The University of Calgary

Code Improvement Through Flow Analysis

by

Keith O. M. Andrews

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

Department of Computer Science

Calgary, Alberta

December, 1987

ISBN 0-315-42483-4

The University of Calgary

Faculty of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Code Improvement Through Flow Analysis" submitted by Keith O. M. Andrews in partial fulfillment of the requirements for the degree Master of Science.

Prof. Harry D. Baecker, Supervisor
Department of Computer Science

Dr. Michael R. Williams
Department of Computer Science

Dr. John W. Heintz
Department of Philosophy

December 21th, 1987

Abstract

Code improvement techniques are used to overcome the inherent inefficiencies present in automatic, machine based code generation. One of these techniques involves the examination of flow patterns within the program to determine where such problems are occurring and to derive suitable corrections.

This thesis describes a code improvement system called GO (for *G*lobal *O*ptimizer which applies the techniques of flow analysis to machine executable object modules. The analysis is performed without benefit of a symbol table, linkage map or any other form of assistance from the compilation system.

Presented are details of the implementation of the GO control flow and data flow analysis subsystems as well as discussions on how the information gathered by these procedures may be used to effect improvements in machine generated code.

Acknowledgements

No product of human endeavour is the result of the efforts of a single person and this thesis is in no way an exception. I would like to express my gratitude to the following people for their contributions.

First and foremost, I am deeply indebted to Professor Harry Baecker, my supervisor, for his excellent counsel and guidance. Equally deserving of gratitude is Dr. Mike Williams, whose extremely well timed "pep talk" got me over the "hump".

My thanks to Dave Mason and Terry Heintz, my coworkers, for their understanding when things were bad, for their encouragement and especially for the talent they seemed to develop for not noticing that I was on occasion not doing any "real work".

My appreciation to Barry Asaph, Bonnie Bray, Dolores Clarkson, Darcy Grant and Steve Jenuth for their concern and interest.

Dave, Dolores and Steve also contributed their valuable time in proofreading this thesis.

Finally, a special mention of Clarence Comsky, for he, of all the creatures in the universe, seems to be the only one never to have asked me *"how's your thesis going?"*!

Keith Andrews

Calgary, December 1987

To my parents,

For without their love
and encouragement,
this thesis would have never been.

Table of Contents

List of Figures

Preface

This thesis describes a code improver called "GO" (for *G*lobal *O*ptimizer) which is currently under development. GO is being implemented in the C programming language and is to run on the DEC VAX 11/780* running the UNIX† operating system. Though it should work with code generated by compilers for almost any language, the improver is designed to accept code generated from C.

In consideration of these facts, all examples of "real" code (as opposed to pseudo-code) given in this thesis will be in C or VAX assembler. For the reader who is unfamiliar with the VAX assembler language, a brief overview is given in the Appendix (page 102).

The reader is also referred to [20] for a description of the C programming language and to [14] and [25] for more detailed information on the VAX assembly language.

*DEC and VAX are trademarks of Digital Equipment Corporation
†UNIX is a registered trademark of AT&T Bell Laboratories

Chapter 1

Introduction

Computer Science is a discipline obsessed with the concept of efficiency. This obsession is a holdover from the infancy of the field, when computers were slow, expensive, unreliable and somewhat limited in their capabilities. Under such constraints a fast (time efficient) and compact (space efficient) program was an absolute necessity.

In more recent times, computers have improved greatly in all areas. The most precious commodity in the computing industry is no longer the computer but its programmer, or more correctly, the programmer's *time*.

It was because of this desire to make the most productive use of this resource that so-called "high level languages" and their associated compilers were developed. Much of the time spent when programming in machine or assembler language is devoted to translating a small number of relatively straight-forward ideas into the elementary and highly repetitive instructions required by a computer. Such tasks as these are the computer's forte and automation of them is an obvious solution to this problem.

High-level languages, then, permit the programmer to compose a rough outline of the solution to a problem and have the computer supply the details of its implementation. The programmer's time may therefore be addressed to more meaningful tasks.

Machine generated code, unfortunately, is usually of poor quality. Why is the quality of machine produced code so low? How can the process be altered to achieve better quality? To answer these questions an examination of how compilers operate is required.

As stated above, the purpose of a high-level language compiler is to translate a given source program into a series of machine instructions. This is conventionally done in two stages. The first phase, *lexical analysis*, involves the partitioning of the source into a series of basic expressions. This is followed by a *code generation* phase in which each of the sequences detected in the first step is matched with a predefined sequence of machine instructions that performs the same computation. The result is a translation of the original input program into machine executable code.

This technique has two desirable traits: it is fast and it is easy to implement. Unfortunately, it also has a major flaw. Because each input expression is treated in isolation, the compiler cannot detect and capitalize on trends that are caused by interactions between segments as a human coder would. The result is poor code.

To solve this problem it is necessary to introduce a *code improvement* or *optimization* phase into the compilation process.

Code improvers (optimizers) operate by manipulating a given program to produce a new program that is equivalent in every respect except that the latter exhibits more desirable characteristics in some facet of its execution (i.e. the original program is translated into a new program that works "better"). There are many aspects of a program that may be improved. These include the amount

of time the program takes in execution, the amount of memory space a program requires and how it interacts with the operating system. The decision on what area should be enhanced is up to the programmer and is usually based on the environment in which the program is to run.

1.1 Code Improvement: An Overview

1.1.1 Methods of Improving Code

The first question that arises in the design of a code improvement system is *how should it go about performing it's task*? In [3], Aho and Ullman point out that the greatest source of program enhancement is achieved through the selection of the appropriate algorithm for the task being undertaken. They give as an example of this the reduction in the amount of computation required when a quick sort is used in lieu of a bubble sort ($O(n \log_2 n)$ as compared to $O(n^2)$). Unfortunately, this type of code improvement must be done by the programmer as it is beyond the scope of most automated systems.

Another important method of improving code—one that may be performed under program control—is the proper allocation and use of CPU registers. Through careful manipulation of a computation, it is possible to reduce the number of memory references by keeping intermediate results in high speed CPU registers. Since the access time for memory is usually several times slower that that of registers, the resultant savings in execution time can be great. Memory reference instructions also tend to be "longer" (require more storage to represent) than corresponding register reference instructions, thus one sees a decrease in program size as well.

A third source of code improvement is peephole optimization. An early paper on the subject by McKeeman [24] describes this technique as "... a method which consists of a local inspection ... to identify and modify inefficient sequences ...". In actuality, the term peephole optimization refers to a broad class of heuristics which all use the tactic of "local inspection". Some of these are listed here.

- Constant Folding

  This entails locating all constant values in a computation and coalescing them into a single term. For example, given the expression

  ```
  x + 2 * 3 - 1
  ```

  an application of constant folding would result in

  ```
  x + 5
  ```

- Redundant Instruction Elimination

  Typically, this involves finding sequences that result in no real change in the program environment. An example of such a situation is as follows. Given the source code:

  ```
  x = y + 1;
  z = x + 1;
  ```

  It is plausible that the following series of machine instructions would be generated:

  ```
  movw   y,r0    # temp ← y
  incw   r0      # temp ← temp + 1
  movw   r0,x    # x ← temp
  movw   x,r0    # temp ← x
  incw   r0      # temp ← temp + 1
  movw   r0,z    # z ← temp
  ```

Here, the fourth instruction is redundant as the value of x is already in the temporary register, thus the instruction may be safely eliminated.

- Instruction Upgrading

  Often a compiler will use only a small fraction of the instructions in a machine's repertoire. This is because some instructions are highly specialized or their functions span what would be several expressions in the source program (this is a classic example of the deficiencies of the "standard" compiler described previously). Instruction upgrading is the process by which the code improver attempts to locate sequences of instructions that may be replaced by a series of more specialized instructions which will execute in less time or occupy less space. (one instruction sequence is "upgraded" to a better one). Consider the following code fragment:

```
do {
        ⋮
    x = x + 1;
} while (x < 10);
```

Compiled on a VAX, the code generated for this may look something like:

```
loop:               # Top of loop
        ⋮           # Loop body
    incw    x       # x ← x + 1
    cmpw    x,$10   # Compare value of x to limit
    blt     loop    # If not at limit, branch to top of loop
```

However, the VAX, like many modern computers, has a loop instruction of the form "add a value to variable, compare result to a second value and branch if a certain condition is met". Thus, the entire code fragment given above could be replaced by:

```
loop:
        ⋮
    acbw $10,$1,x,loop # x ← x + 1,
                       # if x < 10 branch to loop
```

- Jump to Jump Elimination

  Since statements and sometimes parts of statements are treated separately in the compiler described above, it is almost certain that the situation will arise where an instruction causes a transfer of control to another control transfer instruction. All that is involved in the application of this heuristic is to locate such instances and have the first instruction simply refer to the final destination, thereby eliminating the intermediate step.

These examples show that the effect of the peephole optimizer is to "blur" the sharp boundaries between code segments caused by the conventional code generation process. This technique does little to capitalize on the trends that take place over more than a few instructions so the improvements are highly localized. To exploit the trends which take place over larger sections of a program, it is necessary to use another class of code improvement techniques: *global optimizations.*

Global optimizers use a data gathering technique known as *flow analysis* to determine how a program will behave when it is run. The flow analysis process is generally performed in two stages. The first part is the analysis of flow of control. In this phase, a series of graphs are generated that illustrate how the various parts of the program interact (i.e. for any given point in the program, the graphs indicate which points may next gain control). The second step is to use these graphs to determine how information is processed in the program. This

involves determining how variables are used or modified and where these uses or modifications occur. The result of this analysis is a graph of how data flows within the program.

The control and data flow graphs form one of many different representations of a program (the original source program and the compiler's output are two others). In this form, however, the structure of the program and the interactions of its components become more evident. The improvement of code is performed by looking for specific patterns in these graphs and applying some straight forward transformations. The procedures for performing the various analyses and the methods for using the results obtained to achieve improvements in code will be discussed in subsequent chapters.

1.1.2 When to Improve Code

The next decision to be made in the design of a code improver is at what point in the compilation process the code improvement phase should be done. The various stages of the compilation process are outlined in Figure 1.1. As shown, code improvement can be performed at any of three points:

- by manipulating the source code before lexical analysis,
- by altering the intermediate representation of the code as output by the lexical analyzer,
- by transforming the machine executable program image which is the final product of the compiler.

Performing code improvement on the input program before submission to the compiler (i.e. on the source code) has been proposed by Loveman [22] and by

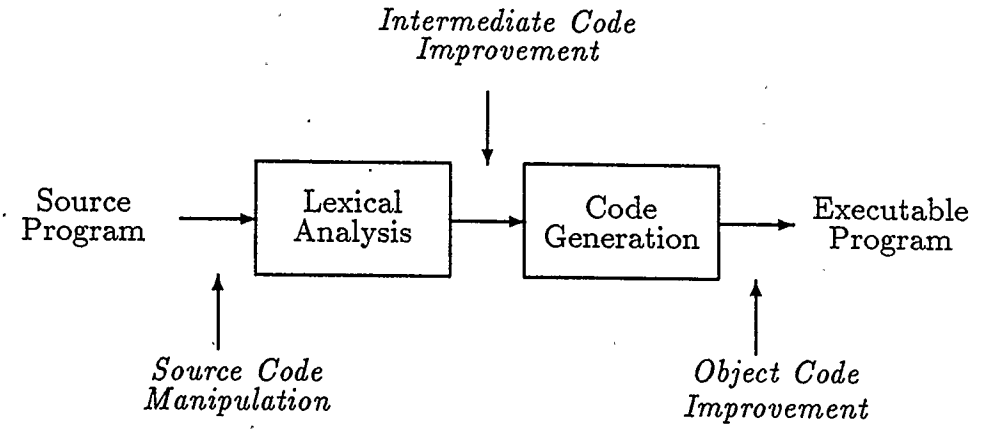


Figure 1.1: Stages of compilation

Standish et al.[27]. Systems using this technique take the programmer's code and generate a new source program for input to the compiler.

Both the Loveman and Standish systems use databases of transformation rules. These rules specify how constructs commonly found in programs may be translated into code and data structures that are converted into more efficient sequences of machine instructions. A simple example of a transformation rule (paraphrased from [27]) is given in Figure 1.2.

The result of the application of this transformation is the elimination of a superfluous *if* statement.

It should be pointed out, however, that an experienced programmer (one who understands the nuances of the target machine and the compiler used) will tend to apply transformations such as these when first coding the program. Therefore, code improvement techniques of this form are best used to correct deficiencies in

Provided $j < b$, a statement of the form:

```
for (k = a; k < b; k = k + 1)
    if (k < j)
        perform operation X
    else
        perform operation Y
```

may be translated into:

```
for (k = a; k < j; k = k + 1)
    perform operation X
for ( /* k = j */; k < b; k = k + 1)
    perform operation Y
```

Figure 1.2: An example source manipulation translation rule

the coding style of the programmer as they do nothing to enhance the quality of code produced by the compiler.

Another advocate of manipulation at this level is Rosen[26] who does data flow analysis on the original source code. This is necessary, Rosen argues, because there is too much loss of information when going from the source level to that of the intermediate code. It is interesting to note that these are the same reasons cited by advocates of intermediate code manipulation over object code manipulation.

Intermediate code manipulation is a much more accepted form of code improvement. This entails the rewriting of the intermediate code generated by the first pass of the compiler using one or more of the methods described previously. There are a several advantages to improving code at this stage:

- The source program has already been broken up into its constituent parts and checked for syntactic correctness.

- The information is complete with regard to the program elements under scrutiny. That is, the intermediate code contains all the information given in the original source (variable types, form of data and control structures, etc.), but in a much more convenient form.

- In a well designed compiler, the intermediate code would be in a target machine independent form. This allows the code improver to be written in a portable fashion.

There are, unfortunately, two major problems associated with performing code improvements at this stage. First, although complete information is available for the section of the program being compiled, there will be no information about the rest of the program. This is most evident if the program has been broken up into many separately compiled modules (as in most large programs) or if extensive use is made of library routines. Lack of this information can be crippling to a code improver using flow analysis as the improver must assume a worst case scenario in such instances (i.e. the missing code modifies all variables and invokes all the routines in the module being compiled). The second flaw with this technique is that improvement of the intermediate code does little to improve the quality of the final output. The code generator will still work with the various sections of the intermediate code in isolation.

Manipulation of the output from the compiler (the object code) is also a popular form of code improvement[13]. Not surprisingly, the areas in which this technique are good are precisely those in which intermediate code improvement

fails. Those facets in which object code improvement is poor are the ones in which intermediate code improvement excel:

- The machine code must be "disassembled" by the improver and encoded into an intermediate form for processing.
- The complete program is available to the object code improver, not just a single module. However, most of the details about variable types and data structures have been lost.
- Although some of the major elements of the code improver can be written in a machine independent way, much of it must be machine specific.
- As this represents the final stage in the generation of the target program, the improver directly controls the quality of code produced.

1.2 The "GO" Code Improver

1.2.1 Design Issues

A survey of the literature shows that, for the most part, global optimizations are done on the intermediate representation and peephole optimizations are performed on the object code. There seem to be no "production" systems which utilize source code manipulations of any form. This may be due to in large part to tradition (code improvement is *always* done on intermediate or object code) but a more likely answer is that such transformations are too far removed from the code generation phase to have a significant effect on code quality.

The timing of the peephole code improvements is primarily because such operations as instruction upgrading and jump-to-jump elimination may only be per-

formed at this stage. The reasons for performing global improvements at the intermediate code stage (instead of after code generation) are less clear.

There are two main arguments for doing flow analysis and the subsequent code improvements between the lexical analyzer and the code generator. First, there is the view that much relevant information is lost in the code generation process and without this data, it is not possible to build the flow graphs correctly [18]. The second point is closely related: it is not possible to distinguish instructions from data in the object program. Both of these objections are unfounded.

The key to understanding this assertion is the realization that the original source program, the intermediate code and the object code are merely representations of the same algorithm. Furthermore, each form details the implementation in precisely the same manner as the others (an analogy to this is the relationship between the numbers 74_{10}, 112_8, $4A_{16}$ and the ASCII letter 'J'). Each form contains the same information as the others but the details of the representation of this information vary. Given that the information is present, it should be possible to extract it and build the appropriate flow graphs.

The point that it is impossible to differentiate instructions from data is clearly false as evidenced by the fact the computer *must* differentiate between the two as it runs the program. The technique for distinguishing between the two is to apply a rule that all assembler programmers learn at the beginning of their careers: if the computer attempts to execute it, it is an instruction.

