Structuring Assembly Programs

Douglas Simon
Supervisor: Cristina Cifuentes

31st October, 1997

An automated assembly to high level language (e.g. C) translator would be of great benefit to maintainers of legacy assembly programs. A component of such a translator would derive higher level constructs such as conditional, loops and sequences from the original source. This can be done by applying an algorithm to structure the control flow graph of the assembly source. This is commonly referred to as a structuring algorithm.

Most existing techniques for control flow structuring not easily adapted to structuring assembly code. During research into the feasibility of a binary decompiler, an algorithm more suited to the task of structuring assembly code was developed. This seminar focuses on a number of modifications to this algorithm that were developed as part of an honour's thesis. In particular we discuss techniques for generating 'better' high level code from unstructured assembly control flow graphs. Unstructured graphs are those that will require the use of gotos in the high level code. The quality of the code can be measured by the number of gotos generated.

In addition to our attempt to improve the functionality of the original algorithm, we also seek to improve the runtime speed of the translator by investigating less expensive data structures. Ideally, any improvement in speed will not involve a trade off with functionality.

We present the results of a prototype assembly structuring tool. The results were generated by running the tool on a number of well known, large benchmark programs used by those in the compiler community. From these results, we discuss what combination of improvements in speed and functionality are optimal in some well defined sense.

Contents

1 Introduction **12**

1.1 Assembly to High-Level Language Conversion 12

1.1.1 Data Flow Analysis 15

1.1.2 Control Flow Analysis 15

1.1.3 HLL Code Generation 19

1.2 Goal and Primary Motivation 19

1.2.1 Other Benefits 20

1.3 Outline for Remaining Chapters 21

2 Related Work **23**

2.1 The Nature of Unstructuredness 23

2.2 Graph Reductions and Transformations 25

2.3 Semantic Preserving Structuring 28

2.4 Assembly Control Flow Graph Structuring 31

3 Assembly to High Level Language **33**

3.1 Assembly Parsing 34

3.2 Control Flow Generation 36
3.2.1 Basic Blocks 36
3.2.2 Graph Theory 37
3.2.3 Control Flow Graphs 38
3.2.4 Unreachable blocks 40

4 Structuring Assembly Graphs 42

4.1 Notation 42
4.2 Graph Structuring 43
4.3 Control Flow Graph Terminology 44
4.3.1 Interval Theory and the Derived Sequence of Graphs . 46
4.3.2 Parenthesis Theory 48
4.4 Structuredness 51
4.4.1 Loops 51
4.4.2 2way Conditionals 53
4.4.3 Nway conditionals 54
4.4.4 Multi-structure headers 55
4.5 Structuring Algorithms 56
4.5.1 Determine Node Ordering 58
4.5.2 Determine Reverse Ordering 59
4.5.3 Immediate Post Dominators 60
4.5.4 Structuring Conditionals 64
4.5.5 Loop Structuring using Derived Sequences of Graphs . 68

4.5.6 Loop Structuring using Parenthesis Theory 76

4.6 Restructuring 2way Conditionals 82

5 Code Generation 88

5.1 Notation and Conventions Used 90

5.2 Generating Code for Loops 91

5.2.1 Generating Code for a Loop Header 92

5.2.2 Generating Code for a Loop Body 92

5.2.3 Generating Code for a Loop Trailer 93

5.2.4 Generating Code for a Loop Follow 94

5.3 Generating Code for 2way Conditionals 94

5.4 Generating Code for Nway Conditionals 96

5.5 Generating Code for Non-Compound Structures 98

5.6 The Complete Algorithm 99

6 Implementation and Results 102

6.1 ast 102

6.1.1 Implementation Parameters 102

6.1.2 Delayed Instructions 103

6.1.3 Preprocessor 105

6.1.4 Data Representation 107

6.2 Testing and Results 107

6.2.1 Test Inputs 107

6.2.2 Comparisons for Quality of Code Generated 108

6.2.3 Resource Usage 114

7 Conclusion **117**

A Attribute definitions **120**

A.1 Base Node Attributes 120

A.2 Predicates 122

A.3 Attributes for Other Objects 122

A.4 Derived Sequence of Graphs Attributes 123

A.5 Parenthesis Theory Attributes 123

List of Figures

1.1 Structure of an assembly to high level language translator along with its inputs and outputs. 13

1.2 A simple assembly program that calculates the greatest power of two less than 50. 14

1.3 The result of translating the source in Figure 1.2 into C code. 14

1.4 Assembly code using conditional jumps for control flow. 16

1.5 The CFG representation of (a) the code in Figure 1.4 and (b) an **if..then** statement. 16

1.6 A derived sequence of graphs. 17

1.7 Sample unstructured assembly code and its associated CFG representation. 18

2.1 The backward path in a loop is just a specialisation of a standard loop. 24

2.2 An example of parallel loops. 25

2.3 An example of the two reductions used for structuring in the Pep, Java to Self bytecode translator. 26

2.4 The graph transformation rule for an **if..then..else** statement. 27

2.5 (a) Original CFG and (b) CFG with extra loop header node added. 29

2.6 A graph with the reach set shown for each node. 30

2.7 Two possible CFG representations for a compound boolean expression. The short-circuited representation is given on the right hand side of the figure. 32

3.1 The phases of an assembly to HLL translation. 34

3.2 A sample assembly program. The basic block partitioning is shown by the table on the right hand side. 40

3.3 The control flow graph for the program given in Figure 3.2 . . 41

4.1 HLL control structures and examples of their control flow graph representations. 44

4.2 Interval building algorithm. 47

4.3 Derived Sequence of Graphs building algorithm. 48

4.4 The Derived Sequence of a graph with the intervals of the initial graph and each derived graph shown by the dotted boxes. 49

4.5 An example of an irreducible graph. The final graph is the canonical irreducible graph. 49

4.6 Algorithm used to set the parentheses for each node. 50

4.7 Sample unstructured loops. 52

4.8 Unstructured 2way conditional graphs. 54

4.9 Unstructured nway conditionals. 55

4.10 (a) A structured and (b), an unstructured graph in which node 1 is the header of two different control structures. 56

4.11 The phases performed during structuring. Branches taken by PT algorithms are labelled 'PT' and branches taken by DSG algorithms are labelled 'DSG'. 57

4.12 Algorithm to establish the reverse post-order within the nodes of the original CFG. 58

4.13 Algorithm to establish the reverse post-order for the nodes within the reverse CFG. 59

4.14 A control flow graph with the nodes identified by their post-order value. The table gives the post-order of each node in the reverse graph. The nodes in the 'Node' column correspond with the nodes in the graph shown. 60

4.15 Algorithm for calculating the immediate post-dominators. . . 61

4.16 Common post-dominator algorithm. 62

4.17 A CFG that has different reverse graphs. 63

4.18 The immediate post-dominators for each node in the graph of Figure 4.14. 64

4.19 A control flow graph with the potential conditional nodes shaded. 65

4.20 Algorithm to Tag the Nodes within an Nway Conditional. . . 67

4.21 Conditional Structuring Algorithm 68

4.22 The intervals of the Control Flow Graph of Figure 4.14 69

4.23 Derived Sequence of Graphs for Figure 4.22 70

4.24 A loop header with two potential latching nodes. 71

4.25 Loop Structuring Algorithm. 72

4.26 A loop that requires the use of intervals to derive the correct membership. 73

4.27 Algorithm to tag the nodes within a loop. 74

4.28 Algorithm for determining the type of a loop. 75

4.29 Algorithm for determining the follow node of a loop. 76

4.30 Loop Structuring Algorithm. 78

4.31 The (a) true and (b) false parenthesis for each node in the graph of Figure 4.14. 79

4.32 Algorithm to tag the nodes within a loop. 80

4.33 An Endless loop. 82

4.34 Algorithm for determining the follow node of a loop. 83

4.35 HLL code generated by an existing code generation algorithm. . . 84

4.36 HLL code generated as a result of 2way conditional restructuring. . 85

4.37 Algorithm to Restructure 2way Conditionals 87

5.1 The HLL structure type determined for each node in the reproduced control flow graph of Figure 4.14 with respect to the DSG structuring algorithms. For 2way nodes, the true branch is the left one. 89

5.2 Algorithm to Generate Code for a Loop Header Rooted Graph 95

5.3 Algorithm to Generate Code for a Loop Header Rooted Graph — (cont) 96

5.4 Algorithm to Generate Code for for 2way Rooted Graph . . . 97

5.5 Algorithm to Generate Code for a Nway Rooted Graph 98

5.6 Algorithm to Generate Code for a Non-compound Rooted Graph 99

5.7 Algorithm to Generate Code for a Control Flow Graph. 101

6.1 CFG demonstrating the semantics of a delayed instruction in the context of the annul flag. 104

6.2 Solution to the delayed instruction dilemma. 105

6.3 Assembly program before (left) and after (right) preprocessing. Each instruction has been given a numeric label for easier reference. 106

6.4 The programs used for testing along with the instruction count and graph node count. 108

6.5 The C style high level code produced by $\mathbf{ast_{PT}}$ 110

6.6 The C style high level code produced by $\mathbf{ast_{DSG}}$ 111

6.7 Number of HLL 2way conditionals generated. 112

6.8 Number of HLL nway conditionals generated. 112

6.9 Number of HLL loops generated. 113

6.10 Number of **goto**s as a percentage of the total HLL control structures generated. 113

6.11 Memory cost of DSGs 115

6.12 Memory cost of DSGs / number of reachable nodes. 115

6.13 Time taken to build the DSGs as a percent of the total time to perform the structuring. 116

Chapter 1

Introduction

In the dynamic area of software development, there is a continual shift to the use of new programming languages. New design paradigms such as object oriented design are a catalyst for the creation of many new languages. Other reasons include advancements in the lower level hardware that allow for more expressive and powerful languages to be developed without compromising efficiency. To ease the task of maintaining software written in older, legacy languages, it is desirable to have an automatic translator that converts the original program into an equivalent program coded in one of the newer languages. The translation could be either from an older high level language (such as Algol) or from a lower level language (such as assembly). This thesis presents and discusses some of the issues involved in the design and implementation of such a translator. We will be concentrating on translating assembly code into a higher level language such as C. In particular, we will focus on the component that translates assembly statements that transfer flow of control into equivalent high level language control flow statements such as loops and conditionals

1.1 Assembly to High-Level Language Conversion

The overall structure of the process of translating assembly to a high level language (HLL) is shown in Figure 1.1. The box represents the represents the translation process as a whole and anything outside it shows its external

inputs and outputs. In this respect, the translation process can be conceived of as a black box into which we can feed assembly programs and it will output HLL programs.

The *parser* reads in the assembly input into memory. We assume the input to be syntactically correct, this is a straight forward task that represents little more than string processing. We therefore do not concentrate on this phase any further from a theoretical point of view. Parser techniques are covered in any compiler's text book such as [14].

The *data flow analysis* phase is described in Section 1.1.1, the *control flow analysis* phase in Section 1.1.2 and the *HLL generation* phase in Section 1.1.3.

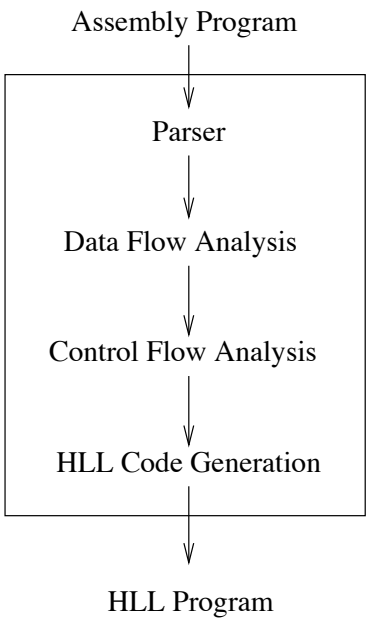


Figure 1.1: Structure of an assembly to high level language translator along with its inputs and outputs.

As a simple example of the functionality embodied within the translation process, consider the source assembly program in Figure 1.2. This is a short program that calculates the greatest power of two less than 50. Due to the somewhat cryptic nature of assembly code, comments have been included to clarify what each instruction is doing. The assembly language used in this example (and all examples hereafter) is the SPARC Version 8 assembly language. Comments follow the ";" symbol.

```
main:   mov 2,%g2       ; %g2 := 2
        mov 50,%g3      ; %g3 := 50
        sll %g2,1,%g2   ; %g2 := %g2 * 2
.LL11:  cmp %g2,%g3     ; %g2 < %b3
        bl .LL11        ; goto .LL11 if %g2 < %b3
        sll %g2,1,%g2   ; %g2 := %g2 * 2
        retl            ; return
        nop             ; do nothing
```

Figure 1.2: A simple assembly program that calculates the greatest power of two less than 50.

From this assembly program, the translator should be able to generate a source listing similar to the one in Figure 1.3. As can be observed, the target program makes use of generic variable names, which assumes a complete absence of any symbolic information within the input program. That is, the assembly input does not contain any names for the memory locations it manipulates. This is not always the case as programmers often use variable names as a means of abstraction in any language, whether it be assembly or some high level language.

```
main()
{
        int loc1;
        int loc2;

        loc1 = 2;
        loc2 = 50;
        while (loc1 < loc2)
                loc1 = loc1 * 2;
}
```

Figure 1.3: The result of translating the source in Figure 1.2 into C code.

Without introducing complex analysis, a reasonable level of semantic-recovery could be performed on assembly programs devoid of any symbolic information by using generic names derived from the type of a variable (e.g. string1, string2, etc.) if the type itself can be determined. Anything more would require an almost artificial intelligence level of analysis. Alternatively, the

process could be semi-automated and make use of human input to derive meaningful names where required.

1.1.1 Data Flow Analysis

The data flow analysis phase attempts to transform the use of machine registers and condition codes into high level expressions and assignment statements. This reason for the transformation is that direct register use is not a feature of any HLL[1] and condition codes are represented as boolean expressions. Data flow analysis also performs a number of other transformations on the source assembly code such as condition code propagation. The conditional branch instructions use condition codes (usually stored as bits in a reserved register) as the means of making decisions about whether or not to branch. Collapsing the setting of the condition codes and their subsequent testing into one instruction can make it easier to generate more meaningful test expressions for the HLL control constructs. Other examples of data flow analysis transformations/optimizations as well as the algorithms used to implement them are given in [5].

