF information to obtain the following information:

IUSE(p)

Set of variables which may be referenced by statements directly contained in procedure p without first being defined by statements in p.

PDEF(s)

Set of variables always defined by statements in the procedure containing call site s, before the call site is executed. This includes definitions due to other call sites in the procedure.

The initial assignment to each procedure p is IUSE(p). The function associated with the edge for call s is

$$f_s(X) = PDEF(s) * f_s(X)$$

The meet over all paths solution to this flow problem is the generalized USE side effect (GUSE(p)). The direct usage side effect DUSE(s) can then be calculated from the GUSE side effect.

$$DUSE(s) = f_s(GUSE(p)) .$$

5.2.2. Incremental Algorithm

The arguments given in this section for incremental flow sensitive calculation are similar to those for flow insensitive side effects. In fact, the calculation for the side effect USE is exactly the same as that for MOD.

In addition to program changes stated for flow insensitive side effects, the USE side effect must also be updated when PDEF of a call site is changed. The first phase of the algorithm computes the USE side effect for the directly affected procedure and call site, it also initializes the worklist (as described for MOD). The algorithm used in the second phase is the same as that for MOD.

The only difference is in the equations used, which are the following:

- $DUSE(s) = (GUSE(s) * GLOBAL(q) + \{BIND(s, Y) \mid Y \in GUSE(q)\}) - PDEF(s)$
- Compute USE(s) from DUSE(s) and aliases.
- $GUSE(p) = EIUSE(p) * GLOPARM(p)$

  where,

  $EIUSE(p) = IUSE(p) + \{USE(s') \mid FROM(s')=p\}$

The incremental calculation for the DEF side effect is slightly different due to the dependence of DEF on must call chains and necessary aliases. The DEF side effect must be updated in the following cases:

(1) Addition of a new call site, which results in addition of a new procedure to MCALL(p).

(2) Deletion of a call site which results in deletion of a procedure from MCALL(p).

(3) Changes in IDEF(p).

(4) Changes in global or reference parameters of P.

With the exception of the way W, the worklist, is constructed, the incremental update DEF algorithm is the same as that for MOD side effect. Since DEF is a must side effect, the worklist is defined as:

$$W = \{s \mid TO(s) \in MCALL(FROM(s))\}$$

The equations used are:

- $DDEF(s) = GDEF(q) * GLOBAL(q) + \{MPASS(q, p, x) \mid x \in GDEF(q)\}$
- Find DEF(s) using DDEF(s) and necessary aliases
- $GDEF(p) = EIDEF(P) * GLOPARM(p)$

where,

$$EIDEF(p) = IDEF(P) * \{DEF(S) \mid TO(s) \in MCALL(p)\}$$

CHAPTER 6

CONCLUSIONS

This Chapter summarizes our work in the development of incremental update algorithms for data flow analysis and suggests areas for future research.

6.1. Summary

The major contribution of this thesis is a set of incremental update algorithms for global and interprocedural data flow analysis.

All algorithms are designed as a two-step process and use a worklist which contains work to be done. In each algorithm, the first phase deals with the data flow solutions of the immediately affected area, removes suspect values from old solution and initializes the worklist. The second phase propagates the immediate changes resulting from a program modification to all affected areas of the graph. The major difference between these algorithms is the way in which the worklist is constructed.

Our incremental global flow analysis algorithms are based on Hecht/Ullman's iterative algorithms. We presented incremental reaching definitions and incremental live variable analysis as examples of forward and backward flow problems,

respectively. With the exception of the worklist, the two algorithms are alike due to the similarities of their exhaustive counterparts. In a forward flow problem, the worklist contains the set of immediate successors of the affected node. The immediate predecessors of the affected node are the members of the worklist in a backward flowing problem.

The incremental interprocedural analysis algorithms presented in this thesis are based on the exhaustive ones designed by Banning. These algorithms are designed to deal with a language with PASCAL like scope rules, pass by reference parameters and recursion.