It is theoretically possible, then, to perform global optimizations on the final output of a compiler. One of the principle areas of investigation in the design and implementation of GO is to see if this theory can be put into practice. Us-

ing only the information available to the computer when it executes a program (the executable binary image of the program itself) GO will determine what the control and data flow patterns within the program are. It will then attempt to "rearrange" the code in order to improve the programs execution characteristics. (An explanation as to *why* one would want to perform global optimizations at this level will be presented in the concluding chapter.)

Another key issue in the design of a code improver is machine independence. The implementation of a code improver is not a trivial task. It is therefore useful to design the code improver so that it may be easily "ported" to different systems. Unfortunately, when dealing with programs at the machine code level, complete machine independence is an impossibility. Nevertheless, by careful partitioning of the tasks in the code improver, it should be possible to isolate most machine dependencies to a few small modules. This is done in GO.

1.2.2 Implementation of GO

GO is implemented as a series of relatively independent modules as depicted in Figure 1.3. The first four modules, the instruction decoder, the data flow preprocessor, the basic block generator and the call graph generator work in concert to produce a usable representation of the target program. The basic block and call graph generators also form the first part of the control flow analysis subsystem. The second part of the control flow analyzer is the interval generation code. The information generated by control flow analysis and by the data flow preprocessor is used in the data flow analysis subsystem. The actual code improvement is performed by a series of modules which utilize the information gathered by the

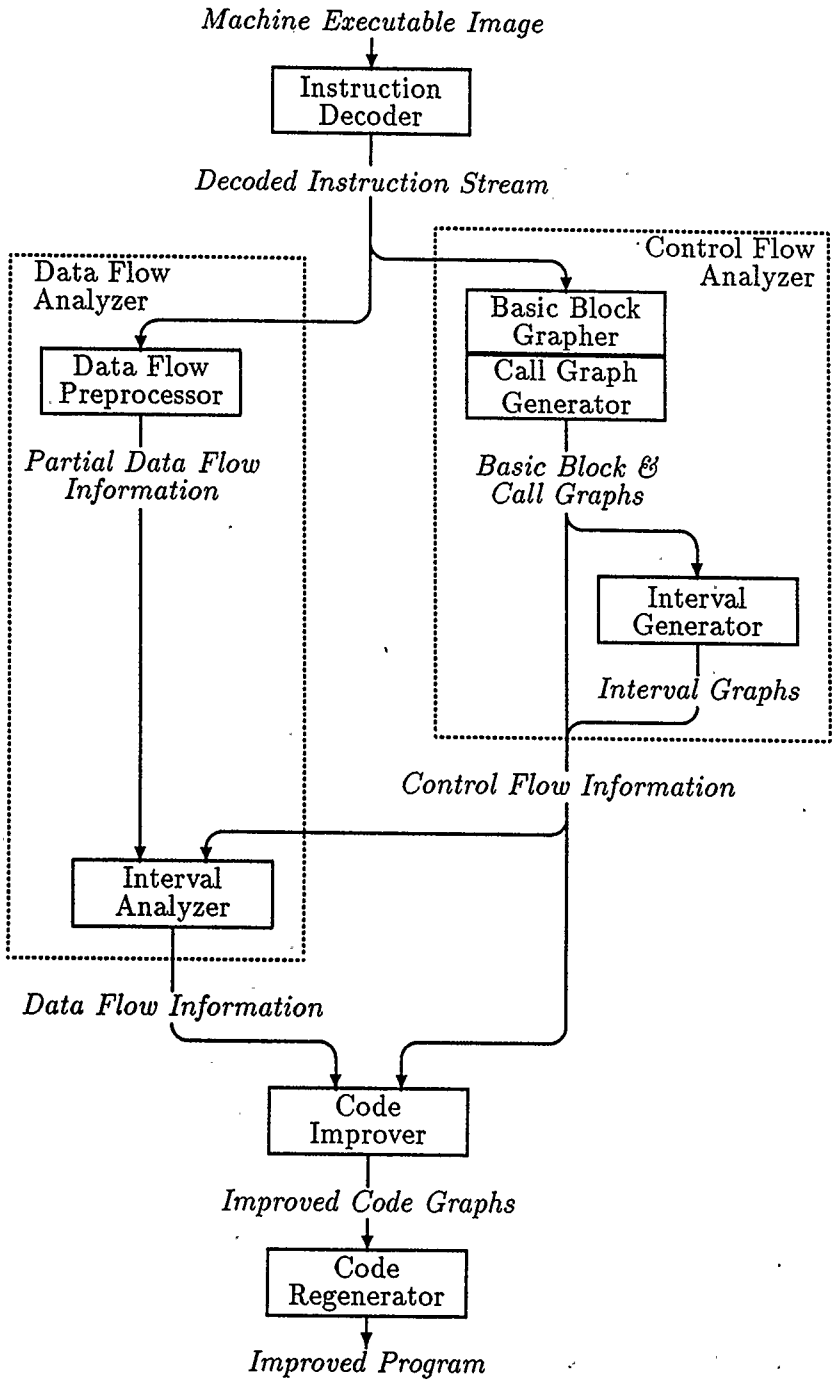


Figure 1.3: Structure of the GO code improver

various analysis phases to identify inefficiencies. Collectively, these modules form the code improvement subsystem. Finally, the output of the code improver is passed to the cleanup module which recodes GO's internal representation of the program into a new machine executable object image.

To achieve machine independence, most of the operations which require a knowledge of the machine architecture are isolated in two modules. These are the instruction decoder and the cleanup module. Machine dependencies are further reduced by performing the operations in these modules in a table driven manner. To do this, a table is generated which enumerates each of the target machines instructions. Each table entry details the basic information about the instruction: number and type of operands, instruction opcode, type of instruction (e.g. test, branch, arithmetic), and so forth. Each entry contains sufficient information for the module to identify and usually completely process the instruction.

1.3 Thesis Overview

The rest of this thesis is an in-depth look at the operation of the various components of the GO system. The discussion starts off in the next chapter with a description of the control flow analysis code. This is followed by details of the operation of the data flow analyzer in chapter three. The techniques for doing the actual code improvement are outlined in the fourth chapter and the concluding arguments are presented in the final chapter.

Chapter 2

Control Flow Analysis

2.1 Overview of the Control Flow Analyzer

2.1.1 Function

The GO control flow analyzer performs several tasks:

- It examines the input program and differentiates the instructions from the data.
- It identifies the subroutines that make up the program and determines how the subroutines are organized with respect to each other.
- It determines the basic flow patterns within each subroutine.
- It identifies the loops within each subroutine.

These tasks are performed in two stages. The first three tasks are dealt with in the first stage and the loop identification is performed in the second stage.

2.1.2 Input

The GO control flow analyzer takes a complete executable program image as input. For the most part, this is all that is required for a complete flow analysis.

2.1.3 Output

The output of the control flow analyzer can be thought of as a hierarchy of graphs, each of which details the flow of control at some level of the program. At

the highest level is the graph of subroutine calls (the *call graph*). This structure serves to outline the operation of the program as a whole. Next comes a series of *interval* graphs that show where the loops are located in each subroutine. The highest level (n^{th} order) interval graph consists of a single node and corresponds to a complete subroutine (i.e. a node in the call graph). The lowest level (1^{st} order) interval graph gives the locations of the innermost loops of the subroutine. The lowest level of the hierarchy is the basic block graph. It details the basic flow patterns within each subroutine.

2.2 Separating Instructions and Data Areas

An executable program image contains at least three types of information. These are a sequence of machine instructions for the computer, data areas which are manipulated by the computer in the course of interpreting the instructions and environmental information.

Environmental information comprises information about the context in which the program is to be run such as memory requirements and program entry point. This data is usually interpreted by the operating system when loading the program. It may not be explicitly present in the program image but must be inferred from it. For example, if the initial memory allocation is not given explicitly, it may be safely assumed that a region of memory equal in size to the image being loaded is required.

If present, the environmental information is usually of a fixed size and in a fixed location in the program image. Unfortunately, this is not true of the instruction

and data regions of the program. Furthermore, it is probable that instructions and data are intermingled throughout the executable image.

The first major task required of the control flow analyzer is to differentiate between the instruction and data regions of the program. To accomplish this, each instruction in the program must be examined in turn starting with the instruction at the program entry point (the location of this instruction is determined from the environmental information). The effect of this process is to "trace" the instructions in the program.

By noting regions of the image into which control is transferred and then recursively examining each of these regions, it is possible to determine which regions of program could possibly be executed. By definition, these are the instruction regions of the program (if the computer attempts to execute it, it's an instruction) and all other areas are data regions

2.3 Call Graph Generation

The process of tracing the instructions within a program will also permit the identification of the subroutines within it and the structure of the subroutines. GO relies on two properties of subroutines to assist in the task of identifying them:

1. At some point in the program, the subroutine must be invoked via a standard subroutine invocation sequence. A "standard" call sequence is defined either by the machine architecture or by language conventions.
2. There are a finite number of paths through the subroutine. Each of these paths begins at a common entry point and ends with a standard subroutine termination sequence.

Algorithm 2.1: GraphSubrCalls

PURPOSE: To generate the subroutine call graph for a program.

INPUT: The address of the start of the main procedure.

OUTPUT: The graph of all subroutine calls.

METHOD: Recursive function as outlined in Figure 2.1.

As shall be shown shortly, GO makes active use of the first property to identify the subroutine entry points and build the call graph. The use of the second property, however, results as a side effect of the analyses of the subroutines' internal structures.

The method GO uses to generate the call graph is outlined in Algorithm 2.1. Informally, the call graph generator operates as follows. If the subroutine under scrutiny has been encountered previously, the call graph node corresponding to the subroutine is located and returned to the calling routine. Otherwise a new call graph node is created and added to the call graph. The instruction stream is then scanned until a subroutine call sequence is encountered (this procedure is outlined below). At this point the call graph generator invokes itself recursively with the address of the called subroutine. The recursive routine repeats the scanning process, except in this instance, the instruction stream is that of the new subroutine. When the end of the subroutine is encountered the recursive call terminates. Upon regaining control the initial instance of the graph generator creates a link in the call graph from the node representing the subroutine under

```
recursive function GraphSubrCalls(integer Addr) : call_graph_node
    call_graph_node ThisNode
    call_graph_node NextNode
    if call graph node for subroutine at Addr exists then
        ThisNode := call graph node for subroutine at Addr
    else
        ThisNode := new call_graph_node
        add ThisNode to the call graph
        while not at the end of subroutine do
            look for a subroutine call sequence
            NextNode := GraphSubrCalls(address of called subroutine)
            add graph arc (ThisNode, NextNode) to call graph
        endwhile
    endif
    return ThisNode
endfunction
```

Figure 2.1: Function to graph subroutine calls

scrutiny to that of the subroutine that has just been processed. The net output from this process is a graph detailing the overall structure of the program. A sample program and its call graph are depicted in Figure 2.2.

2.4 Basic Block Graph Generation

The generation of the basic block graph proceeds in conjunction with the generation of the call graph. In addition to revealing the fine details of flow control within each subroutine, this procedure has two important side effects. First, it performs the instruction/data differentiation and second, it locates the subroutine call sequences for the call graph generator.

```
main()
{
        subr1();
        subr2();
        subr3();
}

subr1()
{
        subr4();
        subr5();
}

subr2()
{
        subr5();
}
```

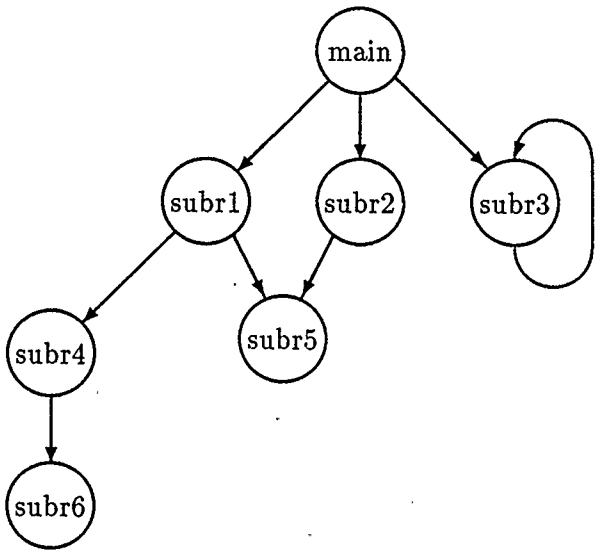
```
subr3()
{
        subr3();
}

subr4()
{
        subr6();
}

subr5()
{
        ⋮
}

subr6()
{
        ⋮
}
```

(a)



(b)

Figure 2.2: A program (a) and its call graph (b)

2.4.1 Identifying Basic Blocks

A basic block is a contiguous sequence of instructions in which execution proceeds linearly from lowest address to highest address and where the instruction at the lowest address is always the first executed.

In [3], an algorithm is introduced which determines basic blocks in compiler intermediate code. This is a two stage process. The first stage is the identification of all *leaders*. A leader is either the first statement, the target of a "goto" (i.e. a control transfer) statement or the statement immediately following a conditional goto statement. The second stage is to divide the code into basic blocks which are delineated by leaders, each section begins with a leader and ends with the statement just prior to the next leader. When dealing with intermediate code, these tasks are straightforward. To accomplish it, one makes a single pass through the code noting the positions of all goto statements and all instructions with labels associated. Unfortunately, when dealing with pure binaries as in GO, the problem becomes more complex.

The difficulty with binary data is that there are no labels conveniently pointing out targets of branch instructions. What appears to be a branch instruction may simply be an instruction's operand. As stated previously, GO surmounts this problem by "tracing" the instructions much the same as the computer would do when it executes the program. This process is actually quite simple and is outlined in Algorithm 2.2.

It should be noted that this algorithm does not require knowledge of what the input to the program will be, nor is the issue of whether the program under scrutiny will ever terminate a factor. This is because what is being looked for

Algorithm 2.2: FindBBlocks

PURPOSE: To partition an instruction sequence into basic blocks.

INPUT: The address of the instruction at which to start partitioning,

OUTPUT: The instruction stream is broken up into a series of basic blocks.

METHOD: Recursive procedure as outlined in Figure 2.3.

are *potential* control transfer points. The question of whether a transfer of control occurs at any given point actually happens when the program is run is not important. What is important is that a transfer of control *may* occur.

As an example of the operation of this algorithm, consider the code to compute Ackermann's function as given in Figures 2.4, 2.5 and 2.6. FindBBlocks would be invoked with the address of the function ack (00) as its parameter. As no instructions have yet been processed, the procedure would create a new basic block (block 1) and begin examining instructions. The initial instruction, a test, does not cause a transfer of control, so it is simply added to the current basic block. The second instruction is also added to the current basic block. As it is a branch, it denotes the end of the current block. The instruction is a conditional transfer, so two recursive calls are made (Figure 2.7a).

In the first recursive call, the address passed is that of the instruction following the branch (05). As this instruction has never been processed, a second basic block is created.

```
recursive procedure FindBBlocks(integer Addr)
    integer NextAddr
    instruction CurInstr
    basic_block CurBBlock

    if Addr is in an existing block then
        if Addr is not the first address of that block then
            split block containing Addr in two at Addr
        endif
    else
        CurBBlock := new basic_block
        NextAddr := Addr
        repeat
            CurInstr := instruction at NextAddr
            add CurInstr to CurBBlock
            NextAddr := address of next instruction
            if CurInstr is a branch then
                if CurInstr is conditional then
                    call FindBBlocks(NextAddr)
                endif
                call FindBBlocks(branch address)
            endif
        until CurInstr is a return instruction ∨
            CurInstr is a branch instruction ∨
            NextAddr is in an existing block
    endif
endprocedure
```

Figure 2.3: Procedure to find basic blocks

$$Ack(n,y) = \begin{cases} y+1 & \text{if } n = 0 \\ Ack(n-1,1) & \text{if } y = 0 \\ Ack(n-1, Ack(n, y-1)) & \text{otherwise} \end{cases}$$

Figure 2.4: Simplified Ackermann's function (from [11])

Processing of this block proceeds normally until the return instruction (address 0A) is encountered. As a return instruction is one of the three conditions which signal the end of a basic block, the repeat loop terminates and the recursive call on FindBBlocks subsequently returns.