1.1.2 Control Flow Analysis

The third phase in the assembly to HLL translation process, control flow analysis, is the primary focus of this thesis. We want to analyse the control flow graph (CFG) of an assembly program and recognise patterns that are equivalent to the patterns representing HLL control structures. As an example, consider the assembly code in Figure 1.4. For clarity hereafter, a row of dots has been used to represent a non-empty, arbitrary length sequence of instructions, not including any control transfer instructions (i.e. jumps, conditional jumps, procedure calls etc.). Graph node boundaries are also shown in the comments as `N`n `begin` and `N`n `end` where n is the identifier of a node. Figure 1.5 shows the CFG for the assembly code sample as well as the standard CFG representation of an **if..then** statement. The `t` and `f` labels on the first CFG denote the branch taken when the condition is true and false respectively. The `t` label on the second CFG denotes that the branch to

[1] C allows the programmer to recommend that the value of a given variable is stored in hardware registers instead of in general memory. However, this does not constitute a direct use of registers.

the **then** clause is the true condition. The observation that the two CFG's are identical enables us to see that the assembly code can be interpreted as an implementation of a higher level construct (i.e. an **if..then** construct).

```
        ...             ; N1 begin
        ble .LL2        ; N1 end
        ...             ; N2 begin
        call printf,0   ; N2 end
.LL2:   ret             ; N3 begin
        restore         ; N3 end
```

Figure 1.4: Assembly code using conditional jumps for control flow.

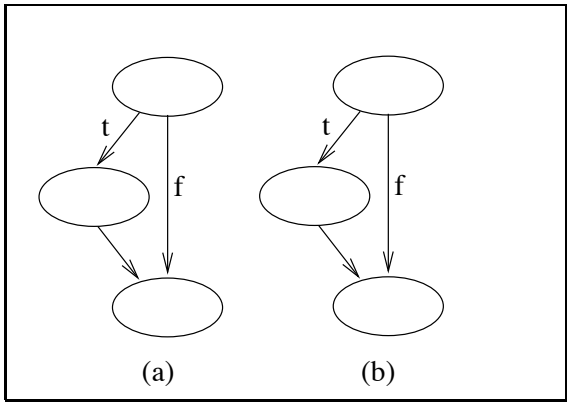


Figure 1.5: The CFG representation of (a) the code in Figure 1.4 and (b) an **if..then** statement.

Control flow analysis is implemented as a set of structuring algorithms that try to discover subgraphs that can be interpreted as the flow graph representation for a set of generic high level constructs such as **if**, **while** and **repeat** statements. There will also be CFGs whose control flow cannot be expressed purely in terms of the HLL constructs. These *unstructured graphs* present a non-trivial problem when trying to structure them and this thesis presents certain techniques that can be employed to generate HLL code from them.

Structured Control Flow Graphs

A control flow graph is *structured* if it is composed entirely of subgraphs that represent a generic set of HLL control structures. In other words, it can be

transformed into a trivial graph (i.e. a one-node graph) by repeatedly replacing instances of the structured subgraphs with a single node. A structuring algorithm is relatively straight forward to design for programs which have a structured CFG representation.

Figure 1.6 shows a sequence of transformations applied to a structured CFG to produce a trivial graph. The original graph is shown in (a). In graph (b), the **if..then..else** structure represented by nodes B, C and D in graph (a) is collapsed to node G. The sequence structure in graph (b) represented by nodes A, G, E and F is then collapsed to the single node H in graph (c). This node now represents a *self* loop which can be collapsed to a single node, I, in graph (d) which is now a trivial graph. The information collected during the transformations now tells us that the original CFG had a loop which enclosed an **if..then..else** structure followed by two sequential nodes.

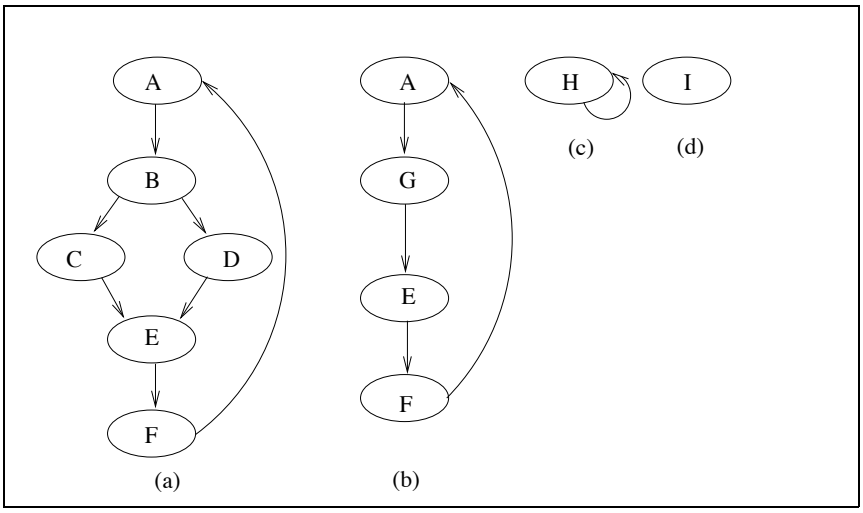


Figure 1.6: A derived sequence of graphs.

Unstructured Control Flow Graphs

We cannot assume that the CFG for all assembly programs will always be structured. In fact, most programs hand written in assembly will not be structured. One of the motivations for writing code in assembly is that the programmer is not as constrained by structured programming techniques as they would be when writing in a HLL. It may also be the only way to interface directly with the hardware. This gives the programmer the option

to generate smaller and faster code but at the expense of structuredness and, quite often, readability. Compilers will also invariably generate unstructured assembly code whenever optimization is performed. Figure 1.7 is an example of unstructured assembly code. The CFG shown for this assembly code is unstructured as the loop containing the nodes 2, 3 and 4 have two entry edges (i.e the two edges from node 1).

```
.LL1:   ...          ; N1 begin
        ble .LL3     ; N1 end
.LL2:   ...          ; N2 begin and end
.LL3:   ...          ; N3 begin
        b .LL4       ; N3 end
.LL4:   ...          ; N4 begin
        b .LL2       ; N4 end
```

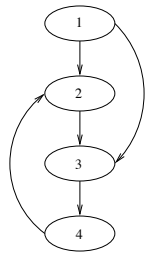


Figure 1.7: Sample unstructured assembly code and its associated CFG representation.

There are a number of possible solutions when faced with unstructured control flow graphs. The first is to duplicate nodes within the graph and continue doing so until you have a structured graph. In terms of the underlying assembly program, this will be reflected by code duplication. Another technique is to introduce boolean variables into the underlying code. Both of these techniques involve altering the semantics of the original source code. For our purposes, this is undesirable as wish to preserve the semantics of the original source code. This is because both code duplication and the introduction of boolean variables usually make the resulting code more unreadable than if we had not used these techniques.

An alternative method is to derive as many high level control structures as possible and allow the use of a **goto** statement when necessary. By using this technique, we do not alter the semantics of the original source code. This takes into account that it may well have been the case that a **goto** would have been the programmers choice for some situations, even if she/he had the option of higher level constructs.

1.1.3 HLL Code Generation

The final phase of the assembly structuring process is the generation of HLL code, based on the CFG after it has been annotated with the relevant structuring information. A type of post-structuring analysis can also be performed during this phase to derive extra HLL constructs such as **continue** and **break** statements.

1.2 Goal and Primary Motivation

We describe herein what the goals of this thesis are and why the research involved in attempting to attain these goals is justified.

As part of the work in [5], a structuring algorithm was designed that uses the last method described in Section 1.1.2. It transformed an arbitrary (structured or unstructured) CFG into a functionally and semantically equivalent flow graph. The HLL control constructs chosen for the structuring represent a set of generic constructs found in most commonly used HLL's. Specifically, it detected simple and compound statements, 2 way conditionals (**if..then..else** [2]), n-way conditionals (**case** and **switch** statements), pre-tested loops (**while** loops), post-tested loops (**repeat** and **do** loops), endless loops and **goto** statements. Each of the loop types was also allowed to have multiple exits as long as every exit jumped to the same statement (i.e. the first statement after the loop). This captured the functionality of the **break** statement found in C.

Apart from the data structures needed to simply represent a CFG, the above algorithm makes use of two extra data structures to do its analysis - *intervals* and a *derived sequence of graphs*. We will not go into the details of these data structures now except to point out that the use of any reasonably sized data structure can add a significant overhead in terms of both space (memory) and performance to an algorithm. Even though a tool implementing these algorithms may be used only infrequently, for large inputs, it is expected that more resource efficient algorithms will give a significant improvement. It is also reasonable to assume that these algorithms may be integrated into the development of tool that provides other functionality as well. For example, a

[2] The algorithm is able to distinguish between the two different forms of an **if..then..else**- the case when the else clause is empty and the case when it is not.

tool that implements the complete translation of assembly programs to high level language programs. A developer of such a tool will want to use the most efficient algorithms available. The testing phase of the development could be severely hindered by slower algorithms.

Realising the benefits for a faster algorithm, an alternative approach has to be taken. The key lies in achieving the same functionality without requiring any expensive data structures. Such an algorithm has been proposed that does not use any data structures apart from those required to actually hold the CFG. This new algorithm has been proven to work in the case of structured assembly code with a much better performance than the original algorithm. This thesis attempts to extend the algorithm so that it will work with arbitrary assembly programs (i.e. both structured and unstructured programs).

1.2.1 Other Benefits

As previously stated, the primary use for a tool implementing the structuring algorithms is for maintaining programs that were originally written in an assembly language. To realise the advantage of programming in a higher level language as opposed to assembly, one just has to attempt to read and understand the source code for an application written in both a HLL and some form of assembly. However, the opportunity for this kind of comparison is quite rare. Instead, it is interesting to look at the results of research that have made such comparisons. Tanenbaum[15] quotes a study comparing the effort required to program in assembly and the amount of effort to program in PL/1. This study showed that the number of lines of debugged code a programmer can produce in either of these languages is approximately 100 to 200 lines in a given time frame. They found that one PL/1 statement is equivalent to about 5-10 assembly statements. Therefore a PL/1 programmer is about 5-10 times more efficient than an assembly programmer. While this ratio is somewhat moderated by the fact that a lot of assembly coding involves using idioms (segments of code with a well known semantics), which offset the time taken to write more code, it is reasonable to say that coding in a higher level language is still more efficient. While this does not make a direct comment on why a faster set of structuring algorithms is beneficial, it does show that there is indeed a need for assembly structuring tools. It would be ideal for these tools to use the fastest algorithms available.

Given that the structuring algorithms are mainly CFG manipulation algorithms, the techniques used are also applicable in other tools that need to manipulate and analyse control flow graphs. For example, the loop unrolling optimization used by many compilers involves some analysis of a program's CFG.

Finally, other possible benefits of the structuring algorithms include their use in the process of binary translation[3]. Also, as part of a complete assembly to HLL program translator, it may be used as an aid for teaching assembly to someone who already has already experience programming in some HLL.

1.3 Outline for Remaining Chapters

The remaining chapters elaborate on the issues introduced in this introductory chapter. Chapter 2 reviews work by others in the field of program structuring and closely related topics.

Chapter 3 gives the overall structure of the assembly structuring process. It also provides greater detail about the less significant components in this process.

Chapter 4 deals with the heart of this thesis. It gives an in depth coverage of the area of program structuring and then proceeds to present both the new algorithms we have developed as well as those against which we seek an improvement.

The issue of generating high level code from the assembly input after it has been structured forms the body of chapter 5. The algorithms used are discussed in detail as well as mentioning how they can be retargeted to output code in different languages.

Chapter 6 firstly discusses an implementation of the algorithms presented in the chapters 4 and 5. The issue of any restrictions placed upon the development is discussed as is the idiosyncrasies of the environment for which the implementation is targeted.

[3]Binary translation is the process of converting a binary object file or executable that will run on one hardware platform into a functionally equivalent program that will run on a different hardware platform

The second half of chapter 6 presents the results of comparisons made between the performance of the old and new algorithms. The tests concentrate on the issues of functionality (what is the quality of high the level code generated) and efficiency (how good is the resource usage).

Chapter 7 leaves the reader with the conclusions reached by this thesis and discusses further work that can be done in this area.

Chapter 2

Related Work

The issues involved in structuring assembly programs are similar to those involved in structuring high level language (HLL) programs. The latter aims to transform one HLL program into another HLL program. This chapter reviews work that has been done in the general area of program structuring and relates it to our work on assembly structuring.

2.1 The Nature of Unstructuredness

As with much of the work on transforming CFGs, the emphasis is on restructuring an unstructured graph into an equivalent but structured graph. Oulsnam [13] describes a method for performing such a translation with the added property that the original program is derivable from the translated program. The motivation behind this is that it allows for comparisons to be made between the original and final programs to determine the additional overheads in space and time that are introduced into the structured program. He states that there are only six fundamental forms that can be used to describe an unstructured graph; jump into or out of a branch of a decision structure (**if..then..else** statement), a jump into or out of a forward path of a loop and a jump into or out of a backward path of a loop. Any graph displaying one or more of these forms will be unstructured and none of the basic forms will occur by itself. That is, the offending edge in one of these forms will either come from or lead to another fundamental unstructured form. He lists the normalised form of six unstructured subgraphs that are a

result of pairing four out of the six basic forms. The remainder of the article describes a set of transforms that can be performed on these unstructured forms and subgraphs to produce a structured graph as the final result. The transforms described involve the introduction of new boolean variables.

The notion of a backward path in a loop does not make sense in terms of the assembly graphs we will be dealing with. The only backward path we can have in our loops is the back edge that defines the loop. For example, graph (a) in Figure 2.1 is the example given by Oulsnam of a backward path in a loop. The backward path includes the nodes C, B and A. However, it is equivalent to the graph shown in (b). The offending edge in the graph (a) is (B, E) where as it is the edge (C, D) in graph (b). This means we can ignore two of his six fundamental forms of unstructuredness. The remaining forms however are valid and give us an indication of the types of graphs we will have to deal with.

The transformations resulting in new boolean variables is irrelevant to our work as we would like to preserve as much of the original semantics and structure as possible.