Interprocedural data flow analysis consists of two problems which have been dealt with separately in this thesis. The first problem considered is the method of updating the possible alias solutions of a previous analysis after a program modification has occurred. We presented two algorithms to deal with insertion and deletion of call sites separately. In these algorithms the worklist consists of pairs of variables associated with the affected call sites. Each pair is either flagged as a candidate for possible aliases or non-aliases depending on the modification type.

Updating the side effects of a procedure call after a program modification is the second problem in incremental interprocedural analysis. We dealt with flow insensitive and flow sensitive side effects separately. With the exception of the data

flow equations, the algorithms for the two types of side effects are similar. The worklist consists of the set of call sites to the affected procedure.

The analysis of these algorithms were found to be very much data structure and machine dependent. The worst case time bound of each of the incremental algorithms were found to be equal to their exhaustive counterparts. The average time bound of all algorithms were also computed using available empirical evidence.

The conclusion that can be drawn from these analyses is that the usual analysis techniques are not suitable for determining the complexities of the incremental algorithms. In many cases the effect of a small program change can not be generalized with the available analysis tools. To precisely analyse the incremental algorithms, empirical evidence is needed for the following :

(1) Expected types of program modifications.

(2) The effects of program modification in various programming languages.

(3) Analysis of different data structures for implementation of these algorithms.

The implementation of our PASCAL source level analyser helped us in determining the complexity results for the incremental algorithms. This incremental system is based on an existing analyser, called SOAP, that was designed for the one

time analysis of PASCAL programs for the purposes of source-level optimization and anomaly detection (Hughes 1981). In order to perform its functions, SOAP scans a program creating for each procedure its internal control and data flow representation. After processing for each procedure is completed, the storage for its internal representation is freed to make room for subsequent procedures. To this base, we added code to incrementally compute reaching definitions, live variables, possible aliases and flow insensitive side effects. The main objective of this implementation was to show the practicality of our algorithms. However, its most important contribution was the insight that it provided for analysing the algorithms. The algorithms were found to be straightforward to implement and easy to maintain.

The analyser as implemented uses bit vectors to represent sets of variables and statements. Use of bit vectors leads to clean and easy to maintain code. However, the lack of support for bit vectors in PASCAL lead to some unexpected timing problems. The other speed limitation was due to the sequential nature of standard PASCAL files. To take care of one line of source change, the entire data file containing the result of a previous analysis had to be read in main memory and eventually written back to the secondary memory. This process is rather time consuming. However, assuming the existence of a good programming environment, the routines for random access use

can be written in another language and incorporated in the system.

6.2. Future Directions

There are a number of ways in which this research can be extended and we conclude with the discussion of some of them.

(1) The first obvious extension of our work is to implement the algorithms using a more suitable programming language and then to gather statistics and find average complexities based on empirical evidence.

(2) A suitable next step is to extend our incremental interprocedural analysis algorithms to deal with all aspects of the PASCAL language.

(3) After analysis of PASCAL is completely understood, it would be appropriate to design incremental algorithms that deal with constructs available in some of the newer languages such as ADA.

(4) Since the creation of an analyser for each and every new programming language is a formidable task, there is a need for the design and implementation of an automated incremental data and control flow analyser generator that is independent of any programming language.

(5) An excellent application of the techniques devised here can be found in the design of incremental update algorithms for metrics and testing purposes based on data flow analysis. However, it should be noted that research into the application of data flow analysis to these areas is still rather new and suitable exhaustive algorithms need to be found first.

(6) A final extension is to implement these tools within a comprehensive programming environment in the manner described in the Appendix.

APPENDIX

APPLICATION TO SOFTWARE DEVELOPMENT SYSTEMS

An overview of how the research reported in this dissertation may be used in the design of a new software development environment based on incremental program analysis is presented here. This environment is discussed primarily to validate the use of our incremental algorithms and to show how these can be clearly integrated into a complete programming environment.

The main goal of any software development environment is the design and development of highly reliable software on schedule and with the minimum life cycle cost. As explained in Chapter 1, the proper achievement of this goal involves consideration of the following two factors:

(1) Design and development of software which is easier to modify, test and maintain.