The third basic block is generated in the second recursive call which starts processing at the branch target address, 0B (Figure 2.7b). In this instance, processing proceeds until the branch at address 0E is seen. Again, this is a conditional branch, so two more recursive calls are made. The first of the second level re-

```
ack(n, y)
int n;
int y;
{
     if (n == 0)
          return y + 1;
     else if (y == 0)
          return ack(n - 1, 1);
     else
          return ack(n - 1, ack(n, y - 1));
}
```

Figure 2.5: Implementation of Simplified Ackermann's function in C

```
ack:    tstl    4(ap)           # test n = 0
        jneq    ack1            # branch if not
        addl3   $1,8(ap),r0     # return value ← y + 1
        ret                     # exit subroutine
ack1:   tstl    8(ap)           # test y = 0
        jneq    ack3            # branch if not
        pushl   $1              # stack ← 1 (2nd recursive call param.)
ack2:   subl3   $1,4(ap),-(sp)  # stack ← n − 1 (1st parameter)
        calls   $2,ack          # recursive call
        ret                     # exit subroutine
ack3:   subl3   $1,8(ap),-(sp)  # stack ← y − 1 (2nd parameter)
        pushl   4(ap)           # stack ← n (1st parameter)
        calls   $2,ack          # first recursive call
        pushl   r0              # stack ← ack(n, y − 1) (2nd param.)
        jbr     ack2            # do second recursive call
```

Figure 2.6: Ackermann's function in assembler

cursive calls generates the fourth basic block. It starts processing at address 10 and terminates with the return at address 1E. The other recursive call starts processing at address 1F and locates the boundaries of the fifth basic block in the function (Figure 2.7c).

Unlike the previous blocks, block five ends with an unconditional branch. Therefore there is only a single recursive call made. The start address in this call is 12. In this instance, the first target instruction has already been processed. Furthermore, this instruction is located in the middle of an existing block. In this situation, no further processing of instructions is performed. Instead, the basic block containing the target instruction is split into two new basic blocks. The first contains all of the instruction from the start of the original block to the

```
00 ack:   tstl    4(ap)
03        jneq    ack
                                  } Block 1

05        addl3   $1,8(ap),r0 [1]
0A        ret
0B ack1:  tstl    8(ap)
0E        jneq    ack3
10        pushl   $1
12 ack2:  subl3   $1,4(ap),-(sp)
17        calls   $2,ack
1E        ret
1F ack3:  subl3   $1,8(ap),-(sp)
24        pushl   4(ap)
27        calls   $2,ack
2E        pushl   r0
30        jbr     ack2
                                  } Unprocessed
```

[N] indicates the position of CurInstr for the N^{th} recursive instance of FindBBlocks. N is 0 for the base level instance.

Current state: Processing of block 1 complete; processing of block 2 about to commence.

(a)

Figure 2.7: Partitioning Ackermann's function into basic blocks

... continued on page 28

```
00 ack:   tstl    4(ap)
03        jneq    ack1 [0]
```
Block 1

```
05        addl3   $1,8(ap),r0
0A        ret
```
Block 2

```
0B ack1:  tstl    8(ap) [1]
0E        jneq    ack3
10        pushl   $1
12 ack2:  subl3   $1,4(ap),-(sp)
17        calls   $2,ack
1E        ret
1F ack3:  subl3   $1,8(ap),-(sp)
24        pushl   4(ap)
27        calls   $2,ack
2E        pushl   r0
30        jbr     ack2
```
Unprocessed

[N] indicates the position of CurInstr for the N^{th} recursive instance of FindBBlocks. N is 0 for the base level instance.

Current state: Processing of blocks 1 & 2 completed; processing of block 3 about to commence.

(b)

Figure 2.7 continued

... continued on page 29

```
00 ack:   tstl    4(ap)
03        jneq    ack1 [0]
```
Block 1

```
05        addl3   $1,8(ap),r0
0A        ret
```
Block 2

```
0B ack1:  tstl    8(ap)
0E        jneq    ack3 [1]
```
Block 3

```
10        pushl   $1
12 ack2:  subl3   $1,4(ap),-(sp)
17        calls   $2,ack
1E        ret
```
Block 4

```
1F ack3:  subl3   $1,8(ap),-(sp)
24        pushl   4(ap)
27        calls   $2,ack
2E        pushl   r0
30        jbr     ack2 [2]
```
Block 5

[N] indicates the position of CurInstr for the N^{th} recursive instance of FindBBlocks. N is 0 for the base level instance.

Current state: Processing of blocks 1 through 5 completed; 3rd level recursive call with address 12 about to be made.

(c)

Figure 2.7 continued

... continued on page 30

```
00 ack:   tstl    4(ap)
03        jneq    ack1 [0]
                                  } Block 1

05        addl3   $1,8(ap),r0
0A        ret
                                  } Block 2

0B ack1:  tstl    8(ap)
0E        jneq    ack3 [1]
                                  } Block 3

10        pushl   $1
                                  } New block 4

12 ack2:  subl3   $1,4(ap),-(sp) [3]
17        calls   $2,ack
1E        ret
                                  } Block 6

1F ack3:  subl3   $1,8(ap),-(sp)
24        pushl   4(ap)
27        calls   $2,ack
2E        pushl   r0
30        jbr     ack2 [2]
                                  } Block 5
```

[N] indicates the position of CurInstr for the N^{th} recursive instance of FindBBlocks. N is 0 for the base level instance.

New block 4 + Block 6 } Original block 4

Current state: All blocks processed; stack unwind about to start with termination of 3rd recursive call.

(d)

Figure 2.7 continued

instruction preceding the target instruction. The second new block contains the remaining instructions (Figure 2.7d).

At this point there are four instances of FindBBlocks active:

1. the original call, generating block 1 with NextAddr set to 05,
2. the call which generated block 3 with NextAddr set to 10,
3. the call which generated block 5 with NextAddr set to 32,
4. the current call, which has created block 6 by splitting block 4.

The current instance returns immediately after having split up block 4, as no further processing is required. When this happens, the third instance is reactivated. In this case, CurInstr is a branch instruction, which causes the repeat loop to terminate and control to be passed back to the second instance of FindBBlocks. This instance also terminates immediately as the instruction just processed was another branch. The original invocation of FindBBlocks is now the only one active. It too has just processed a branch instruction and terminates.

The entire program has now been scanned and all of its component basic blocks have been identified.

2.4.2 Graphing the Basic Blocks

Each recursive call of FindBBlocks corresponds to an arc in the basic block flow graph. The reader may wish to compare the description of the execution of FindBBlocks given previously with the flow graph for Ackermann's function given in Figure 2.8. The generation of the basic block flow graph requires some simple changes to FindBBlocks as outlined in Algorithm 2.3.

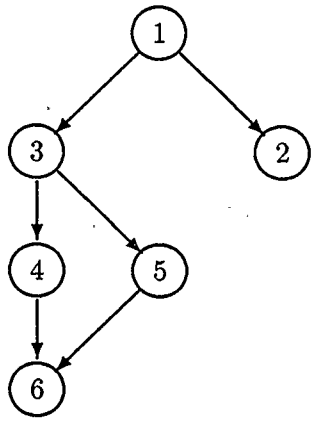


Figure 2.8: Flow graph for Ackermann's function

Algorithm 2.3: GraphBBlocks

PURPOSE: To generate a flow graph of basic blocks in a subroutine.

INPUT: The address of the instruction at which to start partitioning.

OUTPUT: A flow graph of the basic blocks in the subroutine.

METHOD: The procedure of Algorithm 2.2 is changed to a function. The return value of the function is the basic block it generates (NextBBlock). After every recursive call a graph arc is added between the basic block currently being processed (CurrentBBlock) and NextBBlock.

2.5 Call Graphs Revisited

As stated earlier, one of the side effects of the process of graphing the basic blocks was the detection of subroutine call sequences. Because of this, the call graph generation procedure and the basic block graph generation procedure effectively form a pair of coroutines.

The operation of these coroutines is as follows. First, GraphSubrCalls is invoked with the address of the program's entry point, which is considered to be the start of the first subroutine. GraphSubrCalls then invokes GraphBBlocks with the entry point value. Processing of the subroutine proceeds as outlined in the previous section. If a subroutine call sequence is encountered during processing, GraphSubrCalls is invoked recursively with the address of the new subroutine. In turn, GraphSubrCalls invokes a recursive instance of GraphBBlocks and so on. Processing of the program as a whole is complete when the initial instance of GraphSubrCalls regains control.

2.6 Loop Identification

The second phase of control flow analysis in GO is the identification of the loops within individual subroutines. Loop identification is the most important endeavour of control flow analysis as it forms the basis for the subsequent data flow analysis. It also locates local "hot spots" in the program which is of great use in the application of actual code improvement heuristics.

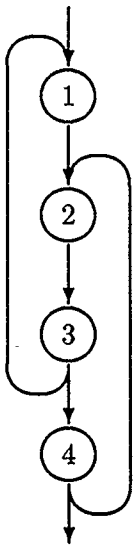


Figure 2.9: Cycles which do not form loops

2.6.1 What is a Loop?

The first task in identifying loops is to define exactly what a loop is. One might be tempted to define a loop as simply a cycle in the flow graph of a subroutine. Unfortunately, this definition is weak as it does not take into consideration the proper nesting of loops. For example, consider the flow graph in Figure 2.9. There are two cycles within the flow graph; the first is defined by the path 1-2-3-1 and the second by the path 2-3-4-2. Neither can be called a loop however due to the fact that the two overlap.

A better model of a loop is given with the *interval* as defined by Cocke and Allen [4,9] which is a refinement of the use of *dominators* in the detection of loops as outlined by Lowry and Medlock [23].

An interval is a portion of a flow graph having the following properties:

1. It is *strongly connected.* That is, there exists a path from every node to every other node.

2. There exists at least one node which is on all paths through the flow graph. Furthermore, this node will be visited before any other node in each iteration of the loop. This node is said to *dominate* all other nodes in the loop and is termed the *head* node of the interval or *interval header.*

3. It is *maximal*—no other interval for the same flow graph contains it as a subset.

A corollary to the second property is that entry to the loop is only via the interval header. This precludes the invalid nesting of loops resulting from the cycle definition.

Hecht [17] gives the following formal definition of an interval:

> The interval with header h, denoted $I(h)$ is the subset of [a flow graph with entry node s] constructed as follows:
>
> $I(h) := \{h\}$
> **while** $\exists$ a node m such that $m \notin I(h) \wedge m \neq s \wedge$ all arcs entering m leave nodes in $I(h)$ **do**
> $\quad I(h) := I(h) \cup \{m\}$
> **endwhile**

In [4], Allen gives an algorithm for the partitioning of a flow graph into intervals. This algorithm is refined and codified by Hecht. Algorithm 2.4 is based on Hecht's procedure.

The algorithm operates by performing a series of breadth first traversals of the flow graph. Each traversal is anchored at a header node. As the first header is defined to be the flow graph entry point node, the ordering of visits is the same

---

Algorithm 2.4: FindIntervals (based on the algorithm in [17])

PURPOSE: To partition a flow graph into intervals.

INPUT: A flow graph.

OUTPUT: A list of intervals.

METHOD: The procedure is given in Figure 2.10. ICurrent, IPending and Headers are lists of graph nodes which represent the current interval, the unprocessed nodes of the current interval and the list of potential headers, respectively. Count[x] is the number of arcs flowing *into* graph node x which have not yet been accounted for. Reach[x] is the header node for the interval containing x.

Note: Hecht specifies that the nodes in IPending and Headers be processed in a first-in-first-out basis. There seems to be no reason for doing this with IPending. Processing Headers in this order, though, results in a breadth first traversal of the graph as opposed to a depth first traversal achieved with last-in-first-out.

---

as the execution order. Once all arcs into a node have been followed, it is added to the current interval. The only case in which all arcs are not followed during the processing of a given interval, is when a *back arc* (i.e. one originating from a node occurring later in the execution order) is present. A back arc to any node save the header in an interval is not possible, so the node must be the header of another interval. Processing terminates when a traversal has been performed from each header.

2.6.2 Interval Ordering

A fundamental property of the interval partitioning algorithm is that it performs a partial topological sort on the nodes contained in each interval. That is, no node is placed in the interval list until all of its predecessors have been placed

---

```
procedure FindIntervals(graph G with n nodes and with start node s)
    list ICurrent
    list IPending
    list Headers
    integer Count[1..n]
    graph_node Reach[1..n]

    foreach node n in G do
        Count[n] := number of arcs entering n
        Reach[n] := undefined
    endforeach
    Reach[s] := s
    Headers := { s }
```

Figure 2.10: Procedure to locate intervals in a flow graph

... continued on page 38

```
    while Headers ≠ ∅ do
        select a node H from Headers
        Headers := Headers − { H }
        IPending := { H }
        while IPending ≠ ∅ do
            select a node H from IPending
            Headers := IPending − { H }
            foreach arc (X, Y) in 𝒢 do
                Count[Y] := Count[Y] − 1
                if Reach[Y] = undefined then
                    Reach[Y] := H
                    if Count[Y] = 0 then
                        IPending := IPending + { Y }
                        ICurrent := ICurrent + { Y }
                    elseif Y ∉ Headers then
                        Headers := Headers + { Y }
                    endif
                elseif Reach[Y] = H ∧ Count[Y] = 0 then
                    IPending := IPending + { Y }
                    ICurrent := ICurrent + { Y }
                    if Y ∈ Headers then
                        Headers := Headers − { Y }
                    endif
                endif
            endforeach
        endwhile
        add ICurrent to list of intervals
    endwhile
endprocedure
```

Figure 2.10 continued

Algorithm 2.5: GraphIntervals

PURPOSE: To generate a graph of intervals from a list of intervals.

INPUT: A flow graph and a list of intervals contained in it.

OUTPUT: An interval flow graph.

METHOD: A function as given in Figure 2.11.

in the list. The only exception is a header node which may have back arcs entering it. In this case, the back arcs are ignored (the header is treated as if it only had a single inflowing arc which originates outside of the interval). This sequence is known as the *interval ordering.* As shall be shown in the next chapter, this property becomes useful in the analysis of data flow.

2.6.3 Interval Graphs

Using the information in the original flow graph and obtained via FindIntervals, it is possible to generate an interval graph. This structure details how the inner most loops of the subroutine are organized. The procedure for generating these graphs is given in Algorithm 2.5. Basically, all that is done is to generate a new graph node for each interval. The arcs flowing out of this node are the collection of arcs flowing out of the original flow graph nodes which comprise the interval.

```
function GraphIntervals(graph G0, list L) : graph
    graph G1
    foreach interval I in L do
        if ∃ a node in G1 for I then
            X1 := the node in G1 representing I
        else
            X1 := new graph_node
            add X1 to G1
        endif
        foreach node X0 contained in I do
            foreach arc (X0, Y0) in G0 do
                J := the interval in L containing Y0
                if J ≠ I then
                    if ∃ a node in G1 for J then
                        Y1 := the node in G1 representing J
                    else
                        Y1 := new graph_node
                        add Y1 to G1
                    endif
                    add an arc (X1, Y1) to G1
                endif
            endforeach
        endforeach
    endforeach
    return G1
endprocedure
```

Figure 2.11: Procedure to graph intervals

2.6.4 Nested Loops

The result of the application of FindIntervals and GraphIntervals to the basic block flow graph is termed the first order interval graph. Nested loops are found by generating higher order flow graphs (Figure 2.12), that is by applying FindIntervals to a previously generated interval graph.

Repeated applications of FindInterval will eventually result in one of two stable graphs (i.e. applying FindInterval to the graph results in the same graph). The first is a graph consisting of a single node that corresponds to the entire subroutine. In this instance, the original graph is termed *reducible.* The second is a graph similar to that depicted in Figure 2.13. If this situation arises, the flow graph is called *irreducible.*

The distinction between reducible and irreducible flow graphs is important because the presence of an irreducible graph makes subsequent data flow analysis using the interval graphs difficult. Fortunately it is possible to transform an irreducible graph to a reducible graph via the technique of *node splitting.*

Studies [21] have shown, however, that irreducible graphs occur very infrequently in "real life" programs. Furthermore, there are indications that the flow graphs of all programs conforming to "structured programming" techniques are reducible. In light of this, GO does not attempt to handle programs whose flow graphs are irreducible.

