INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Cooper, Keith Daniel

INTERPROCEDURAL DATA FLOW ANALYSIS IN A PROGRAMMING ENVIRONMENT

Rice University  Ph.D.  1983

University Microfilms International  300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages √
2. Colored illustrations, paper or print √
3. Photographs with dark background √
4. Illustrations are poor copy √
5. Pages with black marks, not original copy √
6. Print shows through as there is text on both sides of page √
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements √
9. Tightly bound copy with print lost in spine √
10. Computer printout pages with indistinct print √
11. Page(s) _________ lacking when material received, and not available from school or author.
12. Page(s) _________ seem to be missing in numbering only as text follows.
13. Two pages numbered _________. Text follows.
14. Curling and wrinkled pages √
15. Other __________________________________________
RICE UNIVERSITY

INTERPROCEDURAL DATA
FLOW ANALYSIS IN A
PROGRAMMING ENVIRONMENT

by

Keith Daniel Cooper

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

Kenneth W. Kennedy, Professor
Dept. of Mathematical Sciences
Chairman

Robert S. Cartwright, Professor
Dept. of Mathematical Sciences

Francis D. Tuggle, Dean
Jones School of Administration

HOUSTON, TEXAS

April, 1983
INTERPROCEDURAL DATA FLOW ANALYSIS

IN A PROGRAMMING ENVIRONMENT

KEITH D. COOPER

ABSTRACT

This thesis examines three problems arising in the construction of an ambitious optimizing compiler based in a programming environment. The specific problems discussed are the analysis of aliasing patterns, computation of summary data flow information, and assignment of linkage styles to call sites.

A two phase algorithm for alias analysis is developed. Alias introduction analysis, performed during syntax analysis, detects the introduction of aliases and determines which call sites can propagate aliases. Alias propagation analysis uses a version of the classical iterative data flow analysis algorithm to compute potential aliases at each call site. The algorithm retains information to allow efficient updating in response to editing changes.

Two approaches to computing may summary information are presented. The first uses a version of the algorithm developed for alias propagation analysis. The second approach divides the problem into analysis for call-by-reference formal parameters and global variables. The parameter problem is solved using Tarjan's fast path expression algorithm, while the global variable analysis uses a depth first version the iterative algorithm.
An algorithm is developed for assigning linkage choices to each call site in the program. The algorithm compares the relative benefits of generating open, semi-open, semi-closed, and node split linkages at each call site. By using estimates and simplifications, it efficiently produces an assignment which leads to improved run-time behavior under reasonable assumptions.
ACKNOWLEDGEMENTS

I would like to thank my committee, particularly Dr. Ken Kennedy, for their guidance, insight, and support. The National Science Foundation, through grant MCS 81-04006 has directly supported this research. IBM Corporation, through its continued research contracts with the department, has provided support and encouragement for this work. I owe a large debt to my colleagues at Rice, and particularly in the LCSE, for making this an enjoyable experience.

Finally, I must my parents for support and encouragement without which this work would never have begun, and my wife Linda, for hours of proof reading, months of patience, continuous technical insight and moral support.
# Table of Contents

Chapter 1 - Introduction ........................................ 1
Chapter 2 - Notation and Problems ............................ 16
Chapter 3 - Aliasing ............................................. 33
Chapter 4 - Summary Information ............................ 60
Chapter 5 - Another Approach to Summary Information .......... 84
Chapter 6 - Linkage Tailoring ................................ 107
Chapter 7 - Conclusions ........................................ 132
Appendix A - Lattice Proofs ..................................... 141
Appendix B - Graham - Wegman Formulations ............. 156
Bibliography ..................................................... 162
**TABLE OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>Aliasing Example</td>
<td>35</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Banning's Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Leave_site</td>
<td>36</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Leave_node</td>
<td>49</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Alias</td>
<td>50</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Visit</td>
<td>52</td>
</tr>
<tr>
<td>Figure 3.7</td>
<td>Edge_function</td>
<td>53</td>
</tr>
<tr>
<td>Figure 3.8</td>
<td>Propagation Example</td>
<td>58</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Leave_node_mod</td>
<td>71</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Summary</td>
<td>72</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Visit</td>
<td>73</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Project</td>
<td>75</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>MOD_from_DMOD</td>
<td>79</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Example</td>
<td>90</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Simple GMOD Algorithm</td>
<td>92</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Interpreting $P_i \circ P_j$</td>
<td>98</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Interpreting $P_i \cup P_j$</td>
<td>99</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Interpreting $P^*_i$</td>
<td>100</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Node Splitting</td>
<td>120</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Linkage Assignment</td>
<td>124</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Estimation Formula</td>
<td>125</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

For a variety of reasons, sociological, economic, and technical, optimizing compilers have long been constrained to compiling small subsets of the procedures in large programs. User control of source code prevents the compiler from recalling the results of previous compilations. User dictated compilation orderings interfere with optimization. Separate compilation decreases the costs resulting from recompilation when a single procedure is changed. Interprocedural optimization can increase the amount of recompilation required in response to a source code change. The lack of commercially available compilers which perform such optimization can be construed as evidence that the complexities of interprocedural analysis and optimization have been sufficient to frighten off most implementors.

A combination of factors is changing this situation. The development of sophisticated tools to aid in the programming process is making it more palatable for the user to surrender possession and control of the source code for a large program. Once the compiler controls the disposition of the source, it can use knowledge gained in past compilations with assurance that the information accurately reflects the current state of the source. This allows the compiler to achieve some of the advantages of optimizing all of the procedures together, while preserving most of the economic advantages of separate compilation. Our understanding of interprocedural issues is
advancing to the point where an engineered attack on the complex problems of interprocedural optimizations is possible. The time for an ambitious interprocedural optimizing compiler has come.

Recent research in the areas of programming environments and interprocedural data flow analysis has provided the theoretical underpinnings necessary for construction of such a compiler. Research in the area of programming environments [TiRe 80] [BaNo 82] has focused more directly on issues related to source code control, management of large projects, and programmer productivity than on code optimization. This research has established a context in which an excellent interprocedural compiler can operate. Because these environments can perform sophisticated analysis in their structured editors, store the results of that analysis along with their source in long-term databases, and retain records of the procedure versions used in specific executable images, they provide the background continuity and resources required by a compiler which undertakes ambitious interprocedural optimization.

This dissertation examines several issues arising in the design of an optimizing compiler imbedded in a programming environment. The specific problems discussed are:

- analysis of aliasing patterns
- collection of summary data flow information
- tailoring of procedure linkages.

Each of these problems has been discussed in recent literature. In almost all cases, this literature has been written from the perspective of traditional, batch oriented compilation, without
examining the resources and requirements presented by a programming environment. The unique opportunities available in a programming environment context can have a large impact on solutions to these problems.

We anticipate that providing solutions to these problems will both simplify the construction of a compiler based in a programming environment and provide insight into the structure and function of other parts of the environment. Surely, these three problems are not the only outstanding problems involved in the construction of such a compiler. However, solutions to these problems should both clarify the issues involved in construction of the compiler and environment and point out other problems for future research.

1.1. Motivations.

Our interest in extensive interprocedural optimization arises from several distinctly different problems in the implementation of programming languages. In this section, I present three problems where techniques developed for an ambitious interprocedural optimizing compiler can be applied. The problems are:

- the compilation of APL
- management of extensible language features
- optimization of very large programs.

An interprocedural optimizing compiler will not solve all of the problems in any of these areas. Applying a combination of interprocedural analysis and optimization can, however, ameliorate problems faced by implementors in all three of these areas.
1.1.1. Compilation of APL

Highly interactive languages, particularly those with late binding of names to types, have posed a formidable challenge to language implementors. In particular, APL has been relegated to interpretive implementation, where it behaves correctly, but runs slowly. Much attention has been focused on problems involved in the compilation of APL to efficient code [Abra 70] [Mill 78] [Weid 79]. To generate efficient code for a workspace of APL functions, a compiler needs information on the types, shapes and sizes assumed by the program's variables during execution. While it is simple to construct an APL program which defies such analysis, in practice programs do not exhibit this type of behavior. For the most part, APL programs use variables in consistent ways, displaying patterns which are detectable with static analysis [BiCa 76].

Much of the time used interpreting an APL program is spent setting up for the execution of individual statements, rather than performing actual computation. This is true despite the fact that APL provides, and programmers use, explicit syntax for specifying vector operations. Studies have shown that that the average length of vectors in APL programs is between 14 and 28 [Wied 79]. With vector lengths in this range, the setup time for interpreting individual statements appears to require between seventy-four and eighty-five percent of the total time. Much of this time is spent in type and length checking. Bauer and Saal show that a fairly simple static analysis scheme can replace 80% of the run-time checking performed in a naive APL interpreter [BaSa 74]. Their technique
ignores interprocedural effects and assumes the worst about the side
effects of user defined functions. Incorporating interprocedural
techniques into the analysis can only increase the extent to which
static checking can obviate run-time checking.

This suggests that a system which derives precise information
about variable use and compiles code accordingly may be an effective
way to enhance APL performance. Such a system would annotate each
function in the workspace with compiler derived information about its
impact on other functions. This scheme has the potential for allowing
compilation of quick, clean code. While such a system could still be
fooled into poor performance by pathological programs, the same is
true of any scheme for compiling the full APL language. A clever
implementation, with the ability to fall back on straight
interpretation of pathologically behaving functions, might well
perform numerical work at speeds close to those exhibited by current
FORTRAN compilers, for algorithmically equivalent programs [Moru 71].

The heart of such a system lies in the collection, maintenance,
and use of a library of data flow information on each routine in the
workspace. This library of information must be generated, used, and
updated by the compiler. The design of such a system would go a long
way towards providing an efficient APL compiler which generated good
code and preserved the functionality of the language. To be sure,
there are major pitfalls which lie in the way of APL implementors,
but arming an APL compiler with precise interprocedural information
will help it.
1.1.2. Feature Management in Extensible Languages

A classic problem in the design of programming languages is deciding precisely what capabilities, or features, to include in a language. Often, the care and taste with which this is done are the major determiners of the ultimate popularity and utility of a programming language. As an alternative to making such decisions, designers of some languages, like Smalltalk [Gold 83], have elected to provide open ended languages. Rather than attempting to provide a complete set of features for all applications, they provide an initial set of features, along with a mechanism to add user-defined features. Once defined by the user, these features are virtually indistinguishable from those in the initial set, except perhaps in their run-time efficiency. These languages have been termed "extensible" languages, since they contain the inherent ability to extend the language for specific problem domains. For the most part, extensible languages have been implemented in interpretive environments.

Although designed primarily to assist in the programming process, languages like CLU and ADA, with data abstraction facilities, provide similar functionality. These languages allow the user to define and use powerful abstract data objects and operations on them, leading to the construction of very modular, structured programs with well defined interfaces. Programs written in data abstraction languages tend to be composed of many relatively small procedures [Sche 77]. The value of interprocedural optimizations in these languages is already recognized [Sche 77] as a means of
overcoming the implementation inefficiencies inherent in myriad
procedure calls.

A final example where interprocedural optimization can help
implement extensibility in a programming language can be found in the
documents coming out of the ANSI FORTRAN standards committee, X3J3.
As they prepare to publish a new FORTRAN standard, they have been
confronted with many proposals for additions to the language. In
response to these requests, the committee has adopted a formal plan
for feature management, the "core plus modules" approach. Their
scheme is to define a small language, the "core" language, and make
provision for compiling the core language with various combinations
of "applications modules". An applications module is a group of
related features written in the core language which are bundled
together for management purposes. The intent is that a user select
one or more applications modules at compile time, and the compiled
program behaves as if the desired modules are part of the core
language. This single aspect of the proposed FORTRAN standard may
have the effect of forcing adoption of interprocedural compilers, to
allow compilation of the language with any semblance of the run-time
efficiency of current FORTRAN systems. (This approach to compiling
FORTRAN is similar to the philosophy envisioned in the ECS system
done at IBM Research [AlCa 80].) The alternative appears to be
construction of a combinatorial number of compilers, one per
combination of applications modules.

A compiler which performs interprocedural analysis and
optimization, which can construct and maintain libraries of source
language programs, and which can generate highly tailored code for these procedures when used in source language programs, may provide an effective implementation strategy for an extensible language. For the future FORTRAN standard, such an approach to managing language features may well be the most reasonable approach to implementing the entire language.

1.1.3. Optimization of Systems of Programs

A matter of continuing practical interest to both the computer science and scientific programming communities is the optimization of large FORTRAN programs. In spite of advances in the speed of computers, there are still a large number of pragmatically interesting, tractable problems which are beyond the capacity of the fastest available computers. Powerful optimization techniques have the potential for speeding up these computations. Procedure calls present a major barrier to optimization. This has been recognized for some time; Medlock and Lowry pointed it out in their 1969 paper on the FORTRAN II compiler:

For small loops of a few statements, it very often produces perfect code. The efficiency is limited mainly by the (rather unnecessary) restrictions on the types of data organization that can be described in FORTRAN and the inability to combine subprograms in a single compilation. [MeLo 69]

The addition of explicit vector syntax, anticipated in the next FORTRAN standard, will raise the cost of subroutine calls substantially. The proposed array features [ANSI 81] [Paul 82] will necessitate passing arrays with dope vectors. Where subroutine calls have been inexpensive in FORTRAN, the new features will make them
much more expensive.

Separate compilation, a prized feature of FORTRAN implementations, forces compilers to do without accurate knowledge about the interactions of procedures. The side-effects of a procedure call impede optimization in the neighborhood of the call site, while ignorance of details about the called procedure forces the use of very general calling sequences in all cases, even where the full generality is unnecessary. For safety's sake, the compiler must assume the worst behavior on the part of the called procedure. Useful information is not propagated across calls to a subroutine; for example, unused registers may be saved and restored on each side of the call. By discarding valuable information known at the point of invocation, the data flow analysis produces much less precise information leading to inefficient code.

The Perkin-Elmer 3200's FORTRAN compiler demonstrates the potential benefits of applying interprocedural optimization [CW 82] [Holb 82]. The FORTRAN VIIZ compiler includes a source-to-source preprocessor which performs inline substitutions of subroutine bodies for procedure calls. The resulting FORTRAN code is then processed by the global optimizing compiler. This process leads to speed improvements of up to a factor of 4 over separate compilation with the same optimizing compiler. This result demonstrates the possible improvement from such optimization. This figure combines improvement from two distinct sources: elimination of actual code in procedure linkages and utilization of improved data flow information in global optimization.
For FORTRAN to remain as useful and popular as it has been, compiler implementors must overcome the expensive procedure calls inherent in the proposed new standard. An ambitious compiler, using summary information and performing linkage tailoring, can do this, concurrently increasing the impact of global optimization. Interprocedural summary information provides global data flow analysis with more accurate information at procedure calls. Optimization of calling sequences and tailoring of procedures to specific call sites offers the potential for further improvement in the efficiency of the generated code.

While we have discussed this problem in terms of optimization of FORTRAN programs, the same problems arise in any language which permits separate compilation of procedures, and has non-trivial side effects associated with procedures. The problems addressed in an optimizing program management system for FORTRAN are not radically different from those that would be encountered in such a system for ADA or PL/I. As more complex programs, like operating systems, compilers, and large simulations, are written in these languages, the need for good optimizing compilers will increase.

1.2. The Mesh Programming Environment

Since the context in which this work is to be implemented has had a major effect on the shape of the solutions, some familiarity with this project is helpful. A large effort is underway at Rice to construct a sophisticated hardware and software environment to support scientific computing. One of the central software projects
in this research is the development of a programming environment for a superset of FORTRAN 77. This dialect includes a number of features which are expected to be included in the next ANSI FORTRAN standard, particularly features intended to support vector processing. The algorithms presented in this dissertation will be implemented as a part of this FORTRAN programming environment.

The environment will consist of a central database of programs and annotations, along with a collection of tools to manipulate information stored in the database. The tools include a database manager, a version control system, a structured editor for FORTRAN, a debugging interpreter, and an optimizing compiler. Each of these components deserves further discussion.

1.2.1. The Database

The database is the heart of the programming environment. It is the central repository for information under the system's control. This includes procedures, annotations for those procedures, specifications, documentation, version control information, test data, and test results. The system can manipulate the database to create an executable image for a program, determine whether an image is consistent with its source code, produce external documentation on programs and individual procedures, or edit and browse source code. The database provides the compiler with access to any procedure in a program. This latter capability radically changes the range of optimizations available to the compiler; since the compiler has free access to the source and annotations of each procedure, it can
effectively look across procedure boundaries. In this context, the compiler can collect interprocedural information and use it as a basis for optimization.

The database is organized into projects, which are simply collections of modules and programs. A program is represented by a list of procedure names, a designated main procedure, and a call graph. A module is simply a collection of procedures, a unit of source text which can be edited. Modules are stored in intermediate language form, along with system generated annotations and user entered specifications and documentation. Different versions of programs and modules are stored separately, allowing the system to retain context specific information.

1.2.2. Version Control

The environment has an organizational system to track software which takes advantage of all of the information generated by the editor and the compiler. The version control software serves two roles: it performs the bookkeeping needed to track different versions of modules and programs; and it mechanizes the process of updating executable images in response to editing changes to a procedure's source code.

Because the environment's knowledge of the programs it manages is much more exact than that of existing systems, like MAKE [Feld 79], it can be far more precise in deciding which modules must be re-compiled in response to source level editing changes. MAKE must consider any modifications to a file as semantically important
changes, where the R^n environment can determine precisely which other modules in the system must be recompiled. Further, the system can distinguish the difference between a change which makes an optimized module incorrect, versus a change which merely makes it poorly optimized.

1.2.3. Intelligent Editor

The editor is the user's primary interface with the system. It provides basic facilities for creating and maintaining projects, programs, and modules, along with their annotations. The editor computes some of the annotations needed by the compiler and interpreter. Using the editor, the user can examine and modify source, documentation, specification, and test data files.

The editor performs syntactic and semantic checking on entry, replacing the function of a parser. It represents the program in an abstract intermediate form. Analysis performed as the procedure is entered provides the user with direct feedback on errors.\(^1\) On-line documentation is available for parameters to subroutines, and the editor can perform on-line type checking of parameters and COMMON variables. The editor incrementally computes global data flow information allowing timely detection of data flow anomalies, and providing initializing approximations to the interprocedural information gathering processes in the compiler.

\(^1\)There is reason to believe that intelligent use of such information may lead to increases in programmer productivity. Since this notion is the subject of a dissertation in preparation by Kenneth Zadeck [Zade 82], I will explore this issue no further.
1.2.4. Debugging Interpreter

To aid in debugging programs developed with the environment, an interpreter for FORTRAN is included to provide powerful tracing and monitoring facilities. Since the interpreter executes a high-level representation of the program, it can relate errors to the user in terms of the source program; likewise the user can describe debugging actions in source program terms. To allow debugging of realistically sized programs, the environment's interpreter is capable of executing programs composed of both interpreted and compiled code.

1.2.5. Optimizing Compiler

The optimizing compiler produces executable images of programs managed by the system. Since the system retains control over the source for a program, it always has accurate information about all of the existing procedures in a program. The compiler collects information on each procedure in the program, including data flow and register use information, and uses this information to improve the efficiency of the generated code. This improvement in run-time efficiency arises from two distinct sources, decreases in procedure call overhead from linkage tailoring and improved optimization in the vicinity of procedure calls due to sharper information.

1.3. Overview

The remainder of this dissertation presents the three problems in detail. Chapter two introduces some requisite notation and then introduces and motivates each problem. Chapter three presents the
aliasing problem. Chapter four examines the collection of summary data flow information in a programming environment using an approach similar to that applied to the aliasing problem in chapter three. Chapter five presents another method of computing summary information with a much lower time bound than the method of chapter four. Chapter six discusses the generation of tailored subroutine linkages. Chapter seven looks at the relationship between the solutions of the problems, and draws some conclusions.
CHAPTER 2

Notation and Problems

This chapter lays the groundwork for chapters three through six. The first section introduces terminology and definitions needed in the remaining chapters. The second section examines the unique concerns of a programming environment-based compiler. The third section briefly discusses the ordering of events in the program development cycle. The final three sections introduce each of the problems covered in chapters three, four, and six.

2.1. Notation

Before proceeding, I must clarify some notational issues. I will use the term program to refer to an entire executable unit, with procedure specifying an individual subroutine, function, or main program. An annotated procedure is a representation of the program's source, along with some automatically generated information. The program's structure consists of the relationships induced by the procedure invocations in the program.

The structure of a program is represented by a call multi-graph $CG = (V_{CG}, E_{CG})$, where each node represents a procedure and each edge a call site. Since a program has a designated main procedure, the node representing that procedure is called the root of the call multi-graph. For $v_1, v_2 \in V_{CG}$ (represented by similarly named nodes in $V_{CG}$), there is an edge $e = (v_1, v_2) \in E_{CG}$ if and only if $v_1$ invokes $v_2$. While FORTRAN does not allow recursion, we explicitly assume that
CG contains cycles, anticipating the use of these techniques on languages which allow recursion.

All edges in the call multi-graph are directed from a calling procedure to a called procedure. The calling procedure is considered the source of the edge, while the called procedure is the target of the edge. Many of the algorithms presented will assume that these edges can, in fact, be traversed in either direction, but the directions are needed to avoid ambiguity.

In traversing the call graph, I will need two different orderings on the nodes. The depth of a node in the call graph is defined as the maximum number of edges on any acyclic path between the node and the root of the call graph. The depth of the graph is the maximum of the depths of each node in the graph. The height of a node in the call graph is defined as the depth of the graph minus the minimum number of edges on any acyclic path between the node and the root of the call graph. The depth and height of a node are used in various algorithms; it is assumed that they are readily available. They can be computed as the call graph is updated to reflect changes made in an editing session.

To simplify references to components of programs, I will define the following sets, following Barth's notation [Barth 77].

- PP - the set of procedures in a program
- SS - the set of statements in a program
- WV - the set of variables in a program

To describe various aspects of name scoping, I assume, for each p ∈ PP, the existence of the following named subsets of WV:
GLOBAL(p) - variables declared in p which have global scope
REF(p) - the set of reference formal parameters of p
FORMAL(p) - the set of formal parameters of p
ACTUAL(e) - the set of actual parameters of a call site e

Since the problems being discussed all deal with interactions which occur at call sites, it will be necessary to describe some qualitative aspects of the call sites. In particular, several time bounds will involve the number of formal parameters of a procedure. For any \( p \in PP \), \( f_p \) is the number of formal parameters of \( p \). In any program, \( f_{p_{\text{max}}} \) is the maximum value of \( f_p \), over all \( p \in PP \).

In describing data flow frameworks, we often deal with subsets of a set \( X \). The power set of \( X \), denoted \( 2^X \) is the set containing all subsets of \( X \). The denotation for the power set is easy to confuse with an exponentiation; surrounding context should allow the reader to determine whether a set or a number is intended.

Algorithms are presented in a pseudo-Algol notation, unfettered by rules of syntax. The presentation of algorithms is intended to convey intuition and understanding, not to specify a direct implementation.

Throughout the remainder of this work, I presume a moderate familiarity with data flow analysis. A good introduction can be found in either [Kenn 81] or [Bech 77].

2.2. Algorithms for a Programming Environment

Much of the work done in optimization has implicitly assumed implementation in a batch-oriented compiler which operates on one or more procedures to produce "object code", a mix of executable machine
code and linker instruction. In this model, the object code is later linked with other object code to form an executable image of the program. The relationship between the program and the compiler is very transitory in nature. The compiler deduces information about the program, uses it to produce an object program and then discards the information. Compilers which support separate compilation do not even have access to all of the procedures in a program at compile time. In a programming environment, many of these underlying assumptions change.

A compiler imbedded in a programming environment is a much less monolithic structure. The work of translation is distributed over several programs which run at different times. Syntactic and semantic analysis, along with some global data flow analysis, are performed in a structured editor as the program is entered. Optimization and code generation are done in a series of passes over the body of the program; these may occur long after the editor has finished gathering its information. Further, since the compiler has access to the repository holding the source code, it can access information and source code for related procedures at nearly any time during compilation. However, the environment imposes requirements and restrictions which are very different from those found in a batch compiler.

Since the programming environment supports multiple users, the compiler must be capable of arbitratiing conflicting requests from users and dealing with nonsensical situations. The analysis pass spawned by one user's editing session may require access to a
procedure being edited by another user. Similarly, the analysis pass may require a procedure which has yet to be created. To deal with these situations, any algorithms used in the compiler must be able to defer work until needed procedures are available.

Because procedures kept in the environment can change during their lifetime, the compiler must be able to detect and track the impact of these changes. Editing a single procedure may force recompilation of other procedures, particularly when optimizations have been performed based on information about the changed procedure. When this occurs, the compiler must be able to update or replace outdated information with current information. Providing this capability, in an efficient manner, requires the algorithms to retain enough information about their internal state to allow inexpensive detection of changes and quick updating of information.