(2) Detection of errors in the early stages of the life cycle.

It is our belief that a software development environment which facilitates tight supervision based on continuous analysis of source programs and the design code can achieve such goals.

Continuous analysis permits the collection of essential data flow, control flow and metrics informations at both the design and implementation phases of the life cycle. The analysis process should start after the functional specification phase and continue throughout the life cycle. The results of such analysis can be inspected by project leaders in order to find design and implementation flaws, and be used as the basis of various tools by the system.

The proposed environment consists of four major components which together provide an integrated set of development tools. The four components are

- the design analyser,
- the source code analyser,
- the programmer's tool kit, and
- the manager's tool kit.

Both analysers, depend on the incremental data flow analysis and incremental metrics algorithms which have been proposed in this paper. Incremental program analysis is important in carrying out the desired continuous analysis in a reasonable time frame, with minimum cost.

The analysis at the implementation phase reveals data flow anomalies, possible deviations from design and methodology and

errors not detected at the design phase. Tight supervision and continuous comparisons between metrics calculated here and those evaluated in the design phase are essential in detection of all errors at this stage.

The underlying principle in the design of the tool kits is to provide as much assistance as possible to both the programmers and their managers. All the tools will be in some way dependent on the output from one or both of the analysers. Hence, all constituents of the tool kits provide a unified view of the whole system to their user community.

A.1. The Design Analyser

The activities in the design phase begin with the study of the requirement/specification document. A design methodology is then chosen and a software design document is prepared.

The software design document is then coded in a design specification language. Here, we assume the existence of an appropriate design specification language. This language should provide assistance in the clear specification of the following desired characteristics:

- The data structures,
- The breakdown of the procedures/modules,
- The interfaces between procedures/modules, and

- The control flow information.

The purpose of the design analyser is to incrementally parse the design code and collect all the necessary control and data flow information for metric evaluation and graphical representation of the control structure.

The quantitative evaluation of the design document is essential in finding errors generated at this stage. This analysis can result in either some modification to the design document or complete redesign in extreme cases. As explained in Chapter 1, the detection and correction of design errors early in the cycle is beneficial in reducing the total cost of the system. The design analyser is invoked automatically by the system, when the system version of the design code is updated. The output from this component is saved for further interrogation by other tools and by the analyser itself.

A.2. The Source Code Analyser

The source code analyser is automatically invoked by the system when the system version of the source code is updated. Its main function is to incrementally collect the necessary intraprocedural and interprocedural summary data flow information. The analyser also gathers control flow and possibly other information for metric calculation.

The output from the source code analyser is saved for possible modification by itself and for use by other tools.

A.3. The Programmer's Tool Kit

This tool kit consists of a collection of integrated tools that provide assistance to the programmer in the implementation, testing and maintenance phases of the life cycle.

A.3.1. Anomaly Detector

The anomaly detector examines the output produced by the source code analyser and reports on the detected data flow anomalies. Anomalies reported include the following:

- Definitions of variables with no subsequent use.
- Use of variables with no prior definitions.
- Global declarations of loop indices.
- Global variables that are only used locally.
- Loops governed by a condition that is invariant across the loop body because none of the control variables in the loop body are changed.

A.3.2. Optimizer

This is essentially a source level optimizer. It examines the output produced by the source code analyser and reports on code segments which are dead and can be removed.

A.3.3. Documentation Generator

The documentation generator is partly automated and is designed to assist the programmer. By examining the source code analyser's output, it makes a list of all the local and global variables, calling procedures and called procedures for each procedure. The programmer is then prompted for the possible semantic explanation of the procedure. This information is then inserted just before the first statement of the procedure , as comments in the source code.

A.3.4. Test Case Generator

The technique used here is that of data flow path testing. The methodology is that testing should be done incrementally throughout program implementation. Data flow paths are tested symbolically and clearly recorded in a data base. These records consists of the paths tested and symbolic test results for each path.