2.7 Problems with the Control Flow Analyzer

The control flow analyzer as described is capable of processing almost all "real" programs. However, there is one class of programs which it is definitely not able

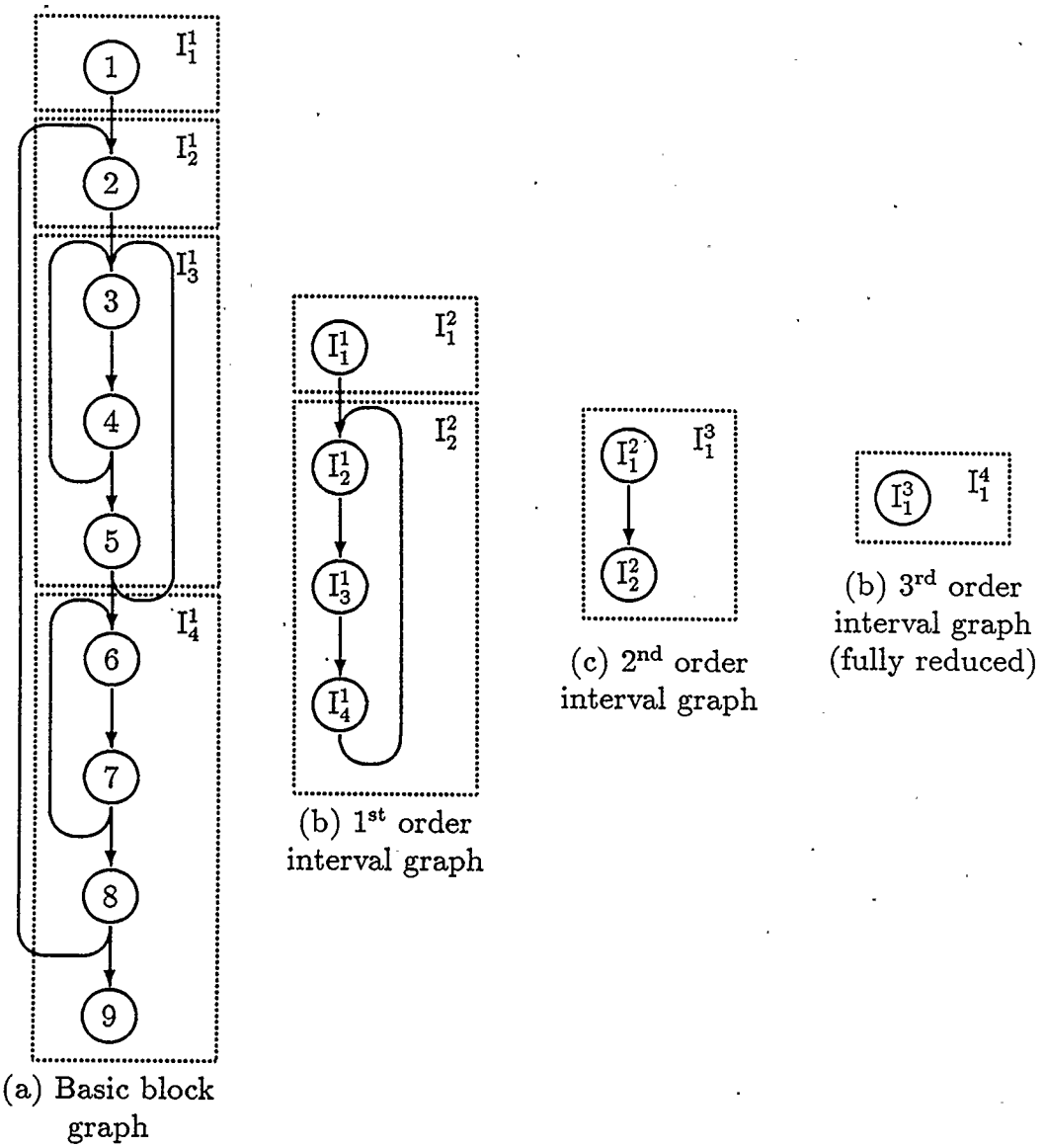


Figure 2.12: A sequence of derived interval graphs

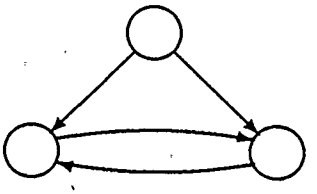


Figure 2.13: An irreducible flow graph

to handle. These are the programs which make use of jump tables or other similar coding techniques.

The inability to handle these programs arises because the control flow analyzer cannot determine the target address of a control transfer instruction which is computed at run time. This problem is also present in conventional systems; however, in those cases, it is not as crippling a problem as there is no need to separate the instructions from data.

There are a number of ways in which to surmount this problem. These are detailed in chapter 5.

Chapter 3

Data Flow Analysis

3.1 Overview of the Data Flow Analyzer

3.1.1 Function

If one thinks of GO's control flow analyzer a tool for generating a "road map" of a program, then its data flow analyzer may be thought of as a tool for studying the traffic patterns along each road. That is, it details how information is moved and used in a program. There are two principle forms of "traffic" which are of interest in this form of flow analysis: variables and expressions.

In GO, any operation which effects the state of the program is considered to be an expression. This state is embodied in the program's variables. To clarify, if a program is thought of as passing through a series of stages during the course of execution, then the program's variables indicate which is the current stage and its expression cause the program to move from one stage to the next.

It should be noted that the definitions given above are subtly different from those normally used. In GO, not only are the instructions for arithmetic and logical operations considered expressions, but so are control transfer instructions. Also implied is the fact the the machine's program counter and condition codes are considered to be variables (however, GO does not attempt data flow analysis on the program counter for obvious reasons).

Close inspection of the variables used in a program will reveal where they are defined (assigned new values) and where they are referenced. Application of data flow analysis techniques to this information yields the "scopes" of both the definitions and references. That is, for each definition of a variable, flow analysis will show where all possible uses of the result of that definition occur. Likewise, analysis will show which definitions are likely to be in force when a given reference to a variable occurs.

Using the information gained about variables it is possible to ascertain a number of interesting facts about the expressions in a program:

- what expressions are computed,
- where each instance of a given expression is computed,
- what the scope of each computation of an expression is (i.e. at which points the result of an instance is no longer valid),
- where the instance of a given expression is redundant (i.e. where the scopes of different instances of the same expression. overlap)

3.1.2 Input

As stated in the previous chapter, the output of the control flow analyzer forms the basis for the operation of the the data flow analyzer. Pure control flow information, though, is insufficient. In order to work, the data flow analyzer needs information about the operations carried out at the very lowest level, that is, within each basic block.

This information is generated during a "preprocess" phase that runs in conjunction with the basic block generator portion of the control flow analyzer. The operation of this module is outlined below.

3.1.3 Output

The output of the data flow analyzer can be thought of as a series of annotations to each basic block of the control flow graphs. These annotations are in the form of a series of sets. Each set enumerates the possible solutions to a given data flow problem.

For example, one set may indicate which variables will have been modified within its associated basic block when control leaves that block. Another set may indicate which expressions are guaranteed to have been computed when control enters a block (that is, the expression has been computed and is valid along each path leading into the block).

3.2 The Data Flow Preprocessor

The main task of the data flow preprocessor is to encode the internal structure of a basic block into a form usable during the actual data flow analysis. This entails three separate operations. The first is the identification of the variables which occur in the block. Second is the identification of the block's component expressions. Finally, there is the task of locating and encoding the specialized information required for data flow analysis.

3.2.1 Identification of Variables

There are a number of variable types which have fundamentally different uses. The principle task of variable identification is to determine the type of each variable and to create a unique descriptor for it, based on the type. In more conventional systems this information would be available from the compiler symbol table. However, as GO deals with executable program images alone, it must resort to inspection of the program code to determine this information.

Classes of Variables

GO distinguishes between four classes of variables. These are global variables, local variables, argument (procedure parameter) variables and temporary variables. The main distinguishing factors between the various classes is the method by which they are addressed and their scope.

The primary feature of global variables is that they may be referenced from any point in the program. References to these variables are through their absolute memory addresses, as displacements off of the program counter or through a base register. Regardless of the method of addressing them their locations are static throughout the execution of the program. It should be noted that the definition of a global variable does not imply that it is referenced throughout a program, merely that the possibility exists for this to be done. Therefore, a variable declared to be "static" in a C procedure, although generally considered to be a local variable, is treated as a global variable in GO.

The local variables are the converse of the global variables. Access to this class is restricted to the procedure in which they are defined. Variables of this

sort logically exist only during the execution of their defining procedure and each instance of this procedure has a unique copy of its set of local variables. In general, local variables are allocated in a special pool (usually the stack) for each instance of a procedure and access to these variables is via a *frame pointer.*

Argument variables are similar to local variables and on some machines are addressed through the frame pointer. For example, on the PDP-11, arguments are transmitted via the stack which also serves as the storage area for local variables. The local variables are addresses with negative displacements relative to the frame pointer. Arguments have positive displacements. On other machines, like the VAX-11, a special register, the *argument pointer*, is maintained but is used in precisely the same manner as the frame pointer.

The last class of variables, the temporaries, are not normally considered variables. As implied by the name, temporary variables have a very restricted scope. They usually do not span more than a few basic blocks and most are only valid for the portion of a single block.

The most obvious temporary variables are the machines general purpose registers. For the most part, these are used to store intermediate results from calculations. On machines with a limited register set the stack may be used for storage of intermediate results. The stack can therefore be thought of as a sequence of temporary variables as well.

Categories of Variables

GO recognizes two categories of variables: normal and pointers. A normal variable is one which contains a binary value. Whether this value represents an

integer, a floating point number or a character is irrelevant. As far as GO (and for that matter the computer) is concerned it is simply a stream of binary digits. Pointer variables contain the addresses of other variables. They can undergo the same operations as normal variables (addition, subtraction, etc.) but at some point the computer will attempt to use the value in the pointer as the address of another variable in memory.

One may think it naive to recognize only two types of variables, given the host of types which are usually supported by high level languages. This is not the case. There are no types in a high level language which can not be expressed in terms of either a normal or pointer variables. For example, an array is simply a series of normal variables. When referenced, a pointer is created to reference the first element. This pointer is then manipulated by addition of the array index to refer to the target element. The target is then referenced indirectly through the modified pointer.

3.2.2 Identification of Expressions

The second major task of the preprocessor is to identify the expressions within each basic block. This is not done directly however; it is a side effect of the conversion of the target program into an internally usable form. This internal representation is a reflection of the overall structure of the basic block.

From a control flow standpoint, a basic block is simply a sequence of instructions which is executed in a strictly linear fashion having no real internal structure. The data flow view is somewhat more complex.

Sequential execution of the instructions in a basic block does not preclude the existence of a number of parallel computations. This concept is illustrated in the following code fragment:

```
addl3   b,c,a     # a ← b + c
movl    s,q       # q ← s
subl3   $5,x,z    # z ← x − 5
divl2   t,q       # q ← q/t
movl    y,r0      # temp ← y
shl     r0,1      # temp ← temp * 2
mull2   r0,z      # z ← z * temp
mull2   d,a       # a ← d * a
```

Casual inspection would seem to indicate that there are six individual expressions calculated in a serial fashion: (note that the fifth expression is carried out in three stages with the intermediate results kept in a temporary variable)

$$\begin{aligned} a &= b + c \\ q &= s \\ z &= x - 5 \\ q &= q/t \\ z &= z * 2y \\ a &= a * d \end{aligned}$$

However, after a simple rearrangement of the instructions (which does not effect the results of the computations), it can be seen that there are really three independent calculations proceeding in parallel:

$$\begin{aligned} a &= b + c * d \\ q &= s/t \\ z &= (x - 5) * 2y \end{aligned}$$

3.2.3 Representation of Basic Blocks

The internal representation of blocks used in GO is derived from those of Allen[7] and Aho and Ullman[3]. A primary feature of the data structure used is that it attempts to record as much information about the relationships between the various components as possible (for example, whether a given expression occurs elsewhere in a basic block or where the operands of an expression where last set) The representation is based on five tables: the global, local and argument variable tables and the global and local expression tables. Separation of the variables into three tables is made necessary by the different mechanisms used to identify them.

The global variable table contains descriptions of each global variable identified during the preprocess stage. The table index for a given variable is the absolute address of the variable in the target address space.

Unlike the global variable table, there are local and argument variable tables associated with each call graph node. This reflects the transient nature of these classes of variables. The index for each of these variables is its offset from either the frame pointer or argument pointer, as appropriate.

The global expression table records all of the instruction/operand combinations which occur in the program. Each descriptor in the table contains a description of the instruction in terms of its function (branch, data manipulation, null operation, etc.) and its operands (number, size of each, whether a given operand is read or written, etc.). Paired with each operand description is a pointer to the description of the actual variable which resides in one of the variable tables. However, if the variable is a temporary, a flag is set to indicate this and the pointer

is treated as a numeric identifier for the temporary. The scope of a temporary is the subroutine in which it occurs, so temporary variable identifiers are reused between subroutines. Constant values are also treated as a special case in the global expression table. The presence of a constant is noted via a flag in the table. The value of the constant is simply stored in place of the variable pointer.

Entries in the global expression table are clustered in groups. All the expressions within a group have identical instruction descriptors. Furthermore all "input" operands for the expressions in a group are the same. However, each expression in the group has a different "result" operand. This organization facilitates easy expression matching when performing common expression elimination (this will be covered in chapter 4).

The global expression table enumerates all the basic expressions which occur in a program, however, it does not give an indication as to where each instruction occurs. This is the function of the local expression tables. There is a local expression table associated with each basic block in a program. It contains a descriptor for each expression occurring in the block. The ordering of descriptors is that of the occurrence of the expressions they describe in the basic block.

Each local expression descriptor contains a pointer to the occurrence of the particular instruction/operands combination in the global expression table for the instruction it describes. In addition to this, there is a pointer corresponding to each operand which indicates which previous instruction in the block which most recently modified that operand (Figure 3.1). If no previous instruction in the basic block modified the operand, the pointer is nil.

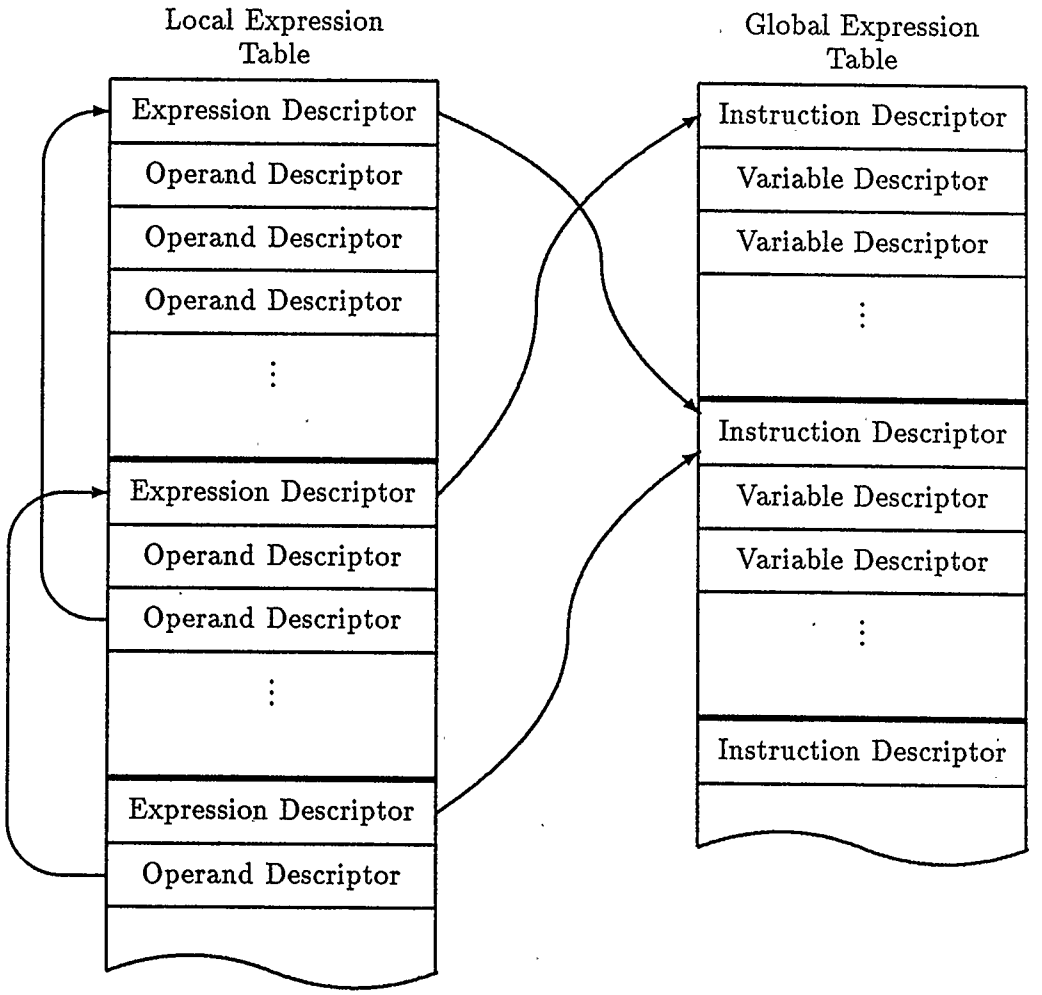


Figure 3.1: Organization of local and global expression tables

3.2.4 Used and Defined Sets

The preprocessor uses two sets, USED and DEFINED to pass the facts it has learned about the variables in a basic block to the main data flow analyzer. USED details which variables are referenced in each basic block whereas DEFINED shows which variables are set (assigned new values). The possibility exists that a variable may be both used and defined within the same basic block. This presents a problem as the main data flow analyzer treats basic blocks as monolithic units. It is therefore necessary for the preprocessor to "filter" out this seemingly contradictory information. For example, a number of different situations exist in the following code fragment.

```
add13    a,b,c    # c ← a + b
add13    c,d,b    # b ← c + d
```

All four variables are referenced and variables b and c are assigned new values. The data flow preprocessor would mark both b and c as being members of the DEFINED set, however, it would place only variables a, b and d in the USED set. This reflects the fact that the use of a variable is only of concern *backwards* along the control flow paths, thus the use of c is obscured by its definition. Variable b, however, is used *before* being set therefore its use is *upwardly exposed* (visible to previous blocks).