Finally, many algorithms proposed for use in compilers are based on an implicit assumption that maneuvering around the program is inexpensive. In a compiler where the entire intermediate representation is kept in main memory, this is a reasonable assumption. If the procedures reside in a database kept in secondary storage, however, there is almost certainly a sizable overhead associated with moving through the program. In a simple implementation, individual procedures might well be stored in individual files, leading to a situation where each traversal of a call graph edge requires at least the opening of a single file. These costs can dominate the cost of analysis if such motion is not carefully ordered.
The programming environment frees the compiler to collect and use precise information about procedures in a program. At the same time, the structure of the environment forces the compiler to work under a different set of constraints and costs than a batch compiler. In analyzing algorithms for use in the environment, we must pay attention to the distribution of work between the editor, analysis, and code generation phases. The algorithms must be able to defer work in response to conflicting requests for access to individual procedures. The algorithms must be able to detect the impact of editing changes on information collected previously, and have the ability to update information outdated by the changes. Finally, the algorithms must perform all of this work quickly enough to allow actual use.

2.3. The Program Development Cycle

The three algorithms discussed in the remainder of this dissertation are used by the programming environment as a normal part of managing the program development cycle. To understand the relationship between the algorithms, both temporal and otherwise, requires a brief discussion of that development cycle.

A program, in the view of the environment, is a collection of specific versions of procedures, along with a designated main entry point. Programs do not spring full grown from the minds of programmers. They are entered, one or more procedure at a time, into the system. This is done with the intelligent editor. As the programmer finishes editing a procedure, the editor spawns a
background process which updates the aliasing information on the procedure. After the aliasing information has been computed, summary information is computed.

In the course of developing a program, the programmer goes through many edit-update cycles. At some point, the programmer asks the system to construct and executable image of the program. In response to this request, the compiler performs linkage tailoring on the entire program, makes several optimization passes over the program, and generates code.

Alias analysis and summary information computations are performed as a result of every procedure editing session. Linkage tailoring, optimization, and code generation are performed only in response to the programmer's request for an executable image.

2.4. Aliasing

One major problem which a compiler performing interprocedural optimization must face is the analysis of patterns of aliasing in a source program. Whenever a procedure can refer to a single storage location using the names of two or more variables, the variables are said to be aliased. Alias analysis attempts to discover the patterns of aliasing which can occur in a given program. Since the FORTRAN standard forbids assignment to aliased variables, it could be argued that this topic can be ignored in our environment. Unfortunately, programmers are unlikely to completely avoid aliasing, so that any compiler which attempts to handle realistic FORTRAN programs must expect to encounter aliased variables.
In the absence of accurate information about aliasing patterns in a program, the compiler has two choices. It can generate code which is efficient but may produce incorrect results in the presence of aliasing, or it can generate extremely conservative code for all references to parameters and global variables. The latter course is certainly friendlier to programmers, but has a potentially high cost in lost opportunities for optimization.

The problem examined in chapter three is the detection of potential aliases introduced by the binding patterns of actual parameters to reference formal parameters at call sites. I will use the term aliasing to refer specifically to this phenomenon. Because aliasing occurs only at call sites, it is an inherently interprocedural phenomenon. Separate compilation schemes cannot detect aliasing because they do not have access to all of the procedures in the program's call graph simultaneously. In the absence of aliasing information, these compilers have little choice except to compile cautious code for all accesses to formal parameters. This caution introduces inefficiencies into the generated code.

Aliasing occurs when a single variable is passed twice to a called procedure, either as two distinct formal parameters or as a global variable and a formal parameter. The number of possible

---

1This same reasoning prevents checking parameter types across procedure calls unless the language provides type declarations for parameters, which FORTRAN does not. Lack of parameter type checking often allows programmers to introduce bugs into programs which are difficult for the programmer to detect, since the code generated for them actually runs, albeit incorrectly.
aliases is large. For a procedure \( p \), let \( n = |\text{GLOBAL}(p) \cup \text{REF}(p)| \). Then, there are \( n(n-1)/2 \) possible alias pairs for \( p \). Computing these possible alias pairs can be expensive. Since aliasing patterns are factored into more than one data flow problem, we compute aliasing information in a separate process. Chapter three presents a formulation of aliasing analysis as a data flow problem on the program's call multi-graph as its flow graph. The data flow formulation allows us to adapt the algorithm to the peculiar surroundings of the programming environment.

The cost of computing aliasing information is greater than that for computing summary information, in the worst case. Because programs with complex aliasing relationships are hard to understand, I do not expect that the algorithm will encounter the worst case in any practical program. Programmers do not and will not create programs where each formal parameter and global variable is a potential alias of every formal parameter. Such a program looks like a mass of procedure calls with permuted arguments! The data flow formulation of aliasing analysis allows us to capitalize on some assumptions about the nature of the aliasing patterns which are encountered in actual programs. With the algorithm, I present a brief discussion of how program characteristics impact the algorithm's behavior.

In the literature for interprocedural analysis, aliasing has received some attention. Spillman, in the computation of "modifies" information in IBM's PL/I Optimizing Compiler, imbeds aliasing analysis directly in the data flow computation [Spil 71]. Since
modifies information is the only interprocedural data flow analysis performed, this approach is reasonable. Rosen, in [Rose 79], handles aliasing by introducing an abstraction, the "data flow index". His data flow index

"specifies a place in the graph and whatever else is needed to ask for a single modification or preservation or usage bit."

Here, there is a bit to represent each variable for which information is being computed. Banning [Bann 78] and Myers [Myer 81] both address the aliasing problem in some detail; the work presented in chapter three uses Banning's work as a starting point.

2.5. Interprocedural Data Flow

A major barrier faced by optimizing compilers has been the lack of precise knowledge about the side effects of procedure calls. A compiler based in a programming environment can overcome this problem by collecting information about the side effects of procedures as it compiles them, and storing this information for use in the compilation of other procedures.

Interprocedural information is useful to a compiler, even in the absence of a powerful optimizer. For example, if the compiler knows that a function invocation in an expression has no side effects, it has much greater freedom to reorder the evaluation of the expression. In the absence of such information, it must retain the relative ordering of all calls and accesses to variables which are visible outside the procedure, since the calls may modify the values of the variables. In the presence of a global optimizer, interprocedural
information allows the compiler to perform many optimizations which must otherwise be treated as unsafe.

Interprocedural information can be used by the programming environment in several ways besides its obvious application in a compiler. For example, it allows tools which automatically construct an executable image, like Feldman's MAKE [Feld 79], to have a much more refined understanding of the meaning of an editing change. Where MAKE must assume a semantic change any time a file is modified, a tool based on good interprocedural information can differentiate between changes which are entirely local in their effects and those which necessitate recompilation of other procedures.

Data flow information gathered in the programming environment can be used to aid the programmer's understanding of programs. The DAVE system pioneered the detection of data flow anomalies as a programmer's aid [FoOs 76]. Larry Masinter's SCOPE system provides an unobtrusive facility for providing users with access to data flow information about a collection of lisp functions [Masi 80]. Other researchers on the R^n project are investigating application of data flow information as an aid to programmers [Zade 82].

The literature on interprocedural analysis has presented a variety of techniques for answering a number of interprocedural questions. Spillman presents a technique for analyzing the modification of variables [Spil 71]. Allen discusses the computation of def, use, and reach sets for programs which contain procedure calls [Alle 74]. Sharir and Pnueli suggest treating the entire
program as a single data flow graph and solving problems on it [ShPn 81]. Barth [Bart 77] and Banning [Bann 78] discuss collecting summary information to encapsulate the side effects of procedure calls.

Chapter four builds on the work of Barth and Banning, deriving an algorithm to compute interprocedural summary information in a programming environment. There are two types of summary information which can be computed, \textit{may} information and \textit{must} information. \textit{May} information, also called \textit{flow insensitive} information, describes data flow events which occur on at least one path through a procedure. \textit{May} summary information describes the set of possible data flow events which can occur in a single execution of the procedure. \textit{Must} information, also called \textit{flow sensitive} information, describes data flow events which occur on all paths through a procedure. \textit{Must} summary information describes the set of data flow events which always happen when a procedure is invoked.

The algorithm given in chapter four computes \textit{may} summary information. Chapter five presents an alternate method for computing the same information. \textit{May} information can be computed efficiently because it does not depend on the control flow structures of individual procedures. Once the algorithm discovers an event on a single path through the procedure, it can conclude that the procedure has that event as a side effect. The \textit{must} summary problems may not have efficient implementations. A \textit{must} summary analysis can only conclude that a procedure has an event as a side effect if that event occurs on every path through the procedure. Myers has shown the must
summary problems to be co-NP complete in the presence of aliasing
[Myer 81]. Thus, these problems have no efficient solution, unless
P=NP.² Since efficiency is a vital concern for a compiler
implementor, the algorithms presented here solve may summary
problems. Aside from being tractable, it is fortunate that may
summary information answers questions which are of interest to an
optimizing compiler.

2.6. Linkage Tailoring

The mechanisms used to invoke procedures are often a source of
inefficiency. The costs vary between languages and compilers, but
there is always an overhead for procedure invocation in a compiler
which supports the separate compilation - linkage editing paradigm.
Adhering to this model of translation restricts the compiler to
following a single set of conventions for procedure invocations. The
inefficiencies directly attributable to subroutine linkages are of
two types. First, the code actually generated to perform the
invocation takes time to execute. Second, in a separate compilation
- linkage editing scheme, the compiler is restricted to generating a
single executable version of each procedure, suitable for any call
site. An interprocedural optimizing compiler can generate customized
linkages between subprograms, tailoring both the code to perform the
invocation and, potentially, the procedure bodies themselves to
specific call sites.

²If these algorithms have an efficient solution, then co-NPC=P.
Since we know that P=co-P, it would follow that co-NPC=P ⇒ P=NP
[GaJo 79].
To address the inefficiencies entailed in procedure linkages, four distinct linkage styles have been proposed for an interprocedural optimizing compiler [AlCo 72] [AlSc 74]. These are closed, semi-closed, semi-open, and open linkages. In addition to these linkages, the compiler can apply a naming transformation to create versions of a procedure optimized for one or more specific call sites. I will refer to the last process as node splitting\(^3\), to describe its impact on the call graph.

**Closed procedure linkages** are the standard calling sequences generated by compilers which perform no interprocedural optimizations. A single established convention is followed to divide responsibilities between the calling procedure and the called procedure. These conventions are enforced via a standard prologue and epilogue code for each procedure and a standard sequence of instructions preceding and following the actual branch instruction which implements the call.

Semi-closed procedure linkages provide more opportunities for optimization while using only a single copy of each procedure. To generate semi-closed linkages, the system compiles called procedures before the procedures which call them. The compiler records information on register usage and linkage conventions for use during code generation of the calling procedures. This technique allows the calling procedure to tailor its register use around that of the

---

\(^3\)This should not be confused with node splitting techniques used to transform irreducible flow graphs into reducible flow graphs [Bech 77].
called procedure, and to pass information in registers. Of all the linkages discussed, the semi-closed linkage requires the most intricate interprocedural coordination on the compiler's part.

Semi-open procedure linkages create specialized copies of the procedure body, while moderating the space growth which can occur with a complete inline substitution. In a semi-open linkage, the calling and called procedures are compiled together. A copy of the called procedure is created which is not visible outside of the calling procedure. This private copy of the called procedure is optimized in conjunction with the calling procedure. The local storage of the called procedure is merged into the calling procedure's local storage and formal parameters are assigned actual storage. Parameter binding at the call site is replaced with actual assignments.

Open procedure linkages are created by inline substitution of the called procedure's body at the call site, under naming transformations which preserve the original meaning. This technique is also called inline substitution or procedure integration. The run-time overhead of procedure linkage is eliminated, and ordinary global optimizations applied to the calling procedure tailor the expanded procedure body to the specific environment at the call site. Since open procedure linkages lead to multiple copies of a single procedure body, the size of the resultant program may grow substantially.
Semi-open and open linkages obtain improvement through the generation of specialized copies of procedure bodies. As a generalization of the semi-open linkage, the compiler can create a specially optimized procedure body for use by some subset of the call sites which invoke that procedure. This procedure can then be accessed via a semi-closed or closed linkage. By grouping together the call sites for a program which allow a certain optimization, or which require a specific inefficiency, the compiler can gain the same benefits found from site specific optimization in a semi-open linkage, for call sites in separate procedures.

Chapter six examines methods for performing linkage tailoring in a compiler. The problem is deciding which linkage style to generate at each of the call sites in a program. If a program has $S$ call sites, there are $|S|^4$ possible assignments of linkages to call sites. If node splitting decisions are included in the process, the number of possibilities increases further.

To tailor linkages, an optimizing compiler must choose an assignment of linkage types to the call sites of a program, in such a manner that the resulting program runs more quickly than a program generated with all closed linkages. Ideally, the compiler should find the assignment which minimizes the program's running time. For large programs, the compiler does not have time to evaluate each of the possible assignments. The algorithm presented uses an engineering approach to finding an acceptable assignment, adopting some assumptions and approximations to simplify the problem.
2.7. Conclusion

The three problems presented, aliasing, summary information, and linkage tailoring, are all important issues in a compiler which attempts ambitious interprocedural optimizations. Each problem deals with events which occur at call site boundaries between procedures. For this reason, compilers which only attempt intraprocedural optimizations have not often dealt with these problems. The three problems are related. The linkage tailoring problem requires accurate data flow information to estimate improvement at a given call site due to a specific style of procedure linkage. Computing interprocedural summary information requires an accurate picture of the aliasing patterns in the program. The following chapters present the problems, in the order in which the analysis must be performed in the compiler.
CHAPTER 3

Aliasing

3.1. Overview of the Problem

A compiler which performs ambitious code optimization needs to understand the patterns of aliasing which can arise in programs it processes. Aliasing occurs when, from a given point in a program, a single storage location can be accessed using two different names. Aliasing can complicate or invalidate flow information by violating implicit assumptions made in the analysis. In order to have confidence in data flow information, the compiler must have accurate knowledge about the aliasing patterns present in the procedure being analyzed.

In the absence of good aliasing information, a compiler has two choices: generate cautious code for all statements which involve reference formal parameters, or generate code which runs quickly but may be incorrect in the presence of aliases. Even simple optimizations like keeping a value in a register for later reference can be invalidated by aliasing.

Aliases arise in a number of ways in a program, depending on the specifics of the programming language. PL/I's DEFINED attribute and FORTRAN's EQUIVALENCE statement both create aliases. The use of

---

1The FORTRAN 77 standard, in fact, encourages this latter course. Any program which stores into an aliased variable is declared to be non-standard conforming, with the results of execution undefined.
pointers in PL/I, C, and PASCAL can create aliases which are more subtle and difficult to detect. Pass by reference parameters can also create aliases.

The EQUIVALENCE problem is well understood, to the extent that there are published algorithms for dealing with it (see [AhoU 79]). The effects of EQUIVALENCE are entirely local in scope and can be handled by translating references to EQUIVALENCEd variables as references to offsets within a larger aggregate data structure. I will assume that the aliasing effects generated by EQUIVALENCE statements are handled at parsing or editing time, by converting aliased names into relative offsets within the appropriate global name. Aliasing effects due to pointers require an in-depth understanding of a specific programming language and its semantics. Because our programming environment is designed for a language which does not allow pointer variables, we leave the problems associated with pointers for others (see [Weih 80]). This chapter focuses on the last problem, aliasing effects arising from call by reference formal parameters.

When a single actual parameter is passed to a procedure in more than one reference formal parameter position, the called procedure has two distinct names for a single storage location. These names are aliases of each other. Aliases also arise when a variable that can be referenced inside both the calling and called procedures is passed as a reference formal parameter. I call such a variable a global variable or name. Figure 3.1 shows an example of a call site which induces an alias. In the example, the formal parameters a and
PROGRAM MAIN
INTEGER x, y

***
10 CALL S1( x, x )  \# causes an alias <a,b>
\# in S1
END

SUBROUTINE S1( a, b )
***
20 CALL S2( a, b )  \# if <a,b> exists then this
\# causes <c,d> in S2
***
END

SUBROUTINE S2( c, d )
***
END

Figure 3.1
Aliasing Example

h of procedure S1 both reference the storage location for variable x
of procedure main when S1 is called from statement 10.

Aliasing information is represented by alias pairs. The alias
pair <x,y> is interpreted "x and y are potential aliases". In the
example, the call in statement 10 creates an alias pair <S1.a,S1.b>.
Notice that <S1.a,S1.b> \(\Leftrightarrow\) <S1.b,S1.a>; the interpretation of an
alias pair is independent of order within the pair.

Since aliasing is intimately involved with the program's control
flow, the problem of computing exactly which aliases hold at a given
call site at a specific point in program execution can require
execution of the program. Instead, we compute the set of aliases
possible at a given call site, taking into account each control flow
path whereby execution can reach that site.
3.2. Previous Work

In his dissertation, Banning presents an algorithm to compute alias information [Bann 78]. His algorithm follows chains of parameter bindings in a depth first fashion to create all of the sets of potential aliases. Banning's algorithm is given in figure 3.2. Initially, there are no potential aliases. Marking is begun by calling visit on each of the trivial alias sets, a variable and itself. The algorithm proceeds, in a depth first style, along the chains of alias pairs passed through call sites in the program. The algorithm builds a collection of alias pairs capable of answering questions of the form: "are S2.c and S2.d potential aliases?".

Banning's algorithm computes all of the potential alias pairs of the program. In practice, several problems arise using this algorithm.

---

Banning's Algorithm

procedure visit( x, y )

begin
if x and y are not potential aliases
then
mark x and y as potential aliases
for each <x',y'> ≠ <x,y>
s.t. ∃ a call to some procedure
s which passes x to x' and y to y'
do visit( x', y' )
end /* visit */

for each variable v do visit( v, v )

---

Figure 3.2
in a programming environment. By examining the problems with Banning's algorithm, we can avoid building them into our own algorithm.

Simply computing the sets of alias pairs is not enough. The aggregate information produced by Banning's algorithm is difficult to update in response to editing changes on individual procedures. The algorithm discards information discovered in intermediate states of calculation. Updating the information is similar to beginning the computation in one of these intermediate states of computation. Retaining some of the information Banning discards is essential to efficient updating.

In general, to update information in response to modification of a procedure $p$, we must:

1. remove all of the effects which the old version of $p$ contributed to the aggregate information;
2. compute the effects of the new version, and add them to the aggregate information.

Using visit, the process of removing the effects of an old procedure requires a duplication of the original work that computed them. To add in the effects of a new version, we must apply visit to the trivial aliases of all the actual parameters of call sites in the modified procedure. This duplicates much work. Further, deleting alias pairs can be tricky. Since an alias can occur on multiple paths into a procedure, and the modified procedure may be on only one such path, alias pairs must be reference counted to allow correct "deletion".
Banning's algorithm assumes that the entire call graph is available. This has two major pragmatic implications in a programming environment implementation. In a programming environment, arbitrary procedures may be unavailable due to the editing activity of other programmers. Further, needed procedures may not yet exist; program development is often an incremental task. Any information gathering process in the programming environment must be able to defer processing due to situations like these. Banning's algorithm cannot postpone processing, precisely because it fails to retain intermediate information. Secondly, an algorithm used in a programming environment must recognize the costs associated with moving around the call graph, between procedures. These costs may dominate the cost of the analysis. Banning's algorithm chases down call chains as it discovers them, switching procedures at each step of each call chain. We would like to minimize such motion, or come close.

Banning's algorithm was designed to work with the entire program structure available and immutable. These are reasonable assumptions in a batch style compiler. In a programming environment, neither of these assumptions necessarily holds true. Since programs are expected to undergo frequent editing changes, the algorithms used must handle changes gracefully. In formulating an algorithm suitable for use in the programming environment requires in depth understanding of the problem, we must keep in mind the shortcomings of Banning's algorithm.
3.3. **A New Approach**

The computation of aliasing information can be divided into two subproblems: analyzing the introduction of aliases and analyzing the propagation of aliases. These are separate and distinct problems. Aliases are introduced by specific binding patterns at call sites. Aliases are propagated by other binding patterns at call sites. Call sites which can introduce no new aliases can propagate aliases; likewise call sites which introduce aliases need not propagate any aliases.

3.3.1. **Alias Introduction Analysis**

Aliases are introduced at call sites when the invoked procedure is given multiple names for a single storage location. This can only occur in two ways.

1. At a single call site, an actual parameter is passed to a procedure in more than one parameter position. This binds the corresponding formal parameters in the called procedure to the same storage location, making them potential aliases.
2. A global variable may be passed to a procedure as an actual parameter. This introduces a potential alias between the corresponding formal parameter and the global variable.

These are the only ways that an alias can be created. A call site which has distinct arguments, none of which is a global variable, can introduce no new aliases.

The editor can examine all of the actual parameters at a call site and construct a set containing all alias pairs introduced by the call. For a call site \( s \), we label this set \( \text{INTRO}(s) \). This set captures all of the information about alias introduction needed for
propagation analysis. INTRO(s) is invariant in the propagation phase of alias analysis; it can only change when the program is edited.

3.3.2. Alias Propagation Analysis

Aliases are propagated at a call site when the calling procedure passes two or more aliased variables as actual parameters. The called procedure then has an alias pair containing the corresponding formal parameters. More precisely, alias propagation occurs when:

(1) Two of the calling procedure's aliased reference formal parameters are both passed as actual parameters in the same call. This action creates an alias between the called procedure's reference formal parameters which correspond to the aliased actuals at the call site.

(2) One of the calling procedure's reference formal parameters which is aliased to a global variable is passed as an actual parameter. This creates an alias between the global variable and the called procedure's reference formal parameter which corresponds to the aliased actual.

Alias propagation requires that reference formal parameters of the calling procedure be used as actual parameters at the call site. Call sites which pass only local variables can propagate no aliases.

To formalize the propagation effects, I will define a function prop$_s$ mapping alias pair $<x, y>$ into the set of alias pairs propagated to the called procedure which are derived from $<x, y>$. The subscript $s$ denotes the fact that prop$_s$ models the binding at a specific call site $s$. To define prop$_s$ requires a model of the reference formal parameter binding which occurs at $s$. I use the function bind$_s$ for this purpose. Bind$_s$(x) maps the name of an actual parameter, $x$, into the resolved name of the formal parameter to which it is bound at the call site. Formally,
\[ \text{bind}_s(x) = y, \text{ if } x \text{ is passed to } y \]
\[ = z, \text{ if } x \text{ is global} \]
\[ = \lambda \text{ otherwise} \]

\text{Bind}(x) \text{ is one-to-many only when a call site } s \text{ binds a variable to more than one formal parameter, in which case INTRO}(s) \neq \emptyset.

Given a set \( A \) of alias pairs which hold true at call site \( s \), the alias pairs which hold at entry to the called procedure, due to alias propagation effects from \( s \), are given by

\[ A' = \bigcup_{a \in A} \text{prop}_s(a) \]

where

\[ \text{prop}_s( \langle x, y \rangle ) = \{ \langle x', y' \rangle \mid x' \in \text{bind}_s(x), \text{ and } y' \in \text{bind}_s(y) \} \]

The edge corresponding to \( s \) is annotated with \( A' \), to represent the propagated aliases.

3.3.3. Combining The Two

Alias analysis consists of detecting alias introductions and tracking alias propagation. In the call graph, we can formulate this problem as a process of annotating the edges with INTRO sets, annotating nodes with the union of information on all incoming edges, and propagating these aliases along all paths through the call graph until the information stabilizes. The algorithm computes successive approximations to the sets of potential aliases at each node, with the union of INTRO(s) along all edges entering a node as the first approximation. This approach is similar to that used in several global data flow analysis algorithms, notably Kildall's iterative algorithm [Kild 73].
Let $\text{NodeAlias}(n)$ be the set potential aliases at node $n$ in the call graph and $\text{EdgeAlias}(e)$ be the set of aliases propagated along edge $e$. The following equations can be used to compute $\text{NodeAlias}$ and $\text{EdgeAlias}$:

\[
\text{NodeAlias}(p) = \bigcup_{s \in p} \text{EdgeAlias}(s)
\]

\[
\text{EdgeAlias}(s) = \text{INTRO}(s) \cup (\text{u}_a \text{NodeAlias}(p) \text{prop}_s(a))
\]

We will create a variation of Kildall's algorithm to solve these data flow equations.

3.4. A Semilattice Formulation

Much of the work done in data flow analysis has relied on lattice theory for its mathematical characterization. In particular, Kam and Ullman [KaUl 76] have provided a set of conditions for a semilattice data flow analysis framework which guarantee quick termination of Kildall's iterative algorithm [Kild 73]. This section formalizes the aliasing problem in such a framework.

3.4.1. Definitions

A **semilattice** is a set $L$ with a binary meet operation $\wedge$ which is idempotent, commutative, and associative. There is a partial ordering, $\geq$, on the set, defined such that for a given semilattice $L$ and elements $a, b \in L$:

\[
\begin{align*}
\text{a} \geq \text{b} &\iff \text{a} \wedge \text{b} = \text{b} \\
\text{a} > \text{b} &\iff \text{a} \wedge \text{b} = \text{b} \text{ and } \text{a} \neq \text{b}.
\end{align*}
\]

By convention $\text{a} \leq \text{b}$ means $\text{b} \geq \text{a}$ and $\text{a} < \text{b}$ means $\text{b} > \text{a}$. A
semilattice has a bottom element, 0 or \( \bot \), if \( \forall x \in L, 0 \land x = 0 \). A semilattice has a top element, 1 or \( \top \), if \( \forall x \in L, 1 \land x = x \). A semilattice is said to be bounded if, for each \( x \in L \) there is a constant \( k_x \) such that each chain beginning with \( x \) has length of at most \( k_x \).