When this tool is invoked, it examines the source code analyser's output and identifies the data flow paths. A comparison is then made between these paths and those in the test data base. A report is generated both graphically and textually which identifies

- the new paths which should be tested, and
- the extensions to the previously tested paths, and the symbolic output of these paths.

The underlying philosophy behind the design of this tool is to provide assistance to the programmer in designing new test cases.

A.3.5. Maintenance Tool

The purpose of this tool is to provide assistance to the programmer in making a valid and reasonable modification during the maintenance phase. It tracks and reports the data flow relationships of an intended change.

Upon invocation, the maintenance tool takes the following steps:

(1) Prompts the user for the intended modification.

(2) Calls on the source code analyser to do a simulated analysis of the modification request. This will be a simulated analysis in the sense that the actual analyser's output is not updated.

(3) Reports the results of the source code analysis. This report consist of the global affects of the intended modification, the use/def history of each of the variables used in the specified change and all the aliasing relationships of them.

The programmer can then make a decision based on this report, rather than on some ad hoc approach.

A.4. The Manager's Tool Kit

The manager's tool kit consists of a collection of integrated tools to aid the project leaders. These tools are designed in accordance with our basic philosophy of providing a user friendly environment as well as tight supervision.

In a large-scale programming environment, a project leader is possibly in charge of several projects and each project involves the collaboration of several programmers. To facilitate automatic supervision in such an environment, a specific project structure is required.

A project directory is created by the project leader, at the time of initiation of each project. Associated with each directory are

- an access list,
- a collection of source programs,

- a collection of modification histories
  for each program, and

- a set of reports on the progress of
  each program.

The access list is created and can only be modified by the project leader and it includes programmer identification and their access rights.

The modification histories and the progress reports are only accessible by the project leader. These will be automatically updated by the system. Their sole purpose is to provide a mechanism for the project leaders to evaluate the progress of each project and that of each programmer in their group. The source program is accessible by both the project leader and the programmer assigned to it.

A.4.1. Modification History

A modification history is kept for each source program and is automatically updated by the system. To provide a friendly environment, the programmers, depending on their access rights, can make a copy of their program. They can work on their copy, but at least once a day they need to update the system version of the source program. When the system version is updated, the following sequence of events takes place:

- The source code analyser is invoked to incrementally analyse the program.

- The new version of the program is compared to that of the old system version and their source code differences are determined.

- The programmer is prompted for some comments on each piece of the difference.

- The differences and the programmer supplied comments are recorded and attached to the modification history list for that program.

On those projects bounded by maximum security and time constraints, the above process can be carried out continuously. Under such circumstances, programmers are not premitted to make copies of their programs and consequently must work on the system version of the source code at all times. The modification history file can be reviewed by the project leaders at their convenience.

A.4.2. Interrogation Facility

The purpose of this tool is to assist the project leader in calculating metrics for the system. The functions provided are as follows:

- Predefined design metric calculation.
- Predefined source code metric calculation.
- Open ended design / program metric calculation.
- Graphical display of the design / program's control flow structure.

The interrogator is a simple interpretor, with calculating capabilities. It interprets queries made and automatically searches the output from one of the analysers (depending on the request). Some metric formulas are predefined in the system and depending on the query made, the interpretor provides the result of that metric calculation.

The interrogator also permits open ended metric calculation, where the formula is provided by the user. Some commands are also available for graphical display of the control structure.

A.4.3. Progress Report Generator

At different milestones in the implementation phase, the system will automatically create a progress report for each source program. This report is added to the progress report list, for later examination by the project leader. The following steps are taken in generating such a report:

- The output from the source code analyser

is examined and data flow anomalies are recorded.

- Some predefined metrics are evaluated and recorded based on the output from the analyser.

- These metrics are compared with those from design. Possible deviations from design and methodology are recorded.

Another function of the report generator is to automatically compute some predefined design and source code metrics and compare the computed metric values against some predetermined bounds. If any of these values exceeds the set bounds, the report generator alerts the supervisors through electronic mail or immediately signals them depending upon the importance of that metric value to the success of the project.