3.2.5 Computed and Obsolete Sets

Two additional sets, COMPUTED and OBSOLETE are defined with respect to expressions as are USED and DEFINED with respect to variables. If an expression is evaluated during the course of execution of a basic block, it is placed

in the COMPUTED set. If one of the terms in an expression is redefined (i.e. a component variable is assigned a new value or a subexpression re-evaluated) then it is placed in the OBSOLETE set.

As with variables, the preprocessor must act as a filter for the main data flow analyzer. In this case the situations to be eliminated are those in which an expression is computed and subsequently made obsolete within the same basic block.

3.3 Intraprocedural Data Flow Analysis

The task of the main data flow analysis code is the investigation of flow patterns within each subroutine. Once the basic infrastructure has been constructed by the data flow preprocessor and the control flow analysis code, it is possible to begin this phase of analysis. The operation of the main data flow analyzer is straightforward. It propagates the information gained by the preprocessor over the graphs generated by the control flow analyzer with a view towards solving a number of predefined data flow problems.

There are many possible data flow problems which may be investigated. However, many are rather specialized (i.e. the information they yield is not useful in a wide range of optimizations). This renders them too expensive to be used. In light of this, the GO data flow analyzer solves only the four most basic data flow problems[17,29]. All four problems are similar, making implementation simple.

Ullman[29] notes that each of these problems may be likened to a system of simultaneous linear equations and that they may be solved through techniques similar to Gaussian Elimination. The similarity arises because each basic block in

the subroutine under scrutiny contributes a single equation to the overall problem. The equation relates a single unknown, the solution to the problem, to the solutions of its neighbouring blocks. There are N equations for N blocks with N unknowns which is a solvable system of equations. The general form of the equation is:

$$X_i = ((\forall a \in A, \mathcal{F}(X_a)) - R_a) \cup G_a$$

Where

- A is the set of basic blocks "adjacent" to block i. Depending on the problem these could be either block i's *predecessors* (those blocks from which i may gain control) or it *successors* (those blocks to which i may pass control).
- $\mathcal{F}(X)$ is determined by the data flow problem under investigation. It is either the union of all sets X or the intersection of those sets.
- X_i is the set of solutions to the problem at block i.
- X_a is the solution to the problem in the adjacent block a.
- R_a is the subset of solutions which are *removed* by the operations in block a.
- G_a is the subset of solutions to the problem *generated* in block a.

3.3.1 Available Expressions

An expression is said to be *available* at a given point in a program, if it has been computed at some time along each of the paths which arrive at that point. Furthermore, none of the component variables of the expression may have been modified since the expression was computed.

As an example, consider the flow graph fragment in Figure 3.2. Of interest is

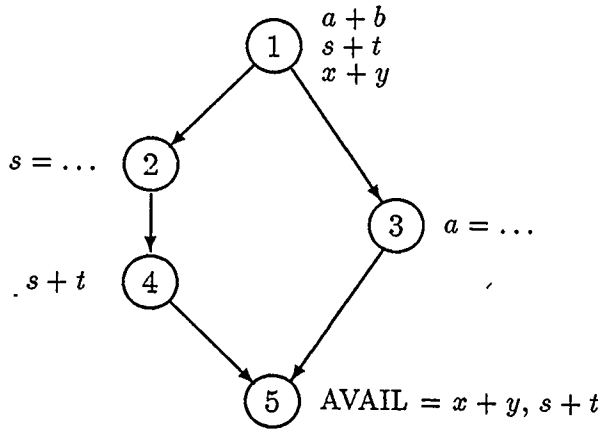


Figure 3.2: Illustration of the available expressions problem

the set of available expressions at block 5. The first expression $x + y$ is computed in block 1. From this block to block 5 are two paths: 1-2-4-5 and 1-3-5. Neither x, nor y is redefined in any of the blocks along either path. Therefore, $x + y$ is said to be available at block 5.

The expression $a + b$, is also computed in block 1. However, in this case, the variable a is redefined in block 2. Since the expression is only valid on one of the paths entering block 5, it is not considered available.

A similar situation is presented with the expression $s+t$. Although in this case the expression is reevaluated in block 4. There is a valid instance of the expression in force on entry to block 5 along either path. Here again, the expression is considered available.

There are three elements to the solution of the available expressions problem:

1. The problem concerns itself with equations, so the solution involves the COMPUTED and OBSOLETE sets defined by the preprocessor.

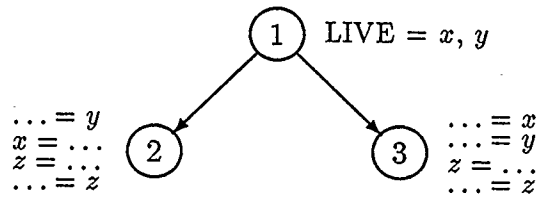


Figure 3.3: Illustration of the live variables problem

2. Information is propagated in the direction of control flow. Thus the data flow equation for this problem must operate over the set of predecessors of the block being processed.

3. For an expression to be considered available, it must have been computed along every path to the block. Therefore the set of solutions must be based on the intersection of the sets of its adjacent blocks.

The equation for solving the available expressions problem is:

$$\text{AVAIL}_i = ((\bigcap_{p \in P} \text{AVAIL}_p) - \text{OBSOLETE}_p) \cup \text{COMPUTED}_p$$

where P is the set of predecessor blocks to i.

3.3.2 Live Variables

Variables are said to be "alive" at a given point in a program if there is a references between that point and the next redefinition of the variable. Consider the skeletal flow graph in Figure 3.3. In the entry block three variables are set: x, y and z. Variable x is used in both blocks which can potentially gain control after the entry block. In both cases, x is used before being assigned a new value. Variable y appears in only one of the subsequent blocks, however, it too is referenced before

being set. Both variables x and y are considered live on exit from the entry block. Variable z, however, is in both cases redefined before it is used. It is therefore considered "dead".

The parameters for this problem are:

1. The sets USED and DEFINED are used as this problem is associated with variables.
2. The flow of information is *backwards* (against the flow of control) as the issue is the disposition of variables after the current block. Thus, the data flow equation operates over the successors to the current block.
3. The function relating all information from the successor block is set union as a use of the variable in *any* successor is sufficient for it to be considered live.

The corresponding equation is

$$\text{LIVE}_i = ((\bigcup_{s \in S} \text{LIVE}_s) - \text{DEFINED}_i) \cup \text{USED}_i$$

where S is the set of successor blocks.

3.3.3 Very Busy Expressions

If an expression is computed in a given block and is subsequently used on all paths originating from that block, it is considered a very busy expression. An illustration of this problem is given in Figure 3.4. The object of the exercise here is to determine the set of very busy expression at the exit points of node 1. The expression $x + y$ is computed along two of the paths leaving node 1, specifically in nodes 2 and 4. However, the expression is not computed in node 3 or any path

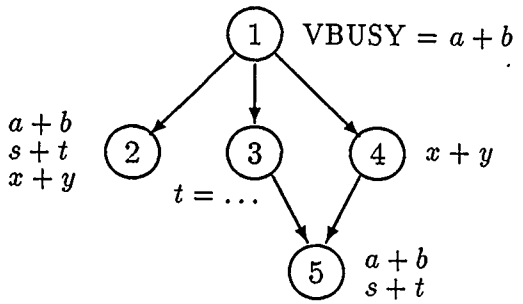


Figure 3.4: Illustration of the very busy expressions problem

leading away from node 3. Therefore, $x + y$ is not a very busy expression with respect to 1. The other expression, $a + b$ is computed in nodes 2 and 5, but not in either of 3 or 4. There exist paths from both of these nodes to node 5, thus an exposed use of the expression is visible along all paths leading away from node 1 and the expression is considered very busy. A similar situation is presented with the expression $s + t$. However, in this case, the redefinition of the variable t masks the occurrence of $s + t$ along the 1-3-5 path. For this reason, the expression is *not* very busy with respect to node 1.

The parameters of the equation for this problem are:

1. This is an expression problem so it operates on the COMPUTED and OBSOLETE sets.

2. The problem relates conditions in the current block to those in its successors.

3. By definition, the solution to the problem is a function of the intersection of the solutions to neighbouring blocks.

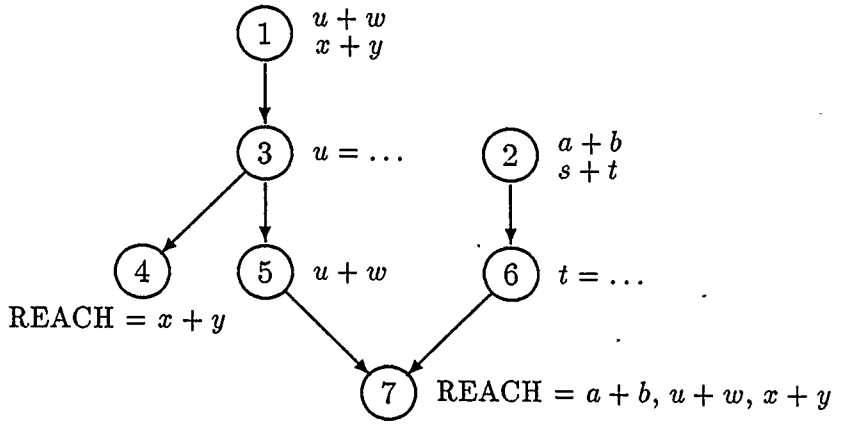


Figure 3.5: Illustration of the reaching definitions problem

which yield the following equation:

$$\text{VBUSY}_i = ((\bigcap_{s \in S} \text{VBUSY}_s) - \text{OBSOLETE}_s) \cup \text{COMPUTED}_s$$

where S is once again the set of successor blocks to i.

3.3.4 Reaching Definitions

The reaching definitions problem gives the "scope" of each expression. It is similar to the available expressions problem in that it gives the history of an expression on entry to the block in question. It differs in that an operation that occurs on any incoming path is of interest as opposed to available expression in which only operations which occur on all paths are of interest.

An example reaching definitions problem is given in Figure 3.5. Of interest in this example are the definitions that reach the nodes 4 and 7. There is only one path leading into node 4. Along this path are computed two expressions: $x+y$ and $u+w$. The former is transmitted through node 3 unscathed. The latter, however,

is made obsolete by the setting of variable u. The set of definitions reaching node 4 therefore contains the single expression $x + y$. Two paths lead into node 7: 1-3-5-7 and 2-6-7. Along the latter path are computed two expressions. An element of one of the expressions is redefined, though, so only one of them, $a + b$ reaches node 7. The 1-3-5-7 path gives the most complex situation. Once again, the one of the expressions computed in node 1 is masked by a redefinition of one of its terms in node 3. The same expression is recomputed in node 5 and is thereby made available at 7. The set of definitions reaching node 7 then comprises three expressions: $x + y$, $u + w$ and $a + b$.

The elements involved in the solution of this problem are:

1. The solution to the problem involves the use of the COMPUTED and OBSOLETE sets.

2. Of interest are the set of expressions which may have been computed at entry to the block, therefore the data flow equation must operate over the predecessor blocks.

3. Definitions tend to accumulate. That is, the solution to the problem for any given block will be based on the *union* of the sets of solutions of its predecessors.

The equation used to solve this particular data flow problem is:

$$REACH_i = ((\bigcup_{p \in P} REACH_p) - OBSOLETE_p) \cup COMPUTED_p$$

where P is the set of predecessor blocks.

3.3.5 Solving the Data Flow Equations

Solution of the various data flow analysis problems is achieved by propagating the data flow information around the control flow graph. The extent of propagation of information is controlled by the various data flow equations.

The process would be trivial if control flow in a program were strictly linear. The solution for the entry block would simply be the values determined by the preprocessor. Solutions for subsequent blocks would be derived by applying the data flow equations. Unfortunately, programs are rarely this simple.

The existence of loops in programs opens up the possibility of "feed back" from blocks which have yet to be processed. This is a classic "chicken and egg" problem: the data flow information for subsequent blocks cannot be determined without first knowing the data flow information for the current block, however, this information cannot be determined without solving the data flow equations for the subsequent blocks.

Earlier on, it was stated that techniques similar to Gaussian elimination could be used to solve the data flow problems. The simplest of these (the *Round-Robin technique*) is to first make some minor assumptions about the solutions for each block based upon the information derived by the preprocessor (i.e. assume that each blocks operates in isolation, thus the solution for each block is the information derived by the preprocessor). Each block is then visited in a round robin fashion. At each visit, the data flow equations are computed. The processing terminates when there has been no change in the solution for any block during one complete pass.

The major problem with this technique is that the solutions to some of the blocks will stabilize before others, therefore reprocessing them is an unnecessary expense. A variant on this method (the *Worklist technique*) surmounts this difficulty by keeping lists of blocks whose solutions are still in transition. Neither technique, however, is very cost effective in terms of processing time.

In GO, the solution of the data flow analysis problems is achieved through a less expensive, though more complex technique known as interval analysis[6,9,12,17,29] which uses the interval graphs generated by the control flow analyzer.

The interval analysis technique (Algorithm 3.1) operates in two phases. In the first phase, data flow information is from the basic block graph (which may be thought of as the 0^{th} order interval graph) through each higher order interval flow graph until the highest order flow graph is reached. The second phase reverses the process and propagates information from the highest order graph back down to the basic block graph.

This technique worksp due to the nature of intervals. All paths in an interval involve the interval header, by definition of an interval. This means that if one could obtain a solution to the data flow equations for the header node, the solutions for the other nodes in each interval may be solved for easily by a single application of the data flow equations.

There are two sources of data flow information entering the interval header. Both must be consulted before generating a solution. The first source is from within the interval, through the back arcs to the header. This information is obtained in the first phase of interval analysis. The second source is from prede-

Algorithm 3.1: AnalyzeDataFlow

PURPOSE: Perform data flow analysis using intervals

INPUT: A derived series $(\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n)$ a basic block graph, $\mathcal{G}_1$ is the first order interval graph, $\mathcal{G}_2$ of flow graphs. Where $\mathcal{G}_0$ is is the second order interval graph, etc. The output of the data flow preprocessor. The set O containing the set of solutions arriving from outside the subroutine. (In GO, this set is always empty.)

OUTPUT: A data flow solution set for each basic block node.

METHOD: The first pass of interval analysis is outlined in Figure 3.6. The second pass is given in Figure 3.7. The driving procedure for both passes is given in Figure 3.8.

cessors to the header which lie outside of the interval. The propagation of this information is handled in the second phase of interval analysis.