3.4.2. The Aliasing Semilattice

The aliasing problem is to construct a set of alias pairs at each node in the call graph, such that a node’s alias pairs represent all aliases that can occur on a path through the call graph which leads to the node. The set of all possible alias pairs is \( AP \). For a given program, \( AP \) can only include pairs constructed from the set of all variables which are used as actual parameters. In most programs, this is a smaller set than \( WV \), but we can construct a program where the set of actuals is exactly \( WV \). Assuming the program has \( n \) variables which are used as actual parameters, there are \( n(n-1)/2 \) potential alias pairs, excluding trivial aliases.

The aliasing semilattice, \( L_A \), can be defined as follows:

\[
L_A = (2^{APd}, \land)
\]

where \( 2^X \) is the power set of \( X \)
\( \land \) denotes lattice meet.

\( \land \) is set union \( (\cup) \)
\( \bot \) is \( AP \), the set of all variable names
\( \top \) is \( \emptyset \), the empty set
\( \geq \) is equivalent to \( \leq \)
\( \leq \) is equivalent to \( \geq \)

An element \( x \in L_A \) associated with a node in the graph is interpreted
to mean that all of the alias pairs in $x$ arise along at least one path into the node. Since set union is idempotent, commutative, and associative, $L_A$ is a semilattice. Since $2^{AP}$ is finite, any chain in $L_A$ is bounded.

There is a function space $F_A$ associated with our semi-lattice $L_A$. Functions in $F_A$ map lattice elements into other lattice elements; $f: 2^{AP} \rightarrow 2^{AP}$, $f \in F_A$. These functions represent alias introduction and propagation at call sites. For analysis, we must characterize $f \in F_A$. Due to the nature of aliasing, any function $f \in F_A$ can be defined as:

$$f(x) = A \cup \{ u_{x \in X} g(x) \}$$

where $A \leq AP$ is a constant set and $g$ is an arbitrary mapping $AP \rightarrow 2^{AP}$.

3.5. Derived Properties of Aliasing

This data flow framework for aliasing allows us to prove some useful properties about the problem. Proofs of the assertions can be found in Appendix A.

Kam and Ullman examined data flow problems couched in semi-lattice frameworks. They identified a class of frameworks for which they could bound the number of iterations required by a modified version of Kildall's iterative algorithm. Admission to this class of semi-lattice frameworks is governed by a set of conditions on the function space associated with the semi-lattice. The admissibility criteria are:

---

$^2$Proofs of assertions about $L_A$ are in Appendix A.
Each \( f \in F \) distributes over \( \wedge \).

\exists \text{ an identity function } i \in F \text{ s.t. } i(x) = x, \forall x \in L.

\( F \) is closed under composition.

\( \forall x \in L, \exists \text{ a finite subset } H \text{ of } F \text{ s.t. } x = \bigwedge_{f \in H} f(i). \)

Our function space, \( F_A \), meets these conditions.

An admissible Kam and Ullman framework is \textbf{rapid} if, for all functions \( f \) in the function space and all lattice elements \( x \):

\[ f(x) \geq x \wedge f(\tau) \quad \forall f \in F, \forall x \in L. \]

Our framework is not rapid. Alias propagation effects map an alias pair into other alias pairs. This fact leads to a simple counterexample. Construct a function,

\[ f_{\text{slow}}(X) = \emptyset \cup \left( \bigcup_{x \in X} \text{prop}_{\text{slow}}(x) \right), \]

by letting \( A = \emptyset \) and \( \text{prop}_{\text{slow}} \) map \( <a,b> \) into \( <c,d> \). Now, \( f_{\text{slow}} \in F_A \), but it does not meet the rapidity criterion when applied to \( x = \{ <a,b> \} \).

The FORTRAN program in figure 3.1, used in our initial discussion of aliasing, has a call site with \( f_{\text{slow}} \) as its propagation function. Statement 10 introduces an alias pair \(<\text{Sl.a},\text{Sl.b}>\). It propagates no aliases, since MAIN has no parameters. Statement 20, perhaps better labelled as statement "slow", introduces no aliases, since all of the actual parameters are distinct. This gives \( A = \emptyset \). Statement 20 can propagate an alias, however, since it uses two formal parameters as actual parameters. \( \text{prop}_{S20} \) maps \( <\text{Sl.a},\text{Sl.b}> \) into \( <\text{S2.c},\text{S2.d}> \). In the example, when Sl is invoked from statement

\[ ^3\text{Kam, J. and Ullman, J.: Global Data Flow Analysis and Iterative Algorithms, JACM; January 1976. See Observation \# 6, p166.} \]
10, statement 20 propagates the alias introduced in statement 10, creating the alias pair \(<S2.c, S2.d>\), to realize \(\bar{f}_{s10}\).

In the example of figure 3.1,

\[
\begin{align*}
\bar{f}_{10}(x) &= \{ <S1.a, S1.b> \} \cup (u \in X \ g_{10}(x)) \\
\bar{f}_{20}(x) &= \emptyset \cup (u \in X \ g_{20}(x)) \\
\text{where} \\
g_{10}(x) &= \emptyset, \text{ for all } x \\
g_{20}(x) &= <S2.c, S2.d> \text{ if } x = <S1.a, S1.b> \\
&= <S2.d, S2.c> \text{ if } x = <S1.b, S1.a> \\
&= \lambda \text{ otherwise}
\end{align*}
\]

In a second paper, Kam and Ullman examine semi-lattices which are not distributive [KaUl 77]. They show several properties which hold on a data flow framework under slightly weaker conditions than their previous admissibility criteria. They call these weaker conditions "monotonicity". Monotonicity is implied by admissibility, so our aliasing framework is monotone.

Monotonicity and distributivity imply several important properties for the algorithm. Since the framework is distributive, the least fixed point solution found by the iterative algorithm will be equivalent to the meet over all paths solution, which is the solution we really want.\(^4\) The iterative algorithm will always halt on an instance of our aliasing problem.\(^5\) The resultant information is independent of the order in which we consider the vertices in the algorithm; the ordering only impacts the amount of time we spend

\(^4\)[KaUl 76] See Observation 3, p160.
\(^5\)[KaUl 77] See Lemma 2, p310.
3.6. The Algorithm

The algorithm is divided into two distinct phases, preparatory work and a propagation pass. Due to the nature of the programming environment, the phases must coordinate their work through a set of common data structures. Before examining the phases themselves, I will describe the data structures.

The program is represented by a call multi-graph. The nodes of the graph represent procedures and the edges represent call sites. Each node has space for a NodeAlias set, a list of call sites relevant to the propagation pass, and lists of incoming and outgoing edges. Each edge has space for an EdgeAlias set, an INTRO set, and a matrix representation of the bind function.

Associated with the call graph are a pair of priority queues, named alias change_q and alias recurse_q. The queue elements are nodes of the call graph. Within each queue, priority is based on the node's depth in the call graph, where the depth of a procedure is the number of edges on the longest acyclic path between it and the root of the call graph, as defined in section 2.1. The following operations are defined on the queues:

---

6[KaUl 77] See Corollary to Theorem 3, p311.
leave_site: procedure;

for each actual parameter p do /* initialization */
    pointer(p) += 0; ref_count(p) += 0

actual_# += 1; relevant += false;
INTRO(s) += λ; ordinal += 1;

for each actual parameter p, in order, do
    if pointer(p) = 0
        then
            pointer(p) += actual_#
            actual_# += actual_# + 1

    /* build bind */
    ref_count(p) += ref_count(p) + 1
    bind(pointer(p), ref_count(p)) += ordinal
    backbind(actual_#) += p
    ordinal += ordinal + 1

    if p is a formal parameter
        then relevant += true /* check propagation */

for i = 1 to actual_# do /* build INTRO */

    if ref_count(backbind(i)) > 1
        then for i = 1 to ref_count(backbind(i)) do
            for j = i to ref_count(backbind(i)) do
                add <bind(i), bind(j)> to INTRO(s)

    if relevant
        then add s to the call site list
    else
        EdgeAlias(s) += INTRO(s)
        add target(s) to alias change_q on disk

end; /* leave_site */

Figure 2.3

(1) add <node> to <queue>; This places <node> in <queue> according to the priority associated with <node>.

(2) <node> = next from <queue>; This removes the highest priority element of <queue> and assigns it to <node>. If the queue is empty, or the node is reserved, it returns λ.
The underlying database manager is responsible for providing safe and synchronized access to all of the data structures.

All of the preparatory work is performed by the editor or parser before it stores a procedure in the database. This work divides into two categories:

(1) work done as call sites are constructed or parsed,
(2) work done as the editor leaves the procedure.

The former is done by leave_site and the latter by leave_node. The names reflect the fact that the editor invokes them as it finishes working on the particular structure, be it call site or complete procedure.

---

**leave_node**: procedure( );

    NodeAlias(node) + φ /* compute NodeAlias */

    for each incoming edge, e, do
        NodeAlias(node) = NodeAlias(node) U EdgeAlias(e)

    for each call site, s, do /* visit each call site */
        temp ← edge_function( NodeAlias(node),
                             bind_s, INTRO(s) )

        if temp ≠ EdgeAlias(s)
            then
                EdgeAlias(s) + temp
                add target(s) to alias change_q on disk

    end; /* leave_node */

---

*Figure 3.4*
alias: procedure( );

/* fetch the queues of work to be done */
changes + alias change_q from disk
recurse + alias recurse_q from disk

node + next from( changes )

while ( node ≠ λ ) do

visit( node, changes, recurse )

if changes is empty
then /* end of this pass */
   changes + recurse
   recurse + λ

node + next from( changes )

/* at this point, we cannot go any further */
/* so, we store the queues and quit */
alias change_q on disk + changes
alias recurse_q on disk + recurse

end; /* alias */

Figure 3.5

Leave_site makes two passes over the actual parameters of the call site. It builds two tables on the first pass, an array representing bind and a table named backbind which maps a the ordinal number for a formal parameter into the name of the actual parameter bound to it at the site. A second pass uses these two data structures to compute INTRO for the site. Leave_site assumes that temporary storage space is available in the symbol table. This is reasonable, since the names will have been reduced to symbol table indices.

Leave_site stores bind and INTRO with the specific edge. Additionally, if the call site can propagate aliases, leave_site adds
the call site to the node's list of call sites for propagation. If the call site cannot propagate aliases, then the value of EdgeAlias for the site is a constant equal to INTRO, so leave_site sets EdgeAlias to INTRO. An optimization which is not shown would be to preserve only those rows of bind representing actuals which are also formal parameters.

Leave_node computes the initial NodeAlias set and annotates any outbound edge which propagates aliases with an EdgeAlias set. Edges which can not propagate aliases have EdgeAlias sets which are always equal to INTRO(s), so leave_site initializes them.

After editing is finished on a given procedure, the editor calls leave_node to compute NodeAlias for the procedure and EdgeAlias for all of its alias propagating call sites. Leave_node tests each new EdgeAlias value against the old value for the edge. If the set has changed, the target node of that edge is added to the changes queue.

Alias propagation is a forward data flow problem [Kenn 81]. The procedure alias is the driving procedure of the propagation phase. It sequences a worklist version of the iterative data flow algorithm. It uses alias change_q and alias recurse_q to traverse the call graph in a series of passes.

To perform the work required at a single node, alias invokes visit, which computes a new NodeAlias value, using the formula derived previously. If this value differs from the stored NodeAlias value, then changes have propagated into the procedure, so the EdgeAlias sets for each potentially effected outbound edge must be
visit: procedure ( node, changes, recurse );

    temp + φ /* compute NodeAlias */

for each incoming edge, e, do
    temp + temp ∪ EdgeAlias(e)

if temp = NodeAlias(node)
    then return

/* we get here if any changes have occurred */
NodeAlias(node) + temp

for each s ∈ the call site list do
    temp + edge_function( NodeAlias(node),
        bind_s, INTRO(s) )

if temp ≠ EdgeAlias(s) /* more changes */
    then
        EdgeAlias(s) + temp
        if depth(node) < depth(target(s))
            then add target(s) to changes
            else add target(s) to recurse

end; /* visit */

Figure 2.6

recomputed. If any EdgeAlias sets change, visit adds the edge's
target node to the appropriate queue for updating. By checking both
the NodeAlias and EdgeAlias sets, visit avoids adding a node to the
worklists unless it changes the information on which the node's
NodeAlias value is based. These checks allow the algorithm to
restrict its attention to that area of the call graph where changes
can occur.

To evaluate the actual propagation functions, visit calls
dge_function. This procedure computes the correct function for the
edge_function:
procedure ( AliasPairs, bind, INTRO );

tempAlias + ∅

for each alias pair <x,y> ∈ AliasPairs do

/* propagation due to aliased formals */
if bind(x) ≠ λ and bind(y) ≠ λ
then for all x' ∈ bind(x) do
    for all y' ∈ bind(y) do
        add <x',y'> to tempAlias

/* propagation due to global name */
if x is globally exposed
then for all y' ∈ bind(y) do
    add <x,y'> to tempAlias

if y is globally exposed
then for all x' ∈ bind(x) do
    add <x',y> to tempAlias

tempAlias + tempAlias u INTRO /* account for INTRO */

return ( tempAlias )

end; /* edge_function */

Figure 3.7

given call site and applies it to the NodeAlias set of the calling procedure. Edge_function uses the bind and INTRO information on the call site previously computed by the editor.

3.7. Time Complexity

To evaluate our method for computing aliasing information, we must examine the time complexity of the individual algorithms. The processing performed by the editor is contained in two procedures, leave_site and leave_node.
If \( \alpha \) is the number of actual parameters passed at a call site, invoking \texttt{leave\_site} there requires \( \mathcal{O}(\alpha + \alpha^2) \) operations. To see this, note that the doubly nested loop adding alias pairs to \texttt{INTRO} can add at most \( \alpha^2 \) alias pairs, from the earlier discussion on alias introduction. The \texttt{bind} matrix can never have more than \( \alpha \) entries containing information, since only one entry can be initialized per iteration of the loop, and the loop body is executed \( \alpha \) times. All of the other loops are executed only \( \alpha \) times, to meet the time bound. An equivalent formulation of this bound is that invoking \texttt{leave\_site} at a call site \( s \) requires \( \mathcal{O}(\alpha + |\text{INTRO}(s)|) \) operations.

For a procedure containing \( \omega \) call sites which is invoked from \( \iota \) call sites, \texttt{leave\_node} requires \( \iota \) set unions to compute \texttt{NodeAlias} and \( \omega \) calls to \texttt{edge\_function}. Similarly, \texttt{visit}, which uses the same basic algorithm, requires \( \iota \) set unions to compute \texttt{NodeAlias} and \( \omega \) calls to \texttt{edge\_function}. \texttt{Visit} only examines call sites which can propagate aliases, while \texttt{leave\_node} needs to examine each call site in the procedure, to initialize all outbound edges of the procedure. So, both \texttt{leave\_node} and \texttt{visit} require \( \mathcal{O}(\omega + \iota) \) operations.

Bounding the time required for the entire data flow algorithm is more complex. The bound below is stated in terms of calls to \texttt{visit}.

In a call multi-graph \( \text{CG} = (V_{CG}, E_{CG}) \), alias can invoke visit at most \(|PP|(|PP|-1)f_{p\text{\scriptsize{max}}} \) times,\(^7\) where \( f_{p\text{\scriptsize{max}}} \) is the maximum number of formal parameters of any procedure in \( PP \).

\(^7\)Note that \(|PP| = |V_{CG}|\).
This bound derives from the following observations. In an acyclic call graph, the longest path over which an alias can be propagated is $|PP| - 1$.

With a cyclic call multi-graph CG, we must account for the longest chain of binding events which can happen in the graph. The binding which occurs at a call site can be represented by a binding pair $(a,f)$ where $a \in \text{ACTUAL}(p)$, $f \in \text{FORMAL}(q)$, and there is a call site $s$ in $p$ which invokes $q$, binding $a$ to $f$. A binding path is a list of binding pairs of the form $[(a,b)(b,c)(c,d)\ldots]$ where $a$, $b$, $c$, and $d$ are fully qualified names of formal parameters, and adjacent pairs must share adjacent variables. Cycles in a binding path are uninteresting to alias propagation, since the pair which completes the cycle can add no new alias pair to the set already known at the node.

Alias propagation requires a pair of binding paths through the call graph, with corresponding binding pairs in each list caused by the same call site. The length of such a pair of paths is bounded by the maximal length binding path in the graph. If a call graph cycle contains $n$ nodes, the longest possible acyclic binding path through the cycle is of length $(n-1)fp_{\text{min}}$ where $fp_{\text{min}}$ is the minimum number of formal parameters of any procedure in the cycle. The largest cycle in the graph can have $|PP|$ nodes, for a length $\leq (|PP|-1)fp_{\text{max}}$, assuming that $fp_{\text{max}} \geq fp_{\text{min}}$.

Given a bound on path length for alias propagation, the number of calls to visit can be bounded. If we consider a pass to consist of
processing each node in the graph once, then a pass requires \(|PP|\) calls to visit. In each pass, every alias pair which can propagate further through the graph moves along at least one call graph edge. (The ordering of the priority queues attempts to maximize the number of edges a given alias pair traverses in a single pass.) Since each pair which can move propagates along at least one edge in each pass, all pairs will finish propagating in no more passes than the number of edges in the maximum length binding path. Thus, the information in the alias sets at each node should stabilize in after no more than \((|PP|-1)f_{p \text{ max}}\) passes. Since there can be up to \(|PP|\) calls to visit per pass, the entire propagation phase can take no more than \(|PP|(|PP|-1)f_{p \text{ max}}\) calls to visit.

This discussion has centered on the cost for a complete analysis of a single call multi-graph. It is important to understand the amount of work which will be done in response to an editing change to a single procedure. If any of the EdgeAlias sets on call multi-graph edges have changed because of modifications to the procedure, then the targets of those edges are added to the alias change.q worklist on disk. The alias propagation analysis will be invoked by calling alias; it will begin with the list of nodes stored in the alias change.q on disk. In processing each of these nodes, the algorithm will only add nodes to the worklists in the event that changes have propagated into those nodes. Since no node is considered until it is known that changes have propagated into it, the algorithm can be said to limit its work to the effected area of the graph.
It is important to understand that a change to a single procedure can possibly propagate into every other procedure. In this case, the effected area of the program is the entire program. Further, other invocations of alias may have been forced to defer computations, leaving nodes in the worklists. In this case, those updates did not examine the entire area of the graph effected by their changes. Averaging over multiple editing sessions, the algorithm will limit its attention to the affected area of the graph, with perturbations in the cost of individual update to reflect database synchronization problems.

3.8. Expected Programs

All of the discussion about aliasing has centered on worst case behavior. It is important to understand that aliasing does not occur in a worst case manner in actual programs. The amount of time required by alias propagation is very dependent on two characteristics of the program, which I call the density of aliasing and the lifetime of alias pairs.

The density of aliasing refers to the raw number of introduced aliases. If there are few introduced aliases, opportunities for alias propagation are few. In practice, I expect that the number of alias introductions per call site will be small, probably much less than one. Although I have no empirical evidence at this time to support this claim, an examination of several large FORTRAN programs appears to support it, particularly in light of the restrictions against aliasing in the standard.
The lifetime of an alias pair is the length of the path over which it can be propagated. Recall that an alias pair can only propagate through a call site if both variables in the alias pair are visible in the called procedure. In the case of an alias pair between two formals, this requires that the procedure use both formals as actuals in the same call site. If the alias pair contains a global and a formal, the procedure must only pass the formal.

I expect that the lifetime of aliases is relatively short, no more than two or three procedures for most alias pairs. Programmers do not normally pass variables into a procedure as formal parameters and out of it as actuals, repeatedly, over long chains of procedures. Global variables, like FORTRAN COMMON, were created to allow programmers to avoid doing exactly this! To achieve the maximum binding path length discussed above would require a procedure to pass a single value around the cycle through every formal parameter in the
procedure with the smallest number of formal parameters. The example given in figure 3.8 shows such a procedure for a single node cycle.

If the density of aliasing is low, and the lifetime of alias pairs is not long, the expected running time of the alias analysis algorithm presented will be much lower than the worst case time bound.

3.9. Conclusions

The algorithms presented to compute aliasing information overcome many of the problems which plague an implementation of Banning's depth first scheme. They divide the computation into work which can be performed at syntax analysis time and that which must be done in a propagation phase. The algorithms can gracefully defer the propagation phase in response to either database synchronization problems or missing procedures.

The algorithms presented take advantage of several subtle optimizations. By identifying the potential of a specific call site to introduce and propagate aliases, the algorithm decides, at syntax analysis time, precisely which call sites will need attention in the propagation phase. The algorithm recognizes that call sites which introduce aliases but do not propagate them can be treated as constant valued edges in the propagation phase. Finally, the algorithm relies on continual equality testing of the information to limit processing to the effected area of the graph.
CHAPTER 4

Summary Information

4.1. Overview of the problem

To understand when it is safe to apply an optimization, a compiler must determine the impact of executing individual statements on the values of variables known to the program. For a given statement, this impact is called the side effect of the statement. In programs which contain no procedure calls, the side effects of statements can be determined by a relatively simple examination of each statement, with a knowledge of the semantics of the source language.

The presence of procedure calls in a program complicates the analysis of side effects in two distinct ways: inherited environmental aliasing effects and cumulative side effects of procedure invocations. This chapter examines the computation of information about the side effects of statements which contain procedure calls. The side effects of such statements may depend on statements inside the called procedure and any procedures invoked by the called procedure. Since the analysis produces information which, in effect, summarizes the analysis of all procedures invoked directly or indirectly from the call site, the information is commonly called summary information.

Previous work on the computation of summary information has largely been directed to computing answers to three specific
questions:

given a variable $v$ and a statement $s$,

Can execution of $s$ modify $v$?
Can execution of $s$ use $v$?
Can execution of $s$ preserve $v$?

These three questions concisely address many of the concerns of an optimizing compiler [Rose 79]. They are often expressed in terms of sets of variable names, where the equivalent questions are:

Is $v \in \text{MOD}(s)$?
Is $v \in \text{USE}(s)$?\(^1\)
Is $v \in \text{PRE}(s)$?

Since the compiler is usually interested in the side effects on all variables which are accessible at a particular point in the program, it is logical to extend the questions to cover sets of variables.

The summary data flow problem has received much attention in recent literature, notably [Bart 77], [Bann 78], [Rose 79], and [Myer 81]. These papers have been written primarily from the perspective of application in traditional compilers. The needs and concerns of such a compiler differ from those of a programming environment in several important ways. Our work is specifically aimed at the development of an algorithm which addresses the problems of the programming environment. To this end, this chapter will examine the computation of \text{MOD} information. The extension to computing \text{USE} is straightforward and given at the end of the chapter. The

\(^{1}\text{USE here has the meaning imparted to it by Barth and Rosen, rather than that given by Banning. Banning considers USE in the context of def-use analysis, calling this side effect REF.}\)
extension to PRE requires more work, since PRE information differs fundamentally from MOD and USE. A brief discussion of the PRE computation is given later in the chapter.

To understand the problems we must address, we should examine the work of previous authors, notably Barth and Banning.

4.2. Barth's Work

Barth's work on the interprocedural summary information problem is an excellent starting point. Barth presents his algorithms as relations on the domains PP x PP and PP x VV. He defines the relations:

- \( (P, Q) \in \text{CALL} \iff P \text{ invokes } Q \)
- \( (P, x) \in \text{DIRECTMOD} \iff P \text{ modifies } x \)

Using these relations, he defines MOD as:

\( (P, x) \in \text{MOD} \iff \text{a call to } P \text{ can modify } x \)

The effects of a call to \( P \) include the effects of any calls made within \( P \). Notice that MOD is flow insensitive, or "may", information.

Barth nicely formalizes the notions of correctness and precision. A may relation \( A \) is correct if the following three conditions are not simultaneously true for any \( v \in VV \).

1. \( v \) is addressable at the calling site for \( P \)
2. calling \( P \) may have effect \( A \) on \( v \)
3. \( (P, v) \notin A \)

Given two sets, \( A \) and \( A' \), both expressing a may relation, \( A' \) is more precise than \( A \) if \( A' \subset A \). A may summary relation is said to be precise up to symbolic execution if it is the most precise
information computable under the assumptions that all of conditionally executed code in the program is executable and all variables in the program are spelled distinctly. In general, this is as accurate as a data flow algorithm can produce, unless it delves into analysis of the possible values of conditional branches in the code.

Given his previous definitions, Barth provides a series of algorithms to compute MOD, using composition and transitive closure of relations. He successively refines the algorithms to account for increasingly more complex name scoping rules. Rather than repeating all of his formulas, I will give only one. This particular formulation is interesting because it gives a good intuitive feel for the problem. This is his MOD/1.1 [Bart 77]:

Given CALL and DIRECTMOD as above:

\[ MOD = CALL \ast DIRECTMOD \]

where \( \ast \) denotes reflexive transitive closure.

This formula only describes MOD effects involving global variables, but nicely captures the interplay between DIRECTMOD and chains of procedure calls.

4.3. Banning's Work

Rather than computing MOD directly, as Barth does, Banning defines two sets of information which, when combined with knowledge of aliasing patterns in the program, suffice to compute MOD. These sets are DMOD, the direct modification side effect of a statement,
and GMOD, the generalized modification side effect of a procedure. He also poses a global data flow problem on the program's call graph to compute these two sets. Banning's approach has two chief advantages:

1. It solves the summary information problem using well understood algorithms;
2. It need not require time proportional to transitive closure, unless the information being computed requires such time.