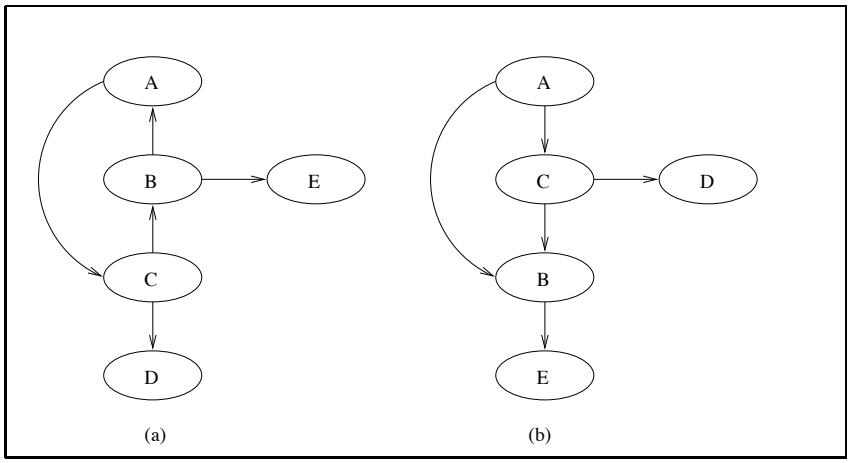


Figure 2.1: The backward path in a loop is just a specialisation of a standard loop.

A similar article on the nature of unstructuredness by Williams [11] also identified a base set of unstructured subgraphs. He varied on the exact base subgraphs given by Oulsnam, deriving four that are equivalent to four of Oulsnams; jump into or out of a branch of a decision structure and a jump into or out of the forward path of a loop. He also distinguishes one further

case; parallel loops. These are loops that share the same entry node as shown in Figure 2.2. The loops induced by the edges (A, B) and (A, C) are parallel loops as they share the same header node, A. A proof is given that these five substructures are the only ones that can lead to unstructuredness.

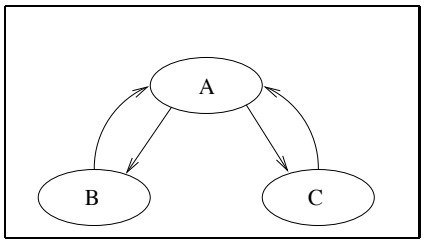


Figure 2.2: An example of parallel loops.

Both of the above papers are closely related to the structuring algorithm in this thesis in that they define structuredness in the same way. That is, the structuredness of a CFG is defined in terms of a set of control structures, and is only structured if the CFG can be decomposed by repeatedly collapsing all instances of these subgraphs into a single node.

We can make use of this similarity by considering a set of canonical unstructured subgraphs derived from the two papers. If our new algorithm can deal with instances of these kinds of unstructuredness, then it should work for any program.

2.2 Graph Reductions and Transformations

An alternative approach is taken by Agesen in [1]. This paper details the implementation of Pep, a Java bytecode to Self bytecode translator. Self is a pure object-oriented language where even the smallest units of data such as integers and booleans are all objects. The translator was built to study the applicability of Self's optimisation techniques to Java code.

Of interest to this work is the section on converting the control flow constructs within Java bytecodes to higher level Self control constructs. The former strongly resemble the control flow constructs found in typical assembly languages while the latter is a good representative of the control constructs found in most HLLs.

The analysis is performed on the CFG representation of the Java bytecodes. The high level constructs are detected by the repeated application of two particular graph reductions — absorption and promotion. Absorption moves a node into its predecessor and can only be applied to nodes with a single predecessor. Promotion 'lifts' a node to the point that covers all of its incoming branches. An example of these two reductions is shown in Figure 2.3 where the first reduction (from (a) to (b)) shows nodes b and C being absorbed into A. The second reduction depicts how D is then promoted into A at the same nesting level.

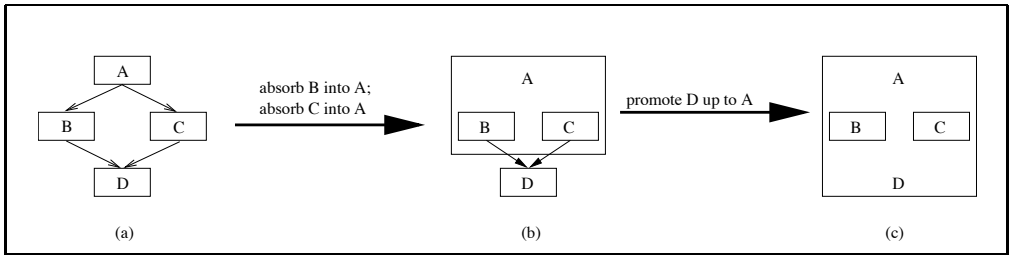


Figure 2.3: An example of the two reductions used for structuring in the Pep, Java to Self bytecode translator.

Agesen's algorithm works very well in the absence of loops. When structuring programs that involve loops, we are forced to fall back on additional techniques to ascertain the information we require about loops. These extra techniques, such as repeated depth first searches of the graph are considerably expensive in terms of execution efficiency. However, these techniques of absorption and promotion may yet prove useful in generalised structuring if these inefficiencies can be resolved.

Lichtblau [10] describes a formalised method of structuring control flow graphs by means of graph transformations. The transformations are performed by using the concept of a context-free flowgraph grammar which is described by rules that transform one graph into another. The transformations always produce a smaller graph as they all involve two or more nodes being collapsed into one node. It therefore follows that a sequence of transformations is finite and he proves that the algorithm terminates in a time linearly depending on the number of underlying vertices in the graph. The generation of high level code is a side effect of these transformations and therefore does not impose any further effort.

A graph transformation rule described by Lichtblau is shown in Figure 2.4.

This example shows the transformation of an **if..then..else** graph pattern into a single node. Diamond shaped nodes indicate a condition node (i.e. 2 out edges), rectangular nodes are either simple nodes or the result of previous transformations. The oval node is any type of node and is not included in the transformation. A dashed line indicates one or more edges. The node resulting from the transformation contains the original nodes from which it was derived embedded in the code skeleton given by that particular transformation.

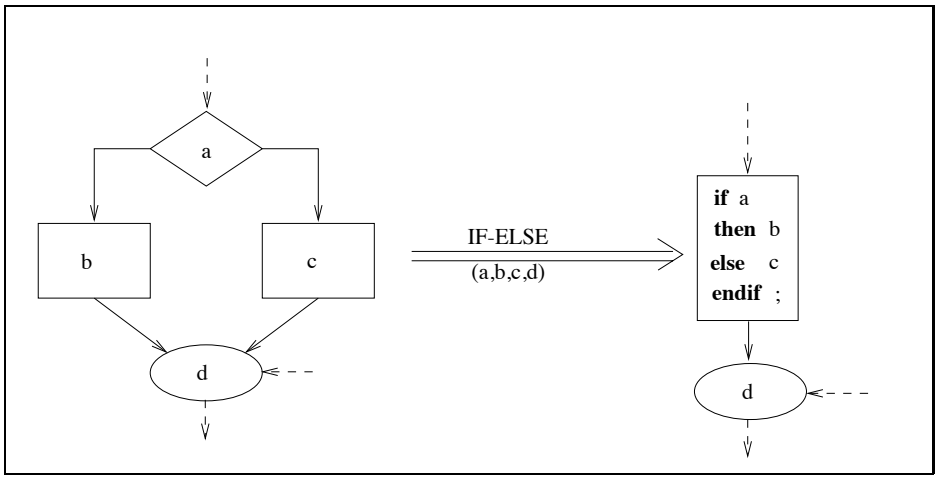


Figure 2.4: The graph transformation rule for an **if..then..else** statement.

If a sequence of transformations results in the single node graph, then the original CFG is said to have been structured. For unstructured graphs, the sequence will stop (i.e. none of the transformation rules can be applied) before reaching the single node graph. In this situation, an edge is removed from the graph and a **goto** is generated at the appropriate position.

The transformation system is shown to be finite Church-Rosser. The implication is that the applicable transformation rules can be applied in any order and the same result will always be obtained.

The graph transformations given by Lichtblau are similar to ours as do not alter the semantics and control flow of the original program. However, he does not include transformation rules for one very common high level control structure — the nway (i.e. ***switch****) statement. It is not clear whether or not adding this to his system would be a trivial task and what effect it may have on the given cost analysis. He also ignores the occurrence in assembly control*

flow graphs of patterns representing the short-circuit evaluation of compound boolean conditions (see Section 2.4).

In the presence of unstructured graphs, Lichtblau's algorithm removes one or more edges in the graph until a transformation rule can be applied. However, he does not elaborate on how to decide which edge(s) should be removed. This may possibly effect whether or not the end result will always be the same.

2.3 Semantic Preserving Structuring

Baker's work [3] differs from the above in that it presents a structuring algorithm for control flow graphs that does not introduce new boolean variables or use code replication. The constructs used for structuring are defined to be those in a given example language, SL. This set is comprised of straight line code, a **stop** statement, **if-then-else**, **repeat**, **goto**, **break(i)** and **next(i)** statements. These should all be familiar to HLL programmer with the possible exception of the last two. The **break** and **continue** constructs statements in C are examples of these parameterised jumps when the value of the parameter is 1. In general, **next(i)** is a multilevel continue and transfers control to the beginning of the i'th enclosing loop. The **break(i)** is similar except that the transfer is to the statement following the end of the i'th enclosing loop.

The algorithm initially orders the nodes in the graph by constructing a spanning tree of the graph in a depth first manner. The details of the depth-first procedure are omitted here but can be found in any book dealing with algorithms and data structures such as [7].

When structuring loops, loop headers are determined to be those with an edge into them from a node lower in the graph. For each of these loop headers an extra node is introduced. This node does not introduce any extra code in the underlying program and is used when tagging all the nodes within a loop. These are determined to be all the nodes between the new loop header (the introduced node) and the loop tail. We do not detail how the tail is determined here as it is not relevant. Figure 2.5 shows the process of adding the extra node to a graph. In graph (a), B is determined to be a loop header and the results of adding the extra node (B') for this header is shown in graph (b). The next step would be to introduce a new node for A.

After adding the extra nodes, the algorithm the defines a predicate *HEAD*

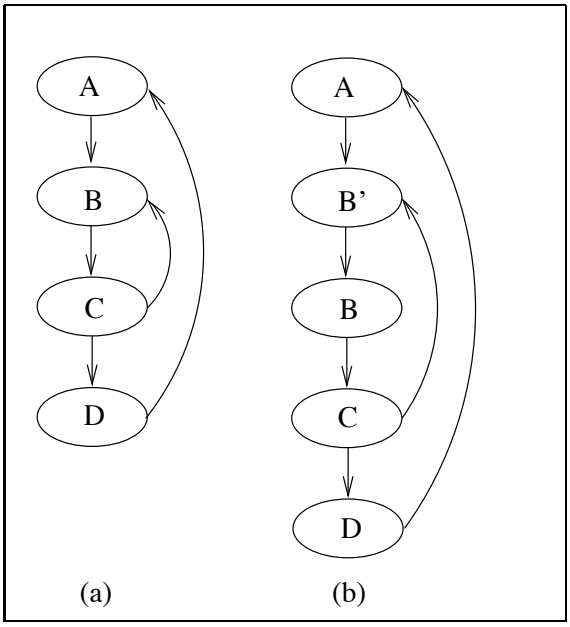


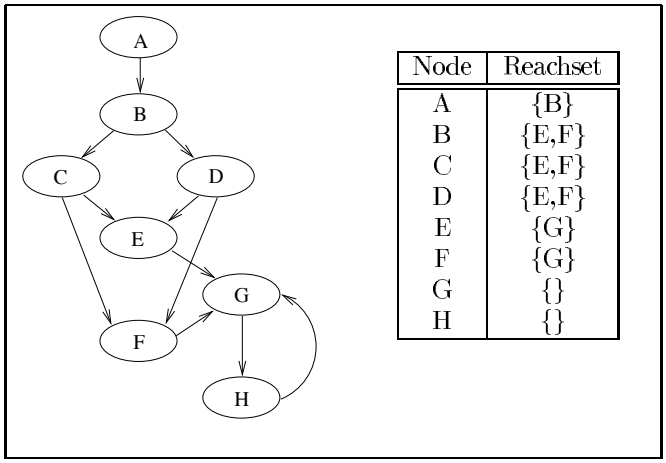
Figure 2.5: (a) Original CFG and (b) CFG with extra loop header node added.

for each node in the graph such that $HEAD(q)$ is the head of the smallest loop enclosing q in the program. If q is not in a loop, then $HEAD(q)$ is undefined.

Having added the extra nodes, we are guaranteed that for loop header p, $HEAD(p)$ will be either the header of an enclosing loop or undefined. This means that nested loop are correctly determined. Loop nesting occurs when one loop is inside another. An example of this is shown by graph (a) in Figure 2.5. The loop induced by the back edge (D, A) nests the loop induced by the other back edge (C, B).

For conditional structuring, a *reach* set is constructed for each node p where the members of this set are all the nodes entered by edges from p or from nodes nested within p (i.e. nodes within a loop headed by p or nodes within the clause of a conditional headed by p). The constraint on the set membership for p is that it cannot contain p or any node nested within p. Figure 2.6 gives an example of a CFG and a table showing the reach sets derived from it.

Using these sets, a node is structured as a conditional header if it has at least



Node	Reachset
A	{B}
B	{E,F}
C	{E,F}
D	{E,F}
E	{G}
F	{G}
G	{}
H	{}

Figure 2.6: A graph with the reach set shown for each node.

two out edges, both of which eventually lead to one of the nodes in the reach set with at least one doing so indirectly. It does this by considering a subset of all reachable statements from each statement where the membership of the subset is well defined. For each conditional detected, the first statement of each branch is recorded as well as the follow node of the whole conditional (i.e. the first common statement reached by the branch of the conditional). The predicates of the branching statements are inverted if necessary to prevent generating **if..then..else** statements where the **then** clause is empty. As an example consider nodes B and C. Node B will be structured as a conditional as both of its out edges lead *indirectly* (via c and D) to the nodes in its reach set. Node C on the other hand, will not be structured as a conditional as both of its out edges lead directly to the nodes of its reach set (i.e. e and F).

The algorithm is extended to handle irreducible CFGs. The above algorithms ensures that unstructured jumps will only take the form of forward jumps into a clause of an **if..then..else** statement or into the body of a loop.

This algorithm gives us the useful concept of imposing an ordering between the nodes within a CFG and how such an ordering can be constructed (i.e. by a depth first search traversal of the graph from the entry node). The immediate dominator technique of structuring conditionals provides the basis for how we perform structuring of conditionals.

An adaptation of the technique of inserting nodes when structuring loops is

used by our structuring algorithms. The direct result is that we can structure the one node to be both a loop header and a conditional header.