There is one problem associated with performing interval analysis as outlined above. Within an interval, there may be a number of different control flow paths. It is unlikely that the set of generated definitions, G and the set of removed definitions, R, would be the same along each path. At the level of the interval, however, the existence of these different paths is masked. An assumption made in the presentation of data flow equations was that data flow information is associated with graph nodes. How then are the different generated and removed sets for each path through the interval be transmitted to the higher order graphs?

```
procedure IntervalAnalysis1(graph G_n, G_n - 1)
    if G_n ≠ G_0 then
        foreach arc i in G_n do
            Locate corresponding arc j in G_{n-1}
                which exits interval header node h
            R_i = R'_j
            G_i = (X_h - R'_j) ∪ G'_i
        endforeach
    foreach arc i exiting the header node do
        R'_i = R_i
        G'_i = G_i
    endforeach
    foreach exit arc i of node N in G_n processed in interval order do
        foreach arc j entering node N do
            R'_i = R'_i F R_j
            G'_i = G'_i F G_j
        endforeach
        R'_i = R'_i - R_i
        G'_i = (G'_i - R_i) ∪ G_i
    endforeach
    /* X_h is an estimate at the solution to the data flow problem
        at the interval header node h */
    X_h = ∅
    foreach back arc i in the interval with header h do
        X_h = X_h F G'_i
    endforeach
endprocedure
```

Figure 3.6: Procedure to perform the first pass of interval analysis

```
procedure IntervalAnalysis2(graph G_n, G_{n+1})
    foreach node N_I in G_{n+1} do
        /* Node h is the header node in G_n
            for the interval represented by node N_I */
        X_h = X_h F X_{N_I}
    endforeach
    foreach node N_I in G_{n+1} do
        foreach arc i exiting the header node h do
            /* X_i^a is the intermediate data flow solution at arc i */
            X_i^a = (X_h - R_i) ∪ G_i
        endforeach
        /* Note: In the following loop, "forward" problems (available
        expressions and reaching definitions) handled by processing nodes in
        interval order. For "backwards" problems (very busy expressions and
        live variables) the notes are visited in reverse interval order */
        foreach node N_J in the interval represented by N_I do
            X_{N_j} = ∅
            foreach arc p entering node N_J do
                X_j = X_j F X_p^a
            endforeach
            foreach arc s exiting node j do
                X_s^a = (X_j - R_s) ∪ G_s
            endforeach
        endforeach
    endforeach
endprocedure
```

Figure 3.7: Procedure to perform the second pass of interval analysis

```
procedure IntervalAnalysis(graph G_0, G_1, G_2, ..., G_n)
    IntervalAnalysis1(G_0, nil)
    foreach graph G_i, i = 1, 2, ..., n - 1 do
        IntervalAnalysis1(G_i, G_{i-1})
    endforeach
    There is a single node in G_n. For this node, X = O
    foreach graph G_i, i = n - 1, n - 2, ..., 1 do
        IntervalAnalysis2(G_i, G_{i+1})
    endforeach
endprocedure
```

Figure 3.8: Driver routine for interval analysis

The solution to this problem is to associate the two sets with the graph arcs exiting each interval instead of the interval node itself. This works because the set of control paths associated with each exit arc would all produce the same answers to the data flow problems.

Each arc in an interval graph at level N corresponds to an arc in the interval graph at level $N-1$. Therefore, the solution to both arcs is the same. The only problematic situation occurs between the first level interval graph and the basic block graph. If the basic block graph is treated as the 0^{th} level interval graph, all that need to be done is to find a mapping of solutions associated with basic block graphs nodes to their exit arcs. This is trivial.

By definition, all exit arcs from a basic block graph node must have the same solutions to the data flow problems. Therefore, each exit arc from a basic block is associated with the solution for the basic block.

Algorithm 3.2: GenerateUDChain

PURPOSE: Determine the use-definition chain for a given use of a variable

INPUT: A basic block flow graph with data flow annotations, the descriptor for the variable for which the chain is to be generated and a pointer to local expression table entry in which the use appears

OUTPUT: A use-definition chain

METHOD: Procedure as outlined in Figure 3.9.

3.3.6 Use-Definition and Definition-Use chains

There are a number of useful pieces of information that may be derived from the various data flow sets. Chief among these are use-definition and definition-use chains[6].

A use-definition chain relates each instance of a variable to the specific instances of the expressions which defined them. That is it relates a point in the basic block graph where a specific variable is used to the points in the graph where it could have been set. The use-definition chain for a variable is determined by examining the REACH, AVAIL and DEFINED sets. The procedure is given in Algorithm 3.2.

The converse of a use-definition chain is the definition-use chain. In this structure, the location of a given expression is related to the locations of the uses of this variable it defines. This structure is found by examination of the REACH, LIVE

```
procedure GenerateUDChain(graph G, vardesc V, localexpr E0)
    if E0.OperandDesc[V] ≠ nil then
        add E0.OperandDesc[V] to the use-definition chain
    else
        foreach E1 ∈ REACH of block containing E0 do
            if E1 defines V then
                foreach path P entering block containing E0 do
                    Locate the node N on P in which E1 ∉ AVAIL
                    then locate the definition of V in node N and
                    add it to the use-definition chain
                endforeach
            endif
        endforeach
    endif
endprocedure
```

Figure 3.9: Procedure to generate a use-definition chain

and USED sets. The procedure is similar to that used to determine use-definition chains.

3.4 Interprocedural Data Flow Analysis

The techniques discussed thus far in this chapter will yield a fairly complete description of the data flow patterns within each subroutine of a program. Subroutines, however, do not operate in complete isolation from each other. It is necessary to also perform some analysis of the data flow *between* procedures [5].

Interprocedural data flow analysis can be performed using the same techniques as intraprocedural analysis. All that is required is the substitution of the call graph for the basic block graph.

For the most part, call graphs are far simpler in structure than basic block graphs. Loops in call graphs exist only in the presence of recursive subroutines and these occur infrequently. In light of this fact, using interval analysis to obtain interprocedural flow information is a waste of effort. More appropriate techniques would be the round robin or worklist methods which were briefly mentioned earlier.

In GO, only a rudimentary interprocedural analysis is performed. Its sole purpose is to determine which variables are referenced or defined in a subroutine. There are USED and DEFINED sets associated with each call graph node to record this information. The technique used is as follows.

Before data flow analysis commences, the basic block and interval graphs for each subroutine are modified. The modification entails the addition of two "dummy" basic block or interval nodes. The first node is added prior to the entry node for the graph in question. The dummy node for each interval graph maps to the dummy node of the immediately inferior graph. The second node is a dummy exit node. Links are added between each real exit node in the graph and the dummy. Like the dummy entry node, each dummy exit node in the interval graphs is mapped to the equivalent node in the inferior graph. The USED, DEFINED, COMPUTED and OBSOLETE sets for all the dummy nodes are initially empty.

A topological sort is then performed on the call graph. Each node in the call graph is processed in reverse topological order. When a node is visited an intraprocedural data flow analysis is performed on the subroutine. The USED set for the call graph node is the LIVE set of the dummy entry node for the

subroutine with all local, argument and temporary variables removed. Likewise, the DEFINED set is the REACH set belonging to the dummy exit node with only the global variables preserved.

In order to make use of the interprocedural information, the basic block graphs for each subroutine undergoes a "fixup" procedure prior to commencement of the intraprocedural analysis. This is required because side effects may result when a subroutine call occurs. Specifically, global variables which may on the surface appear to be untouched may be referenced or redefined in the subroutine. If the intraprocedural analyzer does not know of these, erroneous data flow results will definitely arise.

The fixup procedure involves scanning each basic block node for subroutine calls. If a call is found, the DEFINED and USED sets of the block in which it occurs are merged with the corresponding sets of the called subroutine (i.e. a set union is performed). In addition, the local expression table for the block is searched for operand pointers which span the subroutine call. If any of these links are for variables in the called routines DEFINED set, the links are broken (i.e. the operand pointer is set to nil).

Chapter 4

Heuristics for Code Improvement

4.1 Overview

Control and data flow analysis provide GO with a vast amount of information about a program. Raw information, though, is of limited use. Ideally there would be a simple, algorithmic method of processing this raw data and producing an optimal variant of the program. Unfortunately, no such technique exists that can be guaranteed to run in a finite amount of time. Instead GO resorts to heuristic programming techniques. Basically, all that is done is to search the the flow graphs of the input program for predetermined patterns which can be manipulated in a systematic way in order to produce better code.

For the most part none of these patterns, or the manipulations performed on them, is complex. Nor does any transformation result in a massive improvement in program performance. However, they do occur with great frequency. Thus the cumulative effect may be quite pronounced.

Code improvement, then, is achieved through brute force rather than elegance of design or subtlety of implementation.

4.1.1 Constraints on the Application of Heuristics

A cardinal rule in code improvement is that an alteration may affect the manner in which results are obtained but not the results themselves. There are a

number of facets to this rule which Kennedy[19] summarizes as three constraints on the optimization process:

1. The functional equivalence constraint

   The optimized sequence must generate the same output as the original program for all *legal* input data. Output for illegal input data is not so constrained (the program still fails, but for a different reason).

2. Legal data set constraint

   The set of legal inputs to the optimized program must be the same as, or a superset of, that of the unoptimized program; the new program may handle additional classes of input but it must not handle fewer classes.

3. Safety Constraint

   The code resulting from optimization should not generate errors which the original code did not generate. That is, if a computation was valid under the old program, it must be valid after optimization. However, the reverse need not be true. Also permissible is a change in the timing of errors. That is, an error may occur at a different stage in the execution that it would have in the unimproved program. Furthermore, a great deal of flexibility is available in the application of this constraint. For example, at the discretion of the programmer, it may be acceptable to sacrifice some loss in precision of floating point calculations when an optimization is applied.

All of the heuristics presented below conform to these guidelines.

4.2 Performing Code Improvement

In most code improvement systems each optimization heuristic is implemented as a separate entity. This is not true of GO. One of the major goals in its design was the use of a modular programming discipline in order to reduce the redundancies in the code. The implementation of the code improvement heuristics reflects this goal.

GO's repertoire of optimization heuristics have two common traits. First, the information required to implement each optimization is given directly in the the results of the control and data flow analysis phases. Second, the heuristics are for the most part machine independent. They operate primarily through manipulation of the flow graphs rather than recoding of the program.

The flow graph manipulations required to implement the various heuristics are few in number. The most rudimentary is the creation of new basic blocks. All that is entailed for this is the addition of a basic block skeleton in front of an existing block. The new block takes from the existing block all of the latter's inflowing *forward* graph arcs. A single new arc is then added between the two blocks. Any *back* arcs are retained by the original block. The new node becomes a component of the interval which contains its *predecessor*.

A related manipulation is the splitting of a basic block. When this is done, the local expression table is divided at a specified instruction. The first node gains the first part of the table and all incoming arcs (back arcs included). The second node consists of the rest of the local expression table and retains all of the outflowing arcs of the original. Both nodes are members of the same interval that contained the original node.

Some heuristics require the ability to "create" new expressions. Actually, the process is not so much a matter of creating a totally new expression as making minor modification to an existing one. To do this a copy of an entry in the global expression table is made. One of the operands is then altered to suit the needs of the heuristic. If the altered operand is the one which receives the result of the expression's calculation its new value is usually a fresh temporary variable. For example, a "move value to local variable" instruction may be changed into "move value to temporary variable".

A manipulation which is used in close conjunction with the previous one is the allocation of new temporary variables. During processing of a program, GO acts as if an infinite supply of temporary variables are available for use. A new temporary may be allocated from this pool at will. The actual assignment of temporaries to memory locations and registers is performed by the code regeneration part of the improver.

Another important manipulation is the insertion of expressions into basic blocks. To do this a reference to an entry in the global expression table is provided. A reference to the destination block is also supplied. A new entry that refers to the given global expression is added to the block's local expression table. The appropriate operand pointers are then generated to link the new expression into the existing basic block structure. Finally, the data flow information for the basic block graph must be updated. Each path leading away from the modified block is traversed. As each node is visited, the various data flow sets are updated as appropriate (e.g. the expression will be added to the node's AVAIL set). The traversal of each path terminates when an exit node is reached, when a redefini-

tion of one of the expression's components is found in the visited block or when the expression is found to already be in the node's AVAIL set.

The final important flow graph transformation is the removal of expressions from blocks. All that is needed to accomplish this is to remove the appropriate entry from the local expression table and update and references to it from other expressions in the block. An expression is never removed unless it is unused, that is, the result it produces is never referenced in subsequent expressions. There is no need update the graph's data flow information as any information relating to the expression will be ignored anyway.

4.3 The Heuristics

The catalogue of possible optimizations is constrained only by one's imagination. However, many heuristics have limited applicability and so are not suitable for a general purpose code improvement system. A number of heuristics, though, are almost universally applicable. These heuristics are well covered in the literature [3,7,8].

GO actually has very few code improvement heuristics. This reflects the fact that the code improver is a research tool rather than a production system. The heuristics were chosen in order to show that code improvement is possible under the constraints put forth in the design of the improver.

The heuristics are detailed below. The order of presentation reflects the order in which the heuristics are applied. This order is crucial to the proper operation of the system.

4.3.1 Inline Subroutine Expansion

One of the most expensive operations in terms of both machine cycles and storage is the subroutine call/entry/exit sequence. Because of this, it would seem prudent to limit the number of subroutine calls in a program. Unfortunately, good programming technique dictates the use of many subroutines. Here then, lies an excellent opportunity to affect code improvement[10,15].

What is called for is an optimization heuristic that will convert hierarchies of many small subroutines into, ideally, one large routine. This procedure is known as *inline subroutine expansion.* As the name implies, this heuristic involves the "textual" replacement of a subroutine call with the body of the actual subroutine, thus eliminating the expensive subroutine overhead.

Anatomy of a subroutine call

The invocation of a standard subroutine can be thought of as progressing through five discrete stages. In order of execution, these are:

1. Call set up

   In this stage, the calling procedure arranges for any parameters to be passed to the subroutine. It also records the address to which the subroutine is to return control upon termination. Finally, it does the actual transfer of control to the subroutine.

2. Subroutine entry

   Here, the called subroutine saves any registers which it uses (except, of course, any which are used to transmit information to the calling routine) and any other elements of machine state which it modifies. It also performs

any initialization functions required for the normal operation of the subroutine. This includes such functions as creating a stack frame, allocating space for local variables and setting up dynamic and static links.

3. Subroutine body

   At this point, the actual execution of the subroutine proceeds.

4. Subroutine exit

   This phase is primarily concerned with undoing the effects of the second stage: local variables storage is released and the saved machine state is restored. If the subroutine is a function type (i.e. it sends information back to the caller) the return value is placed into a standard location for the calling routine (usually one of the general purpose registers). Finally, control is passed back to the address recorded in the first stage.

5. Call cleanup

   After regaining control from the called subroutine, the caller releases any storage it allocated for passing of parameters and retrieves any information passed back from the subroutine.

Except for the third stage, all of this processing is directed at maintenance of the machine's operating environment rather than towards the solution of a problem. The goal behind this particular optimization is the elimination of as many of these nonproductive sequences as possible.

Selecting subroutines to expand in line

The first stage in performing the optimization is the selection of the subroutines which are to be expanded. A subroutine is suitable for inline expansion if either of the following criteria is met.

- The subroutine is only invoked at one point.
- The "cost" of the subroutine body is less than that of the subroutine invocation sequence given above.

The concept of cost in the second case is a nebulous one. The cost of a sequence of code is entirely dependent on the situation in which the sequence is computed. For example, if a program were close to the maximum size the computer could handle and the code sequence would cause the program to exceed this limit, then the cost of the segment would be considered unacceptably high.

In general, the cost of a code sequence is determined by the goals of the optimizer. For example, if speed enhancement is the objective, a small increase is program size in exchange for a large reduction in execution time would be acceptable. What constitutes a small decrease in size or a large reduction in time is determined by the programmer in accordance with his needs.

All other things being equal though, the relative costs of two sequences of code may be determined by comparing like characteristics of the sequences. To meet the second criteria for inline expansion then, the subroutine body would have to take up less space and execute in less time than the associate subroutine invocation sequence.

The procedure for expanding subroutines inline is given in Algorithm 4.1.

Algorithm 4.1: InlineSubrExpansion

PURPOSE: Procedure to expand subroutine calls inline

INPUT: The call graph, the basic block graph and a pointer to the instance of a subroutine call to be expanded

OUTPUT: Modified basic block and call graphs with the subroutine expanded inline

METHOD: The procedure is as follows.