DMOD(s) is the direct modification side effect of statement s. For a statement containing no procedure calls, DMOD(s) is the set of variables modified by executing s. DMOD(s) can be computed by inspection for such statements. If s contains a procedure call, DMOD(s) is the set of variables modified by executing s independent of the procedure call, plus all of the variables visible to s which can be modified in a procedure directly or indirectly invoked by s. DMOD(s) is simply MOD(s), minus any aliasing effects.

GMOD(p) is the generalized modification side effect of procedure p. It contains the names of all variables visible outside of p which can be modified in any call to p, ignoring aliasing. This includes DMOD(s) for each call site in p, along with any variables modified by statements in p.

To compute in GMOD(p) for each procedure and in DMOD(s) for each call site, Banning gives the following data flow equations.

\[\text{Unless we consider asynchronous traps and erroneous computation. See [Spil 71] and [Henn 81] for a discussion of these problems.}\]
\[ \text{GMOD}(p) = (\text{GLOBAL}(p) \cup \text{REF}(p)) \cap (\text{IMOD}(p) \cup \{ \text{call sites } s \text{ in } p \, \text{DMOD}(s) \}) \]

\[ \text{DMOD}(s) = (\text{GMOD}(q) \cap \text{GLOBAL}(q)) \cup \{ \text{backbind}_{s}(y) \mid y \in (\text{GMOD}(q) \cap \text{REF}(q)) \} \]

where \( q \) is the target of the edge corresponding to \( s \). \( \text{backbind}_{s}(y) \) maps \( y \) into the name of the actual parameter bound to \( y \) at call site \( s \), and \( \text{IMOD}(p) \) is the union of \( \text{DMOD}(s) \), for all statements \( s \) in \( p \) which are not call sites. \( \text{IMOD}(p) \) can be viewed as an initial approximation to \( \text{GMOD}(p) \). It is completely computable at parse time. In Banning's scheme, we can use any of the standard global data flow techniques to solve these equations.

Although Banning's scheme does not actually compute \( \text{MOD} \), it is awkward to refer to computing \( \text{GMOD}(p) \), \( \text{DMOD}(s) \) and \( \text{IMOD}(p) \). For this reason, I will continue to refer to the problem as a computation of \( \text{MOD} \). Later in the chapter, an algorithm is presented for computing \( \text{MOD} \) from \( \text{DMOD} \).

As a final remark on Banning's treatment, it is pleasing to note that in a data flow treatment of \( \text{MOD} \), the complications introduced by name scoping effects are handled much more naturally and intuitively than in a relational formulation like that of Barth. Where Barth adds more and more notation to deal with more complex scope rules, Banning simply absorbs them into the definition of the set \( \text{GLOBAL} \). In practice, the issues of membership in \( \text{GLOBAL} \) and \( \text{REF} \) for each procedure will have been resolved prior to this analysis. The editor
must determine such things as a part of its normal semantic analysis of each procedure.

4.4. Computing MOD

Our approach to MOD is to adapt Banning's work to the needs of a programming environment. The primary issues raised by the programming environment are algorithm choice, deferred updates, and the ability to limit the algorithm to the affected area of the graph. The algorithm presented here is based on Kildall's iterative algorithm. There are two reasons for this choice, clarity and generality. The iterative algorithm is the most easily understood and coded of the global data flow algorithms. For purposes of explaining the algorithm, it is certainly the obvious choice. Secondly, many of the other well-known global data flow algorithms do not handle irreducible call graphs. While there is good evidence that these rarely arise in ordinary global flow analysis [Knut 71], I know of no similar evidence demonstrating that irreducible call graphs are unusual. The iterative algorithm handles such difficulties as a matter of course.

4.4.1. A Lattice Formulation

Before formulating an algorithm suitable for use in a programming environment, it is helpful to formalize the MOD computation in a lattice theoretic framework. Since much of the underlying theory of data flow analysis has been developed using lattice frameworks, this allows us to prove some properties which are useful in an algorithm to compute MOD. The framework formalizes many
of the crucial aspects of the problem, and provides a basis for the algorithm. The lattice for MOD is:

$$L_{\text{MOD}} = (2^X, \lor, \land)$$

where $2^X$ is the power set of $X$

$\lor$ denotes lattice join, and

$\land$ denotes lattice meet.

$\land$ is set union ($u$)

$\lor$ is set intersection ($n$)

$\bot$ is $\forall v$, the set of all variable names

$\top$ is $\emptyset$, the empty set

$\geq$ is equivalent to $\leq$

$\leq$ is equivalent to $\geq$

Since $\land$ and $\lor$ are idempotent, commutative, and associative, $L_{\text{MOD}}$ is a lattice.\(^3\)

The data flow framework for MOD requires a function space $F_{\text{MOD}}$ associated with $L_{\text{MOD}}$ and a mapping from edges of the call graph into functions in $F_{\text{MOD}}$. Banning uses functions of the form:

$$f(X) = (A \cap X) \cup B \cup \{ u_{c \in X} g(c) \}$$

where $A$ and $B$ are constant sets, $A \leq \forall v$, $B \leq \forall v$, and $g$ is an arbitrary function mapping $\forall v \to 2^{\forall v}$. $F_{\text{MOD}}$ is the collection of all such functions.

To complete the definition of the MOD framework, we must show a mapping from an edge in the call graph into a function in $F_{\text{MOD}}$. Given an edge $e = \langle u, v \rangle$ representing a call site $s$ in $u$ which invokes $v$, the function $f_e \in F_{\text{MOD}}$ associated with $e$ is defined by:

---

\(^3\)All of the theorems I use regarding lattice properties can be found in [Hech 77] pages 40 - 50. Proofs of assertions made about the lattice can be found in Appendix A.
\[ f_e(x) = (A_e \cap X) \cup B_e \cup \{ u_{a \in X} g_e(a) \} \]

where

\[ A_e = (\text{GLOBAL}(v) \cup \text{REF}(v)) \cap \text{GLOBAL}(u) \]
\[ B_e = \text{IMOD}(v) \cap (\text{GLOBAL}(v) \cup \text{REF}(v)) \]
\[ g_e(x) = \{ \text{hackbind}_e(x) \mid x \in \text{REF}(u) \} \]

To simplify the reading of these equations, I will denote the elemental application of a function \( g \) to a set \( X \) as \( g\{X\} \), rather than writing out \( \{ u_{a \in X} g(a) \} \). For example, \( f_e \) would be written

\[ f_e(X) = (A_e \cap X) \cup B_e \cup g_e(X) \]

This notation is used throughout the remainder of the chapter.

### 4.4.2. Properties of \( L_{\text{MOD}} \)

The primary purpose of formulating MOD in a lattice framework is the derivation of properties about the problem which are useful in constructing an algorithm. Kast and Ullman, in a pair of papers ([Kaul 76] and [Kaul 77]), proved a number of helpful results about data flow frameworks like \( L_{\text{MOD}} \). For their results to hold, the function space associated with the lattice must meet a number of conditions. They pose two separate sets of conditions, calling a framework which meets the conditions "admissible" in the case of [Kaul 76] and monotone in the case of [Kaul 77]. \( L_{\text{MOD}} \) is both admissible and monotone.

Since \( L_{\text{MOD}} \) is admissible the least fixed point solution derived by the iterative algorithm is guaranteed to be equivalent to the meet over all paths solution to the problem [Kaul 76]. Since \( L_{\text{MOD}} \) is
monotone, the iterative algorithm always halts when applied to an instance of the MOD problem [KaUl 77]. Finally, the information produced by the iterative algorithm is independent of the order of evaluation of the vertices; ordering can only impact the algorithm's running time [KaUl 77].

Among admissible frameworks, Kam and Ullman identify a class of problems which can be solved quickly using a variant of Kildall’s algorithm. A framework in this class is called a rapid framework. An admissible Kam-Ullman framework is rapid if:

\[ f(X) \geq X \land f(\tau) \quad \forall f \in F, \forall X \in L. \]

There are functions in \( F_{MOD} \) which do not meet this condition, so \( L_{MOD} \) is not rapid.

To prove this, we plug our characterization of \( f \in F_{MOD} \) into the rapidity condition:

\[
\begin{align*}
  f(X) &\geq X \land f(\tau) \\
  f(X) &\geq X \cup ((\land \tau) \cup B \cup g(\tau)) \\
  f(X) &\geq X \cup ((\land \phi) \cup B \cup \phi) \\
  f(X) &\geq X \cup B
\end{align*}
\]

Expanding the left side yields

\[(\land X) \cup B \cup g(X) \geq X \cup B\]

Clearly, \( (\land X) \geq X \) and \( B \geq B \). The difficulty arises if \( g(X) \not\subseteq (X \cup B) \). This condition fails for any \( g \) which takes an element of \( X \) into something outside \( (X \cup B) \). Since \( g \) is an arbitrary map, \( g:V \to 2^V \), there exist \( X, B, \) and \( g \) such that \( g(a) \not\subseteq (X \cup B), a \in X \). The function space is therefore not rapid.

---

\(^4\text{Kam, J. and Ullman, J.; Global Data Flow Analysis and Iterative Algorithms, JACM; January 1976. Observation \# 6, p166.}\)
We should ask, will such functions occur in an actual program? In practice, B will be mapped into IMOD ∩ (GLOBAL ∪ REF) of the called procedure, so B will contain only names visible inside of the called procedure. X will be an approximation to GMOD(p), and therefore completely visible in the called procedure. Consequently, the condition fails to hold for any function which maps reference formal parameters into names which are not visible in the called procedure. In a FORTRAN program, any call site which passes a local variable as a reference parameter violates the rapidity condition.  

It is trivial to show that the rapidity condition holds if we remove the part of the function which models the impact of the reference formal parameters.

Another formulation for data flow problems which is of interest is the Graham-Wegman formulation. Graham and Wegman give an algorithm which runs in nearly linear time for a large class of problems, which they term fast problems. Not surprisingly, MOD is not a fast problem. A proof of this is given in Appendix B. The Kam-Ullman framework is presented here because it gives us several useful additional results about the lattice, whereas the Graham-Wegman formulation would only show that the problem does not meet their time bound.

---

5 In the example shown in Figure 3.1, both call sites S10 and S20 generate functions which are not rapid.
leave_node_MOD: procedure( )

/* this is done incrementally in the */
/* editor, but listed for completeness */
perform global data flow to compute IMOD(p)

/* compute a new GMOD(p) */
   GMOD(p) + IMOD(p)
   for each outgoing edge, e, do
      GMOD(p) + GMOD(p) u DMOD(e)

/* compute new DMOD and EdgeIMOD sets */
   for each incoming edge, e, do
      temp = project( e, GMOD(p) )
      if temp \neq DMOD(e)
         then
            DMOD(e) = temp
            add source(e) to summary changes
            stored on disk

end; /* leave_node_MOD */

Figure 4-1

4.4.3. The Algorithm

The MOD computation is a backwards flow problem [Kenn 81] on the
call graph. The code to perform the analysis is relatively short and
clear. The algorithm is divided into an initialization process and a
data flow propagation phase. As the programmer finishes an editing
session on a procedure, the editor initializes the sets of
information for the procedure and each call site in that procedure.
There are two sets, IMOD and GMOD, associated with each node in the
call graph. A single set, DMOD, is stored with each edge in the call
graph.
The initialization work is performed by the procedure leave_node_MOD, shown in Figure 4.1. Leave_node_MOD computes new values of GMOD for the procedure by computing the meet of the DMOD sets of all of the outgoing edges unioned with the IMOD set for the procedure. It then annotates each incoming edge e with \( f_e(GMOD) \) as DMOD.

The data flow propagation phase is driven by the procedure summary shown in Figure 4.2. This procedure is nearly identical to

```plaintext
summary: procedure( );

/* fetch the queues of work to be done */
changes + summary change_q from disk
recurse + summary recurse_q from disk

node + next from( changes )

/* iterate until we exhaust the list, or hit */
/* an unavailable node */
while ( node ≠ λ ) do

  visit( node, changes, recurse )

  if changes is empty
    then /* end of this pass */
      changes + recurse
      recurse + λ

  node + next from( changes )

/* at this point, we cannot go any further */
/* so, we store the queues and quit */
summary change_q on disk + changes
summary recurse_q on disk + recurse

end; /* summary */
```

Figure 4.2
visit: procedure ( node, changes, recurse );

    temp + IMOD    /* compute GMOD */

    for each outgoing edge e, do
        temp + temp u DMOD(e)

    if temp = GMOD(node)
        then return    /* quick exit! */

    /* we get here only if changes have occurred */
    GMOD(node) + temp

    /* compute new DMOD's */
    for each incoming edge e do
        temp + project( e, GMOD(node) )

        if temp ≠ DMOD(e) /* more changes */
            then
                DMOD(e) = temp
                if depth(node) > depth(source(e))
                    then add source(s) to changes
                    else add source(s) to recurse

    end; /* visit */

Figure 4.2

the procedure propagate used in the alias analysis. It relies on a
pair of queues to coordinate the data flow computation, storing them
on disk to allow for deferred updating in a natural fashion. These
queues are handled identically to the queues in the alias algorithm,
except that the queue is ordered by height in the call graph, as
defined in section 2.1. This reversal accounts for the fact that MOD
is a backwards flow problem, and is an attempt to increase the
distance over which information propagates in a single pass. Summary
is invoked by the editor, after leave_node_MOD, if either of the
summary information queues are non-empty. Operations on the queues
are defined as they were for the aliasing algorithm, so that summary
will halt if it encounters either an empty queue or a procedure which
it cannot access. It uses a procedure visit to perform the actual
analysis at each node.

Visit performs the needed work at each node of the call graph
(see figure 4.3). It recomputes GMOD from the DMOD sets of all of
the incoming edges and IMOD of the node. If GMOD has not changed, it
exits without adding any new nodes to the queues. If GMOD has
changed, it recomputes DMOD for each incoming edge. If the DMOD set
of an edge changes, visit adds the source of that edge to the
appropriate queue for later updating. Visit uses equality tests on
both GMOD and DMOD to limit the set of nodes being updated to
precisely those which can have their GMOD set changed.

Project computes $f_e$ of a set, given the name of an edge $e$. It
simply implements the choosing and application of the appropriate
function from $F_{MOD}$. It is shown in Figure 4.4. The two sets, GLOBAL
and REF, used in project model the name scoping behavior of the
source programming language. These sets can be computed by the
editor as a normal part of its semantic checking, and are assumed to
be stored with the procedure. It is important to notice that all of
the impact of name scoping rules on the algorithm are encapsulated
into the set GLOBAL.

4.5. Time Complexity

The cost of computing MOD information must be analyzed from two
different perspectives: the cost of a complete analysis and the cost
project: procedure( edge, namelist )

set + λ

/* iterate over the set */
for each n ∈ namelist
doi
if n ∈ GLOBAL(target(edge))
    then set + set ∪ n
if n ∈ REF(target(edge))
    then set + set ∪ backbind(n)
end

/* at this point, set contains */
/* (GLOBAL n namelist) ∪ */
/* {backbind(y) | y ∈ REF n namelist} */
return (set)
end; /* project */

Figure 4.4

of updating the database in response to an editing change. By
analyzing the procedures, in a bottom-up ordering, we can develop an
understanding of the costs involved in each of these situations.

Project is the innermost procedure in the propagation phase. It
is shown in figure 4.4. Visit calls project to recompute DMOD for
edges where GMOD set from which DMOD was computed has changed.
Project has a single main loop, executed once for each element in the
GMOD set of the node being visited. The two conditional assignments
in the loop are mutually exclusive, since REF ∩ GLOBAL = Ø. Project
performs, in the worst case, |GMOD| unions. Since a GMOD set can
have at most |V| elements, project requires O(|V|) operations.
Visit is the second most deeply nested procedure in the propagation phase. Visit is called by summary to update a node's GMOD set and the DMOD sets of all outgoing edges. Assume the node has 0 outgoing edges, and 1 incoming edges. Computing a new value of GMOD requires 0 unions. If visit detects a change in GMOD, it computes new DMOD values for each of the 1 incoming edges, calling project 1 times. The total cost for visit is 0 unions and 1 function evaluations performed by calls to project. Visit requires \( O(o+1|VV|) \) unions.

Summary is the driving routine of the propagation phase. It is more difficult to analyze than either visit or project, since the bounds on its iteration are not as easily determined. We must bound the number of iterations for the MOD information to stabilize.

The basic quantum of MOD information is the knowledge that a single variable is modified by executing a call site. This knowledge is propagated along binding paths through the call graph. As in the aliasing algorithm, the maximum length binding path in the call graph is \( \leq (|PP|-1)f_{p_{\max}} \) edges. If we consider a pass of the algorithm as visiting each node in the call graph once, then in any pass, any MOD information which can propagate further will propagate along at least one edge in the call graph. In \( (|PP|-1)f_{p_{\max}} \) passes, all MOD information at the nodes will have stabilized, since any single quantum of MOD information will have had time to propagate along its longest possible path.
Notice that a pass corresponds exactly to summary exhausting its
changes_q worklist. Since the queues are maintained in order of
decreasing depth in the call graph, nodes which have already been
visitation will be added to the recurse_q worklist. So, the
information propagates across the call graph in \(|\text{PP}| - 1|\text{fp}|\) passes,
each of which can require no more than \(|\text{PP}|\) calls to visit. Thus the
entire analysis requires \(O(|\text{PP}|^2 \times |\text{fp}|)\) calls to visit. (The
worst adversary ordering will require \((|\text{PP}| - 1)|\text{fp}|\) passes. The
intent of ordering the queues in order of decreasing depth is to
maximize the distance over which information is propagated in each
pass. In an acyclic graph, information should propagate through in a
single pass.)

Consider the cost of applying summary to a call graph which has
no annotations, except those provided by the editor. Recall that
visit costs \(O(c + 1|VV|)\) unions. If we are performing complete passes
across the graph, we can simplify the expression for complexity by
noting that, in a pass over the graph, the algorithm examines each
each edge once as an incoming edge and once as an outgoing edge in a
single path. The sum over all \(p \in \text{PP}\) of \(c\) will be \(|E|\), as will the
sum over all \(p \in \text{PP}\) of 1. This allows us to state that the pass
requires \(O(|E|(|1+|VV|))\) operations. This makes the total cost for
computing summary information \(O(|E|(|1+|VV|)) \times |\text{PP}| \times |\text{fp}|\) operations, in the worst case. Restating this, computing summary
information requires
\(O(|\text{PP}| \times |\text{fp}| \times |E|(|1+|VV|))\)
operations.
An equally important concern, from a programming environment point of view, is the cost of updating in response to an editing change in a single procedure. How much work will be done when an individual user finishes editing his procedure? As the editor finishes with a procedure, it invokes \texttt{summary} if either \texttt{changes\_q} or \texttt{recurse\_q} is non-empty. This causes a propagation phase which potentially updates the entire graph. The simple answer to the question is $\mathcal{O}(|PP| \times \log_{\max} \times |E|/(1+|VV|))$. This answer is misleading.

There are two distinct reasons why the updating queues can be non-empty.

1. the editor just changed the current procedure in a manner which modified its side effects;
2. a previous update was unable to complete because of some synchronization problem on a procedure involved in that update.

Either of these can be true, or both. Recall that \texttt{summary} halts if it requires a node which is unavailable for any reason. This premature halting behavior leaves nodes in the queues. In either case, the computation of new edge values is limited to edges entering nodes whose GMOD sets have changed. It accomplishes this with the comparison of the new GMOD set against the old, performed in \texttt{visit}. The check on new values of DMOD in \texttt{visit} ensures that the only nodes added to the queues are those which have GMOD sets dependent on DMOD sets which have changed.

When called to update, \texttt{summary} examines only the effected area of the graph. \texttt{Summary} does not achieve a completely equitable distribution of costs to those who cause the changes. For a variety
of synchronization reasons, a user's update may perform work deferred from another user's editing session. Since that work is limited to the effected region of the graph, the total updating time, over a series of updates beginning with empty queues and ending with empty queues, will be proportional to the changed region of the graph. Any individual update, however, may not reflect the costs attributable to the editing session which invoked the update.

4.6. Solve for MOD

The algorithm, as presented, annotates each node in the call graph with a GMOD set, and each edge with a DMOD set. Recall that we were actually looking for a MOD set for each statement in the program. The edge sets DMOD are a subset of the desired information. MOD can be formed from DMOD by expanding DMOD to account for aliasing. Assuming that the aliasing analysis described in chapter three has already been performed, this can be done by applying

```
MOD_from_DMOD: procedure( e, OtherEffects )
    temp ← DMOD(e)  /* get DMOD */
    for each v ∈ temp do /* add aliases */
        for each w s.t. <v,w> ∈ NodeAlias(target(e)) do
            temp ← temp ∪ w
    /* add in purely local effects */
    temp ← temp ∪ OtherEffects
    /* at this point temp = MOD! */
    return(temp)
end
```

Figure 4.5
procedure MOD_from_DMOD to the DMOD set on each edge. MOD_from_DMOD is shown in Figure 4.5. MOD_from_DMOD takes two arguments, an edge and a set of names. The set of names is intended to include all variables modified by the statement, other than those modified by direct or indirect calls to procedures.

4.7. Using MOD to Compute Transitive Closure

As noted in section 4.4.2, the MOD computation is neither rapid nor fast. In fact, the MOD computation can be used to compute the transitive closure of a relation. Barth shows this by providing a formula which computes MOD using transitive closure, and by providing a transformation for a relation which imbeds the relation's transitive closure in the program's MOD information [Bart 77].

Barth's transformation is interesting, in that the program generated does not have a particularly complex call graph. Given a relation M, we can represent M in a matrix form. This form can also be interpreted as the adjacency matrix of some graph. Assume we have

\[ p_i: \text{procedure();} \]
\[ x_i + 1 /* \text{modify} x_i */ \]
\[ \text{call} P_j \text{ for each } j \text{ such that } M[i,j] = 1 \]
\[ \text{end;} \]

Figure 4.6
such a relation which has an adjacency matrix which has no recursion. Given this matrix, construct a set of procedures \( P_i \) which manipulate a set of variables \( x_i \) following the model given in figure 4.6.

In Barth's relational notation, it is obvious that \( P_i \) MOD \( x_j \) if and only if \( M^*[i,j] = 1 \). So, the MOD relation for the collection of procedures encodes the transitive closure of \( M \). Notice that all of the variables involved in the computation are global variables. This example takes advantage of the fact that the MOD computation is dependent on the number of call sites, or edges in the graph. Barth's generated program potentially has \( n^2 \) edges, where \( n \) is the number of variables in the relation. It also has \( n \) procedures, so it is not surprising that transitive closure can be imbedded in the MOD information of such an example.

4.8. **Computing USE**

The extension of this algorithm to compute USE information is simple. Like MOD, USE is a many summary problem using union as meet. The process of computing USE is identical to the process for MOD, except that the algorithm must start from sets of initial information which describe USE side effects. If we modify the initial global flow analysis used in the editor to compute IMOD so that it computes IUSE, and change the names MOD, GMOD, and DMOD to USE, GUSE, and DUSE, the same treatment solves USE.
4.9. Computing PRE

The computation of PRE is different from that of MOD and USE. Where the MOD and USE computations use data flow equations which are structurally identical, PRE uses equations based on a framework with set intersection as its meet function. The following equations relate GPRE, DPRE, and IPRE, the analogs of GMOD, DMOD, and IMOD.

\[
\text{GPRE}(p) = (\text{GLOBAL}(p) \cup \text{REF}(p)) \cap (\text{IPRE}(p) \\
\cap \{ \text{call sites } s \in p \cap \text{DPRE}(s) \})
\]

\[
\text{DPRE}(s) = (\text{GPRE}(q) \cap \text{GLOBAL}(q)) \\
\cup \{ \text{back} \text{bind}_{s}(y) \mid y \in (\text{GPRE}(q) \cap \text{REF}(q)) \}
\]

Notice that GPRE is the intersection of DPRE of its outgoing edges, where GMOD is the union of its outgoing edges.

Computing PRE requires that we develop a separate version of the MOD algorithm. Since \( F_{\text{PRE}} \) contains the same functions as \( F_{\text{MOD}} \), the same basic set of properties apply. The difference between MOD and PRE lies in the equation for the generalized side effect of the procedure, not the direct side effect of a statement containing a call. For the sake of brevity, the PRE algorithm is not given, but its development follows that of the MOD computation.

4.10. Conclusions

The algorithm presented is an on-line algorithm for solving Banning's MOD framework. It incrementally updates a database containing a collection of procedures annotated with previously computed summary information. It initializes the summary information
for each procedure with information derived by an intraprocedural
global flow analysis performed by the editor. Like the aliasing
algorithm, it uses a pair of queues to synchronize the deferred
computations forced on it by other activities in the programming
environment.

The algorithm computes MOD information, ignoring aliasing
effects, for an unannotated program in $O(|FP| \times fp_{max} \times |E|(1+|VV|))$
time. An algorithm is given which changes the MOD information for a
specific statement to reflect aliasing patterns. This algorithm
requires time proportional to the number of aliases and the number of
variables in MOD. When called upon to update existing summary
information in response to one or more editing changes, the algorithm
limits its consideration to that area of the call graph where changes
have actually occurred. The algorithm may be forced to recompute the
complete set of MOD information for the program, but only in cases
where the editing changes impact the side effects of all procedures
in the program.
CHAPTER 5

Another Approach to Summary Information

5.1. Splitting the Problem

In chapter four, I presented a treatment of the summary information problem in a lattice theoretic framework. That work combines a formulation of the problem similar to Banning's with the version of the iterative algorithm developed in chapter three for the aliasing problem. Using these methods, computing flow insensitive summary information requires $O(|V_{CG}|^2 \times |E_{CG}| x f_{p_{max}})$ time. This chapter presents a more efficient method for solving that problem. The treatment given in this chapter requires a familiarity with chapter four.