Baker's algorithm does not take into account the occurrence of nodes which have more than two out edges (i.e. ***case*** *and* ***switch*** *statements). She states that adapting the algorithm to handle these cases is straightforward but this is not the case. The reason being that when structures headed by one of the mulit-way nodes overlaps with a loop, there is a need for extra analysis to decide which structure will be generated in the HLL code and which will results in* ***goto*** *statement(s). The restriction of* ***goto*** *jumps being only forward is also unacceptable when attempting to minimise the number of* ***goto*** *statements generated. For example, in the case where there is a jump back into the subgraph of a* ***switch*** *statement, it is desirable to structure that backwards jump as a* ***goto*** *rather than as a loop with multiple forward* ***goto*** *jumps into it.*

2.4 Assembly Control Flow Graph Structuring

The structuring work performed by Cifuentes [5] as a component of a binary decompiler[1] includes most of the algorithms upon which our work is based. The algorithms used for structuring rely upon a number on control flow techniques available in the literature dealing with the analysis and manipulation of CFGs such as successors, predecessors, depth first search and more. The definitions of these concepts is postponed until Chapter 4. The control flow analysis is particularly relevant as it is performed on the control flow graphs of assembly-like programs.

Loop structuring is performed through the use of *intervals* and *derived sequence of graphs* which had previously been used in both compiler control flow and data flow analyses. Conditional structuring was performed by using the *immediate dominator* information for each node in the graph.

An algorithm is also given for structuring short-circuit evaluated graphs. These graphs result from the efficient evaluation of compound boolean conditions. For example, the graph shown on the left hand side of Figure 2.7

[1] A binary decompiler is a tool that translates a binary program (executable) into a high level language program.

representing conjunction in a HLL program, can be implemented by the CFG on the right hand side of the same figure. This type of low level implementation is even guaranteed by some high level languages (such as C) and, for example, allows a programmer to test a pointer before dereferencing it without needing to use a lengthier conditional statement.

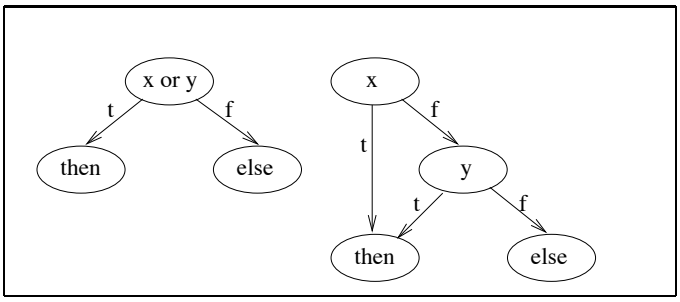


Figure 2.7: Two possible CFG representations for a compound boolean expression. The short-circuited representation is given on the right hand side of the figure.

The algorithms used for structuring 2way and nway conditionals were given separately in this work. We can modify how this structuring is done so that the two can be performed together. The modification should not only give better performance but also improve the quality of the generated code.

As mentioned previously, the loop structuring algorithm presented by Cifuentes is a major focus of this thesis. We attempt to preserve the functionality of the algorithm presented in this research while improving its execution time by using an alternative to the two data structures she used.

We do not include the structuring of short-circuit evaluated graphs as to do so requires data flow analysis. This is quite feasible in the context of developing a tool (such as a decompiler) that already involves data flow analysis. However, as we are concentrating purely on control flow analysis, we have neglected this type of structuring.

Chapter 3

Assembly to High Level Language

The task of structuring assembly programs involves more than just designing some structuring algorithms. We need to deal with the issue of getting the program into its CFG representation which is required by the structuring algorithms. We also need to provide some means of generating the HLL code from the CFG after it has been augmented with the structuring information. This chapter describes these broader issues of assembly structuring. For the components that are not central to the translation process, a detailed discussion of their algorithmic aspects is given here instead of devoting a whole chapter to them.

The structure of an assembly to HLL translation is similar to standard compilation on a conceptual level (although significantly simpler) as it consists of a series of *phases* that transform the input (an assembly program) from one representation to another. The phases involved are shown in Figure 3.1. The *parser* performs both syntax analysis and a simple form of semantic analysis on the input assembly. Any implementation of this phase is architecture dependent because assembly languages are dependent on a particular machine. The *CFG generation* phase builds the CFG representation of the program. As a result of semantic analysis performed in the parser phase, this phase is both architecture and HLL target independent. The third phase is the *structuring* process. Both it and the last phase, *HLL code generation*, are architecture independent but are dependent on the HLL chosen as the target. These last two phases are the subject of the following two chapters

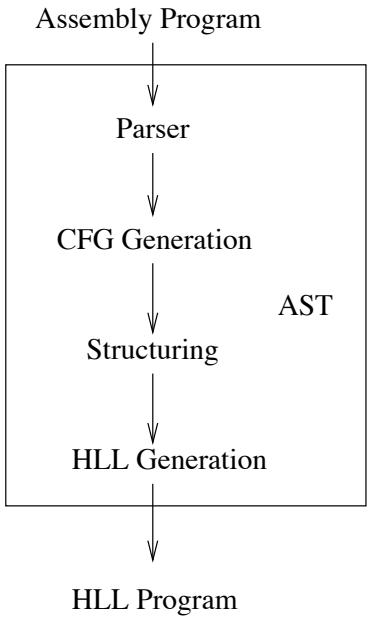


Figure 3.1: The phases of an assembly to HLL translation.

respectively and are the core of the structuring process.

3.1 Assembly Parsing

An assembly language is in essence just a symbolic syntax for a given machine's binary instruction set. This explains why any given assembly language is machine dependent and programs written in it are not portable to another architecture without significant translation. Only pure binary programs, which are further restricted by binary file formats and operating systems, are less portable. The implications of this architecture dependence for any implementation of the translator is that it will only be able to process assembly programs from one architecture. However, given that the later phases are independent of the given assembly syntax, porting the implementation to a different architecture should only involve rewriting the first phase module.

The semantic analysis performed by the parser is the categorisation of each instruction parsed into one of two categories; *transfer instructions* and *non-*

transfer instructions. Knowing what category an instruction is assists in the next phase; control flow graph generation. We partition the instruction set of any architecture into the following two categories:

- Control Instructions (TI): this is the set of instructions that change the value of the program counter. That is, they explicitly set the address of the instruction to be executed. For convenience, we refer to any instruction within this set as a control transfer instruction (CTI). These instructions are:
  - Unconditional branches[1]: the flow of control is transferred to the target branch address.
  - Conditional branches: depending on the value of some condition code(s) set by preceding instructions, the flow of control is transferred either to the target branch address or the next instruction in the sequence.
  - Indexed jumps: the target of the jump is defined by the value of a register and is therefore a range of values, one of which the flow of control is transferred to.
  - Subroutine call: the flow of control is transferred to the routine given in the appropriate operand of the instruction. For our purposes, we assume that the flow of control is transferred back to the next instruction in the sequence after successful completion of the subroutine and only this flow of control is represented in a control flow graph.
  - Subroutine return: the flow of control is returned to the caller subroutine that invoked the current subroutine.
- Non transfer instruction (NTI): the set of instructions that implicitly transfer control to the next instruction in the sequence. That is, all the instructions that are not in TI.

Parsing the assembly input is little more than text file processing. Checking for the correctness of the assembly program is considered to be outside the domain of the AHT process. This phase only reads in the instructions and build a representation of the instruction sequence within memory. When

[1]Different architectures use the terms 'jump' and 'branch' to mean the same thing. We adhere mainly to the latter but the two are used interchangeably hereafter.

parsing a control transfer instruction (CTI), the internal representation is tagged as being a CTI as well as being augmented with the location(s) of the transfer control. This simple semantic analysis is tightly coupled to the host architecture but the information it renders is architecture independent. The locations are stored as positions within the internal instruction sequence. Essentially, they are array indices. A parser for assembly in any architecture could be used here instead as long as it performs the architecture independent categorisation of the instructions as discussed above.

3.2 Control Flow Generation

As a preliminary to discussing the generation of a control flow graph, we define the concept of a *basic block*. Working from this definition, we then proceed to formalise the concept of a control flow graph as well as introducing terminology used when discussing control flow graphs.

3.2.1 Basic Blocks

This section formally defines the concept of basic blocks which are also the building blocks of a CFG. The definition is given in terms of the sequence of instructions that were parsed into memory by the parser phase.

Definition 3.1 *A* **basic block** $b = [i_1, .., i_{n-1}, i_n, i_{n+1}]$, $n \geqslant 1$ *is an instruction sequence that satisfies the following conditions:*

1. $[i_1, .., i_{n-1}, i_{n+1}] \in NTI$
2. $\forall\, i \in \{i_2 \,..\, i_{n+1}\} \bullet i$ *is not the target of a CTI*
3. $i_n \in TI$

OR

1. $[i_1, .., i_{n+1}] \in NTI$
2. i_{n+2} *is the target of a CTI*

The first alternative in Definition 3.1 specifies that a basic block is a sequence of instructions such that all but the second last are from the *NTI* set. Additionally, only the first instruction may be the target of a CTI and the second

last instruction must be a CTI. The second alternative defines a basic block that is not delimited by an instruction that explicitly transfers control. In this case, the last instruction has the property that its successor is the target of a CTI. All the instructions in this type of basic block will be from the *NTI* set.

The above definition is unfortunately dependent upon the underlying architecture. The one we have given is in terms of the SPARC architecture which includes the concept of delayed instructions. These are the instructions that immediately follow most CTIs. This instruction is executed, conditionally in some cases, before the transfer of control takes place. For simplicity, we therefore define these delayed instructions to be a member of the same basic as their preceding CTI. Chapter 6 deals further with the idiosyncrasies of these delayed instruction further when discussing how they were handled by an implementation of our structuring algorithms.

3.2.2 Graph Theory

Before defining a control flow graph, we formalise the notion of a graph and its relevant properties. The definitions presented are those given in [5] and are similar to those found in any text dealing with graph theory.

Definition 3.2 *A* **graph** *G is a tuple (V,E,h) where V is a set of nodes, E is a set of edges, and h is the root of the graph. An* **edge** *is a pair of nodes (v,w) such that v,w* $\in$ *V.*

Definition 3.3 *A* **directed graph** *G = (N,E,h) is a graph that has directed edges; i.e. each* $(n_i, n_j) \in E$ *has a direction which is from* n_i *to* n_j*.*

Definition 3.4 *A* **path** *from* n_1 *to* n_m *in graph G = (N,E,h) , represented by* $n \rightarrow n_m$*, is a sequence of edges* $(n_1, n_2), (n_2, n_3), .., (n_{m-1}, n_m) \in N, m \geqslant 1$*.*

Definition 3.5 *If G = (V,E,h) is a graph such that* $V = \{h\}$ *and* $E = \{\,\}$*, then G is a* **trivial graph** *(i.e. a graph with only one node and no edges).*

Definition 3.6 *If G = (N,E,h) is a graph and* $\forall n \in N, h \rightarrow n$*, then G is a* **connected graph** *(i.e. every node in the graph is reachable from the head of the graph).*

3.2.3 Control Flow Graphs

The following definition formally shows that a control flow graph is constructed from the basic blocks of a program. There is one CFG for each subroutine in the program and these CFGs do not overlap[2]. However, the CFGs do not necessarily partition the basic blocks of a program. Since the edges within a CFG represent the flow of control between the blocks, any unreachable blocks will not be included in any CFG.

Definition 3.7 *A* **control flow graph** *G = (N,E,h) for a procedure P is a connected, directed graph such that:*

1. $\exists f : \mathcal{B} \rightarrowtail\!\!\!\!\!\rightarrow N$ *where $\mathcal{B}$ is the set of basic blocks for the procedure.*
2. $\forall\, e = (n_i, n_j) \in E$, *$e$ represents a flow of control from one basic block to another and $n_i, n_j \in N$.*
3. $\exists\, b_i \in \mathcal{B} \bullet$ *h is the first instruction in b_i.*

Part 1 states that there is a partial one-to-one function mapping basic blocks to nodes in the graph. That is, for every node there is a corresponding unique basic block. However, due to the aforementioned unreachable basic blocks, the reverse is not necessarily true. Part 2 equates the control flow between two basic blocks with an edge in the graph. The third part of the definition defines the head of the graph to be the node containing an instruction that is an entry to a subroutine. This instruction will either be the first instruction in the program or the target of a subroutine call.

For certain types of analysis, it is important that a control flow graph has only one exit node. That is, there is a unique node in the graph such that it is the only one that has no successors. If there is more than one of these in the original graph, we conceptually add a new node and add an edge from each of the candidate exit nodes in the original graph to this new node. This new node does not introduce any code into the underlying program.

[2] As we are only concerned with control flow analysis, we do not consider the transfer of data between subroutines. For this reason, we do not explicitly show the transfer of control from one subroutine to another as an edge in a graph. It suffices for us to simply show the points at which a subroutine call is made.

Definition 3.8 *The* **exit** *node, e, of a graph is such that* $\exists\, e \in N \bullet \nexists n \in N \bullet (e, n) \in E$

As with the instructions, the basic blocks are also classified into different types for the purpose of control flow analysis. The type is determined by the delimiting instruction of a block. Under the SPARC architecture (with its delayed instructions), the delimiting instruction of a basic block will be the second last one in the case where the block is defined by the first part of Definition 3.1. Otherwise it will be the last instruction in the block. The types are:

- Unconditional basic block: the second last instruction in the block is an unconditional branch instruction. This block will have one out-edge.
- Conditional basic block: the second last instruction is a conditional branch. This block will have two out-edges.
- Nway basic block: the second last instruction is a indexed jump. The block will have n out-edges, one for each possible target of the jump.
- Call basic block: the second last instruction is a call to a subroutine. This block will have only one out-edge which will be to the lexical successor of the block's last instruction.
- Return basic block: the second last instruction is a return instruction. This block will have no out-edges.
- Fall through basic block: the block is not delimited by a CTI.

The nodes in the graph representing basic blocks[3] of the above types are given the following names respectively: ucond, cond, nway, call, ret, fall.

To illustrate the process of basic block partitioning and CFG generation consider the example shown in Figure 3.2. The input to the example is an assembly program[4]. Instructions 9 and 12 are examples of unconditional branches which delimit an unconditional basic block while instruction 6 demonstrates

[3]Given this representational relationship between nodes and basic blocks, the two terms are used interchangeably hereafter.