A.5. Prototyping an Environment

The content of this Appendix is meant to suggest how our research can lead to a new environment for software development. In such an environment both programmers and project managers are provided with tools that automatically aid them in their jobs. Much work is left to be done before such an environment can be a reality. Even so, existing environments such as

UNIX can be extended immediately to include some of the tools discussed here. This is one of the more immediate goals of our future research/development effort. Through such a prototyping activity we expect to learn more about the best directions to extend this research.

GLOSSARY

AS(r), is the set of aliases associated with each reference parameter r.

B, is the total number of bits in a word.

$\mathbf{b_s}$, is the total number of bindings for a call.

$\mathbf{b_a}$, is the total number of bindings for each actual parameter.

BIND(s, X), is a partial mapping $(S-\{ \mathbf{s} \}) \times V_r \rightarrow V$. BIND(s, X) gives the actual parameter which is bound to formal parameter X by call site s.

BINDLIST(a), is a set of pairs of reference parameters and call sites associated with each actual parameter a.

d, is the loop-connectedness parameter of a reducible flow graph. It is the largest number of back arcs on any cycle-free path.

DDEF(s), is the direct definition side-effect of a call site s.

DEF[B], has a meaning that varies with the context of its use. In the case of live variable analysis, DEF[B] is a set of variables assigned values in B, prior to any use of that variable in B. The **DEF** side-effect of a call site is the set of variables whose values must be defined by every execution of the call site.

DFN[B], is the depth first order number associated with each node B of a flow graph.

DMOD(s), is the direct modification side-effect of a call s.

DUSE(s), is the direct usage side-effect of a call site.

EIDEF(p), is the extended definition side-effect of procedure p. This set includes the effects of the called procedures.

EIMOD(p), is the extended modification side-effect of a procedure p. This set includes the effects of the called procedure.

EIUSE(p), is the extended usage side-effect of a procedure. This set includes the effects of the called procedures.

FROM(s), is a mapping from elements of (S-{ **s** })-> P. FROM(s) is the procedure from which the call associated with s is made.

GDEF(p), is the set of generalized definition side-effect of a procedure.

GEN[B], is the set of definitions generated within B that reach the end of block B.

GLOBAL(p), is the set of objects global to procedure p according to the rules of block-structured programs.

GLOPARM(s), is the set of variables global to the procedure called by s plus the set of reference parameters of that procedure.

GMOD(p), is the generalized modification side-effect of a procedure p.

GUSE(p), is the generalized usage side-effect of a procedure.

IN[B], has a meaning that varies with the context of its use. In the case of reaching definitions problem, it is the set of definitions reaching the point just before the first statement of block B. **IN[B]**, is the set of variables live at the point immediately before block B for live variable analysis problem.

IDEF(p), is the set of variables defined by statements directly contained in p along every path through p. The effects of procedures called by p are excluded.

IDENT(R), is the set of necessary aliases associated with the reference parameter r.

IMOD(p), is a mapping from P into subsets of V. IMOD(p) specifies the variables which may be assigned by the execution of statements in procedure p.

IREF(p), is a mapping from P into subsets of V. IREF(p) specifies the variables which may be referenced by the execution of statements in procedure p.

IUSE(p), is the set of variables which may be referenced by statements directly contained in procedure p without first being defined by statements in p.

$K_d(k)$, is the set of all nodes in a minimal length, d definition clear path of nodes from k', the source of definition d, to some node k.

KILL[B], is the set of definitions outside of B that define identifiers which also have definitions within B.

OUT[B], has a meaning that varies with the context of its use. In the case of reaching definitions problem, it is the set of definitions reaching the point just after the last statement of block B. It is the set of variables live at the point immediately after block B for live variable analysis problem.

m, is the average number of statements in a procedure.

n, is the total number of nodes in a flow graph.

N, is the total number of statements in a program.

N_a, is the total number of actual parameters in a program.

N_p, is the total number of procedures.

N_r, is the total number of reference parameters in a program.

N_s, is the total number of call sites in a program.