The propagation functions in the data flow framework for MOD model two distinct sets of side effects, those on global variables and those on reference formal parameters. Separating the analysis of global variables from that of reference parameters yields a more efficient solution to the summary information problem. This chapter presents:

1) a framework for computing MOD for globals
2) a framework for computing MOD for parameters

The framework for global variable summary information is much simpler than that for $L_{MOD}$, allowing for a more efficient solution than the algorithm of chapter four. The technique developed for reference formal parameters can be used to efficiently compute an extremely
conservative approximation to the aliasing information computed in chapter three.

5.2. Summary Information for Global Variables

May summary information for global variables can be computed with a much simpler framework than that presented in chapter four. Restricting the problem to global variables removes the renaming effects which arise at call sites, simplifying the problem greatly. This section presents a data flow framework for annotating each procedure in a program with GMOD information about the program's global variables. After this computation has been performed, the GMOD information must be combined with that produced for parameters, and this combined information used to compute DMOD and MOD as in chapter four.

This treatment assumes that normal syntactic and semantic analysis has already been performed, producing a set GLOBAL(p) for each procedure p. GLOBAL(p) contains the names of all variables which are used in p but not defined as local variables of p.

5.2.1. A Semilattice Formulation

The computation of flow insensitive summary information for global variables is a subproblem of the complete treatment presented earlier. As might be expected, the data flow framework is similar to L_{MOD}. By simply removing the effects of reference formal parameters from the propagation functions and contracting the domain of the problem, L_{MOD} is transformed for the new problem. By restricting the
problem to the global variables, the semilattice elements actually used will be subsets of the set

$$GV = \{ \text{GLOBAL}(p) \mid p \in PP \}$$

rather than subsets of $VV$. To differentiate the new framework from $L_{MOD}^*$ and to recall its derivative nature, I call the new framework $L_{MOD-G}$. Formally,

$$L_{MOD-G} = (2^{GV}, \vee, \wedge)$$

where $\wedge$ denotes lattice meet

$\wedge$ is set union ($\cup$)
$\vee$ is set intersection ($\cap$)

$\perp$ is $GV$, the set of all variable names
$\top$ is $\emptyset$, the empty set

$\preceq$ is equivalent to $\leq$
$\preceq$ is equivalent to $\geq$

Again, $\wedge$ and $\vee$ are idempotent, commutative, and associative, as in $L_{MOD}^*$.

The data flow framework requires an associated function space, $F_{MOD-G}$, and a mapping from edges of the call graph into the function space. Each function $f \in F_{MOD-G}$ is in the form:

$$f(X) = (A \cap X) \cup B$$

where $A$ and $B$ are constant sets, $A \preceq GV$, $B \preceq GV$. $F_{MOD-G}$ is the collection of all such functions.

To complete the framework, a mapping from an edge $e \in E_{CG}$ into a function $f_e \in F_{MOD-G}$ is needed. For an edge $e = \langle u, v \rangle$, the function representing propagation of information along $e$ is defined by:
\[ f_e(X) = (A_e \cap X) \cup B_e \]

where

\[ A_e = \text{GLOBAL}(v) \cap \text{GLOBAL}(u) \]

\[ B_e = \text{IMOD}(v) \cap \text{GLOBAL}(v) \]

These constants reflect our intuitions about the problem. At a call site, the changes propagated into the calling procedure from the called procedure are:

1. those changes made from procedure calls out of the called procedure, which effect variables global to both the called procedure and the calling procedure, \( X \cap (\text{GLOBAL}(v) \cap \text{GLOBAL}(u)) \);

2. those changes local to the called procedure which impact global variables, \( \text{IMOD}(v) \cap \text{GLOBAL}(v) \).

It is important to note that these rules model the situation which arises in a language which has only call by value formal parameters.

5.2.2. Properties of \( L_{\text{MOD-G}} \)

Formulating \( \text{MOD-G} \) as a semilattice framework gives us several useful properties. The same properties shown for \( L_{\text{MOD}} \) hold for \( L_{\text{MOD-G}} \), since \( F_{\text{MOD-G}} \) is simply the subspace of \( F_{\text{MOD}} \) where the mapping function \( g(x) = \emptyset \forall x \in VV. \)

\( L_{\text{MOD-G}} \) is an admissible data flow framework, in the sense of [KaUl 76]. Because of this, the least fixed point solution derived by the iterative algorithm is guaranteed to be equivalent to the meet over all paths solution to the problem [KaUl 76]. The iterative algorithm will always halt when applied to

\[ ^1 \text{Proofs of assertions about the semilattice may be found in Appendix A.} \]
an instance of the MOD-G problem [KaUl 77]. Finally, the information
produced by the iterative algorithm is independent of the order of
evaluation of the vertices; ordering can only impact the algorithm's
running time [KaUl 77].

Among admissible frameworks, Kam and Ullman identify a class of
problems which can be solved quickly using a depth first variant of
Kildall's algorithm. A framework in this class is called a rapid
framework. An admissible framework is rapid if:

\[ f(X) \geq X \land f(T) \quad \forall f \in F, \forall X \in L. \]

Every function in \( F_{\text{MOD-G}} \) meets this condition, so \( L_{\text{MOD-G}} \) is rapid.
The depth first version of the iterative algorithm given in [KaUl 76]
will halt on an instance of the MOD-G problem in \( d(G)+2 \) passes.\(^3\)
Knuth's results [Knut 71] lead to an expectation that \( d(G) \) is small,
often \( \leq 3 \).

The basic algorithm formulated for solving alias propagation and
may summary information can be adapted to solve this problem. Using
the approach developed in chapter three results in an algorithm which
easily handles deferred work and limits its attention to the affected
area of the graph. By visiting the procedures in the order specified
in [KaUl 76], the algorithm can achieve the \( d(G)+2 \) pass time bound.

\(^2\)See [KaUl 76], observation 6, page 166.
\(^3\)\( d(G) \) is the loop connectedness of the graph, defined to be the
maximum number of back edges in any cycle free path in the graph,
with respect to a depth first spanning tree. For a reducible graph,
\( d(G) \) is independent of the spanning tree chosen.
5.3. Computing MOD for Reference Parameters

Computing may summary information for reference parameters is a conceptually more complex task. The results of the previous section are evidence that reference formal parameters are the complication which increased the cost of the algorithm in chapter four. This section develops a different approach to computing MOD information for the call-by-reference formal parameters of a program. Since the algorithm only computes summary information on reference parameters, I will assume for the remainder of the discussion that the only variables present in any procedure are either formal parameters or are used as actual parameters. Any references to \( W \), the set of all variables, are actually to this subset of \( W \).

We need a means of referring to each of the parameters used in a call. For a procedure \( p \), \( \text{formal}_i(p) \) represents the name of the \( i \)th formal parameter of \( p \). For an edge \( e = <q,p> \in \text{E}_G \), \( \text{actual}_i(e) \) denotes the name of the variable passed to \( \text{formal}_i(p) \). The syntax analysis in either the editor or the parser can initialize these sets trivially. Finally, there is an index \( \text{ordinal} \) which maps actual parameters at a call site into formal parameters of the calling procedure. For an edge \( e = <q,p> \),

\[
\text{ordinal}_i(e) = j \text{ if } \text{actual}_i(e) = \text{formal}_j(q)
\]

\[= 0 \text{ otherwise.}\]

Again, this information can be computed trivially at parsing time.

To compute summary information, we first build a mapping from a procedure's formal parameters to those formal parameters to which
they can be bound. The relation map is defined over \( \mathcal{W} \) as follows:

\[
a \text{ map } b \iff \exists e = \langle u, v \rangle \in E_{\mathcal{G}} \text{ such that } a = \text{ actual}_i(e) \text{ and } b = \text{ formal}_i(v).
\]

The reflexive, transitive closure of \( \text{map} \) over the entire program, denoted \( \text{map}^* \) can be used to compute summary information. Since \( \text{map}^* \) specifies all of the variables to which \( \text{formal}_i(p) \) can be bound, \( \forall p \in \mathcal{P}, 1 \leq i \leq n \), we can use \( \text{map}^* \) to compute MOD side effects arising due to reference formal parameters.

We can represent \( \text{map} \) as an \( n \times n \) bit matrix, where \( n \) is the sum over all procedures of the number of formal parameters, and call this

---

**PROGRAM MAIN**

<table>
<thead>
<tr>
<th>COMMON // .N</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER I, J, K, N</td>
</tr>
</tbody>
</table>

\( N = 3 \)

**CALL S1(N, I, J, K)**

**END**

**SUBROUTINE S1(A, B, C, D)**

<table>
<thead>
<tr>
<th>COMMON // .N</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER A, B, C, D, E</td>
</tr>
</tbody>
</table>

\( D = 0 \)

**CALL S1(E, A, B, C)**

**END**

**Figure 5.1**

---
matrix MAP. With each row and column of MAP is associated the name of a formal parameter of some procedure. For convenience, assume that the rows and columns of MAP are arranged so that the formal parameters of each procedure occupy adjacent positions. An entry MAP(i,j) is set to true if and only if the formal corresponding to i can be passed to the formal corresponding to j, directly through a single call. An entry in MAP*, the reflexive, transitive closure of MAP, is set to true if and only if the formal corresponding to i can be passed to the formal corresponding to j, either directly through a single call or indirectly through a chain of calls. A row vector specifies all of the formals to which the corresponding parameter can eventually be bound, along some series of calls in the program. A column vector specifies all of the formals which can be bound to the corresponding parameter.

Figure 5.1 shows an example program, along with both MAP and MAP* for the program. Procedure MAIN has no formal parameters, so MAP is defined only in terms of the parameters of procedure S1. The call site in MAIN passes no formal parameters as actuals, so it doesn't give rise to any entries in MAP. The call site in S1 passes A to B, B to C, and C to D, creating the three true bits in MAP. Since these are the only two call sites, MAP is as shown. MAP* is simply the reflexive, transitive closure of MAP. Inspection of MAP* shows that parameter A can be passed to B, C, and D, parameter B to C and D, and parameter C to D. MAP* completely describes the parameter interactions through the recursive call site in S1.
/** initial information - no modified names **/
GMOD = vector of false

/** "or" in the affected names to GMOD **/
for i = 1 to n
  do
    if IMOD[i]
      then GMOD = GMOD ∨ MAP*[*,i]
    end

Figure 5.2

Given MAP* for a program and initial summary information for each procedure, like the IMOD sets of chapter four, a simple algorithm can be used to compute summary information. MAP* is assumed to be an n x n matrix, GMOD an n x l vector, and IMOD a l x n vector, where n = f_max. The IMOD vector is arranged to correspond to the layout of MAP*. I will use MAP*[i,*] to refer to the i_th row of MAP* and MAP*[*,i] to refer to the i_th column of MAP*. The algorithm of figure 5.2 computes a bit vector GMOD whose layout corresponds to that of IMOD. GMOD can be partitioned to form GMOD(p) for the reference parameters of each procedure p. After the algorithm of figure 5.2 is used, this information is combined with the GMOD information on global variables to yield GMOD sets equivalent to those produced by the algorithm of chapter four. These sets are then processed, as in chapter four, to produce MOD sets for each call site in the program.

Unfortunately, computing MAP* directly is expensive. Using Warshall's algorithm takes O(|PP|f_max^3) time [AHU 74]. While there exist faster methods for computing the reflexive transitive
closure of a relation, these methods are not sufficiently fast. The summary information algorithm of chapter four requires $O(|P| \times |E| \times f_{p_{\text{max}}})$ time, assuming a bit-vector representation of the GMOD sets. To improve significantly on that time bound requires a means of computing MAP* which takes advantage of the underlying structure of the problem.

5.4. A Swift Method for Computing MAP*

To more efficiently produce the needed information, we formulate the computation of MAP* as a data flow problem. Given only information which the editor can efficiently compute, annotate each procedure $p$ with an $n$-tuple $\text{BOUND}(p)$, representing the formal parameters to which each of $p$'s $n$ formals can be bound. $\text{BOUND}$ must account for all binding chains in the call graph. The elements of the $n$-tuple correspond to the procedure's formal parameters. $\text{BOUND}_i(p)$, the set for the $i$th parameter, is defined as follows:

$$\text{BOUND}_i(p) = \{ v \mid \exists e = \langle q, p \rangle \in E_{CG} \text{ such that}$$
$$v = \text{actual}_i(e) \text{ or } v \in \text{BOUND}_j(q)$$
$$\text{and } j = \text{ordinal}(\text{actual}_i(e)) \}.$$  

Our algorithm will compute $\text{BOUND}$, derive MAP* from it, and then compute GMOD using the algorithm given in figure 5.2.

To simplify the formal definition of the problem, assume that, for each procedure $p$, $\text{BOUND}(p)$ is an $m$-tuple, where $m = f_{p_{\text{max}}} + 1$. The elements of the tuple are numbered zero through $f_{p_{\text{max}}}$. $\text{BOUND}_0(p) = \emptyset$, for all procedures $p$. For $1 \leq i \leq f_p$, $\text{BOUND}_i(p)$ will correspond to
the column of MAP* representing \text{formal}_i(p). If \( f_p \leq f_{p_{\text{max}}} \), the remaining columns of \text{BOUND}(p) will be set to \( \emptyset \). In an actual implementation, only \text{BOUND}_i(p), 1 \leq i \leq f_p \) would ever be represented, but the propagation functions would be represented, but the propagation functions would be slightly more complex.

To compute \text{BOUND} efficiently, we formulate it as a single-source problem on the call graph. Tarjan has done extensive work on fast algorithms for solving such problems [Tarj 81a] [Tarj 81b] [Tarj 79]. The single source path expression problem is defined as:

given a graph \( G = (V,E) \) and a distinguished source vertex \( s \), find a regular expression \( P(s,v) \) \( \forall v \in V \) such that \( P(s,v) \) represents all paths from \( s \) to \( v \) in the graph \( G \).

The \text{BOUND} computation fits this model because all valid control flow paths through the program must begin with the program's main procedure, corresponding to \( s \), and proceed along some series of call graph edges through the program. The regular expressions, \( P(s,v) \) \( \forall v \), will naturally specify the set of possible control flow paths. By interpreting the operators used to construct the regular expressions \( P(s,v) \) as operations on the program's \text{BOUND} tuples, an algorithm to solve the single source path expression problem can be made to compute \text{BOUND} tuples for each vertex of the call graph.

Tarjan presents a two step algorithm based on Gaussian elimination to solve single source problems [Tarj 81b]. The first step is an elimination algorithm which computes a path sequence for the graph, an ordering in which to traverse the edges of the graph and update the regular expressions being computed. The second step is
to apply a solver which uses the path sequence to compute regular expressions $P(s,v) \forall v \in V$. For reducible flow graphs, the elimination algorithm requires $O(m\alpha(m,n))$ time, where $m = |E_{CG}|$, $n = |V_{CG}|$, and $\alpha$ is the functional inverse of Ackerman's function. It produces a path sequence of length $O(m\alpha(m,n))$. The solver requires $O(n+r)$ time, where $r$ is the time required for regular expression manipulation.

To apply Tarjan's algorithm to computing BOUND, the elimination algorithm is used to compute a path sequence. The solving phase is modified to interpret the regular expression operators in a manner which computes BOUND($v$) along with the regular expressions, $P(s,v)$. Using Tarjan's algorithm requires a continuous data flow framework for the problem and a mapping from regular expression manipulations into operations on the BOUND tuples.

5.4.1. The Framework

Tarjan defines a continuous data flow framework as a pair $(L,F)$ where $L$ is a complete lower semilattice and $F$ is a function space mapping $L \rightarrow L$. In Tarjan's formulation, the functions in $F$ must satisfy the following axioms:

1. $F$ contains an identity function $f_i$ such that $f_i(x) = x$, $\forall x \in L$.
2. $F$ is closed under meet, function composition, and closure,

   where $(f \land g)(x) = f(x) \land g(x)$ and $f^\ast(x) = \bigwedge\{f^i(x) | i \geq 0\}$.
3. $\forall f \in F$ and nonempty $X \subseteq L$, $f(\land X) = \bigwedge\{f(x) | x \in X\}$.

To apply the single source algorithm to the reference parameter
summary information problem requires a formulation of the problem as a continuous data flow framework.

The semilattice for reference parameter summary information is \( L_{\text{MOD-R}} = (B, \land) \), where \( B \) is the set of \( n \)-tuples whose elements are drawn from \( 2^W \), \( n = 1 + \ell p_{\text{max}} \), and \( \land \) is defined as the elemental union of the tuples. For example, \( B \land C \) is a tuple whose elements are the unions of the corresponding elements of tuples \( B \) and \( C \). There is a partial ordering \( \leq \) on \( L_{\text{MOD-R}} \), defined such that

\[
x \leq y \iff x \land y = x, \forall x, y \in L_{\text{MOD-R}}.
\]

\( L_{\text{MOD-R}} \) is complete, since every non-empty subset \( X \subseteq L \) has a greatest lower bound, \( \land X \). If \( X \) contains a single tuple, then \( \land X = X \), otherwise, if \( X = \{x_1, x_2, \ldots x_i\} \), \( \land X = \{x_1 \land x_2 \land \ldots \land x_i\} \). \( L_{\text{MOD-R}} \) has a bottom element, the tuple whose elements are all \( W \). This tuple is denoted \( \bot \) or \( \land L \).

Associated with \( L_{\text{MOD-R}} \) is a function space \( F_{\text{MOD-R}} \). Functions in \( F_{\text{MOD-R}} \) map elements of \( L_{\text{MOD-R}} \) into \( L_{\text{MOD-R}} \). Each function is of the form:

\[
f(X) = <s_0(X), s_1(X), \ldots, s_{n-1}(X)> \land B
\]

where the \( s_i \) are selector functions on \( X \) which return the meet of selected elements of a tuple, \( B \in L_{\text{MOD-R}} \) is an arbitrary constant, and the angle brackets \( (\ <, > ) \) denote an \( n \)-tuple constructor.

The mapping from a specific edge in the call graph to a function \( f_e \in F_{\text{MOD-R}} \) is provided by a definition of the selector functions and the constant \( B \). For a given edge, \( e = <q,p> \in E_{CG} \), \( s_i(X) \) selects only the element in position \text{ordinal}(i) \ of \( X \), and \( B_e \) is a tuple whose
ith element, \( E_i \), is formal \( \text{ordinal}(i)(q) \). Thus, the ith element of the tuple resulting from \( f_e(\text{BOUND}(q)) \) would be

\[ \text{BOUND} \text{ordinal}(i)(q) \cup \text{formal} \text{ordinal}(i)(q). \]

This is just the set of names which can reach the calling procedure's formal parameter, along with the formal parameter, itself. This function space satisfies Tarjan's three axioms, given above. 4

As a point of notational clarity, for the remainder of the chapter, I will use the symbol \( \wedge \) to represent logical and, \( \vee \) to represent logical or, and use the word meet to denote the lattice meet operation.

As defined by Tarjan, a \textit{continuous data flow problem} is a triple \((G,C,M)\), where \( G = (V,E,s) \) is a graph with a distinguished source vertex, \( C = (L,F) \) is a continuous data flow framework, and \( M \) is a mapping from \( E \) into \( F \). The call graph, augmented by distinguishing the node representing the main procedure, combined with the data flow framework \((L_{\text{MOD-R}},F_{\text{MOD-R}}')\), and the mapping from edges to functions defined previously, constitute a continuous data flow problem.

5.4.2. Interpreting the Regular Expressions

To use Tarjan's algorithm to compute \text{BOUND} tuples for each procedure requires an interpretation for each of the elements of the regular expressions \( P(s,v) \) on the \text{BOUND} tuples. The overall time complexity of the resulting algorithm depends on the implementations of these operations. Specifically interpretations are required for:

\[ \text{Proof in Appendix A.} \]
\[ \Lambda \] - the empty string
\[ e \] - a single edge
\[ R_1 \circ R_2 \] - concatenation
\[ R_1 \cup R_2 \] - union
\[ R_1^\ast \] - reflexive, transitive closure

The empty string is interpreted with the identity function, \( f_1 \). This function takes a tuple \( x \in L \) and returns \( x \). It can be formed by defining \( \text{ordinal}(i) = i, \ 1 \leq i \leq f_{p_{\text{max}}^p} \). A single edge is interpreted by application of the function \( f_e \in F_{\text{MOD-R}} \) as described earlier. The remaining three operations require more detailed explanation.

To model the concatenation of two path expressions, elements of the \text{BOUND} tuple of the first expression must be propagated through the second path. This can be done efficiently. For two path expressions, \( P_i \) and \( P_j \), \( P_i \circ P_j \) can be computed directly by examining

---

**Interpreting** \( P_i \circ P_j \)

```plaintext
/** compute the joining edge **/
let temp_tuple = f_e(BOUND(w))

/** propagate changes across P_j **/
for i + 1 to f_{p_{\text{max}}}
do
    for j + 1 to f_{p_{\text{max}}}
do
        if formal_i(x) \in BOUND_j(y)
            then BOUND_j(y) = BOUND_j(y) \cup temp_tuple(i)
    end
end
```

---

**Figure 5.3**
the edge which joins the two paths into a single path. \( P_i \) represents a set of paths from \( v \) to \( w \), with \( P_j \) representing a set of paths from \( x \) to \( y \). The junction of the two paths is an edge \( e = <x,w> \in \mathcal{E}_{CG} \). A method for augmenting \( \text{BOUND}(y) \) based on the addition of information from path \( P_i \) is shown in figure 5.3. The algorithm simply computes the \( \text{BOUND} \) information added by the new edge, then uses the knowledge contained in \( P_j \) to directly update \( \text{BOUND}(y) \). The function application takes \( \mathcal{O}(f_{p_{\text{max}}}) \) unions, and the loop uses \( \mathcal{O}(f_{p_{\text{max}}}^2) \) unions, so the algorithm requires \( \mathcal{O}(f_{p_{\text{max}}}^2) \) unions.

Interpreting the union of two paths requires an element-wise union of the tuples representing the paths. This is shown in figure 5.4, and requires \( \mathcal{O}(f_{p_{\text{max}}}) \) unions.

Computing the reflexive, transitive closure of a path expression on the \( \text{BOUND} \) tuples is more complicated. Given a path expression \( P_i \), its reflexive, transitive closure is just itself, unless the path described is a cycle. In the case of a cycle, \( P_i \) begins and ends at a node \( v \). Membership in the new \( \text{BOUND}(v) \) is controlled completely by the binding chains which map a formal parameter of \( v \) back into another parameter of \( v \). Using this fact, the algorithm given in

\[
\text{Interpreting } P_i \cup P_j \\
\text{for } i = 1 \text{ to } f_{p_{\text{max}}}
\begin{align*}
\text{do} \\
\text{result}(i) + \text{BOUND}_i(w) \cup \text{BOUND}_i(y)
\end{align*}
\text{end}
\]

\textbf{Figure 5.4}
Figure 5.5 computes a side data structure, an $f_{p_{\text{max}}} \times f_{p_{\text{max}}}$ array $\beta$, and takes its reflexive, transitive closure. After this, the algorithm can directly compute the new $\text{BOUND}(v)$. The dominating costs are the computation of $\beta^*$ and the building of a new $\text{BOUND}(v)$. The former operation requires $O(f_{p_{\text{max}}}^3)$ time, using Warshall’s algorithm, while the latter requires $O(f_{p_{\text{max}}}^2)$ unions, where each set is potentially of size $f_{p_{\text{max}}} \times |PP|$.

---

**Interpreting $\beta^*$**

/** assuming a cycle from $v \rightarrow v$ **/
/** build $\beta$ to represent cycle **/
for $i \leftarrow 1$ to $f_{p_{\text{max}}}$
    for $j \leftarrow 1$ to $f_{p_{\text{max}}}$
        do
            if $\text{formal}_i(v) \in \text{BOUND}_j(v)$
                then $\beta(i,j) \leftarrow \text{true}$
                else $\beta(i,j) \leftarrow \text{false}$
        end
/** reflexive, transitive closure **/
let $\beta^* = \beta$
/** use $\beta^*$ to build new $\text{BOUND}(v)$ **/
for $i \leftarrow 1$ to $f_{p_{\text{max}}}$
    for $j \leftarrow 1$ to $f_{p_{\text{max}}}$
        do
            if $\beta(i,j)$
                then $\text{BOUND}_i(v) \leftarrow \text{BOUND}_j(v) \cup \text{BOUND}_i(v)$
        end

Figure 5.5
It is important to note that if the $\beta$ matrix contains either zero or one true entries, it is its own reflexive, transitive closure. In practice, this may prove to be an important special case.

The cost of performing these interpretations determines the cost of the entire algorithm. Assuming that $f_{p_{max}}$ is a small constant for any program, then the interpretations for the empty set and a single edge are $O(1)$. Likewise, concatenation and union require $O(1)$ unions. Interpreting $P_i^*$ requires initializing $\beta$ and computing the reflexive, transitive closure of $\beta$, both of which are $O(1)$ operations, along with the $O(1)$ unions. Each union involves at most $|PP| \times f_{p_{max}}$ elements. Using bit vectors to represent the elements of the BOUND tuples, the unions can be performed in $O(1)$ time, with bit vectors of length $|PP| \times f_{p_{max}}$. The total time for interpreting a single regular expression operation is $O(1)$, assuming bit vector operations of length $|PP| \times f_{p_{max}}$, so the time for the complete solve step is proportional to the length of the regular expressions.