[4]A number of non-transfer instructions have been removed from the code to cut down the size of the example.

an indexed jump[5] which delimits an nway basic block. The graph resulting from these basic blocks and the flow of control between them is shown in Figure 3.3.

```
1    main:  save %sp,-120,%sp
2           or %o0,%lo(.LLC0),%o0
3           call scanf,0
4           or %o0,%lo(.LL3),%o0
5           ld [%o1+%o0],%o0
6           jmp .LL3 .LL12 .LL4
7           nop
8    .LL3:  ld [%fp-20],%o1
9           b .LL14
10          or %o0,%lo(.LLC1),%o0
11   .LL4:  ld [%fp-20],%o1
12          b .LL14
13          or %o0,%lo(.LLC1),%o0
14   .LL12: sethi %hi(.LLC2),%o0
15          or %o0,%lo(.LLC2),%o0
16   .LL14: bne .LL15
17          nop
18          call printf,0
19          mov 0,%i0
20   .LL15  ret
21          restore
```

Label	Type	Instructions
A	call	1 - 4
B	nway	5 - 7
C	ucond	8 - 10
D	ucond	11 - 13
E	fall	14 - 15
F	cond	16 - 17
G	call	18 - 19
H	ret	20 - 21

Figure 3.2: A sample assembly program. The basic block partitioning is shown by the table on the right hand side.

3.2.4 Unreachable blocks

One last refinement has to be performed to the CFGs before they are ready for structuring. We must remove the aforementioned unreachable basic blocks.

To remove the unreachable blocks, a depth first traversal if performed from each node representing the entry point to a subroutine, tagging each node as it is traversed. Upon completion of the traversals, it is simply a matter of removing the untagged nodes.

[5]The range of the possible index values have been resolved by a preprocessor.

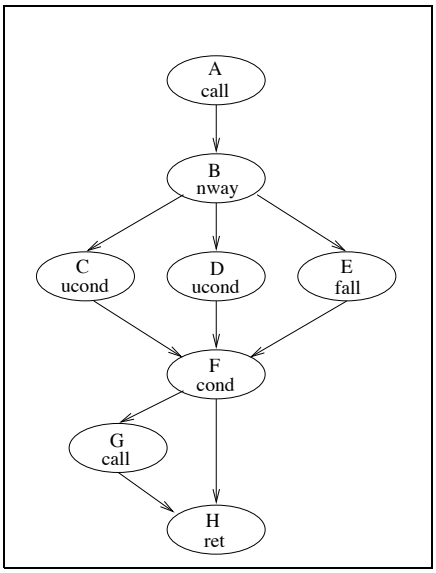


Figure 3.3: The control flow graph for the program given in Figure 3.2

Chapter 4

Structuring Assembly Graphs

A control flow graph a representation of the underlying assembly program which does not contain any information that cannot be read directly from the assembly code. To generate high level code from these control flow graphs requires further analysis. The use of *structuring* algorithms will give us the required information. As a result of these algorithms, the graphs will be annotated with the information describing what high level control constructs are represented in the graph. This chapter presents the control flow analysis performed by the structuring algorithms to attain the high level control flow constructs.

4.1 Notation

This chapter includes a number of algorithms presented in pseudocode. While this style is mostly intuitive, the following conventions are used:

- attributes of objects are written (with a lower case first letter) as a function call on the object. For example, `iPDom(x)` accesses the node `x`'s immediate post-dominator and `isBackEdge(v,w)` returns whether or not (v, w) is an edge.
- procedure calls are written in their usual style with the distinguishing feature of having an uppercase first letter. Return values are implemented as **var** (i.e. reference) parameters.

- the nesting levels are represented through the use of indentation.
- a test for equality uses '=' and assignment uses the ':=' symbol.

While it is presumed that the meaning of each node attribute is self-explanatory, a reference for all these attributes is provided in Appendix A.1. This reference relies upon the definitions given in the following sections. Section A.2 defines any predicates used while Section A.3 defines an attribute of a loop when referred to as an object.

4.2 Graph Structuring

To structure a control flow graph, a set of high level constructs is required to give the structuring some context. For example, the structuring of control flow graphs could be performed very easily if we were only aiming to describe the flow of control using two-way conditionals (**if..then..else**) and **goto** statements. However, it is questionable whether this leaves us much better off than the original assembly program.

The set of high level constructs chosen for structuring needs to be general enough to cater for some of the more common high level languages. This allows any implementation of the assembly structuring process to be easily retargeted for any high level language that at least provides the set of constructs used to do the structuring. Further analysis could then be performed on the output to use additional constructs provided by the language in question. Alternatively, the algorithms could be modified slightly to support new constructs.

A comprehensive review of some common high level languages and the control flow mechanisms they provide is given on pages 138 - 140 in [5]. The constructs embodied in each of the languages is a subset of the constructs shown in Figure 4.1 along with their control flow graph representations. Note that some of the other common control structures such as **for** loops have not been shown. These omitted structures can usually be implemented using those already shown. For example, in the case of the **for** loop, it can be implemented by using a sequential statement followed by a **while** loop.

The set of structures used for our purposes include all those shown in Figure 4.1 except for the last two, multilevel exits and multilevel cycles. Instead,

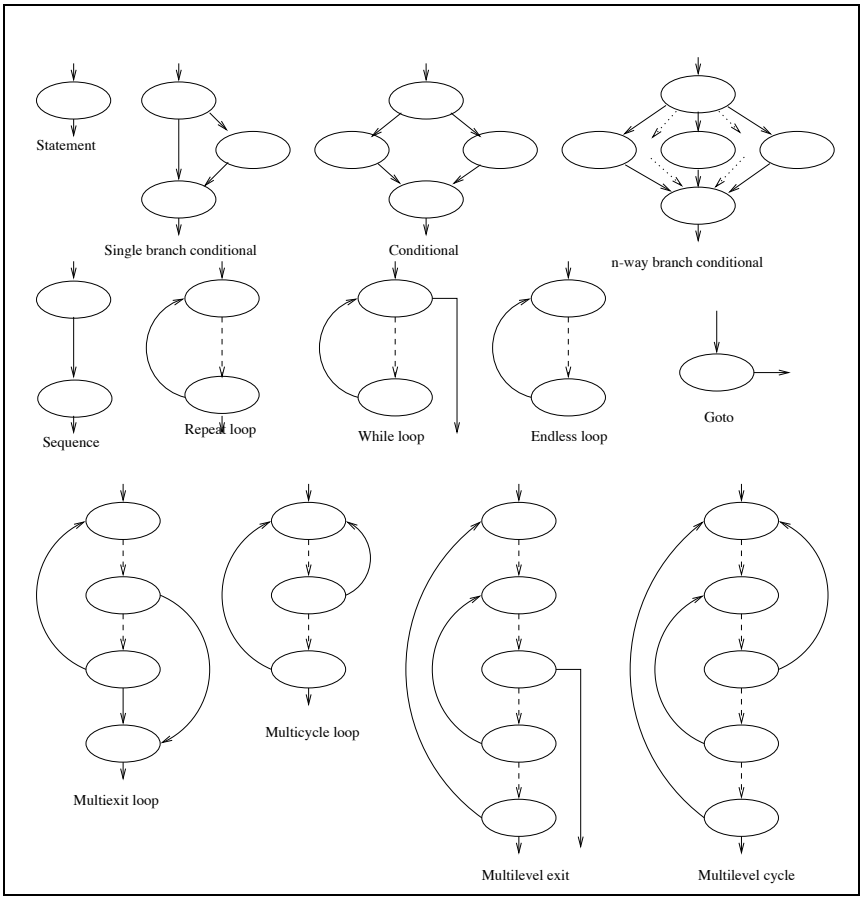


Figure 4.1: HLL control structures and examples of their control flow graph representations.

goto's will be used to emulate these structures should they appear as most languages provide the former but very few (e.g. Ada) provide the latter. However, the inclusion of **goto**s also excludes at least one common language, Modula2.

4.3 Control Flow Graph Terminology

Before discussing how we gather information about the underlying structures present in a control flow graph, we first define the relevant terminology and definitions used. These definitions are taken directly from [5] (apart from

the post-dominator definitions which are a modification of the immediate dominator definitions given in the same piece of work) and are given in terms of a directed graph $G = (N, E, h, e)$. This is an extension of the normal representation of a graph in that we now explicity mention the exit node (see Definition 3.8).

Definition 4.1 *A* **closed path** *or* **cycle** *is a path* $n1 \rightarrow n_v$ *where* $n_1 = n_v$.

Definition 4.2 *The* **successors** *of* $n_i \in N$ *are* $\{n_j \in N \mid n_i \rightarrow n_j\}$. *The* **immediate successors** *of* $n_i \in N$ *are* $\{n_j \in N \mid (n_i, n_j) \in E\}$.

Definition 4.3 *The* **predecessors** *of* $n_j \in N$ *are* $\{n_i \in N \mid n_i \rightarrow n_j\}$. *The* **immediate predecessors** *of* $n_j \in N$ *are* $\{n_i \in N \mid (n_i, n_j) \in E\}$. *A node* $n_i \in N$ **post-dominates** *a node* $n_k \in N$ *if* n_i *is on every path* $n_k \rightarrow e$.

Definition 4.4 *A node* $n_i \in N$ **immediately post-dominates** $n_k \in N$ *if* $\nexists n_j \in N \bullet n_j$ *post-dominates* n_k *and* n_i *post-dominates* n_j *(i.e.* n_i *is the closest post-dominator of* n_k*).*

Definition 4.5 *A* **strongly connected region** *(SCR) is a subgraph* $S = (N_S, E_S, h_S) \bullet \forall\, n_i, n_j \in N_S \bullet \exists\, n_i \rightarrow n_j$ *and* $n_j \rightarrow n_i$.

Definition 4.6 *A* **strongly connected component** *of* G *is a subgraph* $S = (N_S, E_S, h_S) \bullet S$ *is a strongly connected region and* $\nexists S_2 \in SCR(G) \bullet S \subset S_2$ *(i.e. it is the smallest connect region involving* h_S*).*

Definition 4.7 *A* **depth first search** *(DFS) is a method of traversing a graph that selects edges to traverse from the most recently visited node that still has unvisited nodes on at least one of its edges. A* **depth first spanning tree** *(DFST) of a flow graph* G *is a directed, rooted, ordering spanning tree of* G *grown by a DFS algorithm.*

As described on page 482 of [7], the DFST T of a graph partitions the edges of the graph into four categories; tree edges, forward edges, backward edges and crossedges. For the purpose of structuring CFGs, the differentiation between tree, forward and cross edges is uninteresting and so we collapse them all to the one category, forward edges. We end up with the following two categories:

1. *Backward edges* $= \{(v, w) \mid (v, w) \in E \wedge w \rightarrow v \in T\}$.
2. *Forward edges are all the remaining edges.*

The backward edges (also referred to as back edges) are used when structuring loops. The forward edges are used for structuring all other constructs as well as loops.

4.3.1 Interval Theory and the Derived Sequence of Graphs

For the purpose of the describing the structuring algorithm we aim to improve upon, we define the two data structures it uses. The first structure was defined by Cocke in [6] and the algorithm used to build it developed by Cifuentes in [4].

Definition 4.8 *Given some node, $n \in N$, the* **interval** $I(n)$ *is the maximal, single-entry subgraph in which n is the only entry node (i.e. the only node with a predecessor outside the subgraph) and in which all closed paths contain n. The unique interval node n is called the* **interval head** *or simply the header node.*

We can use intervals to partition the nodes of G into a unique set of disjoint intervals $\mathcal{I} = \{I(h_1), .., I(h_n)\}$ for some $N \geqslant 1$. The algorithm presented in Figure 4.2 is used to derive the sequence of intervals for a graph. This algorithm is derived from work done by Allen [2] in the area of control flow analysis. Defining $\mathcal{I}$ as a sequence imposes an ordering between the intervals of a graph which is required when building the other data structure required for this algorithm, the derived sequence of graphs.

Interval theory alone is not sufficient for the complete structuring of loops. We use an additional concept called the *derived sequence of graphs* which was described by Allen [2] and is based upon the intervals of a graph. The construction of the graphs is an iterative method that collapses intervals into nodes. The nodes formed by this process are then the nodes of the next order graph. The edges for this next order graph are both the in edges to the header of the interval from which the new node was derived as well as the out edges from nodes within the same interval whose destination was not

```
procedure FindIntervals (G = (N, E, s), var I)

* Pre: G is a graph
* Post: the intervals of G are contained in the sequence I

   I:= ⟨⟩
   H := {h}
   for (all unprocessed n ∈ H) do
      I(n) := {n}
      repeat
         I(n) := I(n) + {m ∈ N | ∀ p ∈ pred(m) • p ∈ I(n)}
      until (no more nodes can be added to I(n))
      H := H + {m ∈ N | m ∉ H ∧ m ∉ I(n) ∧ (∃ p ∈ pred(m) • p ∈ I(n))}
      I:= I + I(n)
end procedure
```

Figure 4.2: Interval building algorithm.

within the interval. The process of building the intervals is then performed on the new graph after which the whole process starts again. This continues until either the trivial graph is derived or the graph derived is the same as the graph from which it was derived. If the latter is the case, then the graph is said to be **irreducible.**

An adaptation of the algorithm given by Cifuentes in [4] is shown in Figure 4.3. The adaptation is a result of using an object oriented design. Under this paradigm, an interval is represented by a class derived from the class used to represent an ordinary graph node with a few additional attributes (see Appendix A.4). For readability, we do not explicitly show the necessary type conversion required when treating a sequence as a set. The nodes and edges of the derived graph G^i are denoted by N^i and E^i respectively.

Examples of the derived sequences for two programs are shown in Figures 4.4 and 4.5. They demonstrate how the intervals are defined for each graph within the sequence. Figure 4.5 is an example of an irreducible graph. The final graph in its derived sequence is the canonical irreducible graph. All irreducible graphs will reduce to containing only or more instances of this graph. An irreducible graph will occur when there is a forward edge into a loop subgraph.

For convenience, we use the term **DSG** (derived sequence of graphs) when describing the loop structuring algorithm that uses these two data structures.

```
procedure BuildDerivedSequences(G = (N, E, h), var 𝒢)

* Pre: G is a graph
* Post: the derived sequence of graphs G^1, .., G^n, n ⩾ 1 has
* been constructed and is returned in 𝒢

   G^1 := G
   𝒢:= ⟨G^1⟩
   i := 2
   repeat /* Construction of G^i */
      FindIntervals(G^{i-1},ℐ)
      N^i := ℐ
      h^i := ℐ (1)
      E^i := { }
      if (#N^{i-1} ≠ # ℐ) then
         for (all I ∈ ℐ) do
            for (all n ∈ nodesOf(I)) do
               for (all m ∈ (succ(n) - nodesOf(I))) do
                  E^i := E^i + (I, inInterval(m))
         𝒢:= 𝒢 + G^i
         i := i + 1
   until (G^i is the trivial graph or #N^{i-1} = # ℐ)
end procedure
```

Figure 4.3: Derived Sequence of Graphs building algorithm.