1. A duplicate copy of call graph node for the subroutine and all its associated data structures (local and argument tables, basic block graph, etc.) is made. All modifications to the subroutine are made to this copy, thus any other calls to the routine will not be affected.
2. The subroutine's local variable table is adjusted. This entails adding the size of the caller's stack frame to the offsets of each entry in the local variable table.
3. The subroutine's local variable table is merged with the calling routine's local variable table and the calling routine's stack frame is enlarged by the size of the subroutine's stack frame.
4. A mapping of procedure argument variables to the actual parameters in the calling routine is established.

... continued on page 82

Algorithm 4.1: InlineSubrExpansion (continued)

5. Each instance of a procedure argument variable in the subroutine's basic block graph is replaced with the actual parameter. Once all of the procedure argument variables have been processed in this manner, the argument variable table is discarded.
6. The names of the temporary variables in the subroutine are made unique with respect to the caller's temporaries.
7. The basic block which invokes the subroutine is split in two at the subroutine call. The subroutine invocation code in the caller and the entry/exit code in the called routine are discarded. If the calling routine performs a stack clean up after the subroutine returns, the clean up code is also discarded. Any code in the called subroutine which is used to return a value to the caller and the corresponding code to retrieve the value in the caller is retained. However, this code is probably redundant and will be eliminated during subsequent manipulation of the program.
8. The basic block graphs for the two routines is then merged. A single graph arc is added from the first part of the newly split block to the entry node of the called subroutine basic block graph. Graph arcs are also added between each exit in the called subroutines graph and the second new block.

4.3.2 Invariant Computations in Loops

A well known maxim of computer science is the 80/20 rule: eighty percent of the time is spent executing 20 percent of the code. This is merely an observation that most useful work in a program is done in loops. The code which exists outside of any loop is generally involved in loop maintenance: setting up for entry to the loop and cleaning up after exit. Also contained in these sections are the various operations involved with maintenance of the program as a whole (i.e. variable initialization, subroutine entry and termination, etc.). As most "real work" is done inside loops, the greatest improvement in code quality would be realized by optimizing the code within loops[16].

The most obvious loop optimization is to locate invariant computations and move them out of the loop. An invariant computation is one which will result in the same value on each iteration of the loop. For example, the following code:

```
for (a = 0; a < 10; a++)
    for (b = 0; b < 10; b++)
        c = a * 3 + b;
```

would likely be compiled into:

```
        movl    $0,a        # a ← 0
loop1:  movl    $0,b        # b ← 0
loop2:  mull3   $3,a,c      # c ← 3 * a
        addl2   b,c         # c ← c + b
        acbl    $10,$1,b,loop2
                            # b ← b + 1; if b < 10, branch
        acbl    $10,$1,a,loop2
                            # a ← a + 1; if a < 10, branch
```

The third instruction in this case is invariant with respect to the second loop and may be moved out of it. The procedure for performing this optimization is

Algorithm 4.2: InvariantCodeMotion

PURPOSE: To perform loop invariant code motion.

INPUT: A series of interval graphs and the corresponding basic block graph.

OUTPUT: The basic block graph is modified so that all invariant computations are moved out of loops.

METHOD: Procedure as outlined in Figure 4.1.

given in Algorithm 4.2. When this algorithm is applied to the above example, the following code sequence results:

```
        movl    $0,a        # a ← 0
loop1:  movl    $0,b        # b ← 0
        mull3   $3,a,c      # added instruction
loop2:  mull3   $3,a,c      # c ← 3 * a
        addl2   b,c         # c ← c + b
        acbl    $10,$1,b,loop2
                            # b ← b + 1; if b < 10, branch
        acbl    $10,$1,a,loop2
                            # a ← a + 1; if a < 10, branch
```

A new instance of the multiply instruction has been placed before the body of the second loop. However, the original instance is *not* removed. The new instance renders the old one redundant, though, so it will be removed by the common expression elimination heuristic.

4.3.3 Code Hoisting

There are instances in which a given expression is computed along all paths leading away from a given basic block. This situation is indicated by the presence

```
procedure InvariantCodeMotion(graph G1, G2, G3, ..., Gn)
    foreach Gi, i = 1, 2, 3, ... n do
        foreach interval I in Gi do
            foreach block B in I do
                /* Intially, all expressions in B are unmarked */
                foreach expression E in B do
                    Examine the use-definition chain for each
                    variable in E. If no definition of the
                    variable occurs in a block which is in I,
                    then mark E.
                endforeach
            endforeach
            if any expression was marked then
                Create a new block B in front of the header node of I.
                Add an instance of each marked expression to B
            endif
        endforeach
    endforeach
endprocedure
```

Figure 4.1: Procedure to perform invariant code motion

of the expression on the VBUSY set of the block. In such cases, an optimization that may be applied is to have one instance of the expression in the given block and make the result available along each exit path. This is known as code hoisting. The procedure for code hoisting is given in Algorithm 4.3. As in the case of the loop invariant expression heuristic, the net effect of this algorithm is the introduction of a new instance of a given expression which will render the subsequent instances redundant. However, there is a subtle feature of this algorithm which should be noted. The feature is illustrated in Figure 4.2.

Algorithm 4.3: CodeHoist

PURPOSE: To perform code hoisting.

INPUT: A basic block graph with VBUSY data flow annotations.

OUTPUT: The basic block graph modified in accordance to the code hoisting heuristic.

METHOD: Each basic block in the graph is examined. If any expressions are designated as very busy with respect to that block, a copy of the expression is inserted at the end of the block.

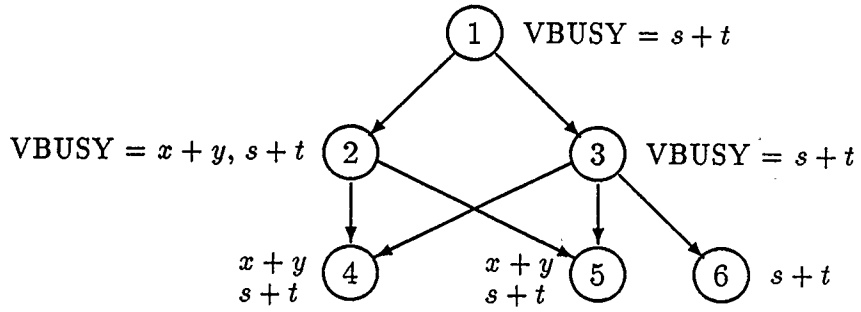


Figure 4.2: Illustration of problems in code hoisting

In this situation we have the expression $x + y$ present in nodes 3 and 4 and the expression $s + t$ present in nodes 3, 4 and 5. As a result, $x + y$ is considered very busy with respect to node 1 and $s + t$ is very busy with respect to nodes 0, 1 and 2. Thus, after application of the code hoisting procedure, $x + y$ will be duplicated in node 1. Careful inspection reveals that moving this expression from nodes 3 and 4 to node 1 as would seem to be indicated is unsafe as the expression would not be computed should control pass through node 2. This, however, is not the case. The common expression elimination heuristic, which is responsible for eliminating redundant expressions will recognize that an unsafe conditions exists and will not attempt to use the expression in node 1 to eliminate those in nodes 3 and 4. The net result is that the new expression in node 1 will be unreferenced and thus be eliminated in later processing.

A second subtlety exists with expression $s + t$. As this expression is very busy with respect to all three higher level nodes, it will be duplicated in all three. In this case, the duplicates in nodes 1 and 2 will be, in turn, found to be redundant with respect to the instance in node 0. Thus the expression will be correctly hoisted two levels to node 0 and all other instances will be eliminated.

4.3.4 Common Expression Elimination

All of the heuristics discussed thus far have involved the insertion of code into various points in basic block graphs. In the last two heuristics, the result was to deliberately introduce redundancies. There is a high probability that the first heuristic, inline subroutine expansion, would also have introduced redundancies. It is the task of the common expression elimination heuristic to set the stage

for the removal of these redundancies[1,2,12,28]. (Note: in most systems, the heuristic for common expression elimination does the actual removal of code. In GO, the heuristic merely changes the flow graphs so that later instances of common expressions are unreferenced. It is left up to the dead expression heuristic (discussed later) to actually remove expressions.)

The preceding heuristics, however, are not the only source of redundant expressions. Consider the following program fragment:

```
a = b + c * d;
b = a + b + c;
c = a + c * d;
d = a + c * d;
```

The code generated would likely be:

```
mull3   d,c,r0    # temp ← c * d
addl3   b,r0,a    # a ← b + temp
addl3   c,a,r0    # temp ← a + c
addl3   d,r0,b    # b ← d + temp
mull3   d,c,r0    # temp ← c * d
addl3   a,r0,c    # c ← a + temp
mull3   d,c,r0    # temp ← c * d
addl3   a,r0,d    # d ← a + temp
```

Here an obvious optimization would be to reuse the value computed for the subexpression $c * d$. This is the function of common expression elimination.

There are two variants of the common expression problem. The first deals with the elimination of redundancies *within* basic blocks and the second with elimination *between* basic blocks. The former is applicable in the above example.

The procedure for common expression elimination within a basic block is outlined in Algorithm 4.4. The procedure for performing this optimization between

Algorithm 4.4: CommonExprElim

PURPOSE: To eliminate common expressions within a basic block.

INPUT: A basic block.

OUTPUT: The block is modified so that any redundant computation is left unreferenced.

METHOD:

1. Locate the common expressions is the block under scrutiny. (Call the first instance E_1 and the second E_2) Common expressions have the following characteristics:
   (a) They belong to the same group in the global expression table.
   (b) None of the input operands are redefined between the occurrence of E_1 and E_2. This can be determined by examining the result operands of the interving instructions.
2. If the result operand of E_1 is redefined before the occurrence of E_2, create a new expression (E_n) based on E_1. The result operand of E_n is a fresh temporary. E_n is inserted into the basic block immediately after expression E_1 . Each expression on E_1's definition-use chain is then modified to reference E_n instead. Thus, E_1's definition-use chain becomes E_n's definition-use chain. E_1's chain is then set to empty.
3. Each expression in E_2's definition-use chain is then modified to reference E_1 (or E_n if it was created).

blocks is identical except that common expressions are identified by examining the AVAIL set.

When this procedure is applied to the above example, the resultant would be: (T is a fresh temporary variable).

```
mull3   d,c,r0    # temp ← c * d
mull3   d,c,T     # added instruction
addl3   b,r0,a    # a ← b + temp
addl3   b,T,a     # added instruction
addl3   c,a,r0    # temp ← a + c
addl3   d,r0,b    # b ← d + temp
mull3   d,c,r0    # temp ← c * d
addl3   a,r0,c    # c ← a + temp
mull3   a,T,c     # added instruction
mull3   d,c,r0    # temp ← c * d
addl3   a,r0,d    # d ← a + temp
```

As each of the added instructions masks the result of the instruction it is intended to replace, the result of the old expression is unreferenced and will be removed by the dead variable/expression heuristic.

4.3.5 Dead Variable/Dead Expression Elimination

The final code improvement heuristic performed is the elimination of useless expressions and variables. The primary sources of these are the other improvement heuristics. This heuristic can therefore be thought of as the final cleanup phase of the code improver.

Dead variables and expressions also occur "naturally" in unoptimized code. This is caused by the fact that subroutines often proceed through "stages" of execution. Each stage usually comprises a single loop or a series of nested loops. In each stage, two types of information are generated: that which will be used in

subsequent stages and "support" information which is used to actually implement the loops and carry out the necessary computations. The usefulness of the latter type ends with the termination of the stage it is associated with and may be safely discarded.

A variable is considered dead at the end of a block if no use is made of it between that point and the next definition of the variable. To determine whether a variable is alive or dead, one simply has to consult the LIVE set for the block. The presence of the variable in this set indicates that a subsequent use of the variable does occur and it should be left untouched.

The possibility exists that a variable will be dead on exit from a block, but still be alive between the exit point and its last definition *within* the block. To determine if this is the case, the local expression table is searched in reverse order until the definition of the variable or a reference to it is encountered. If a reference is encountered, no further processing is required at this stage. If no reference is found, then the variable is truly dead and the dead expression heuristic may be applied.

If the variable is indeed dead, then so is the expression that defines it. This is known to be true because if a later instance of the expression occurs before one of the expression components has been redefined, common expression elimination would have removed the later evaluation in favour of the current one. As this situation does not exist, there is no need to retain that particular instance of the expression, so it is removed from the local expression table.

4.4 Improvements During Recoding

Once the code improver has completed execution, all that remains is to recode the flow graphs into machine instructions. Strictly speaking, this is a minor utility task—GO's main objective has already been accomplished. However, there are a few code improvements that occur as a side effect of the recoding process.

The simplest of these is jump-to-jump elimination. As stated in the first chapter, this is normally considered a peephole optimization. However, one finds that it also fits nicely into the realm of control flow analysis. To perform this optimization, the recoding procedure need only watch out for basic blocks containing only an unconditional control transfer instruction. By definition, such constructs represent redundant jumps. To eliminate the branch, all that needs to be done is to replace the branch target address of the original transfer instruction with that of the redundant instruction.

Another major "free" optimization is the optimal use of local and temporary variables. The code improver is very generous in the allocation of temporary variables. It is not feasible to allocate real storage for each temporary. To circumvent this problem, the recoder generates a "map" of when a given variable contains valid information. It then attempts to "overlay" the various maps in order to find the best possible fit. In this way, it locates sets of variables whose valid periods are mutually exclusive. Such variables may be coalesced into single variable that is in use almost all of the time. The result is the best use of variable space possible.

Chapter 5

Summary

5.1 Incidental Points

The preceding chapters have discussed in broad terms the various mechanisms used to do code improvement in the GO system as well as descriptions of the overall implementation. There are some details of the implementation, however, that have not been addressed. These are discussed here.

5.1.1 Treatment of Pointers

A major problem in data flow analysis is the existence of *aliases*. Aliases exist if there are two variables which reference the same memory location. The problem with this situation is that a definition of one variable will result in a definition of both, however, this fact is not apparent to the data flow analyzer.

In GO, aliases manifest themselves in the form of pointer variables. GO cannot know if two pointers reference the same memory location or if a pointer references a given variable. An indirect reference through a pointer, then, could result in a hidden definition of or reference to some other variable.

Given this set of circumstances GO treats the pointer as though it references *all* variables. If a memory location is defined indirectly through a pointer, the result GO assumes that all variables in the program may have been set. If an indirect reference to a location is encountered, all variables are assumed to have been referenced.

This mechanism, of course, severely limits the usefulness of the data flow analysis process. A possible remedy to this problem is given below.

5.1.2 Treatment of Calls to the Operating System

A similar problem arises with calls to the operating system. How does one determine the side effects they cause? In GO, this is done by handling system calls as a special case of the general subroutine call mechanism. When GO is started, it builds a series of "dummy" call graph nodes, one for each operating system service. As the operations carried out in a system call are rigidly defined, it is possible to supply each system call node with complete data flow information. Thus it is possible to account for the effect of each system call.

5.2 Evaluation

Unfortunately, at the time of writing the GO code improver was still undergoing development so a number of key questions regarding performance must go unanswered. However, a number of qualitative observations may be made:

- In its current form, GO is strictly an experimental tool. Its deficiencies would render it useless in any production environment as it can only handle the simplest of programs.

- The emphasis in the design of GO is simplicity and generality. The implementation of GO stresses modularity of code. Each module is made as general as possible so as to allow its use in a number of capacities. This was accomplished by making use of "canned" utility functions and data

structures. The penalty for this is increased memory usage and execution time.

- Another major design decision was to attempt to incorporate as much machine independence as possible. This was accomplished by using a table driven instruction decoder and defining a set of machine independent "basic operations" which were implemented as small machine dependent routines. An unexpected result of this decision was a restriction on the number and type of optimizations which could be performed. This is primarily because without knowledge of what each instruction in a block actually does, all code improvement must be done through simple pattern matching.
- The design decision that makes the greatest impact, however, is the decision to attempt to directly optimize machine executable code. It effects almost every facet of the system. The merits of this decision will be discussed later. Regardless of any difficulties this decision caused, though, the basic philosophy is sound: flow analysis of a executable program using no supplemental information from the code generation system *is* feasible.

5.3 Possible Extensions to GO

As noted above in an attempt to make GO relatively machine independent, the ability to perform a number of optimizations was lost. This is due to the fact that GO cannot interpret what each instruction does, and thereby discover its effect, except in the most general terms.