N_v, is the total number of variables in a program.

NEWDMOD, is a temporary set of variables representing the direct modification side-effect of a call site.

NEWGMOD, is a temporary set of variables representing the generalized modification side-effect of a procedure.

NEWIN, is a temporary set of definitions in incremental reaching definitions computation. It is a temporary set of variables in incremental live variable analysis computation.

NEWOUT, is a temporary set of definitions in incremental reaching definitions computation. It is a temporary set of variables in incremental live variable analysis calculation.

NUM(X), is the number associated with variable X.

MBIND(p, v), is the set of variables which will be bound to v by some call from p during every execution of p.

MCALL(p), is the set of procedures which must be called during every execution of p.

MOD(s), is the set of variables whose values may be modified by an execution of s.

P, is the set of procedures in the program.

p, is the main procedure, an element of P.

PASS(s, x), is the variables passed to x by call site s.

PDEF(s), is the set of variables always defined by statements in the procedure containing call site s, before the call site is executed. This includes definitions due to other call sites in the procedure.

PRED[K], is the set of predecessors of node K.

r, is the total number of arcs in a flow graph.

REF(s), is the set of variables whose values may be inspected or referenced by an execution of s.

RFG, is a reducible flow graph.

S, is a set of call sites in the program.

s, is the member of S that calls the main program.

$\mathbf{s_r}$, is the set of reference parameters associated with a call site s.

SUCC[K], is the set of f successors of node K

TO(s), is a mapping from elements of S->P. TO(s) is the procedure which is called by call site s.

USE[B], has a meaning that varies with the context of its use. In the case of live variables analysis, it is a set of variables used in block B, prior to any definition of that variable in B. The **USE** side-effect of a call site is the set of variables whose values may be inspected by an execution of that call before being defined.

V, is the set of variables in the program.

V_r, is a subset of V. It is the set of reference parameters of the program.

VISIBLE(p), is the set of objects accessible to procedure p.

W, is the worklist.

LIST OF REFERENCES

Adrion, W. R.; Branstad, M. A.; and Cherniavsky, J. C. "Validation, Varification, and Testing of Computer Software." *ACM Computing Surveys* 14 (June 1982) : 159-192.

Allen, F. E., and Cocke, J. "A Program Data Flow Analysis Procedure." *Communications of the ACM* 19 (March 1976) : 137-147.

Aho, A. V., and Ullman, J. D. "Node Listings for Reducible Flow Graphs." *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, Albuquerque, New Mexico, May 1975, pp. 127-185. New York: ACM, 1975.

Aho, A. V., and Ullman, J. D. *Principles of Compiler Design.* Reading, MA : Addison-Wesley, 1977.

Banning, J. P. "A Method for Determining the Side Effects of Procedure Calls." Ph.D. dissertation, Stanford University, 1978.

Banning, J. P. "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables." *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, TX, January 1979, pp. 29-41. New York: ACM, 1979.

Barth, J. M. "A Practical Interprocedural Data Flow Analysis Algorithm and its Applications." Ph.D. dissertation, University of California, Berkeley, 1977.

Barth, J. M. "A Practical Interprocedural Data Flow Analysis Algorithm." *Communications of the ACM 21* (September 1978) : 724-736.

Budd, T. A.; Lipton, R. J.; Sayward, F. G.; and DeMillo, R. A. "The Design of a Prototype Mutation System for Program Testing." *Proceedings of AFIPS National Computer Conference*, Arlington, Va. : AFIPS Press, 1978. Vol. 47, pp. 623-627.

Boehm, B. W. "The High Cost of Software." In *IEEE Tutorial: Software Testing & Validation Techniques*. Edited by Miller, and Howden, New York : IEEE, 1978.

Babich, W. A., and Jazayeri, M. "The Method of Attributes for Data Flow Analysis: Part I: Exhaustive Analysis." *Acta Informatica* 10 (March 1978) : 245-264.

Babich, W. A., and Jazayeri, M. "The Method of Attributes for Data Flow Analysis: Part II: Demand Analysis." *Acta Informatica* 10 (March 1978) : 265-272.