4.3.2 Parenthesis Theory

The proposed alternative loop structuring algorithm does not require any additional data structures beyond the control flow graph itself. However, it requires an extra property to be given to each node within the graph. Given that this extra property can also be used to make another improvement through 2 way conditional *restructuring* to the structuring algorithms, we define it for the nodes in the graph irrespective of what loop structuring algorithm is used.

We define this property in terms of a graph $G = (N, E, h)$. We also rely on a DFS traversal of the graph from h to increment a 'time' counter (initialised to 1) after the first and last visit to each node during the traversal. The first of these properties was given by Cormen *et al.* on pages 477-481 of [7]. The other two are derived from this first one.

Definition 4.9 *A* **parenthesis** *of $n \in N$ is denoted $t_1 \leftrightsquigarrow t_2$ and t_1 is the*

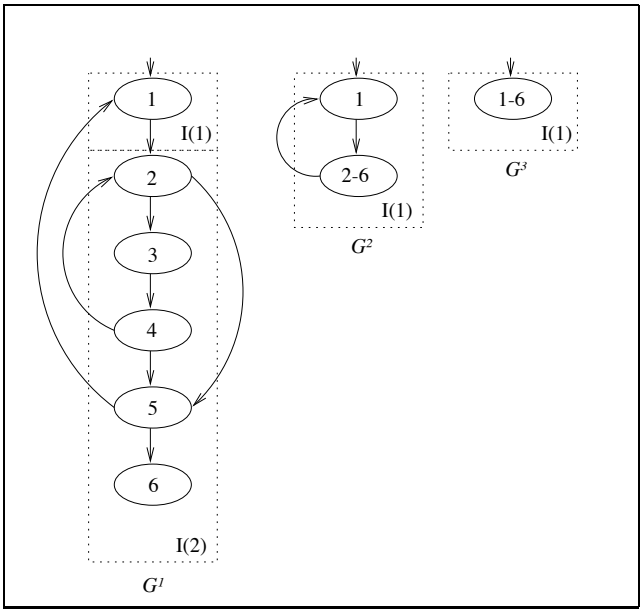


Figure 4.4: The Derived Sequence of a graph with the intervals of the initial graph and each derived graph shown by the dotted boxes.

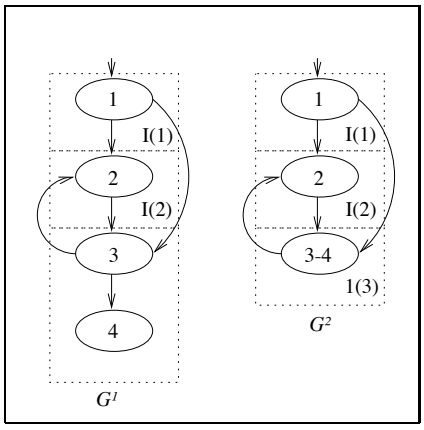


Figure 4.5: An example of an irreducible graph. The final graph is the canonical irreducible graph.

time at which n was first visited during a DFS traversal from h and t_2 is the time at which n was last visited during the same traversal.

Definition 4.10 *A parenthesis $n_1 \leftrightsquigarrow n_2$ is* **enclosed** *by parenthesis $m_1 \leftrightsquigarrow$*

m_2 iff both were defined during the same traversal of a graph and $m_1 < n_1 < n_2 < m_2$.

Definition 4.11 $\exists\, n_1, n_2 \in N \bullet (n_1 \succ n_2 \equiv (\exists\, P_1, P_2 \bullet P_1$ *is a parenthesis of* $n_1 \wedge P_2$ *is a parenthesis of* $n_2 \wedge P_1$ *is enclosed by* $P_2))$. *The* $\succ$ *relationship is pronounced "is in".*

It is relevant to note that the above definitions imply that there is more than one possible DFS traversal of a graph from the same starting point. This is important in our case as will be shown later. It is sufficient to state now that the two (or more) different traversals are a result of the ordering defined between the out edges of a node. Recall that for a 2way (or **cond**) node, the edges represent the flow of control taken when the condition is true or false respectively. This gives us a means of performing two symmetrical DFS traversals of the graph, one traversing the true branches first, the other traversing the false branches first. For an nway node, the symmetrical case involves working through the out edges in reverse.

Figure 4.6 shows the recursive algorithm used to set the parenthesis of each node in a graph. Note that it has a boolean parameter, indicating which edges are to be traversed first. For an nway node, when doing the 'true' traversal, we work through the sequence of out edges in ascending order and descending order for the 'false' traversal. In the context of parenthesis theory, the nodes have a few extra attributes which are described in Appendix A.5.

```
procedure SetParenthesis(G = (N, E, h), brch,var time)

* Pre: G is a graph
* Post: The brch traversal parenthesis for h has been determined.

   traversed(h) := true
   parenthesis_brch(h)(1) := time
   time := time + 1
   for (n ∈succ(h) ∧ ¬ traversed(n)) do
      SetParenthesis(G, brch, time)
   parenthesis_brch(h)(2) := time
   time := time + 1
end procedure
```

Figure 4.6: Algorithm used to set the parentheses for each node.

For convenience, we use the term **PT** (Parenthesis theory) when describing the loop structuring algorithm that uses the parenthesis properties of nodes.

4.4 Structuredness

Having defined the context for structuring, we can now classify control flow graphs as either structured or unstructured. Structured graphs are those in which the flow of control represented can be explained completely in terms of a set of structured control structures. All of the structures in our chosen set (see Section 4.2) apart from the **goto** are structured control structures. Therefore, any control flow graph we can reduce in terms of these structures is a structured graph. In the context of assembly programs, the control flow graphs will quite often be unstructured. We are not recognising subgraphs representing short circuit evaluation of compound conditions[1], so the graphs containing these subgraphs will appear to be unstructured.

To deal with cases of unstructuredness in the CFGs, it is important to first identify the types of unstructured cases that we may come across. In the following sections, we describe the categories of structures we want to detect and what information we aim to derive for instances within each of these categories. The type of unstructuredness that may arise in each category is mentioned.

4.4.1 Loops

The structured forms of loops are subgraphs that display a single entry/single exit nature. This includes the **while**, **repeat** and **do** loops. We also include multiexit and multicycle loops in this case but this decision is qualified in the following section on unstructured loops.

Examples of structured loops are the pretested, postested, endless, multiexit and multicycle loops shown in Figure 4.1.

For any loop, we want to derive the following information:

header - the node at the entrance to the subgraph representing the loop.

latch - the node in the loop whose edge to the header defines the loop.

follow - the first node reached after termination of the loop (if any).

[1]See Section 2.4 for an explanation of why we make this restriction.

loop nodes - the nodes that are within the subgraph representing the loop.

loop type - is it a preTested (**while**), postTested (**repeat**) or endless (**do**) loop?

We introduce the notation $h \circlearrowright l$ to represent a loop with the header node h and the latching node l.

Unstructured Loops

Some of the categories defined by Williams [11] that were discussed in Chapter 2 are shown in Figure 4.7. We ommit his category of parallel loops as we can structure these as multicycle loops. This is influenced by the target language we have chosen, C. However, both **break**'s and **continue**'s within a loop are only detected when generating the high level code. This means that we consider them as forms of unstructuredness when doing the structuring. When modifying the assembly structuring process for another high level language that provides neither of these control structures, all that is required is a small modification to the code generation phase. The type of control flow transfer occurring in these two structures is often considered to be a restricted form of unstructuredness. That is, they can both be seen as **goto**s with a limitation on the destinations to which control can be transfered.

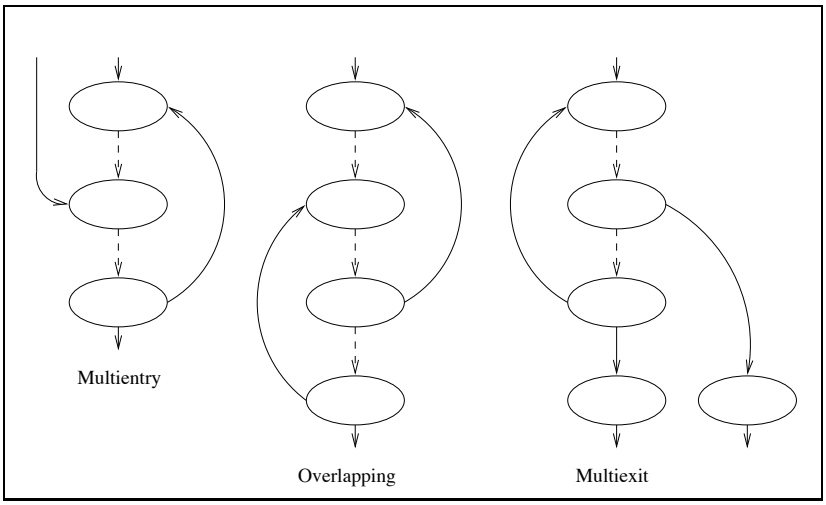


Figure 4.7: Sample unstructured loops.

4.4.2 2way Conditionals

The control flow graph representation of structured 2way conditionals will also display a single entry/single exit nature. For each 2way conditional header, there will be two branches (one for each out edge from the header). For a structured 2way conditional the nodes reachable along each of the branches will be disjoint from each other except for the one node upon which they must both converge.

Examples of structured 2way conditionals are the two middle graphs in the first row in Figure 4.1.

For any 2way conditional, we need to derive the following information:

header - the node at the entrance to the subgraph representing the conditional.

follow - the unique node at which all paths from each branch converge.

conditional type - is it a single clause (**if..then**) or double clause (**if..then..else**) 2way conditional?

We introduce the notation $h \diamond f$ to represent a 2way conditional with the header node h and the follow node l.

Unstructured 2way conditional

Unstructured 2way conditionals are those that have an abnormal entry or exit to or from a branch. Both of these cases are represented in the canonical abnormal selection path graph, shown in Figure 4.8. Graph (a) contains two potential **if..then..else** structures, headed by nodes 1 and 3. The former is shown in graph (c) and contains an abnormal exit from its right branch whereas the latter is shown in graph (b) which has an abnormal entry into its left branch. Both types of abnormal exit can also be along a back edge. In that case, the conditional will be overlapping a loop subgraph.

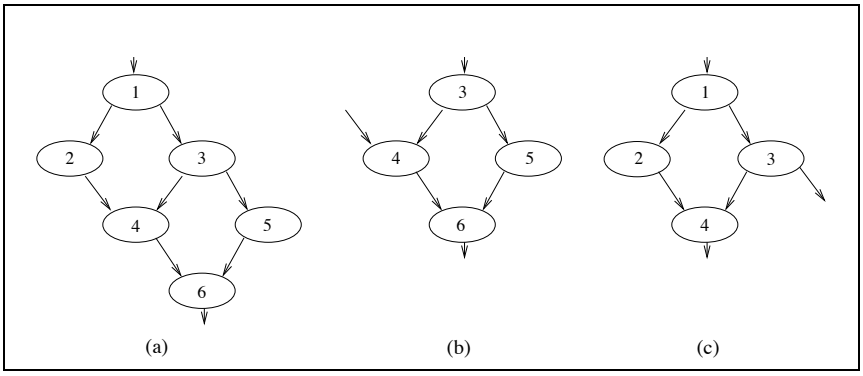


Figure 4.8: Unstructured 2way conditional graphs.

4.4.3 Nway conditionals

Similarly, an nway conditional is structured if the nodes reachable from each of the branches corresponding to the out edges from the nway header are all disjoint. Once again, there will also be a unique follow node on which all branches converge. The same set of properties is required for an nway conditional as is required for a 2way conditional plus one additional property. As with loops, we also want to know the set of nodes within the nway subgraph. This helps with unstructured cases.

We use the same notation to represent an nway conditional as a 2way conditional.

Unstructured Nway conditionals

Being a generalisation of a 2way conditional, an unstructured nway conditional is similar to an unstructured 2way conditional. The three forms of unstructured nway conditionals we can encounter is shown in Figure 4.9. In graph (a) the nway conditional $A \diamond E$ has an unstructured entry from the back edge (F, D). Graph (b) has an unstructured exit from $B \diamond F$ via the back edge (C, A). The nway conditional $B \diamond F$ in graph (c) has an unstructured entry via the edge (A, C). We cannot have an unstructured exit from an nway conditional via a forward edge. If we did, then the node reached by that edge would be lower in the graph than the immediate post-dominator of the nway header. As this contradicts the definition of an immediate post-dominator, such a case cannot occur.

Graphs (a) and (b) raise an interesting issue that must be resolved. The back edge in both of these can potentially be structured as the defining back edge for a loop. At the HLL level, overlapping nway conditionals (**switch**) and loops will result in a large number of **goto**s being used. To prevent this, we must make a choice between one or the other when they overlap. We choose to prioritise nways above loops as an nway can have a large number of clauses. If we were not to structure these as clause of an nway, then each one would involve generating a **goto**. Loops, on the otherhand, will only involve generating one **goto** for the backward edge.

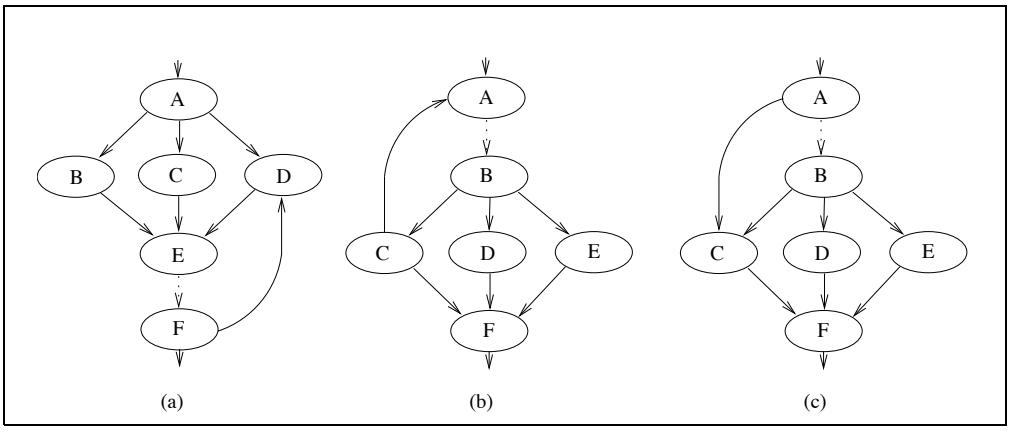


Figure 4.9: Unstructured nway conditionals.