The most obvious method of enabling GO to "understand" the expressions it sees is to code some knowledge of the target machine instruction set into the

code improvement heuristics. Unfortunately this would make the system almost completely machine specific. An elegant alternative technique exists which would preserve some measure of machine independence by sacrificing efficiency.

This technique involves supplementing GO's instruction description tables (which are used to properly decode the input program) with descriptions of how each instruction operates. One way in which this could be done would be to supply a simple stack machine description of each instruction. This information would be used by a stack machine emulator which operates as part of the data flow preprocessor.

The major benefit resulting from this mechanism is the ability to completely determine the values of computations involving constants. This permits the removal of some of the more severe restrictions placed on the system due to the presence of pointers. To understand how this is possible requires some consideration of the manner in which pointers are used.

In general, pointers have two uses. The first use is as a position marker. This may be done explicitly by the programmer to implement complex data structures such as linked lists or implicitly as with array and string references. The second use is to introduce generality into code. By using a pointer, it is possible to merge two or more sections of code which do the same computations, but on different sets of data. A classic example of this is a "pass by reference" procedure parameter.

In either of these cases, *the contents of the pointer is a value that is determined at compile time.* The number of applications in which the value of a pointer could be legitimately obtained from an external source is very limited. Furthermore, *all computations involving pointers deal invariably with constants.* However, this

might not always be readily apparent. For example, if an array bounds is read in from an external source, good programming technique dictates that a range check be performed before attempting to use it. The range check values must be constants. Thus they implicitly define the values which may be accepted as the index.

In the discussion of control flow analysis, it was pointed out that a major flaw in the control flow analysis procedure GO employs is its inability to handle computed control transfer addresses. Using this stack machine system and capitalizing on the above observations, the solution to this problem becomes trivial. One need only use the techniques for propagation of constants to determine what the possible values of any given pointer are at any time during the execution of a program. This technique is equally applicable to pointers to variables. Thus it is possible to circumvent the problems with aliases that were outlined earlier in this chapter.

5.4 Concluding Remarks

The need for code improvement systems and research into such systems is obvious. That this should be accomplished via flow analysis of machine executable object modules is less obvious. There are, however, valid reasons for doing it in this manner. To understand these reasons, one must consider the history of the computing industry.

The information processing sciences and related disciplines are relatively new with respect to the more established areas of study. It has only been in the

past decade, however, that any really fundamental change has been made in the industry.

At one time, everything associated with computers was big: they were expensive, large and slow. Support in terms of software for computers came principally from two sources, the manufacturer and the system support staff. If any third party software support was to be found, the software was written in either assembler or in a primitive high level language for which the manufacturer supplied the development system. As the language compilers and optimizers were supplied by one party, it was possible to devise intricate optimizers which were an integral part of the overall system.

The computer industry of today is a vastly different. The advent of microelectronics and inexpensive "personal" computers has given rise to a situation where systems produced by different manufacturers have essentially the same design. The emphasis has passed from hardware manufacturer supplied software to third party support. For any given system, it is possible to find any number of compilers by different software manufacturers. Unfortunately, in such a case as this, there are only two sets of standards to which the compiler has to adhere: the standards for the input language and those of the machine on which the program has to run. And even the former need not be strictly adhered to. It is highly unlikely that the intermediate representations of programs generated by any two third party compilers is the same. Furthermore, as it is now profitable to market software systems alone (as opposed to producing software as an incentive to the selling of hardware as was the case in the past) it is not in the best interests of the compiler writer to reveal the details of his implementation. This includes

information about the intermediate code, symbol table and linkage information. An individual writing a code improvement system, then, has very little to work with.

There is only one common denominator with respect at all the compiler systems available. That is the architecture of the machine for which all the systems must produce code. In other words, the final executable image is the only form of the program for which there is guaranteed to be information available.

It is for this reason that investigation of the direct optimization of machine executable code, as in GO, is a necessary and inevitable endeavour.

References

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[3] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Reading, MA, 1978.

[4] F. E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970.

[5] F. E. Allen. Interprocedural data flow analysis. In J. L. Rosenfeld, editor, *Proceedings IFIP Congress*, pages 398–402, International Federation for Information Processing, North-Holland, Amsterdam, 1974.

[6] F. E. Allen. A method for determining program data relationships. In *Lecture Notes in Computer Science Volume 5: International Symposium on Theoretical Programming*, pages 299–308, Springer-Verlag, Berlin, 1974.

[7] F. E. Allen. Program optimization. In M. I. Halpern and C. J. Shaw, editors, *Annual Review in Automatic Programming, Volume 5*, pages 239–307, Pergamon Press, Headington Hill Hall, Oxford, 1969.

[8] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, Prentice Hall, Englewood Cliffs, NJ, 1972.

[9] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[10] J. E. Ball. Predicting the effects of optimization on a procedure body. *SIGPLAN Notices*, 14(8):214–220, August 1979.

[11] J. M. Brady. *The Theory of Computer Science.* Chapman and Hall, London, 1977.

[12] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970.

[13] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, October 1984.

[14] Digital Equipment Corporation. *VAX Architecture Handbook.* 1981.

[15] M. C. Er. Optimizing procedure calls and returns. *Software—Practice and Experience*, 13:921–939, 1983.

[16] A. Fong, J. Kam, and J. D. Ullman. Application of lattice algebra to loop optimization. In *Second ACM Symposium on Principles of Programming Languages*, pages 1–9, Association for Computing Machinery, 1975.

[17] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, NY, 1977.

[18] M. Hopkins. An optimizing compiler design. In C. V. Freiman, editor, *Proceedings IFIP Congress*, pages 391–396, International Federation for Information Processing, North-Holland, Amsterdam, 1971.

[19] K. Kennedy. Safety of code motion. *International Journal of Computer Mathematics*, 3(2):Sec. A, 117–130, 1971.

[20] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[21] D. E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1:105–133, 1971.

[22] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[23] E. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[24] W. C. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, July 1965.

[25] R. F. Reiser and R. R. Henry. Berkeley VAX/UNIX assembler reference manual. In *UNIX Programmer's Manual, Volume 2C*, University of California, Berkeley, 1983.

[26] B. K. Rosen. High-level data flow analysis. *Communications of the ACM*, 20(10):712–724, October 1977.

[27] T. A. Standish, D. C. Harriman, D. F. Kibler, and J. M. Neighbors. Improving and refining programs by program manipulations. In *Proceedings of the ACM Annual Conference*, pages 509–516, Association for Computing Machinery, 1976.

[28] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3):191–213, 1973.

[29] J. D. Ullman. A survey of data flow analysis techniques. In *Proceedings of the 2nd USA-Japan Computer Conference*, pages 335–342, AFIPS Press, Montvale, NJ, 1975.

Appendix

An Overview of VAX Architecture

The VAX series of computers comprise a range of machines of varying power which are generally classed as "supermini" computers. All of the machines in the series share the same instruction set.

A.1 Principal Architectural Features

A.1.1 Instruction Set

The VAX is described as a complex instruction set computer (CISC) with a 32 bit architecture. In addition to the basic arithmetic/logical, test and branch instructions, it has a number of specialized instructions for high level language support (loop implementation, string manipulation and array handling), operating system support (queue manipulation) and scientific calculations (polynomial evaluation). The instructions take from zero to six operands and all of the arithmetic operations have two operand ($x = x + y$) and three operand ($x = y + z$) forms. The arithmetic operations are also orthogonal with respect to the integer and floating point data types (i.e. any arithmetic operation can be performed on any numeric data type).

A.1.2 Data Types

The basic data types supported are integers, packed decimal strings, character strings and floating point numbers. The principle sizes of integers are byte (8

bits), word (16 bits) and long word (32 bits). The main floating point data types are F-floating (single precision — 32 bits: 1 bit sign, 23 bits mantissa and 8 bits excess-128 exponent) and D-floating (double precision — 64 bits: 1 bit sign, 55 bits mantissa and 8 bits excess-128 exponent). Both floating point formats represent the mantissa as a normalized binary fraction (i.e. high order bit is always a one) with the high order bit not stored. The effective size of the mantissa is therefore increased by one bit (24 bits for F floating, 64 bits for D floating).

A.2 Programming Features

A.2.1 The Stack

A principle feature of the VAX architecture is the *stack*. The stack is a last-in first-out data structure which is used for transmission of arguments between subroutines, for subroutine linkage and for temporary storage. The stack "grows downwards". That is, new elements are placed on the stack at memory addresses which are less than older members. The machine maintains a *stack pointer* (see below) which indicates the location of the most recently "pushed" value. Under the UNIX operating system, the stack is allocated at the highest point in the user's address space and is automatically extended as more items are place in it.

A.2.2 Registers

The VAX has 17 CPU registers which are of interrest to most programmers. The processor status word (PSW) is a 16 bit register which controls the operation of the CPU and records various aspects of CPU state (e.g. if an arithmetic overflow condition exists). The remaining 16 registers are known as the "general purpose"

registers and are all 32 bits wide. They are general purpose in that that the instruction set is orthogonal with respect to their use as opposed to true generality of use. The first twelve registers (R0 through R11) can be termed "all purpose" registers. They are available for any function desired by the programmer, however, some of the specialized instructions (especially those which manipulate strings) alter one or more of the first six registers (R0 through R5) indiscriminately. The other four registers have the following functions assigned to them by the hardware:

Argument Pointer (AP or R12)

The specialized procedure call instructions use this register to point to the list of arguments being transmitted to a subroutine. It is automatically saved on procedure entry and restored on exit.

Frame Pointer (FP or R13)

This register is set up on procedure entry to show the location on the stack of the local (automatic) variables and other data structures which are unique to a particular instance of a procedure. Like the argument pointer, it is also preserved across procedure calls and is maintained by the procedure call and return instructions.

Stack Pointer (SP or R14)

The SP points to the last item placed on the machine stack.

Program Counter (PC or R15)

The program counter points to the next instruction memory which is to be executed (during instruction processing, it points to the next byte in memory of the current instruction).

A.2.3 Addressing Modes*

One of most appealing aspects of the VAX architecture is its rather elegant set of addressing modes. The addressing modes are, for the most part, orthogonal with respect to all instructions and all of the general purpose registers. That is, any address mode may be used for any operand of any instruction (with certain obvious limitations; for example, it is not possible to use any of the "constant-contained-in-instruction-stream" modes to specify an operand which is to be written). The addressing modes are summarized below.

Literal/Immediate

The operand is a constant value which is specified in the instruction stream. In the case of a literal, the constant is a small value which is encoded in the addressing mode specification itself. Immediate values are kept in the memory locations following the instruction. (Immediate mode is actually a shorthand notation for autoincrement mode used with the PC.) Syntax: `$`*constant*

Register

The operand is contained in register *n*. Syntax: `R`*n*

Register Deferred

The *address* of the operand is contained in register *n*. Syntax: `(R`*n*`)`

*Note: The syntax for the various addressing modes used is that of the standard VAX/UNIX assembler (see [25]) and not the ones used by the manufacturer as given in [14].

Autodecrement

The value of register n is decremented by the size of the operand (1 for byte, 2 for word, etc.) and the value is then used as the address of the operand. Syntax: `-(Rn)`

Autoincrement

The operand's address is contained in register n. After accessing the operand, the value of the register is incremented by the size of the operand. Syntax: `(Rn)+`

Autoincrement Deferred

The same as autoincrement, except that the register contains the address of the address of the operand (two levels of indirection). Syntax: `*(Rn)+`

Byte/Word/Longword Displacement

The address of the operand is the value of the sum of the (byte, word or longword sized) displacement and the contents of register n. Syntax: *X' disp*`(Rn)` where *X* is 'B' for byte displacements, 'W' for word displacements or 'L' for longword displacements.

Byte/Word/Longword Displacement Deferred

The same as the regular displacement modes, except that the value of the sum is the address of the address of the operand. Syntax: **X' disp*`(Rn)`

Absolute

Address is the absolute address in memory of the operand. This is actually a shorthand notation for autoincrement deferred mode with the PC. Syntax: **$address*

Byte/Word/Longword Relative

The instruction stream contains the offset of *Address* from the current PC value. This is a shorthand notation for the regular displacement modes used with the PC. Syntax: *X' disp*

Byte/Word/Longword Relative Deferred

The same as the regular relative modes except that an additional level of indirection is imposed. This is a shorthand notation for the deferred displacement modes with the PC. Syntax: **X' disp*

A.2.4 Machine Instructions

Listed below are most the VAX instructions which may be found in the examples in this thesis. A complete list of VAX instructions can be found in [14]. Details on features unique to the VAX/UNIX assembler can be found in [25].

MOVE

The move instruction transfers a the value one operand to another operand. There are several variants of the move instruction, one for each data type. The mnemonic for the move instruction `mov` with a single character suffix to specify operand type. Examples: `movb x,y` means "move a byte sized value from x to y and `movw $1,a` means "move the constant value 1 to the variable a".

PUSH LONG WORD

The push long word operand moves the value of its single operand onto the stack. This instruction is typically used to move constant values and

the contents of simple variables onto the stack as a prelude to subroutine invocation. The mnemonic for this instruction is `pushl`. Examples: `pushl $123` means place the value 123 onto the stack and `pushl xyz` means place the value of variable *xyz* on the stack. Note: this operation may also be performed using a move instruction: `movl r0,-(sp)`.

ADD

There are a number of variants of the add (perform two's complement addition) instruction. The two principle forms are add two operand and add three operand. The two operand instruction adds the value of its first operand to the value of its second operand, leaving the result in the second operand. The three operand variant leaves the result in its third operand. For both of these forms, there are variants for each type of operand. The mnemonic for the add instruction is `add` with an additional character indicating type of operand and either a "2" or a "3" to indicate number of operands. Examples: `addl2 a,r0` means add the value of a to the value in register zero and leave the result there and `addl3 $2,x,y` means add the constant value 2 to x with the result placed in y.

SUBTRACT

The subtract operation has the same set of variants as add instruction. The mnemonic for subtract is `sub` with the appropriate affixes. Examples: `subl3 s,t,u` and `subl2 (r0),abc`.

INCREMENT

This instruction simply adds the constant one to its single operand, leaving

the result there. It has the same variants with respect to type as the other arithmetic/logical instruction. The mnemonic is `inc` with the appropriate suffix for type. Example: `incw r9`.

DECREMENT

The same as the increment instruction, except the constant one is subtracted. Mnemonic: `dec` with suffix. Example: `decb -(r6)`.

TEST

The value of the single operand is compared to zero and the machine condition code set appropriately. No other action is performed. The instruction has variants for all data types. Mnemonic: `tst` with type suffix. Example: `tstw r5`.

COMPARE

The values of the two operands are compared to each other. The condition codes are set as appropriate with no other action taken. There are variants for each data type. Mnemonic: `cmp` with type suffix. Example: `cmp $43,foo`.

JUMP IF NOT EQUAL

This instruction causes a conditional branch (based on the current condition code settings) to the target address specified by the operand if the result of the last arithmetic operation or comparison was non-zero (not equal). Unlike the operands for most other instructions, the operand is always a program counter relative displacement. Mnemonic: `jneql`. Example: `jneql label`.

JUMP IF EQUAL

The same as the `jneql` instruction except that the branch conditions are reversed. Mnemonic: `jeql`.

UNCONDITIONAL BRANCH

This instruction always causes a transfer of control to the target address specified by its operand. The same restrictions with regard to operands exist with this instruction as with the other branch instructions. Mnemonic: `jbr`.

Note: The branch instructions detailed above are actually pseudo instructions to the UNIX assembler. There are actually a number of variants of each instruction with differing branch displacement sizes. The assembler chooses the correct displacement size based upon distance to the target address.

CALL PROCEDURE WITH STACK ARGUMENTS

A standard VAX procedure call is initiated. This entails the construction of a new stack frame, the saving of registers and the set up of the argument pointer. The arguments for the subroutine are placed on the stack prior to execution of this instruction. Two operands are supplied. The first is the number of arguments which are to be transmitted. The second operand is the address of the target subroutine. Mnemonic: `calls`. Example: calls $2,subr.

RETURN FROM PROCEDURE

This instruction terminates a procedure call which was initiated by a `calls` instruction. The calling routines registers are automatically restored and

current stack frame is discarded. All elements of saved machine state are also restored. In addition, the space on the stack taken up by the procedure parameters is automatically recovered. This instruction takes no operands, all necessary information is retrieved from the stack frame. Mnemonic: `ret`.