Using these algorithms in the solver step to compute the BOUND tuples at each node, the total cost for the computation with bit vector operations of length $f_{p_{max}} \times |PP|$ is $O(m\alpha(m,n))$, where $n = |E_{CG}|$ and $m = |PP| = |V_{CG}|$, assuming that $f_{p_{max}}$ is a small constant. As noted earlier, the $MAp^*$ array is contained within the BOUND tuples. Combining this method of computing $MAp^*$ and the algorithm given in figure 5.2 gives an algorithm for computing $C_{MOD}$ in $O(m\alpha(m,n)+n)$ time, for reducible graphs. This is a significant improvement over the algorithm of chapter four.
5.4.3. Incremental Updating

Since this algorithm is intended for use in a programming environment, it is important to consider the costs of updating the GMOD sets in response to an editing change. The presence of the MAP* array simplifies the process of updating, since it concisely expresses the relationship between parameters of different procedures. Updates can be classified as either non-structural or structural changes, depending on whether or not they modify the call graph and binding relationships.

Non-structural changes, from the perspective of summary information, can only change the IMOD information of the edited procedure, p. Consider a single bit change in IMOD(p). If the bit changed from false to true, then the GMOD sets of all variables which can be bound to that parameter must be set. If the changed bit corresponds to the i⁰th column of MAP*, then the required updating is performed by:

\[ \text{GMOD} \rightarrow \text{GMOD} \odot \text{MAP}^{*}[i]. \]

If the bit changed from true to false, the process is slightly more complex. Since a single GMOD bit can depend on many IMOD bits, the update can not simply set to false all GMOD bits arising from the changed bit. Instead, it must set those bits false, and then recompute their values from scratch. This can be done by:
$$GMOD = GMOD \land \text{complement}(MAP^{*}[*,i])$$

for each true bit in $MAP^{*}[*,i]$
  where $j$ is its position
  do
    $temp = IMOD \land MAP^{*}[j,*]$
    if $temp$ is all false
      then $GMOD[j] \leftarrow \text{false}$
    else $GMOD[j] \leftarrow \text{true}$

Note that if we assign true the value one, and false the value zero, then the body of the loop can be efficiently computed by setting

$$GMOD[j] \leftarrow ((IMOD \land MAP^{*}[j,*]) \neq 0)$$

since most machines can test for zero values efficiently.

Structural changes are like the non-structural changes in that adding a call site can only add true bits to $GMOD$, while deleting a call site can only remove true bits from $GMOD$. Unfortunately, both of these updating techniques are complex. The update for addition of a call site involves adding the new bindings for the call site, recomputing the reflexive, transitive closure for the calling procedure, and then propagating changes forward and backward in the $MAP^{*}$ matrix. After the structural changes are accounted for, the update follows the form of that for adding a true bit to $IMOD$.

Deleting an edge from a call site is more complex. Since the information retained by the algorithm is $MAP^{*}$, it is not easy to back out the impact of an edge. We have not yet formulated a direct updating technique for $MAP^{*}$ to handle edge deletions. Application of an iterative algorithm like that of chapter four will certainly perform the update. More research is needed to find an efficient updating technique for edge deletions.
5.5. Combining $L_{\text{MOD-G}}$ and $L_{\text{MOD-R}}$

Given the GMOD sets for globals and reference parameters, the analyzer must produce MOD sets for each statement containing a call site. It is important to notice that the two problems, GMOD for globals and for reference parameters, are completely independent, except when there is an actual $l_i(e) \in \text{GLOBAL}(q)$ for some $e = <q,p>$, $1 \leq i \leq f_p$. Fortunately, this case corresponds precisely to alias introduction by a global variable. In this case, the algorithm given in chapter four for extending DMOD to MOD by accounting for aliasing effects will extend the appropriate GMOD sets to account for a modification of either the parameter or the global.

To produce MOD information for each call site, it suffices to union the GMOD sets produced by the global and parameter analyses to form a GMOD set for each procedure. The algorithms given in chapter four are then applied to project DMOD sets out of the GMOD sets, for each edge in the call graph, and to extend the DMOD sets to account for aliasing.

5.6. An Approximate Aliasing Algorithm

Chapter three presents a data flow framework for the aliasing problem. The formulation of the iterative algorithm developed to solve that problem is applied in chapter four to solve the summary information problem. Given the similarity between the algorithms for alias propagation and summary information, it is reasonable to examine the application of the technique presented in this chapter to the aliasing problem. While a direct application of this technique to
the alias propagation problem may not be reasonable, the swift method for computing MAP\* can be used to compute a very conservative approximation of the potential alias sets of each procedure.

The application of Tarjan's algorithm to the summary information algorithm is made practical by the assumption that $f_{p_{\text{max}}}$ is a small constant. The alias propagation problem is not defined on $VV$, but on the set of pairs of variables in $VV$. Alias propagation analysis requires the tracking of each possible pair of parameters at a call site, along with each parameter singly. For a procedure $p$, there are $f_p + f_p(f_p-1)/2$ distinct potential alias pairs being considered. While it may be reasonable to consider $f_{p_{\text{max}}}$ a small constant, the assumption becomes a little tenuous when the number is $f_{p_{\text{max}}} (f_{p_{\text{max}}} - 1)/2$.

In a straightforward bit-vector implementation, the space requirements for storing the information may well become unreasonable. For $|PP| = 100$, and $f_{p_{\text{max}}} = 10$, the BOUND tuples for the aliasing problem would be 55-tuples, where the summary information problem would use 10 tuples. Each reflexive, transitive closure would be computed on a 55 by 55 matrix, as opposed to a 10 by 10 matrix for summary information. The sheer size of the problem seems to make this approach unattractive, particularly if aliasing is an infrequent phenomenon.

Alternatively, the analyzer can adopt a very conservative approach to aliasing. By relaxing the definition of alias propagation to allow an alias to propagate if both of its constituent names
propagate along any path, rather than along the same path, the analysis can produce a set of alias pairs using an algorithm based on the same MAP* matrix used in the GMOD computation. The set of alias pairs will be less precise than that produced by the analysis of chapter three, but will still contain all of the potential aliases. If alias introduction is truly an infrequent event, the imprecision may be acceptable, in return for faster computation of the information. Empirical studies of the information produced by the two algorithms are needed in order to realistically assess the impact of using this rough approximation in a compiler.

5.7. Conclusions

This chapter presents a second approach to the computation of summary information. The problem is divided into the analysis of side effects on global variables and side effects on reference parameters. The problem of computing summary information for global variables is shown to be rapid in the sense of Kam and Ullman; a depth first version of the iterative algorithm will halt on an instance of this problem after \(d(G) + 2\) passes over the call graph. The problem of computing summary information for call-by-reference formal parameters is formulated as a single source path expression problem, allowing the problem to be solved in \(O(m\alpha(m,n)+n)\) time for reducible call graphs.
CHAPTER 6

Linkage Tailoring

6.1. Overview of the problem

Performing linkage tailoring in a compiler is an ambitious undertaking. The compiler must assign a linkage type to each call site in a manner which decreases the total running time of the program. There is a direct tension between the compiler's need to deliberate on the available choices and its need to decide quickly. Choosing the optimal assignment of linkages to call sites may require consideration of an exponential number of possible assignments. This is actually a constrained integer minimization problem; we do not know how to do this in a manner which is fast enough to be acceptable in a compiler. This chapter presents an algorithm which uses simplifications and approximations to produce an assignment. The algorithm is a pragmatic compromise between run-time speed of the generated code and run-time speed of the compiler. The algorithm efficiently produces an assignment which should decrease the total running time of the generated code.

The linkage tailoring problem can be divided into an estimation problem and a choice problem. To provide a basis for deciding between alternative linkages, the compiler needs accurate estimates of the size and speed of the code which will be generated at a call site, given a linkage choice and a set of specific knowledge about the values of variables at the call site. This estimation problem is
difficult, since it is entirely dependent on very specific details of the compiler's implementation. The estimator must account not only for the selection of optimizations present and their relative interplay, but also for the effectiveness with which they are implemented and applied. The estimator must answer the question, given a set of facts, how well will this particular compiler do? Providing an answer to this question for a specific compiler will undoubtedly require some tuning of any estimation algorithm.

Given an assignment of linkage types to call sites, the impact of the linkage choice on the total running time is the sum of the improvements at each call site. The improvements at a call site can be computed by multiplying the estimated decrease in running time by the expected number of executions of the call site in a run of the program. The multiplicative impact of repeated executions makes accurate estimation of execution frequency crucial. The execution frequencies may well dominate improvement estimates enough to mask most inaccuracies in the improvement figures, making frequency estimates a major concern.

Given a set of estimates for procedure body sizes and execution frequencies, the compiler must assign linkages to each call site. Since the decisions made at each step of the assignment algorithm impact the potential for improvement at other call sites, the problem of choosing the next site to assign, as well as the linkage for that site, is difficult. The straightforward method of choosing an optimal assignment, with a given set of sites and estimates, is to generate all assignments which do not violate the constraints and
pick the assignment with least estimated running time. Unfortunately, there can be an exponential number of assignments which must be checked.

This chapter sets out an algorithm to assign linkage types to call sites. Recognizing the tension between the complexity of the problem and the need for compile-time efficiency, the algorithm generates the assignment quickly. The technique presented produces an assignment which will improve the program's run-time behavior, under reasonable assumptions. The algorithm does not minimize the program's running time; it only attempts to improve it.

The second section of this chapter reviews previous work in the area. The third presents the linkage tailoring algorithm. Before describing the algorithm, it discusses estimating improvement, estimating execution frequency, and implementation of the node-splitting transformation introduced in chapter two. The fourth section points out the impact of linkage tailoring optimizations on other parts of the programming environment.

6.2. Previous Work

Work in the area of linkage tailoring has ranged from early descriptions like Ershov [Ersh 66] and the Allen-Cocke catalogue [AlCo 72], through implementation in optimizers [Mech 77] [Ball 79], through discussions of construction methodologies for compilers [AlCa 80] [Harr 79]. Three particular treatments are important to the work presented in this chapter: Scheifler's discussion of procedure integration in CLU, Ball's work on estimating improvement
due to integration, and Hecht's work on procedure integration in a code improver.

6.2.1. Scheifler's Work

Robert Scheifler discusses inline substitution in a compiler for CLU [Sche 77]. CLU is a structured programming language which supports data abstraction. Such languages give rise to programs which consist of a large number of small procedures. The overhead of procedure calls is one source of inefficiency in a naive implementation of CLU. Scheifler presents an algorithm for selecting call sites for integration.

Scheifler formalizes the problem of deciding which procedures to integrate as follows:

Substitution Problem: Given a program, a subset of all invocations, a maximum program size, and a maximum procedure size, find a sequence of substitutions that minimizes the expected program execution time.

The upper bounds on program and procedure size arise due to limits on machine address space and compiler table sizes, respectively. He gives a reduction of the substitution problem to the knapsack problem, demonstrating that the substitution problem is NP-hard [GaJo 79]. Since the problem is NP-hard, the substitution problem has no efficient solution, unless P=NP.

Scheifler's work is simplified by the fact that his compiler does not perform global optimization, so that compiler generated estimates of improvement need not account for tailoring effects in the procedure body. He uses a very simple method to approximate code
size. He assumes that an n-ary operator expression requires n+1 words of memory, plus space for any operands which are expressions themselves. Under this measure, the space change caused by an inline substitution of procedure q into procedure p is simply SIZE(p) + SIZE(q) - SEQUENCE, where SIZE is an estimator for procedure size, and SEQUENCE is the size of the standard calling sequence. If the last use of a procedure is integrated, the extra copy of the called procedure's body can also be subtracted.

Scheifler insists on using actual run-time statistics to determine relative execution frequencies. This is because of the multiplicative effect of execution frequencies. To further simplify matters, he assumes that the number of calls generated in a procedure is independent of the path by which the procedure is invoked.

Scheifler's algorithm works in three passes. It first performs any substitutions which do not increase total program size. These integrations may increase the amount of available space. Next, the algorithm integrates as many of the remaining call sites as possible, within the size constraints. Sites are ranked in order by ratio of execution frequency to basic size change, with highest ratios chosen first. Finally, the algorithm checks to determine if the preceding substitutions have made possible any additional substitutions. All such substitutions are made.

Scheifler's algorithm results in integration of roughly ninety percent of the call sites in his sample, when program and procedure growth is limited to twice the original size. His substitutions lead
to a small improvement in execution speed. Scheifler speculates that the impact of global optimization after substitution may well dominate the benefit directly attributable to the substitution itself.

6.2.2. Ball's Work

Eugene Ball describes a technique for estimating potential improvement due to procedure integration [Ball 79]. His algorithm computes information about the underlying data flow structure of the procedure, and uses this information to predict improvement at a specific call site based on knowledge available about the values of parameters. His work assumes that improvement is due to constant propagation, test elision, and elimination of the calling sequence. The work is based on the underlying observation that name scoping rules restrict the extent to which knowledge at a call site can impact execution of a called procedure.

As an example, consider common subexpression elimination. In a procedure $p$, the only expressions which can be common subexpressions are those constructed entirely from variables in the set $\text{GLOBAL}(p) \cup \text{REF}(p) \cup \text{ACTUAL}(p)$. Any variables not in this set can not be accessed by procedures other than $p$, and so can not be involved in common subexpressions outside $p$. Interprocedural common subexpression elimination can only involve parameters and variables which have global scope.

Ball's technique provides estimates of the impact of procedure integration on the size and speed of programs. Like Scheifler, he
assumes an execution profile for the procedure, on representative
test data. The estimation technique proceeds in two phases. The
first phase computes information on the relationship between values
computed in the procedure and the values of parameters and global
variables at the call site. The second phase uses this information
to estimate the potential improvement in speed and size due to
constant folding and test elision after procedure integration.

The first phase of the algorithm solves a global data flow
problem to determine the extent to which each expression in the
procedure is a direct function of values known on entry to the
procedure. Each expression in the procedure is tagged with a
parameter dependency set characterizing this relationship. A
parameter dependency set can be treated as a tuple: $(\text{flag}, \text{set})$, where $\text{flag}$ is a boolean assuming values of "strong" and "weak", and
$\text{set}$ contains names of parameters and global variables. A tuple with a
"strong" flag indicates that the initial values of the variables
named in the tuple's set fully determine the associated expression's
value. Similarly, a weak flag merely indicates that the variables
named in the tuple's set are involved in the computation of the
expression. The meet operation on two tuples is the logical \text{and} of
the flags and the \text{union} of the sets, where strong has a value of
"true", and weak is "false".

Ball formulates computation of these sets as a global data flow
problem. After solving the data flow problem, each statement in the

\footnote{Ball did not express the problem in this form. His form is en-
tirely equivalent, but much less concise.}
program is tagged with a parameter dependency state, the collection
of parameter dependency sets which hold at the statement. These sets
concisely express the relationship between the values of variables at
the statement and the values of parameters and global variables on
entry to the procedure.

The second phase of Ball's algorithm uses the parameter
dependency states of each statement to estimate the size and run-time
speed of a procedure based on the knowledge available at a specific
call site. For each statement, an unoptimized size estimate is
constructed, based on generation of a standard code sequence. From
this estimate, execution speed is computed and then multiplied by
execution frequency. If an expression has a strong parameter
dependency set, the time and space can be saved at call sites where
the values of the corresponding variables are analytically known in
the calling procedure. Further, if an expression with a strong
dependency set controls the execution of a block of statements, the
time and space for the entire block can be saved if:

(1) the conditional expression has a strong parameter dependency
    set;
(2) the conditional expression evaluates to false under the
    known parameters;

By recording the unoptimized estimates and the potential
improvements, the second pass constructs estimates for space and
speed improvement.
6.2.3. Hecht's Work

Matthew Hecht, in his "Modest Quad Improver for SIMPL-T" [Hech 77] implements a simple procedure integration method. His work is encouraging, in that it demonstrates that even simple strategies of attacking the problem have the potential for producing gains in execution time efficiency. Hecht's work was implemented at the intermediate code level, with the program being represented by a list of quadruples (three address code, see [Grie 71]).

This context limits the integration scheme in several ways. SIMPL-T supports separate compilation, and has no goto statement. Both of these aspects of the language impact the integration rules. The code improver performs inline substitution on any procedure which:

1. is called only once;
2. is not the main procedure;
3. is neither an external or an entry procedure;
4. has no embedded return statements.

The third restriction is introduced by separate compilation, while the last restriction is necessary because the intermediate language has no mechanism for representing an arbitrary transfer of control. Still, these criteria permitted integration of at least twenty percent of the calls in a large sample of SIMPL-T procedures. Hecht observed that the integrations resulted in a decrease in the number of quads by one to two percent. Hecht speculates that timing measurements would show expansion to be an important part of a code improver.
6.3. The General Strategy

Linkage tailoring is an optimization which can improve run-time efficiency. If implemented correctly, the benefits should outweigh the costs. However, it is easy to envision simple implementations where the increase in compile-time far outweighs the benefits for many cases encountered in practice. In approaching this optimization, it is important to remember that the resultant improvement arises from two distinct sources: reduced call sequence overhead and tailored procedure bodies. In assessing the application of this type of optimization in a particular compiler, consideration must be given to the relative expense of a procedure call and the ability of the compiler to tailor procedure bodies, both with and without linkage tailoring activity. For a compiler which generates very inexpensive call sequences, the improvement generated by the optimization may not justify the expense of performing it.

For the compiler envisioned in the \( R^n \) programming environment, the calling sequences will be complex, due to language extensions which support vector computation. With the information available to the compiler, it will be able to tailor procedure bodies to improve performance on a site by site basis. Linkage tailoring appears to be a potentially valuable optimization in this compiler.

\[ ^2 \text{Another relevant concern of the compiler may be paging behavior over the local area network supporting the project. If this is the case, the improvement in locality achieved by generation of open and semi-open linkages might singlehandedly justify the overhead of the analysis and transformations. Studies of both network behavior and improvement will be necessary to decide this point.} \]
6.3.1. Estimating Improvement

The problem of estimating the improvement which the optimizer can effect with a given linkage at a specific call site is dependent on details of the optimizer and its implementation. For this reason, I will only outline the estimator, omitting much implementation specific detail. The technique described is an extension of Ball's work.

The estimator works in three passes. The first two passes are called on exit from the editor, to compute information required in the third pass. These two passes are performed once per procedure. The final pass estimates improvement from call site specific knowledge. It must be invoked once per call site. Further, since it depends on knowledge about both the calling and called procedure, it can not necessarily be performed until the compiler is actually invoked. Until that time, there is no assurance that all of the requisite procedures exist.

The first pass converts control dependence to data dependence using the technique described in [AKPW 83]. This procedure transforms the program into a series of conditionally executed assignment and branching statements. Each statement is provided with an expression which determines the conditions under which it is executed. This gives a syntactic expression to the relationships between variable values and the execution of individual statements. This transformation allows the estimator to avoid the need for special case techniques to analyze control flow constructs in the
source language, simplifying each of the remaining two passes.

Pass two is almost identical to the first pass of Ball's algorithm. It solves a forward global flow problem to compute a parameter dependency set for every expression in the procedure.\(^3\) These sets summarize the extent to which the value of the expression depend on values of global variables and actual parameters. The details of the data flow problem are covered in [Ball 79]. The transformations applied in pass one and the tuple based formulation given in section 6.5.2 simplify the implementation of this pass.

The final pass uses the information derived in pass one and pass two to estimate the improvement attainable by the optimizer. The estimator considers each statement individually, in a single pass over the procedure body. For each statement, it computes a code size and speed based on a simple model template for each statement type. Using the information available at the call site, it determines an estimated run-time speed under the assumption of integration. The transformations applied in pass one make explicit which statements, if any, can not be executed due to control flow decisions based on known constant values.

Pass three must consider at least constant folding and test elision optimizations. Dongarra's study of the BLAS routines in

---

\(^3\)The issue of summary information comes into play in this problem, as in almost all global data flow problems. At a call site \(s\), the estimator needs stored summary information to determine the impact of the call site on the parameter dependence sets of all variables in \(\text{GLOBAL}(p) \cup \text{ACTUAL}(s)\). If the variable is in \(\text{MOD}(s)\), its parameter dependence set must be given the value \(<\text{weak},\emptyset>\). If it is not in \(\text{MOD}(s)\), then its set remains intact.
LINPACK strongly suggests that it should examine opportunities for loop unrolling due to constant valued loop step sizes [Donn 80]. The estimator should also look for opportunities for code motion optimizations. If a call site is nested in a loop, moving invariant code from the called procedure out of the loop has the potential for being very profitable.\footnote{Detecting when this can be done safely is non-trivial. One case we should look for is moving strongly dependent expressions out of the called procedure and surrounding loop. If the expression is strongly dependent on values which are constant at the call site, it is, by definition, loop invariant.}

6.3.2. Estimating Execution Frequency

Data on the relative execution frequency of each call site is needed for the algorithm to assess the improvement due to a single linkage tailoring decision. Scheifler and Ball both assume that this requires actual execution data. It may be that the impact is too great for the inaccuracy inherent in estimation to be acceptable. Knuth is also emphatic on the value of execution profiling as an aid to both optimization and debugging [Knut 71]. It is certainly much better if the programming environment, over the course of debugging and formal testing, can record execution profiles. Unfortunately, such numbers may be either unrepresentative or unavailable.

In the event that actual data is available, the linkage tailoring software should scale the profile counts to some reasonable range and use them. In a compiler, it may be impossible to provide actual execution profiles.
In the absence of actual profiles, the compiler must generate numbers for execution frequency. I suggest using the following simple formula:

Estimated frequency = \( 1 \times 10^n \times i \)

where \( n \) is the loop nesting level and
\( i \) is the number of call sites
invoking the current procedure

This is a crude estimate, and practical experience may lead to much better "rule of thumb" estimates. This estimate is, however, inexpensive to compute and reflects well the dramatic impact of loop nesting.

6.3.3. Node Splitting

In chapter two, I discussed a linkage tailoring optimization which I labelled "node splitting". Node splitting is simply a

```c
/* find a set of equivalent call sites based */
/* on constant valued parameters */

/* first create a unique set for each site */
for each edge \( e \) do
    create set(\( e \)) with a single member \( e \)

/* now, UNION similar sites */
for each edge \( e \) do
    for each edge \( f \) do
        if set(\( e \)) \( \neq \) set(\( f \))
            and constant(\( e \)) = constant(\( f \))
            then union(set(\( e \)), set(\( f \)))
```

Figure 6.1
generalization of the semi-open procedure linkage. It creates a separate copy of a procedure body when it can detect that a subset of the call sites for a given procedure share characteristics which have strong potential for allowing improvement of the resulting code. The transformation performs a renaming at each affected call site, so that the procedure call references a copy of the procedure body which is shared only among the sites with common characteristics.

Node splitting can lead to improved run-time through customization of procedure bodies. The opportunities for applying this transformation should be evaluated against those for generating customized linkages, since generation of open and semi-open linkages potentially detracts from the profitability of node splitting. The algorithm presented in the next section does this, pitting node splitting transformations against the other linkages considered in the decision algorithm. The difficult part of performing node splitting lies in discovering which call sites have similar properties with respect to improvement by the optimizer.

The analysis to find potential sites for node splitting is nicely localized. To determine whether a node should be split, the algorithm need only examine the calls to that procedure. To give the flavor of the comparisons which must be done, I present an algorithm for finding the subsets of calls to a procedure which have identical constant valued parameters. To do this, I assume that the editor has annotated each call graph edge e with a list, constant(e). This list contains the value of each actual parameter at the call, if the actual parameter is a constant. Otherwise it contains a designated
value don't know. Given this assumption, figure 6.1 shows a method for partitioning the call sites into equivalence classes based on constant parameter values. Using the \(O(n \log n)\) UNION-FIND algorithm [AHU 74], the cost for unions in the algorithm will be \(O(1)\), where 1 is the number of incoming edges. This is because there are at most 1 sets and 1 unions, and \(O(i)\) will be less than 6 for any call site encountered in practice. The loop overhead will be \(O(1^2)\), but there will be at most 1 UNIONS.

The analysis performed by the estimator can be used to suggest a set of parameters which are worth examining. These parameters will be those which control the execution of a large percentage of the code, values for loop steps, and such. By allowing the estimator to single out parameters or sets of parameters as important, the compiler can focus on areas with a higher likelihood of improvement. The estimator, in effect, acts as a filter which only passes possible node splits with a good potential for code improvement.

### 6.3.4. Linkage Tailoring

The linkage tailoring algorithm has a conceptually simple task. For each call site in the program, it must decide which linkage type to generate. This requires good estimators for improvement and execution frequency, but the choice can be made easily. To actually compute an optimal set of choices is difficult; the problem is identical to Scheifler's substitution problem and therefore unlikely to be solved in polynomial time. Further, since the algorithm presented is intended for application in a compiler, it must be
reasonably efficient. Given the previously described estimators, the efficiency requirements of a compiler, and a constraint on growth for the entire program, the algorithm presented in this section will assign linkages to each call site.