4.4.4 Multi-structure headers

Unlike the original algorithm given by Cifuentes in [4], a node in a control flow graph is not restricted to being a header of only one type of control structure. However, the only case in which this can happen is when a node is the header of a **repeat** or **do** loop header and is also the header of a 2way conditional. This exception can occur in both structured and unstructured graphs as shown in Figure 4.10. Node 1 in graph (a) is both a **do** loop and a single-branch conditional header. This intuitively makes sense as the header of an endless loop does not actually control the loop iterations. It is merely the first node *inside* the loop. If this kind of multi-structuring was not enabled for a single node, a **goto** would have to be used instead. The same applies to graph (b). Node 1 here can be the header of both a **repeat** loop and a 2way conditional. The edge (2,5) will generate a **goto** in this case

but without the multi-structuring, another **goto** would also be required for the edge (4,1).

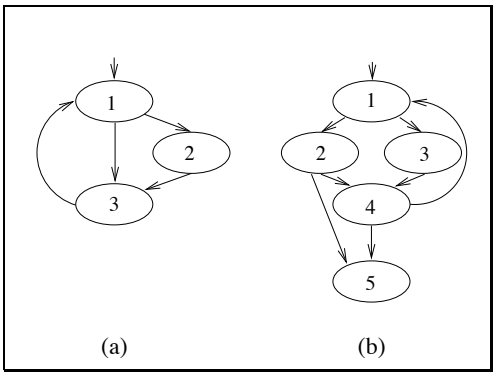


Figure 4.10: (a) A structured and (b), an unstructured graph in which node 1 is the header of two different control structures.

4.5 Structuring Algorithms

This section presents the algorithms designed to determine the underlying control structures of an arbitrary control flow graph (i.e. structured or unstructured, reducible or irreducible). The nodes within the graph are augmented with the structuring information determined which is used during the following phase when generating the high level code.

As we are considering two variations on the algorithm used to structure loops, the treatment of both algorithms is given in separate sections. It also means that the phase diagram in Figure 4.11 describing the order in which the structuring algorithms are performed is not the usual simple sequence. Instead, there is a branch in the path followed by the structuring process when it uses different algorithms depending on how it does the loop structuring.

The *determine ordering* phase imposes an ordering between the nodes of the graphs. The second phase, *determine reverse ordering*, uses the concept of a *reverse graph* and imposes a similar ordering between the nodes in this graph. The *set parenthesis* phase prepares the nodes for loop structuring under the PT algorithm of Section 4.3.2. However as mentioned earlier, the parenthesis of a node can be used to make improvement to both quality of the generated code when 2way conditionals overlap with either loops or

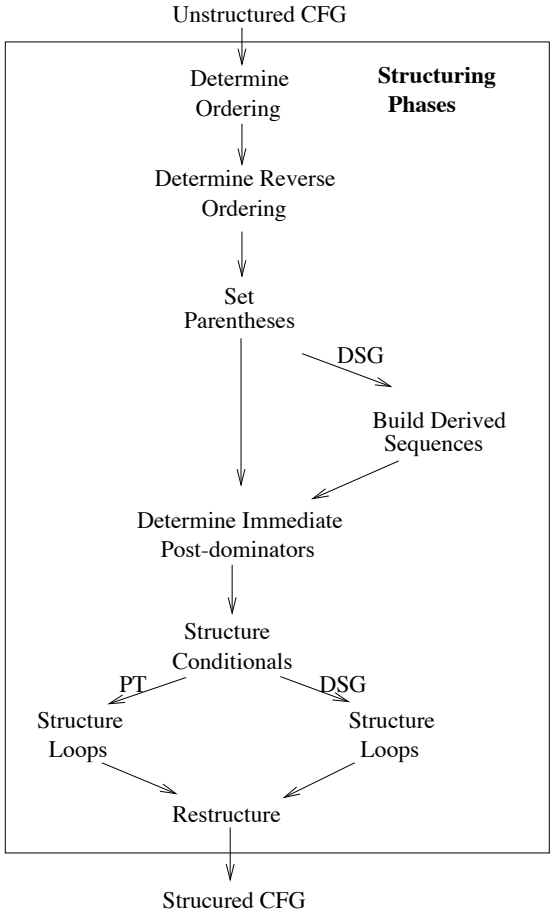


Figure 4.11: The phases performed during structuring. Branches taken by PT algorithms are labelled 'PT' and branches taken by DSG algorithms are labelled 'DSG'.

nway conditionals and so this phase is common to both variations of the overall structuring algorithms. The *build derived sequences* phase, builds the data structures required for loop structuring under the DSG algorithm Section 4.3.1. The next phase, *determine immediate post-dominators* does exactly that for each node in the graph. The *structure conditionals* phase detects the potential conditionals represented within the CFG. The phases diverge again here depending on which loop structuring algorithm is being used. Each alternative, *structure loops* (PT or DSG), detects the loops represented within the CFG using one of the two loop structuring algorithms. Finally, the *restructure* phase gives the aforementioned improvement to the overall structuring process. It reanalyses the 2way conditionals detected in

the conditional structuring phase to see if they can be restructured when they overlap with an nway conditional or a loop. Each phase is next explained in detail.

4.5.1 Determine Node Ordering

The first step in structuring a CFG is to define an ordering between the nodes within the graph. Apart from other uses, this provides an efficient means of traversing the graph in a linear but partially-ordered fashion. The ordering we define is the *post-order* of the nodes. The node that was last visited first during a DFS traversal will be first in the ordering and the start node will have an order equal to the number of nodes within the graph. That is, the nodes are ordered according to when they were last visited. A traversal of the nodes according to this ordering can be visualised as starting from the bottom of the graph and working to the top of the graph.

During the traversal used to determine this ordering, an ordering is also imposed on the in edges to the node from its predecessors. This is the ordering used when later algorithms include a traversal of the in edges in ascending order.

The algorithm designed to perform this traversal is given in Figure 4.12.

```
procedure DefineOrdering (G = (N, E, s), var ord)

* Pre: G is a graph
* Post: the reverse post ordering has been determined

   traversed(s) := true
   for (n ∈ succ(s))
      append s to the end of the predecessor ordering for n
      if (¬ traversed(s))
         DefineOrdering (G = (N, E, n), ord)
   order(s) := ord
   ord := ord + 1
end procedure
```

Figure 4.12: Algorithm to establish the reverse post-order within the nodes of the original CFG.

4.5.2 Determine Reverse Ordering

The reverse graph $G_{rev} = (N_{rev}, E_{rev}, h_{rev})$ of the graph $G = (N, E, h)$ can be defined by the following set of equations (we assume that a unique eit node has been added if it did not already exist):

1. $N_{rev} = N$
2. $E_{rev} = \{(n, m) \mid (m, n) \in E\}$
3. $\exists_1 \, h_{rev} \in N \bullet \nexists n \in N \bullet (h_{rev}, n) \in E$

The order between the predecessors established by the algorithm in Figure 4.12 gives us an ordering between the successors of each node in the reverse graph. The importance of having this relationship between a node's predecessors in the original graph and it successors in the reverse graph becomes apparent when determining the immediate post-dominator of each node. This process relies upon the reverse ordering of the nodes. There are multiple possible reverse orderings and so the aforementioned relationship enables us to constrain the reverse ordering to be directly related to the forward ordering.

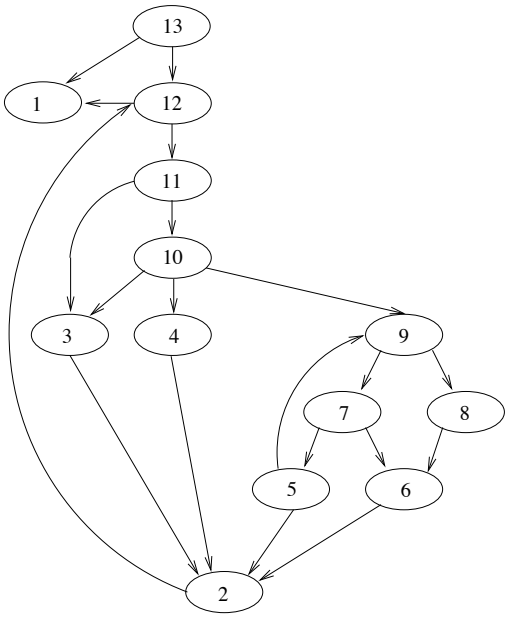
The algorithm for determining the post-ordering of the nodes in the reverse graph in shown in Figure 4.13. Note that the input to this algorithm is the reverse graph. We never actually need to construct this graph as it is just the original graph, viewed in a different way (as shown by the above equations).

```
procedure DefineReverseOrdering (G_rev = (N_rev, E_rev, h_rev), var ord)

* Pre: G_rev is the reverse of graph G and the ordering
* between the predecessors of each n ∈ G has been established
* Post: the reverse post ordering has been determined

   traversed(s) := true
   for (n ∈ succ(h_rev) in ascending order)
     if (¬ traversed(h_rev))
        DefineReverseOrdering (G = (N_rev, E_rev, n), ord)
   revOrder(s_rev) := ord
   ord := ord + 1
end procedure
```

Figure 4.13: Algorithm to establish the reverse post-order for the nodes within the reverse CFG.



Node	Post Order in Reverse Graph
1	13
2	11
3	4
4	5
5	8
6	10
7	7
8	9
9	6
10	3
11	2
12	12
13	1

Figure 4.14: A control flow graph with the nodes identified by their post-order value. The table gives the post-order of each node in the reverse graph. The nodes in the 'Node' column correspond with the nodes in the graph shown.

The control flow graph shown in Figure 4.14 shows how the post-ordering of the nodes enables a top to bottom (or bottom to top) traversal of the graph. The nodes are labeled by their post-order value. This convention of identifying nodes by their post-order value will be followed for the graphs presented hereafter. In this graph, the true edges for a 2way node are on the left. The accompanying table displays the post-order for each node in the reverse graph.

4.5.3 Immediate Post Dominators

The intuitive interpretation of Definition 4.4 is that the immediate post-dominator, *ipd*, of a node, n, is the closest node to n such that all paths from n to the graph's exit node pass through *ipd*. Hecht and Ullman in [12] give an algorithm for finding the dominator tree of a control flow graph. If we consider the graph to be the reverse of another graph, then this algo-

rithm gives us the post-dominator tree. As we are only concerned with the immediate post-dominators, a simplification of their algorithm is fine for our purposes. This simplified version is given in Figure 4.15 and Figure 4.16.

```
procedure FindImmPostDominators(G = (N, E, s), RevG)

* Pre: RevG is the reverse of G and the post-ordering of its nodes has been determined
* Post: the immediate post-dominator for each node has been determined.

   /* first pass */
   for (all n ∈ N in descending order in RevG)
     iPDom(n) := undefined
     for (all c ∈ pred_rev(n))
       if (revOrder(c) > revOrder(n))
         iPDom(n) := CommPostDom(iPDom(n),c)

   /* second pass */
   for (all n ∈ N in ascending order in G • # succ(n) > 1)
     for (all c ∈ succ(n))
       iPDom(n) := CommPostDom(iPDom(n),c)

   /* third and final pass */
   for (all n ∈ N in ascending order in G • # succ(n) > 1)
     for (all c ∈ succ(n))
       if (isBackEdge(n, c) ∧ order(iPDom(c)) < order(iPDom(n))
         iPDom(n) := CommPostDom(iPDom(n),iPDom(c))
       else
         iPDom(n) := CommPostDom(iPDom(n),c)
end procedure
```

Figure 4.15: Algorithm for calculating the immediate post-dominators.

The first algorithm, *FindImmPostDominators*, makes use of the ordering defined between the nodes of the reverse graph. To make sure that the immediate-post dominator for each node is correct with respect to the original CFG, we require three passes over the nodes in the graph.

The first pass traverses the nodes using the ordering defined within the reverse graph. Intuitively, we traverse the nodes in the reverse graph from top to bottom. For each node, we consider its predecessors that are higher in the reverse graph (i.e. they are not reached via a back edge). If there is only one such predecessor, then it is determined to be the immediate post-dominator. Otherwise, the immediate post-dominator is initialised to be the common post-dominator of all the predecessors. Finding this common post-dominator is performed by the *CommPostDom* algorithm. The result of this

```
procedure CommPostDom (var ipd, s)

* Pre: ipd is the current immediate post-dominator and s is a successor of the
* node being considered and is lower in the graph
* Post: ipd is the immediate post-dominator of the node under consideration.

   if (ipd = undefined)
      ipd := s
   else if (s = undefined)
      ipd := ipd
   else
      while (ipd and s are defined ∧ ipd ≠ s)
         if (revOrder(ipd) > revOrder(s))
            s := iPDom(s)
         else
            ipd := iPDom(ipd)
end procedure
```

Figure 4.16: Common post-dominator algorithm.

first pass is that every node except for the exit node has now been given an immediate post-dominator.

The second pass is required because the immediate post-dominator information determined in the first pass is not necessarily correct with respect to the original control flow graph. This occurs only when the original graph includes loops, which is the case for most control flow graphs. The CFG shown in Figure 4.17 is an example where we cannot rely upon the immediate post-dominator information gained from only one pass. The three graphs depict the state of each node after each pass. Each node is given an alphanumeric identifier. The parenthesised data for each node is its position in the reverse ordering of the graph and its immediate post-dominator after the relevant phase. Shaded nodes are those whose immediate post-dominator changed after the previous pass.