Carter, L. R. *An Analysis of Pascal Programs*. Ann Arbor, Michigan : UMI Research Press, 1982.

Clarke, L. A. "A System to Generate Test Data and Symbolically Execute Programs." *IEEE Transactions on Software Engineering* 2 (September 1976) : 215-222.

DeMillo, R. A.; Lipton, R. J.; and Sayward, F. G. "Hints on Test Data Selection: Help for the Practicing Programmer." *IEEE Computer* 11 (April 1978) : 34-43.

Demers, A.; Reps, T.; and Teitelbaum, T. "Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors." *Conference Record of the Eight ACM Symposium on Principles of Programming Languages*, Williamsburg, Va, January 1981, pp. 105-116. New York : ACM, 1981.

Fosdick, L. D., and Osterweil, L. J. "Data Flow Analysis In Software Reliability." *ACM Computing Surveys* 8 (September 1976) : 305-330.

Goodenough, J. B., and Gerhart, S. L. "Toward A Theory of Testing: Data Selection Criteria." In *Current Trends in Programming Methodology, Vol. II, Program Validation*. Edited by Yeh, Englewood Cliffs, N.J. : Prentice-Hall, 1977.

Gilb, T. *Software Metrics*. Cambridge, MA : Winthrop, 1977.

Glass, R. L. *Software Reliablity Guidebook*. Englewood Cliffs, N. J. : Prentice-Hall, 1979.

Ghezzi, C., and Mandrioli, D. "Incremental Parsing." *ACM Transactions on Programming Languages and Systems* 1 (July 1979) : 58-70.

Graham, S. L., and Wegman, M. "A Fast and Usually Linear Algorithm for Global Flow Analysis." *Journal of the ACM* 23 (January 1976) : 172-202.

Habermann, A. N. "An Overview of the Gandalf Project." *CMU Department of Computer Science Research Review*, 1978-79.

Halstead, M. H. *Elements of Software Science*. New York : Elsevier North-Holland, 1977.

Hecht, M. S. *Flow Analysis of Computer Programs*. New York : Elsevier North-Holland, 1977.

Hecht, M. S., and Ullman, J. D. "Flow Graph Reducibility." *SIAM Journal of Computing* 1 (June 1972) : 188-202.

Hecht, M. S., and Ullman, J. D. "Characterization of Reducible Flow Graphs." *Journal of ACM* 21 (July 1974) : 367-375.

Hecht, M. S., and Ullman, J. D. "A Simple Algorithm For Global Data Flow Analysis Problems." *SIAM Journal of Computing* 4 (December 1975) : 519-532.

Henry, S. M. "Information Flow Metrics for the Evaluation of Operating Systems' Structure." Ph.D. dissertation, Iowa State University, 1979.

Henry, S. M., and Kafura, D. "Software Structure Metrics Based on Information Flow." *IEEE Transactions on Software Engineering* 7 (September 1981) : 510-518.

Howden, W. E. "Reliability Of The Path Analysis Testing Strategy." *IEEE Transactions on Software Engineering* 2 (September 1976) : 37-44.

Howden, W. E. "Symbolic Testing and the DISSECT Symbolic Evaluation System." *IEEE Transactions on Software Engineering* 3 (July 1977) : 266-278.

Huang, J. C. "Error Detection Through Program Testing." In *Current Trends in Programming Methodology, Vol, II, Program Validation*. Edited by Yeh, Englewood Cliffs, N. J. : Prentice-Hall, 1977.

Hughes, C. E. "Automated Symbolic Optimization and Anomaly Detection Within PASCAL Programs." Unpublished Report to National Bureau of Standards, Gaithersburg, MD, 1981.

Kennedy, K. W. "Node Listings Applied To Data Flow Analysis." *Conference Record of the second ACM Symposium on Principles of Programming Languages*, Palo Alto, Ca., January 1975, pp. 10-21. New York : ACM, 1975.