The algorithm must deal with open, semi-open, semi-closed, and closed linkages, as well as node splitting. I assume that the user specifies whether a given compiler is for debugging or production purposes. For debugging activity, all linkages generated are closed linkages, to minimize the recompilation liability of changes to individual procedures.\(^5\) In production compiles, there is little reason to expect many changes, so semi-closed linkages are the default linkage, and other linkage choices are investigated.

The key simplifications we can make in the linkage tailoring algorithm are:

(1) generation of default, open, and semi-open linkages, along with the node splitting transformation;
(2) limit estimate updating in response to a decision to the procedure containing the call site;
(3) limit a call site to involvement in only a single node splitting transformation.

These simplifications decrease the computational overhead associated with each decision. The first halves the number of choices by one. The second bounds the amount updating which must be done in response

\(^5\) The only real advantage of a closed linkage over a semi-closed linkage is the lower liability for recompilation posed with a closed linkage scheme. Since semi-closed linkages impose a compilation, they can drastically increase the number of procedures which must be recompiled in response to an editing change.
Pass 1: for all call sites where open linkage results in reductions in net space, assign the call site an open linkage;

Pass 2: choose a call site, and linkage which does not violate the space constraints and assign them; repeat until no more possible transformations;

Pass 3: assign open linkages to any remaining call sites where open linkage results in reductions in net space;

Pass 4: assign default linkages to all remaining call sites.

Figure 6.2

to each linkage decision. The third limits updating overhead from call sites involved in node-splitting.

The tailoring algorithm is shown in figure 5.2. The four pass algorithm is similar to Scheifler's scheme. The first pass assigns an open linkage anywhere that it does not increase total space. This includes all procedures which are only called once, as well as any procedure whose integrated body is smaller than the default calling sequence. The second pass assigns linkages to the remaining call sites in order of estimated improvement. Since these transformations result in space increases, this can continue only until the space constraints are reached. The order of assignment is determined by a choice function described below. The first two passes can alter the structure of the program sufficiently so that integration of some remaining call sites does not increase total space. In particular, procedures which are left with only a single call can be integrated.
The third pass assigns open linkages to such remaining call sites. The final pass assigns default linkages to any remaining unassigned call site.

At the start of pass two, the choice function constructs a priority queue of choices at the outstanding call sites. The queue is ordered by potential improvement. The function examines each remaining call site and computes a score reflecting the potential improvement available from procedure integration. As the algorithm identifies remaining call sites for generation of open linkages, it also examines the unassigned incoming edges to the procedure, looking for opportunities for generation of either semi-open or node split linkages. Any time it finds multiple calls from a single procedure, an estimate for a semi-open linkage is computed and entered. For node splits identified by analysis sketched above, an aggregate

---

**Estimation Formulas:**

**open linkages:**  \[ \text{score} = \text{improvement} \times \text{frequency} \]

**semi-open:**  \[ \text{score} = \text{tp}_1 \times \text{improvement} \times \text{frequency} \]

where improvement is computed using the meet of the information at all call sites, and \( \text{tp}_1 \) is a tuning parameter reflecting lost optimization opportunities.

**node-split:**  \[ \text{score} = \sum \text{improvement} \times \text{frequency} \]

where improvement is computed using the common set of information

**Figure 6.3**
improvement score is computed and entered. Formulas for computing the estimates are given in figure 5.3. Only open, semi-open, and node split linkages are entered, since the algorithm assigns default linkages in pass four. On each subsequent call, the choice function returns the first entry in the queue which does not violate the space constraints, and updates any affected entries in the queue.

Updating the queue in response to a single assignment is simple. Since each call site can have at most three entries in the queue, the choice function keeps a table of queue entries by call site. After each choice, it removes the entries for the alternative choices: open linkages simply require deleting, whereas node splits and semi-open linkages require some numerical adjustment, since they represent aggregate scores for multiple call sites. The choice function must compute the contribution of the call site to the aggregate score, remove the element, adjust its score, and re-enter the element with its new score.

Finally, the choice function must account for the change in expected performance of the procedure containing the call site just assigned. The scores of queue entries are estimated improvements; the improvement estimate for the calling procedure is increased by the estimated improvement for the call site. By the second simplification, only procedures which directly invoke the calling procedure require updating. This can be done by computing the estimated improvement at the source of each incoming edge to the calling procedure and updating the corresponding queue elements. This results in at most the number of incoming edges to the calling
procedure updates.

Over the course of the entire algorithm, a single procedure will need to update estimates along each of its incoming edges once for each call site in the procedure. If \( \imath \) is the number of incoming edges to the procedure and \( \o \) is the number of outgoing edges, then there will be at most \( \imath \times \o \) updates in the procedure. Over the entire run of the algorithm, this can require at most

\[
\sum_{p \in \mathcal{PF}} \imath_p \times \o_p \leq |E_{CG}|^2
\]

updates, each of which involves only minor arithmetic and queue manipulation. The remainder of the costs in the choice function are either \( \mathcal{O}(|E_{CG}|) \) or \( \mathcal{O}(|E_{CG}| \log_2 |E_{CG}|) \), the updating costs dominate the algorithm. Ignoring the costs of estimation, the choice algorithm requires \( \mathcal{O}(|E_{CG}|^2) \) time over the entire linkage tailoring process. This estimate is generous, in that it assumes that every call site is involved in a semi-open or node splitting linkage, which may not be true in a given program.

The key issue in a discussion of the time complexity of linkage tailoring is that estimation and queue updating dominate the other costs of the algorithm. The algorithm presented distributes the estimation costs over local editing sessions in the procedure. This distribution of cost over editing makes some intensive analysis palatable in an environment where each user has a workstation dedicated to supporting his editor session. The editor performs both pass one and pass two of the estimator. The third pass of the estimator requires a linear pass through the called procedure at each
call site.

Updating costs in the choice algorithm are potentially $O(|E_{CC}|^2)$, due to the presence of semi-open and node split linkages. In practice, not every procedure will be eligible for such transformations, decreasing the costs from this upper bound.

The question must be answered: "Can this optimization be applied on an incremental recompilation due to editing changes?". The answer is: certainly; the algorithm will only consider the available call sites. It must be noted that application to a subset of the call sites will favor local optimization over global optimization, and the results produced will undoubtedly be much different than those coming out of a recompilation of the entire program.

To allow global optimizations to tailor the bodies of procedures with specialized linkages, this optimization should be applied before other optimizations. Much of the improvement measured by the estimation method described above depends on early application of linkage tailoring.

### 6.4. Implication: for Other Software

The mere fact that a compiler performs linkage tailoring has profound effects on other software in the programming environment. The implications can be found in areas as diverse as the structured editor, the database system, and the interpreter/debugger. The other software must understand these issues if it is to successfully interact with the compiler.
One class of problems arising out of linkage tailoring is a
proliferation of site specific variants of procedures. If the choice
algorithm ever recommends node splits or semi-open linkages, there
will be copies of the procedure which are optimized for one or more
call sites. These procedure bodies may behave incorrectly if used
from other call sites. For a procedure used in multiple versions of
multiple programs, there may be tailored versions for multiple call
sites in each program/version combination, leading to an explosion in
the number of mutants in the database. The database system must
understand not only that such mutants exist, but also that they
should never be visible to any piece of software other than the
compiler. Further, the system must maintain an unoptimized version
of each procedure, for editing and pretty-printing purposes, and be
able to treat the specialized procedures in a manner consistent with
the source code in the unoptimized version, particularly for analysis
of dependences related to recompilation.

Since procedure integration can erase entire levels of a
program's hierarchal structure, the optimization has the potential
for disguising the source of run-time errors. Both the compiler and
the interpreter/debugger must understand this issue; the former must
leave around enough information to allow the latter to decipher what
has occurred. The issue extends beyond the simple matter of naming
the current procedure and line number; open and semi-open linkages
severely distort the storage mapping of a procedure. When a user
asks the interpreter for a variable's value, the interpreter must be
able to determine what the actual address for the variable is at the
current point in the program's execution. With linkage tailoring, this becomes a much more difficult problem than under a separate compilation scheme for FORTRAN.

One of the major advantages of the proposed programming environment is that the version control software can automate the process of reconstructing executable images of a specific version of a program. Key tools which the environment provides for making decisions about recompilation are the dependences represented in the call graph and summary data flow information on individual procedures. The generation of open and semi-open linkages changes this situation. After a procedure has been integrated, a discrepancy exists between the generated code and the call graph of the program. A change to the called procedure necessitates recompilation of the entire integrated procedure body, even though the edge in the call graph which represents the call has been removed. Similarly, the software must consider all nodes resulting from node splits of a single node as if they are a single node for purposes related to recompilation. Finally, the summary information which is true about a procedure may become more precise when the procedure is integrated in the presence of knowledge about the values of actual parameters. The recompilation decisions must be wary of such information, using more general information in response to changes in either the calling or called procedures.
6.5. Conclusions

The algorithm presented assigns linkages to all of the call sites in a program. It assigns open, semi-open, and default linkages, as well as deciding when it is profitable to generate specially optimized versions of a procedure and use them with default linkages. The design of the algorithm limits its costs to make them acceptable in an actual compiler; consequently the algorithm generates a less than optimal assignment.

The relative profitability of linkage tailoring is a function of the source language and the specific compiler under consideration. In a language like C where procedure calls are relatively cheap, the expense involved in estimation and assignment may not be justified. In a situation where procedure calls are expensive, like IBM's PL/I or the next FORTRAN standard, it seems likely that the optimization will easily pay for the effort.
CHAPTER 7

Conclusions

This dissertation has examined three problems arising in the construction of a compiler based in a programming environment. The problems, alias analysis, summary information, and linkage tailoring, have all been previously addressed in the literature. As the previous chapters have shown, the surroundings of a programming environment impact the solutions to these problems, by providing the compiler new resources and information and by presenting different costs. This chapter summarizes the work presented in previous chapters, evaluates them against criteria proposed in chapter two, and proposes some directions for future research.

7.1. Alias Analysis

The analysis of the aliasing patterns present in a program is posed as a global data flow analysis problem on the program's call multi-graph. The problem is divided into two subproblems: detection of alias introduction and propagation of alias pairs. The former problem is solved during syntax analysis by the editor or parser in an efficient manner. The latter problem is solved using a variant of the iterative global data flow analysis algorithm. The data flow pass requires at most $|PP|(|PP|-1)^{f_{max}}$ node visitations before halting. The introduction analysis recognizes edges which are irrelevant to the entire alias analysis, as well as edges which are constant valued throughout the propagation phase. This decreases the size of the
graph used by the propagation phase. The propagation algorithm defers computation gracefully in response to unavailability of needed procedures. At each step of the algorithm, equality checks on set values enable the algorithm to limit its attention to nodes whose alias sets have possibly changed.

7.2. Summary Information

Two approaches to the computation of summary information are presented. The first is adapted from the framework given in Banning's dissertation. This treatment is a straightforward application of the algorithm devised to solve the aliasing propagation problem. The editor computes initial information for the problem, which is used as input to a data flow pass over the program's call graph. The algorithm gracefully defers work in response to availability or synchronization problems. It limits its work to that area of the graph where summary information can change. The information computed is identical to that computed by Banning. While the development is done in terms of MOD information, extensions are given to USE and PRE information.

The second approach to summary information splits the problem into two subproblems. The first, computing summary information for global variables, is shown to be rapid in the sense of Kam and Ullman. A depth-first version of the iterative algorithm will solve an instance of this problem in $d(G) + 2$ passes over the graph. The second subproblem, computing summary information for reference formal parameters, is solved as a single source path expression problem.
Taking advantage of the assumption that $f_{\text{max}}$ is a small constant, we can use Tarjan's algorithm with path compression to solve this problem in $O(mc(m,n) + n)$ time, where $m$ is the number of call sites and $n$ is the number of procedures, and we assume bit-vectors of length $|\text{PP}| \times f_{\text{max}}$. This time bound is a significant improvement over the first approach.

7.3. **Linkage Tailoring**

The linkage tailoring problem is divided into two problems, estimating improvement and assigning linkage types. While estimation techniques are heavily dependent on details of the actual implementation, a general framework for estimation is presented. The algorithm uses data flow analysis techniques to assess the extent to which the values computed in a procedure are functions of values known at the invoking call site. The estimation process computes values for potential improvement due to the generation of open and semi-open linkages, as well as a generalization of the semi-open linkage, which I have termed the node-split linkage.

An algorithm is given for assigning linkage types to call sites. Since Scheifler has shown that this problem is NP-hard, it is likely that there is no efficient means of computing the optimal solution. The algorithm presented uses simplifications and approximations to efficiently produce an assignment. Local improvement is used to assess the relative benefit of specific decisions, ignoring the global impact of decisions in order to render the problem tractable. The algorithm considers open, semi-open, semi-closed, and node-split
linkages. Group linkages, like sem-open and node-split, are evaluated by computing the aggregate improvement over all involved call sites. The algorithm presented requires $O(|E_{CG}|^2)$ time to compute an assignment.

7.4. Relating the Three Problems

While the problems have been presented in three independent treatments, they are, in fact, closely related. The aliasing problem requires the presence of all procedures in the program for solution. Our algorithm uses the long-term database of the programming environment to facilitate the updating process and to ease synchronization problems. The summary information algorithms are simplified by the fact that alias analysis has been factored out as a separate analytic step. These algorithms, like the aliasing algorithm, require the presence of all procedures and use the database as a tool to ease their tasks. The presence of the database is an implicit requirement for a compiler which attempts to perform linkage tailoring. The linkage tailoring optimizations presented take advantage of the compiler's free access to all procedures, the presence of summary information and performance estimates, and the ability to create and maintain mutant copies of procedure bodies. The linkage assignment algorithm uses information derived from constant propagation, parameter dependence analysis, and the summary information algorithms.

The algorithms are also related by their information requirements. The aliasing algorithm computes and uses bind sets for
each call site. The summary information algorithm requires backbind sets for each call site. The algorithm given in figure 3.3 to compute bind also computes backbind. Computing parameter dependence analysis requires the use of summary information, which relies on aliasing information.

7.5. Evaluating the Algorithms

In chapter two, three criteria were presented for algorithms applied in a programming environment. These criteria were:

(1) the ability to defer work;
(2) inexpensive updating in response to editing changes;
(3) limited maneuvering around the call graph.

It is reasonable to examine each of the algorithms presented against each of these criteria.

The aliasing algorithm performs well under these criteria. Alias introduction analysis is inexpensive, and easily performed during syntax analysis. Posing the propagation problem as a data flow problem lead to an algorithm which retains sufficient information to allow deferred updates and to limit its attention to precisely those nodes in the graph which editing changes can impact. By performing as much processing as possible at each node visit, the algorithm tries to limit its motion in the call graph. Unlike Ranne's algorithm, the alias propagation algorithm traverses the graph in a predictable and disciplined fashion.

The first summary information algorithm is an application of the alias propagation algorithm to the computation of GMOD. This
formulation follows from Banning's work. By building on the alias propagation algorithm, the algorithm retains the good properties of that algorithm. It observes a queue discipline to allow for orderly traversal of the call graph, easy handling of deferred work, and limiting updates to consideration of the effected portion of the graph.

The second summary information algorithm uses a pair of algorithms to compute GMOD for a program. Computing GMOD for global variables is done in a manner very similar to the first summary information algorithm. Removing the call-by-reference parameters simplifies the problem sufficiently to guarantee an efficient solution using a depth first version of the iterative data flow algorithm. The alias propagation algorithm can be adapted to this task. Computing GMOD for reference formal parameters requires use of Tarjan's algorithm for solving single source problems. The resulting algorithm has a much lower time complexity than the algorithm of chapter four, for reducible call graphs. Direct techniques are shown for efficiently updating the resulting summary information in response to non-structural editing changes in the program. More work is needed to show efficient updating techniques in response to changes in the structure of the call graph.

The linkage tailoring algorithm is inherently different than the other algorithms presented. Linkage tailoring decisions are made as the compiler builds an executable image of the program. The algorithm can not defer work, since it must have all of the procedures present and available to generate code. The algorithm
presented can be invoked on a subset of a program's call sites, but the assignments produced are likely to vary greatly from those produced by considering the complete program. The assignment algorithm requires only an initial pass over the program, after which it deals only with numerical estimates rather than procedure bodies. Little physical storage is needed for these estimates, offsetting the algorithm's need for essentially random access to the estimates. The assumptions made to simplify the problem explicitly limit motion through the call graph; unfortunately the algorithm almost always requires multiple visits to a node.

The aliasing and summary information algorithms do well under the criteria of chapter two, while the tailoring algorithm does not. The failures of the tailoring algorithm lie in the nature of the operation, rather than in the formulation presented. This optimization is likely to be expensive to perform, but the overhead must be balanced against its potential for improvement in run-time behavior. It seems likely that the optimization will be profitable for languages with expensive procedure call overhead, like the next FORTRAN standard.

A final criteria suggested in chapter two was careful division of labor between the editing and compiling processes. Each of the algorithms presented has a clear division between those tasks which are appropriate for the editor to perform and work better done in a compilation pass over the call graph.
7.6. Suggestions for Future Research

The solutions presented to the aliasing and summary information problems are suitable for implementation. An implementation of the linkage tailoring scheme will require careful tuning to achieve good results in a specific compiler.

Based on this work, several areas of further inquiry suggest themselves.

The approximative aliasing technique suggested in chapter five should be evaluated against the precise technique of chapter three. A study of the actual density of aliasing and longevity of alias pairs in actual programs could provide a much clearer picture of both the expected case time requirements and the impact of using the approximate solution. If aliasing is sparse and alias pairs short-lived, the alias propagation algorithm requires much less time than given time bound. Aliasing is the factor which makes the interprocedural must summary problems co-NP complete [Myer 81]. Better knowledge of the aliasing patterns encountered in actual programs can help us assess whether algorithms can be devised which will solve must summary problems in reasonable amounts of time for real programs.

Further work is clearly needed on update techniques for the swift algorithm for computing MAP*. The existence of an efficient algorithm to account for deletion of call graph edges would greatly improve the suitability of the algorithm for actual implementation.
The linkage tailoring algorithms and optimizations are too complex for evaluation without an implementation. However, the potential for program improvement that they present should make that experimentation worthwhile. A particularly interesting possibility is combining linkage tailoring with extensive storage optimization [Fabr 79] [Thab 81]. The linkage tailoring algorithms should be implemented, applied to a large selection of programs, and the resulting code studied. In particular, the formulas used for computing improvement will need tuning for the algorithm to achieve good assignments. Such a project will require many man years of effort.
APPENDIX A

Lattice Proofs

The Aliasing Lattice

Lattice Meet Function

The meet function is set union \((\cup)\). It is:

- idempotent: \(a \cup a = a\)
- commutative: \(a \cup b = b \cup a\)
- associative: \(a \cup (b \cup c) = (a \cup b) \cup c\)

It is important to observe that \(\geq\) is equivalent to \(\leq\) for \(\wedge = \cup\).

Kam and Ullman's Admissibility Criteria

A function space \(F\) is said to be admissible [Kaul 76] with respect to a semilattice \(L\) iff:

1. Each \(f \in F\) distributes over \(\wedge\), the semilattice meet function.
2. \(\exists\) an identity function \(i \in F\) s.t. \(i(x) = x \forall x \in L\).
3. \(F\) is closed under composition.
4. \(\forall x \in L, \exists\) a finite subset \(H\) of \(F\) s.t. \(x = \wedge_{f \in H} f(i)\).

The functions in the associated function space \(F_A\) are of the form:

\[ f(x) = A \wedge g(x) \]

where \(A\) and \(B\) are constant sets, \(A \leq W^W\), and \(g\) is an arbitrary function mapping \(W^W \rightarrow 2^{W^W}\).

Claim: Our function space \(F_A\) is admissible.
(1) Distributivity follows from the characterization of $f \in F_A$.
Each function is of the form

$$f(x) = A \cup g(x)$$

with $A$ an arbitrary constant set, $A \leq A^P$, and a map $g: A^P \to 2^{A^P}$.
Recall that $\land$ is simply $\cup$, and the $\{\}$ notation describes the
elemental extension of a function to a set. We must show that:

$$f(a \land b) = f(a) \land f(b), \forall a, b \in L_A : \forall f \in F_A.$$  

Substituting the characterization of $f$ into this equation gives:

$$f(a \cup b) = (A \cup g(a)) \cup (A \cup g(b))$$

$$f(a \cup b) = A \cup g(a) \cup g(b)$$

Observing that $g(a) \cup g(b)$ is equivalent to $\{\cup_{\alpha} g(\alpha) \cup \{\cup_{\beta} g(\beta)\}$ leads to

$$f(a \cup b) = A \cup g(a \cup b)$$

Now expanding the left hand side yields:

$$f(a \cup b) = A \cup g(a \cup b)$$

which is equivalent to the right hand side.

(2) The identity function $i(x)$ can be generated by setting $A_i = \emptyset$
and defining a map $g_i(x) = x, \forall x \in A^P$. Now,

$$i(x) = A_i \cup g_i(x).$$

(3) Closure under composition follows from the characterization of
$f \in F_A$. Consider two arbitrary elements of $F_A$,

$$f_1(x) = A_1 \cup g_1(x)$$

$$f_2(x) = A_2 \cup g_2(x).$$
We must show that there is a function \( f_{102} \in F_A \) such that
\[
f_{102}(x) = f_1(f_2(x)).
\]
Expanding the right hand side of the desired equality:
\[
f_1(f_2(x)) = f_1(A_2 \cup g_2(x))
\]
\[
f_1(f_2(x)) = A_1 \cup g_1(A_2) \cup g_1(g_2(x))
\]
If we let
\[
A_3 = A_1 \cup g_1(A_2)
\]
\[
g_3(x) = g_1(g_2(x))
\]
then \( f_3(x) = f_{102}(x) = f_1(f_2(x)) \in F_A \).

(4) For any specific \( x \in L_A \), we can generate a single function \( f_x \in F_A \) such that \( f_x(\bot) = x \). To generate \( f_x \), let \( A_x = x \), and let \( g_x(a) = \emptyset, \forall a \in L_A \). Then,
\[
f_x(y) = A_x \cup g_x(y).
\]
Applying \( f_x \) to \( \bot \), the empty set, yields \( x \).

Taken together, these demonstrate that \( F_A \) meets the Kam and Ullman criteria for admissibility.

**Monotonicity**

A function space \( F \) associated with a semilattice \( L \) is monotone [KaUl 77] iff:

(1) \( (\forall f \in F)(\forall x, y \in L) f(x \land y) \leq f(x) \land f(y) \).
(2) \( \exists \) an identity function \( i \in F \) s.t. \( i(x) = x \ \forall x \in L \).
(3) \( F \) is closed under composition.
(4) \( L = \) the closure of \( \bot \) under meet and application of \( f \in F \).
Claim: \( F \) is monotone.

To demonstrate monotonicity, simply observe that (2), (3), and (4) are equivalent to the corresponding admissibility conditions. Now, (1) is implied by the fact that functions \( f \in F_A \) distribute over \( \land \), which is just the first of the four admissibility criteria.

\( \land \) is monotone.
The **MOD Lattice**

The meet function is set union ($\cup$). The join function is set intersection ($\cap$). They are both:

- **Idempotent:** $a \cup a = a$
  $a \cap a = b$
- **Commutative:** $a \cup b = b \cup a$
  $a \cap b = b \cap a$
- **Associative:** $a \cup (b \cup c) = (a \cup b) \cup c$
  $a \cap (b \cap c) = (a \cap b) \cap c$

Note that $\geq$ is equivalent to $\leq$ for $\wedge = \cup$.

**Ram and Ullman's Admissibility Criteria**

A function space $F$ is said to be admissible [KaUl 76] with respect to a semilattice $L$ iff:

1. Each $f \in F$ distributes over $\wedge$, the semilattice meet function.
2. There exists an identity function $i \in F$ s.t. $i(x) = x \forall x \in L$.
3. $F$ is closed under composition.
4. $\forall x \in L$, $\exists$ a finite subset $B$ of $F$ s.t. $x = \bigwedge_{f \in B} f(1)$.

The functions in the associated function space $F_{MOD}$ are of the form:

$$f(x) = (A \cap x) \cup B \cup g[x]$$

where $A$ and $B$ are constant sets, $A \subseteq \mathcal{V}$, $B \subseteq \mathcal{V}$, and $g$ is an arbitrary function mapping $\mathcal{V} \rightarrow 2^{\mathcal{V}}$.

**Claim:** Our function space $F_{MOD}$ is admissible.

1. Distributivity follows from the form of the functions. We must show that:

$$f(a \wedge b) = f(a) \wedge f(b), \forall a, b \in L_{MOD}, \forall f \in F_{MOD}$$
Substitution gives us
\[
f(\text{sub}) = (A \cap a) \cup B \cup g(a) \cup (A \cap b) \cup B \cup g(b)
\]
\[
f(\text{sub}) = B \cup (A \cap a) \cup (A \cap b) \cup g(a) \cup g(b)
\]
\[
f(\text{sub}) = B \cup (A \cap \text{sub}) \cup g(a) \cup g(b)
\]
Observe that \(g[a] \cup g[b] = \{u_{a \in a} \, g(a)\} \cup \{u_{b \in b} \, g(b)\}\). This is equivalent to \(\{u_{a \in (\text{sub})} \, g(a)\}\), which is just \(g[\text{sub}]\). So,
\[
f(\text{sub}) = B \cup (A \cap \text{sub}) \cup g[\text{sub}]
\]
Expanding the left hand side yields
\[
(A \cap \text{sub}) \cup B \cup g[\text{sub}] = B \cup (A \cap \text{sub}) \cup g[\text{sub}]
\]
which is equivalent. In \(L_A\) and \(F_A\), function application distributes \(\land\).