Kennedy, K. W. "A Comparison of two Algorithms For Global Data Flow Analysis." *SIAM Journal of Computing* 5 (March 1976) : 158-180.

Kennedy, K. W. "A Survey of Data Flow Analysis Techniques." In *Program Flow Analysis: Theory and Applications*, Edited by Muchnick and Jones, Englewood Cliffs, N. J. : Prentice-Hall, 1981.

Kam, J. B., and Ullman, J. D. "Global Data Flow Analysis and Iterative Algorithms." *Journal of the ACM* 23 (January 1976) : 158-171.

Kam, J. B., and Ullman, J. D. "Monotone Data Flow Analysis Frameworks." *Acta Informatica* 7 (March 1977) : 305-318.

Knuth, D. E. "An emprical Study of FORTRAN Programs." *Software-Practice and Experience* 1 (April 1971) : 105-134.

Lomet, D. B. "Flow Analysis in the Presence of Procedure Calls." *IBM Journal of Research and Development* 21 (November 1977) : 559-571.

McCabe, T. J. "A Complexity Measure." *IEEE Transactions on Software Engineering* 2 (December 1976) : 308-320.

Medina-Mora, R. and Feiler, P. H. "An Incremental Programming Environment." *IEEE Transactions on Software Engineering* 7 (September 1981) : 472-482.

Miller, E. In "Introduction to Software Testing Technology." In *Tutorial: Software Testing & Validation Techniques*. Edited by Miller, and Howden, New York : IEEE Computer Society Press, 1981.

Myers, G. J. *The Art of Software Testing*. New York : John Wiley & Sons, 1976.

Myers, E. W. "A Precise Interprocedural Data Flow Algorithm." *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Williamsburg, Va., (January 1981), pp. 219-230. New York : ACM, 1981.

Osterweil, L. J., and Fosdick, L. D. "DAVE - A Validaion Error Detection and Documentation System for Fortran Programs." *Software Practice and Experience* 6 (December 1976) : 473-486.

Osterweil, L. J. "Using Data Flow Tools in Software Engineering." In *Program Flow Analysis: Theory and applications*. Edited by Muchnick and Jones, Englewood Cliffs, N. J. : Prentice-Hall, 1981.

Oviedo, E. I. "Control Flow, Data Flow, and Program Complexity." *Proceedings of IEEE COMPSAC 80*, Chicago, October 1980, pp. 146-152. New York : IEEE Press, 1980.

Reps, T. "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors." *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*. New York : ACM, January 1982.

Reps, T. "Static-Semantic Analysis in Language-Based Editors." *Digest of Papers of the Twenty-Sixth IEEE Computer society International Conference: Intellectual Leverage For The Information Society Spring COMPCON 83*, San Francisco, Ca. : COMPCON IEEE Computer Society, 1983.

Ramamoorthy, C. V., and Ho, S. F. "Testing Large Software With Automated Evaluation Systems." *IEEE Transactions on Software Engineering* 1 (March 1975) : 46-58.

Rosen, B. K. "High-Level Data Flow Analysis." *The Communications of the ACM* 20 (October 1977) : 712-724.

Rosen, B. K. "Data Flow Analysis for Procedural Languages." *Journal Of the ACM* 26 (April 1979) : 322-344.

Rapps, S., and Weyuker, E. J. "Data Flow Analysis Techniques for Test Data Selection." *Proceedings of the Sixth International Conference on Software Engineering*, Tokyo, Japan, September 1982, pp. 272-278. Long Beach, Ca. : IEEE Computer Society Press, 1982.

Ryder, B. G. "Incremental Data Flow Analysis Based on a Unified Model of Elimination Algorithms." Ph.D. dissertation, Rutgers University, 1982.

Taylor, R. N., and Osterweil, L. J. "Anomaly Detection in Concurrent Software by Static Data Flow Analysis." *IEEE Transactions on Software Engineering* 6 (May 1980) : 265-278.

Weyuker, E. J., and Ostrand, T. J. "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE Transactions on Software Engineering* 6 (May 1980) : 236-246.