(2) The identity function \(i(x)\) can be generated by setting \(A_i = \emptyset\), \(B_i = \emptyset\), and letting \(g_i(x) = x, \forall x \in \mathbb{V}\). This gives a function
\[
i(x) = (A_i \cap x) \cup B_i \cup g_i(x)
\]
\[
i(x) = \emptyset \cup \emptyset \cup g_i(x)
\]
\[
i(x) = g_i(x)
\]
This function simply returns its argument, the desired effect.

(3) To demonstrate closure under composition, we must show that for any two functions \(f_1, f_2 \in F_{\text{MOD}}\), there is a function \(f_3 \in F_{\text{MOD}}\) such that \(f_3 = f_1 \circ f_2\). To show this, consider two such functions:
\[
f_1(x) = (A_1 \cap x) \cup B_1 \cup g_1(x)
\]
\[
f_2(x) = (A_2 \cap x) \cup B_2 \cup g_2(x)
\]
To discover the nature of \( f_3 = f_{102} \), we expand \( f_1(f_2(x)) \), giving:

\[
f_1(f_2(x)) = f_1((A_2 \cap x) \cup B_2 \cup g_2 \{x\})
\]

\[
f_1(f_2(x)) = (A_1 \cap A_2 \cap x) \cup (A_1 \cap B_2) \cup (A_1 \cap g_2 \{x\})
\]

\[
\cup B_1 \cup g_1 \{(A_2 \cap x) \cup B_2 \cup g_2 \{x\}\}
\]

\[
f_1(f_2(x)) = (A_1 \cap A_2 \cap x) \cup (A_1 \cap B_2) \cup (A_1 \cap g_2 \{x\})
\]

\[
\cup B_1 \cup g_1 \{B_2\} \cup g_1 \{(A_2 \cap x) \cup g_2 \{x\}\}
\]

Let \( A_3 = (A_1 \cap A_2) \)

\( B_3 = (A_1 \cap B_2) \cup B_1 \cup g_1 \{B_2\} \)

\( g_3(x) = (A_1 \cap g_2 \{x\}) \cup g_1 \{(A_2 \cap x) \cup g_2 \{x\}\} \)

Now, \( f_3(x) = f_1(f_2(x)) = f_{102}(x) \). We need to show that \( f_3 \in F_{\text{MOD}} \). Since \( A_1 \) and \( A_2 \in V_V \), it follows that \( A_3 \in V_V \). So \( A_3 \leq V_V \). To show that \( B_3 \leq V_V \), observe that \( A_1 \cap B_2 \leq V_V \), \( B_1 \leq V_V \), and \( g_1 \{B_2\} \leq V_V \) since \( g_1 : V_V = 2^V_V \) and all the elements of \( 2^V_V \) are subsets of \( V_V \), by definition of \( 2^V_V \). The function \( g_3 \) is simply the union of applications of \( g_1 \) and \( g_2 \), each intersected with various constants. From the conditions on the original functions, \( f_1 \) and \( f_2 \), we can conclude that \( g_3 \) fits the characterization. So, \( f_3 \in F_{\text{MOD}} \) and the function space is closed under composition.

(4) For any \( x \in L_{\text{MOD}} \), we can define a function \( f_x \in F_{\text{MOD}} \) such that \( x = f_x(\tau) \). For a given \( x \), let \( A_x = \emptyset \), \( g_x(a) = \emptyset \), \( \forall a \in V_V \), and \( B_x = x \). Then,
\[ f_x(y) = A_x \cup B_x \cup g_x(y) \]
\[ E_x(y) = \emptyset \cup \emptyset \cup g_x(y) \]
\[ F_x(y) = g_x(y) \]
\[ f_x(y) = x \]

Applied to any argument, including \( T \), this function returns \( x \).

**Monotonicity**

As observed in the proof for \( F_A \), any lattice which meets the admissibility criteria of [KaU1 76] is also monotone, under the definition given in [KaU1 77]. \( F_{MOD} \) is monotone.
\( L_{\text{MOD-G}} \)

The lattice for the MOD-G problem is presented in chapter five. It is given here for convenience.

\[
L_{\text{MOD-G}} = (2^X, \lor, \land)
\]

where \( 2^X \) is the power set of \( X \)
\( \lor \) denotes lattice join, and
\( \land \) denotes lattice meet.

\( \land \) is set union \( (\cup) \)
\( \lor \) is set intersection \( (\cap) \)

\( \bot \) is \( \mathbb{VW} \), the set of all variable names
\( \top \) is \( \emptyset \), the empty set

\( \preceq \) is equivalent to \( \leq \)
\( \succeq \) is equivalent to \( \geq \)

**Kam and Ullman admissibility**

(1) Distributivity: We need to show that lattice meet distributes over function application.

\[
\begin{align*}
\tilde{f}(a \land b) &= \tilde{f}(a) \land \tilde{f}(b) \\
\tilde{f}(a \lor b) &= ((\tilde{f}(a) \land \tilde{f}(b)) \lor (\tilde{f}(a) \lor \tilde{f}(b))) \\
\tilde{f}(a \lor b) &= ((\tilde{f}(a) \land \tilde{f}(b)) \lor (\tilde{f}(a) \lor \tilde{f}(b))) \\
\tilde{f}(a \land b) &= (\tilde{f}(a) \lor \tilde{f}(b)) \land (\tilde{f}(a) \land \tilde{f}(b)) \\
\end{align*}
\]

(2) Identity: The identity function can be generated by setting \( A = \mathbb{VW} \), and \( B = \emptyset \). For this choice of constants, the function becomes:

\[
\begin{align*}
\tilde{f}_i(x) &= (\mathbb{VW} \cap x) \cup \emptyset \\
\tilde{f}_i(x) &= (\mathbb{VW} \cap x)
\end{align*}
\]

For any \( x \in \mathbb{VW} \), \( \tilde{f}_i(x) \) is the identity function.
(3) Composition Closure: To show closure under composition, we must show that \( \forall f_1, f_2 \in F_{\text{MOD}-G^*} \) there is a function \( f_3 \in F_{\text{MOD}-G} \) such that \( f_3 = f_1 \circ f_2 \). Consider two such functions:

\[
\begin{align*}
 f_1(x) & = (A_1 \cap x) \cup B_1 \\
 f_2(x) & = (A'_2 \cap x) \cup B'_2
\end{align*}
\]

We expand \( f_3 = f_1(f_2(x)) \), giving:

\[
\begin{align*}
 f_1(f_2(x)) & = f_1((A_2 \cap x) \cup B_2) \\
 f_1(f_2(x)) & = (A_1 \cap A_2 \cap x) \cup (A_1 \cap B_2) \cup B_1
\end{align*}
\]

Let \( A_3 = (A_1 \cap A_2) \)

\( B_3 = (A_1 \cap B_2) \cup B_1 \)

Now, \( f_3(x) = f_1(f_2(x)) = f_1 \circ f_2(x) \). We need to show that \( f_3 \in F_{\text{MOD}} \). Since \( A_1 \) and \( A_2 \in \mathcal{VV} \), it follows that \( (A_1 \cap A_2) \in \mathcal{VV} \), so \( A_3 \leq \mathcal{VV} \). To show that \( B_3 \leq \mathcal{VV} \), observe that \( (A_1 \cap B_2) \leq \mathcal{VV} \) and \( B_1 \leq \mathcal{VV} \), so \( B_3 \leq \mathcal{VV} \). Since \( A_3 \) and \( B_3 \) meet the conditions for membership in \( F_{\text{MOD}-G^*} \), \( f_3 \in F_{\text{MOD}-G} \), and the function space is closed under composition.

(4) Finite Subset Closure: To show this, we can simply construct a function, \( f_x \in F_{\text{MOD}-G} \) which returns \( x \) irrespective of its argument. Let \( A_x = \emptyset \) and \( B_x = x \). Now, \( f_x \) returns \( x \) for any argument in \( L_{\text{MOD}-G} \). Since \( x \in L_{\text{MOD}-G} \), \( x \leq \mathcal{VV} \). Clearly, \( \emptyset \leq \mathcal{VV} \). So, \( f_x \in F_{\text{MOD}-G} \), and, for any \( x \in L_{\text{MOD}-G} \), we can construct a single function such that \( f_x(1) = x \).
Since $F_{MOD-G}$ meets each criteria, it is admissible.

**Rapidity**

For admissible data flow frameworks which meet their **rapidity**
criteria, Kam and Ullman give a depth first version of Kildall's
iterative algorithm which will solve an instance of the problem in
d$(G)+2$ passes over the graph. The criteria for $L_{MOD-G}$ is that:

$$f(x) \geq x \wedge f(\tau) \quad \forall f \in F_{MOD-G} \quad \forall x \in L_{MOD-G}.$$  \(^1\)

It is easy to show that this criteria holds for every $f \in F_{MOD-G}^*$

$$f(x) \geq x \wedge f(\tau)$$
$$f(x) \geq x \cup f(\emptyset)$$
$$f(x) \geq x \cup B$$
$$(A \cap x) \cup B \geq x \cup B$$

This last inequality holds for all choices of $x \in L_{MOD-G}$, $A \leq VV$, and
$B \leq VV$. $L_{MOD-G}$ is a **rapid** framework.

\(^1\)See [Kaul 76], Observation 6, page 166.
$L_{\text{MOD-R}}$

The semilattice for the MOD-R problem is presented in chapter five. It is given here for convenience.

$L_{\text{MOD-R}} = (B, \wedge)$

where $B$ is the set of $n$-tuples with elements drawn from $2^V$

$\wedge$ denotes lattice meet

$\wedge$ is the elemental union of tuples

$\bot$ is the tuple whose elements are all $V$

Functions in the associated function space, $F_{\text{MOD-R}}$ are of the form:

$f(x) = <s_0(x), s_1(x), \ldots, s_{n-1}(x)> \wedge B$

where the $s_i$ are selector functions on $X$ which return the meet of selected elements of a tuple, $B \in L_{\text{MOD-R}}$ is an arbitrary constant, and the angle brackets $(<,>)$ denote an $n$-tuple constructor.

To fit Tarjan's formulation, the functions in $f$ must satisfy the following axioms:

(1) $F$ contains an identity function $f_i$ such that $f_i(x) = x, \forall x \in L$.

(2) $F$ is closed under meet, function composition, and closure,

where $(f \wedge g)(x) = f(x) \wedge g(x)$ and $f^*(x) = \wedge \{f^i(x) | i \geq 0\}$.

(3) $\forall f \in F$ and nonempty $X \subseteq L$, $f(\wedge X) = \wedge \{f(x) | x \in X\}$.

Proving that these three axioms hold for $F_{\text{MOD-R}}$ is not difficult.

(1) Identity function: The identity function, $f_i \in F_{\text{MOD-R}}$, is generated by setting $B[j] = \emptyset, 0 \leq j \leq n-1$, and defining $s_j$ as the selector which takes only the $j$th element of its argument. Given this assignment, $f_i(x) = x, \forall x \in L_{\text{MOD-R}}$. 
(2) Closure under meet, composition, and closure: To show closure under meet, we must show that for any functions $f$ and $g \in F_{\text{MOD-R}}$, there exists $h \in F_{\text{MOD-R}}$ such that $h(x) = (f \land g)(x) = f(x) \land g(x)$. Since $f$ and $g$ are defined as:

$$f(x) = <s_{f_0}(x), \ldots, s_{f_{n-1}}(x)> \land Bf$$
$$g(x) = <s_{g_0}(x), \ldots, s_{g_{n-1}}(x)> \land Bg$$

we can construct $h$ by letting $Bh = Bf \landBg$, and $s_{h_i}$ be a selector function which returns the meet of $s_{f_i}$ and $s_{g_i}$ applied to the same argument. Because the $s_{\cdot}$ simply return the meet of selected elements, $s_{h_i}$ can clearly be constructed from $s_{f_i}$ and $s_{g_i}$.

To show closure under composition, we need to show a rule for constructing $h = f \circ g(x) = f(g(x))$ for $h, f, g \in F_{\text{MOD-R}}$. Recalling the structure of $f$ and $g$ discussed previously, we can see that

$$Bh = Bf \land <s_{f_0}(Bg), \ldots, s_{f_{n-1}}(Bg)>.$$  

For given $f$ and $g$, this is a constant. It remains to define the selector functions $s_{h_i}$, $0 \leq i \leq n-1$. Consider that each selector function returns the meet of selected elements of its arguments. Then, $s_{h_i}$ must simply return the meet of the composition of the selections made by $s_{f_i}$ and the selector functions $s_{g_j}$ corresponding to each element $j$ selected by $s_{f_i}$. While complicated to describe, this is easily constructed, yielding a function

$$h(x) = <s_{h_0}(x), \ldots, s_{h_{n-1}}(x)> \land Bh$$

which is precisely $f \circ g$. 
To show that $F_{MOD-R}$ is closed under closure, we must show that for any $f \in F_{MOD-R}$ there is a function $f^* \in F_{MOD-R}$ such that

$$f^*(x) = \wedge\{f^i(x) \mid i \geq 0\}.$$ 

Consider the form of functions in $F_{MOD-R}$. Clearly, the constant term $B$ is no problem. No matter how many times we "meet in" $B$, the portion of the result due to $B$ will remain unchanged.

The selector functions, $s_j$, simply choose elements of the argument tuple and return the meet of the selected elements. Consider the composition of selectors appearing in $f^1, f^2, f^3$, and so on. Since each tuple has but $n$ elements, all combinations of those elements which are going to be returned by the selectors will have been returned by $f^{n-1}$. The argument for this is exactly analogous to the binding paths argument in the time bound for the alias propagation analysis. We can represent the set of selectors in $f$ with a graph, nodes for the selector positions and edges indicating selection. The longest acyclic path in the graph will be $n-1$ edges long. Cycles are uninteresting to this analysis since they simply "meet in" the same set multiple times, to no effect.

So, $f^*(x) = \wedge\{f^i(x) \mid n \geq i \geq 0\}$. Since this is a finite collection of compositions and meets, and $F_{MOD-R}$ is closed under composition and meet, $f^* \in F_{MOD-R}$.

(3) To show this, we simply expand each side of the equation. The left hand side yields:

$$f(\wedge X) = \langle s_0(\wedge X), \ldots, s_{n-1}(\wedge X) \rangle \land B.$$
Assuming \( X = \{x_1, x_2, \ldots, x_k\} \), the right hand side expands as

\[
\land\{f(x) \mid x \in X\} = \\
<\text{s}_0(x_1), \ldots, \text{s}_{n-1}(x_1)> \land B \land \\
<\text{s}_0(x_2), \ldots, \text{s}_{n-1}(x_2)> \land B \land \\
\ldots \land \\
<\text{s}_0(x_k), \ldots, \text{s}_{n-1}(x_k)> \land B
\]

From the nature of the selector functions, it is clear that \( \land \) will distribute over application of the selector function, with the resulting simplification to

\[
\land\{f(x) \mid x \in X\} = \\
<\text{s}_0(\land x_1 \land x_2 \land \ldots \land x_k), \\
\ldots \\
\text{s}_{n-1}(\land x_1 \land x_2 \land \ldots \land x_k)> \\
\land B
\]

which is just

\[
<\text{s}_0(\land x), \ldots, \text{s}_{n-1}(\land x)> \land B.
\]

So, \( f(\land x) = \land\{f(x) \mid x \in X\} \), and the three axioms are satisfied.
APPENDIX 3

Graham-Wegman Formulations

Graham and Wegman present an algorithm for solving global data flow problems. Their algorithm is based on a series of reductions similar to Hecht's T1-T2 transformations.

Definitions

Before proceeding, a few definitions are in order. A flow graph, $G = (N,E,u)$, is a graph with a designated node $u$ such that there is a path from the designated node to each other node in the graph.

For two functions, $f$ and $g$, we define:

- **intersection** $h = f \cap g$ means $h(x) = f(x) \cap g(x)$
- **composition** $h = f \circ g$ means $h(x) = f(g(x))$
- **containment** $f \subseteq g$ $\iff f(x) \subseteq g(x) \forall x$

Further, a function is **monotonic** if $x \leq y \iff f(x) \leq f(y)$. A function space is **monotonic** if all of the functions in the space are monotonic.¹

A function is **fast** $\iff f(x) \cap x \subseteq f(f(x))$. A function space is **fast** if all of the functions in the space are fast.

A function space $F$ is an **information propagation space** if it is monotonic and it is closed under composition and intersection.

¹Note that if function application distributes over intersection, $F$ is monotonic. Some authors denote a framework where function application distributes over meet with the title a "distributive" framework.
Graham and Wegman describe a data flow algorithm which runs in nearly linear time on a large class of problems. They pose their algorithm on an underlying framework which they call an information propagation problem. An information propagation problem is a tuple \( IP = (G, F, X, M) \), where

- \( G = (N, E, s) \) is a flow graph.
- \( F \) is an information propagation space.
- \( X \) is a set, the domain of \( f \in F \).
- \( M \) is a mapping \( E \rightarrow F \).

There is a class of problems which the Graham-Wegman algorithm solves in nearly linear time. These problems are characterized by the fact that their information propagation spaces are fast. Such problems are termed fast problems.

**A Generalized Fastness Condition**

The formulation given by Graham and Wegman studiously avoids the terminology of lattice theory. Unfortunately, their definitions only include problems based on intersection. Both of our problems are based on union. To show that our two frameworks are not fast problems, we need a more general statement of the fastness condition.

An information propagation problem, as defined by Graham and Wegman, can be viewed as a lattice \( L_{GW} = (2^X, \lor, \land) \), where \( \lor = \cup \) and \( \land = \cap \). As shown in Appendix A, both intersection and union are idempotent, associative, and commutative. There is a partial ordering defined by the lattice,

\[
\quad a \geq b \iff a \land b = b \iff a \geq b
\]

Thus, in \( L_{GW} \), \( \geq \) is equivalent to \( \geq \), and \( \leq \) is equivalent to \( \leq \).
In this framework, an information propagation space is a function space which is monotone, closed under meet, and closed under composition. An information propagation space is fast if and only if every function meets the fastness condition. Using the lattice notation, we can restate the fastness condition as:

\[ \forall X \in L, f(X) \wedge X \leq f(f(X)). \]

In this form the condition is suitable for application to the lattices for the aliasing and summary information problems.

The Aliasing Lattice

The aliasing semi-lattice is:

\[ L_A = (2^{AP}, \wedge) \]

where meet (\( \wedge \)) is set union (\( \cup \))

\[ \bot = AP \text{ and } \top = \emptyset \]

\( \geq \) is equivalent to \( \leq \)

\( \leq \) is equivalent to \( \geq \)

For \( L_A \), the Graham-Wegman fastness condition becomes:

\[ \forall X \in L_A, f(X) \cup X \geq f(f(X)) \]

There are functions in \( F_A \) where this condition does not hold.

Demonstrating this requires substituting the function characterization into the fastness condition.

\[ f(X) \cup X = A \cup g\{X\} \cup X \]

\[ f(X) \cup X = A \cup g\{X\} \cup X \]

\[ f(f(X)) = A \cup g\{A \cup g\{X\}\} \]

\[ f(f(X)) = A \cup g\{A \cup g\{X\}\} \]
Substituting these expansions into the fastness condition produces:

\[ A \cup g\{X\} \cup X \supseteq A \cup g\{A\} \cup g\{g\{X\}\} \]

Clearly \( A \supseteq A \), so we need not concern ourselves with that term. The problems arise in with the terms \( g\{A\} \) and \( g\{g\{X\}\} \). For \( A \triangleleft \mathbb{V} \), \( \alpha \in A \), \( \beta \notin A \), \( g(\alpha) = \emptyset \), the function is not fast when applied to the top element, \( \emptyset \). Plugging it into the condition yields the following condition:

\[ A \cup \emptyset \cup \emptyset \supseteq A \cup g\{A\} \cup g\{\emptyset\} \]

Since \( \emptyset \notin A \), the condition does not hold.

The \( g\{g\{x\}\} \) term can also cause problems, for a mapping \( g \) which carries an \( \alpha \in X \) into \( \beta \notin X \), where \( g \) carries \( \beta \) into \( \gamma \notin (A \cup g\{X\}) \). \( F_A \) is not fast.

**The MOD Lattice**

The lattice for MOD is:

\[ L_{\text{MOD}} = (2^{\mathbb{V}}, \vee, \wedge) \]

where join (\( \vee \)) is set intersection (\( \cap \))
and meet (\( \wedge \)) is set union (\( \cup \))

\[ 1 = \mathbb{V} \] and \( T = \emptyset \)
\[ \supseteq \text{ is equivalent to } \leq \]
\[ \subseteq \text{ is equivalent to } \geq \]

\( F_{\text{MOD}} \) is the function space consisting of all functions of the form:

\[ f(X) = (A \cap X) \cup B \cup g\{X\} \]

where \( A \) and \( B \) are constant sets, \( A \leq \mathbb{V} \), \( B \leq \mathbb{V} \), and \( g \) is an arbitrary function mapping \( \mathbb{V} \to 2^{\mathbb{V}} \).
For $L_{\text{MOD}}$, the fastness condition becomes:

$$
\forall X \in L_{\text{MOD}}, \ f(X) \cup X \supseteq f(f(X))
$$

There are functions in $F_{\text{MOD}}$ where this condition does not hold, so the framework is not fast.

Demonstrating this requires some algebraic manipulation:

$$
f(X) \cup X = ((\text{An}X) \cup B \cup g\{X\}) \cup X
$$

$$
f(X) \cup X = B \cup X \cup g\{X\}
$$

$$
f(f(X)) = f((\text{An}X) \cup B \cup g\{X\})
$$

$$
f(f(X)) = \text{An}((\text{An}X) \cup B \cup g\{X\}) \cup B \cup g\{f(X)\}
$$

$$
f(f(X)) = (\text{An}X) \cup (\text{An}g\{X\}) \cup B \cup g\{f(X)\}
$$

So, the condition for $L_{\text{MOD}}$ becomes:

$$
B \cup X \cup g\{X\} \supseteq (\text{An}X) \cup (\text{An}g\{X\}) \cup B \cup g\{f(X)\}
$$

Now, it is clear that $X \supseteq (\text{An}X)$, $B \supseteq B$, and $g\{X\} \supseteq (\text{An}g\{X\})$. The problem lies with the term $g\{f(X)\}$. This term breaks down as:

$$
g\{f(X)\} = g\{AnX\} \cup B \cup g\{g\{X\}\}.
$$

The first term of this expansion poses no problem, since it is contained in the $g\{X\}$ term of the left hand side of the inequality. The latter two terms cause the function space to not be fast. If our function space contains a function where

$$
B \cup X \cup g\{X\} \nsubseteq g\{B \cup g\{X\}\}
$$

then it is not fast. If we can choose $A$, $B$, and $X$ such that $\text{An}B = \emptyset$, and $B \notin X \cap B$, and pick a function $g$ such that for some $\beta \in B$, $\beta \notin X$, and $Y \notin (B \cup X \cup g\{X\})$,

$$
g(\beta) = Y,
$$
then the condition does not hold. Similarly, we can construct a $g$ that violates the condition with the $g(g(x))$ term. Let $a = x, \beta \notin (X \cup g), \gamma \notin (B \cup X \cup g(x))$, for some $x$. Then, if $g(a) = \beta$, and $g(\beta) = \gamma, (B \cup X \cup g(x)) \notin g(g(x))$.

The two means of violating the condition are interesting, in that they arise from two distinct sources in the problem. The former situation arises due to IMOD effects combined with reference parameter bindings, while the latter arises due to propagated GMOD effects coupled with reference parameter binding. $F_{MOD}$ is not fast.
BIBLIOGRAPHY

[Abra 70]

[AhoU 79]

[AHU1 74]

[AKPW 83]

[AICo 80]

[AICo 72]

[Alle 74]

[AlSc 74]

[ANSI 81]

[Ball 79]

[Bann 78]
[Bar77]

[BaSa74]

[BiCa76]

[CW82]

[Donn80]

[Ers66]

[Fabr79]

[Feld79]

[Fos76]

[GaJo79]

[GrWe76]

[Greg71]

[Gold83]
A. Goldberg et al., Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

[HaNo82]
A. Habermann, and D. Notkin, "The Gandalf Software Development

[Harr 79]

[Bech 77]

[Henn 81]

[Holb 82]

[KaU] 76]

[KaU 77]

[Kenn 81]

[Kild 73]

[Knut 71]

[Masi 80]

[MeLo 69]

[Mill 78]

[Moru 71]

[Myer 81]
E. Myers, "A Precise Interprocedural Data Flow Algorithm", 

[Paul 82]

[Reps 82]

[Rose 79]

[Sc Ko 80]

[Sche 77]

[ShPr 81]

[Spil 71]

[Tarj 79]

[Tarj 81a]

[Tarj]

[Thab 81]

[TiRe 80]
[Wied 79]

[Weih 80]

[Zade 82]
F. Zadeck, Research Proposal, Department of Mathematical Sciences, Rice University, Oct. 1982.

Abbreviations:
CACM Communications of the Association for Computing Machinery
JACM Journal of the Association for Computing Machinery
POPL SIGPLAN Symposium on the Principles of Programming Languages