The immediate post-dominators determined for nodes B,C and E after pass 1 are wrong. For node B, there is a path (B,D,H) to the exit node (H) that does not pass through C. Therefore, C is not a post-dominator of B, let alone the immediate post-dominator. The reason for this error is that the reverse ordering algorithm is 'confused' by the presence of loops. For instance, one would expect node D to have a higher reverse order than both A and B as it is 'higher' than both of these in the reverse graph. The problem arises from the fact that it is impossible to include in the reverse ordering algorithm any

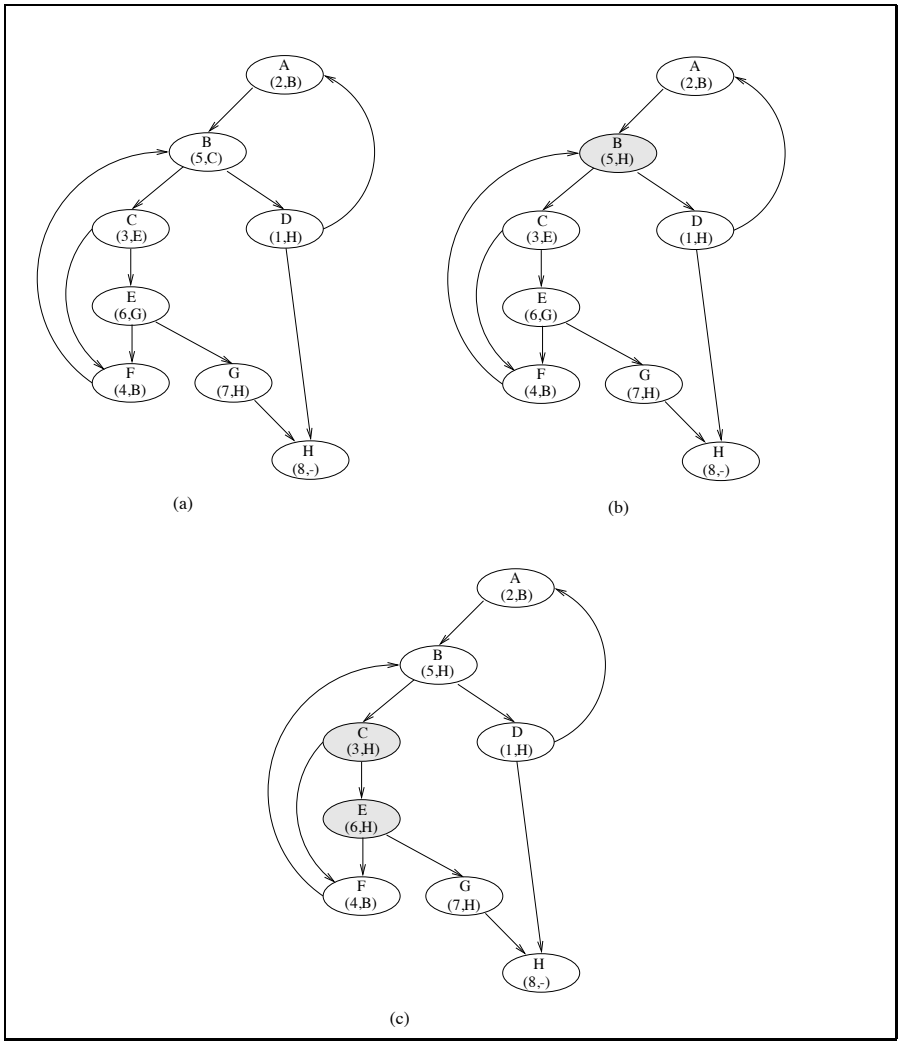


Figure 4.17: A CFG that has different reverse graphs.

capabilities for dealing with loops in the reverse graph as it does not know what are back edges until the reverse ordering has been completed.

The second pass attempts to correct the remaining errors by doing another traversal based on the original ordering of the CFG. This time, only nodes that have 2 or more out edges are considered as the immediate post-dominators determined for single out edges in pass one will be correct (there is no other choice of immediate post-dominators for these nodes). In the example, only node B's immediate post-dominator is corrected. Nodes C and

E are left uncorrected as they were traversed in the second pass before B was and they both rely on B to be correct.

After the second pass, there is still a possibility (as shown in the example) that some nodes within a loop (including the latching node) will still have the incorrect immediate post-dominator. The third pass corrects these remaining nodes. Preliminary analysis seems to indicate that a third pass is only required when the graph contains overlapping loops such as the two loops in the example induced by the back edges (D,A) and (F,B). The table in Figure 4.18 gives the immediate post-dominators determined for the nodes in the graph of Figure 4.14.

Node	1	2	3	4	5	6	7	8	9	10	11	12	13
ImmPDom	-	12	2	2	2	2	2	6	2	2	2	1	1

Figure 4.18: The immediate post-dominators for each node in the graph of Figure 4.14.

4.5.4 Structuring Conditionals

When structuring conditionals, we are analysing nodes that have more than two out edges. For the subgraph headed by one of these nodes, there will be a unique node such that all paths from each of the out edges from the conditional header converge upon it. As mentioned in Sections 4.4.2 and 4.4.3, this is referred to as follow node. Defined in this way, the follow node corresponds with the conditional header's immediate post-dominator and is already determined once this immediate post-dominator is known.

The set of nodes on all paths starting from some successor of a conditional header is called a *branch*. This set is defined in terms of the conditional $h \diamond f$ and one of the succesors of h such that it is not f.

Definition 4.12 *For* $\exists\, s \in (succ(h) \backslash \{f\})$, *a* **branch** *of the conditional* $h \diamond f$ *is* $\{n \in N \mid n \text{ is on a path } s \rightarrow f \wedge n \notin \{h, f\}\}$.

Note that the above definition implies that a branch may be empty. If a subgraph represents a structured conditional $h \diamond f$, then it will satisfy the following two properties:

1. $\forall\, b_i \in \textit{branches of } h\diamond f \bullet \bigcap_i (b_i) = \{\,\}$

2. $\forall\, b \in \textit{branches of } h\diamond f \bullet (\nexists n \in b \bullet (m,n) \in E \wedge m \notin (\{h\} \cup b))$

The first property states that the membership of the branches must be mutually exclusive (i.e no node may belong to more than one branch of a given conditional). The second property says the predecessor of any node within a branch must be either another node within the same branch or the conditional header. An unstructured conditional will violate at least one of these properties.

Whenever, conditional subgraphs are nested within each other, it may be the case that they share the same follow node. This will mean that both the outer and inner conditional headers will share the same immediate post-dominator. The same will also be true for overlapping (unstructured) conditionals.

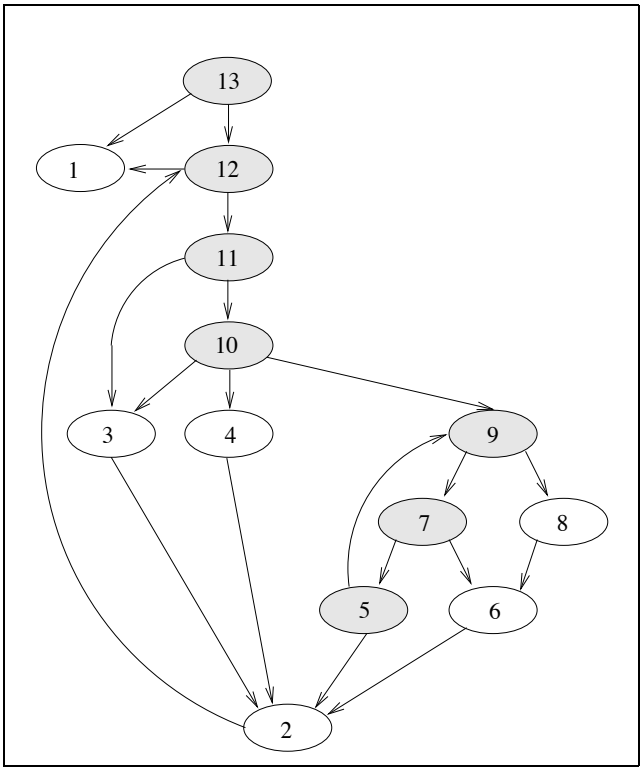


Figure 4.19: A control flow graph with the potential conditional nodes shaded.

Consider the graph in Figure 4.19. The shaded nodes indicate the potential conditional header nodes. Even though some of these may later be structured as loop headers or latching nodes, we still consider them when structuring conditionals. The reason for this is that a node may be both a conditional and a loop header (as discussed in Section 4.4.4).

Determining the type of a 2way conditional

For a 2way conditional, we seek to determine exactly what type of 2way conditional it is. The possibilities are:

- **if..then..else** - neither successor of the header is the follow node or at least one successor is reached by a back edge.
- **if..then** - the false successor is the follow node.
- **if..else** - the true successor is the follow node.

Figure 4.19 gives an example of these 2way conditional types. The conditionals (13◇1) and (12◇1) are **if..else** type of 2way conditionals as their true branches (i.e. the left branches) lead directly to their follow nodes. When generating HLL code for this type of conditional, we have the option of either generating an **if..then..else** that has an empty **then** clause or we can negate the condition in the header node and generate an **if..then** statement. The latter is a more natural looking statement and so we choose it. The conditionals (7◇2), (9◇2) and (11◇2) are **if..then..else**'s as neither of the successors of the respective header nodes are the follow node. The conditional (5◇2) is an interesting case as one of its out edges is a back edge. This type of node will always be initially structured as an **if..then..else** header. As conditional structuring is performed before loop structuring, there is still a chance it may be detected as the latching node for a loop.

Determining the nodes within an nway conditional

For an nway conditional, we want to tag all the nodes on a branch of the conditional with the header node of their most enclosing nway conditional. This information will be used later when performing loop structuring.

The algorithm used to tag each node within a given nway conditional $h\diamond f$ is essentially a modified depth first traversal from h. All nodes (except for the nway header node and its follow) visited during the traversal are tagged with h as their nway header unless it has already been tagged with another nway header node or it was reached via a back edge. The base case for the recursive call is when the follow node is reached. If $h\diamond f$ encloses another nway conditional, $h_2\diamond f_2$ (where f_2 may be f), then all the nodes within $h_2\diamond f_2$ will already have been tagged. This is because we use the post-ordering of the nodes to traverse the graph in such a way that inner nway conditional will be structured first. Therefore, when we come across the header of an inner (or nested) nway conditional, we can jump directly to the follow node of the inner conditional. This means that each node will only be traversed once during all the nway tagging traversals for the complete CFG. The only nway conditional in Figure 4.19 is $(10\diamond 2)$. The nway tagging traversal for this conditional will tag nodes 3 . . 9 as having node 10 as their most enclosing nway conditional header. The algorithm that performs this traversal is shown in Figure 4.20.

```
procedure TagNodesInCase (G = (N, E, n), h, f)

* Pre: (h◇f) is an nway conditional and n = h or is in a branch of (n◇f)
* Post: all the nodes within the branches of (h◇f)
* that are not already members of another nway conditional have been
* tagged as belonging to this nway

   if (n ≠ h)
      caseHead(n) := h
   if (nodeType(n) = nway ∧ condFollow(n) ≠ f)
      TagNodesInCase(G = (N, E,condFollow(n)), h, f)
   else
      for (all c ∈ succ(n) ∧ caseHead(c) is undefined)
         if (c ≠ f ∧ ¬ isBackEdge(n, c)
            TagNodesInCase(G = (N, E, c), h, f)
end procedure
```

Figure 4.20: Algorithm to Tag the Nodes within an Nway Conditional.

Complete Conditional Structuring Algorithm

The complete algorithm for structuring both types of conditionals is given in Figure 4.21. The nodes of the graph are traversed by an ascending post-

order traversal. This ensures that inner conditionals are structured before outer ones. Only nodes with at least two out edges are analysed during the traversal.

```
procedure StructConds (G = (N, E, h))

* Pre: The immediate post-dominator has been determined for each node
* Post: The conditional headers have been tagged, their follows have been determined.
* Each node has been tagged with the header of its most enclosing nway conditional
* and the type of a 2way conditional has been determined.

   for (all n ∈ N in ascending order • #succ(n) > 1)
      structType(n) := Cond
      if (∃ m • (n, m) ∈ E ∧ isBackEdge(n, m))
         condType(n) := ifThenElse
      else
         condFollow(n) := iPDom(n)

         if (n is an nway node)
            condType(n) := Case
            TagNodesInNway(n,condFollow(n))
         elseif (succ(n,Then) = condFollow(n))
            condType(n) := ifElse
         elseif (succ(n,Else) = condFollow(n))
            condType(n) := ifThen
         else
            condType(n) := ifThenElse
end procedure
```

Figure 4.21: Conditional Structuring Algorithm

4.5.5 Loop Structuring using Derived Sequences of Graphs

This section describes how loops are structured using the DSG theory. It is based upon the work on loop structuring given in [5]. Slight modifications have been made as a result of further development on the original algorithm.

The first step in structuring loops is to determine how a loop is represented within a CFG. A loop must involve a back edge from its latching node to its header node. Therefore, we want to know both of these nodes for any loop. We also want to know which nodes are within a given loop as well as a nesting order between the loops. Hecht in [8] made the point that the

representation of loops by cycles is too fine as not all loops are properly nested or disjoint. The use of strongly connect components is too coarse as there is no nesting order whereas strongly connected regions do not provide a unique coverage of the graph and does not cover the whole graph. The interval and derived sequence of graphs theory discussed in Section 4.3.1 provides us the information we require: one loop per interval and a nesting order given by the derived sequence of graphs.

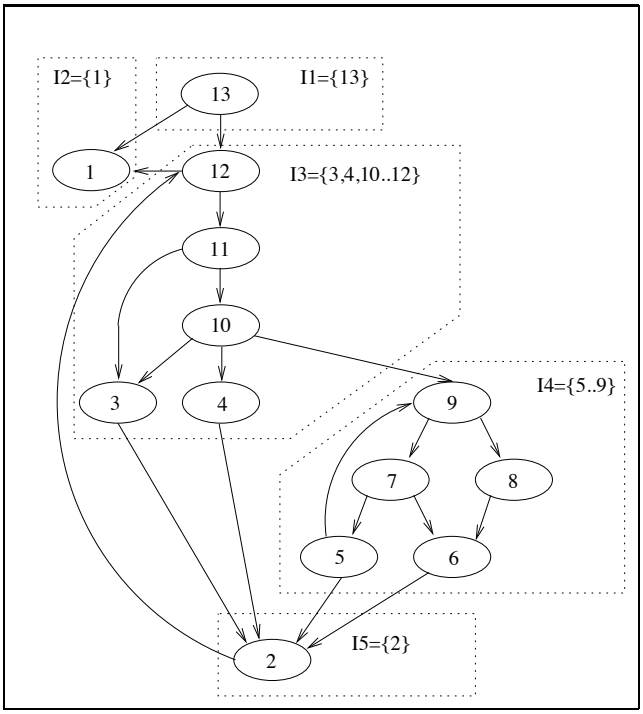


Figure 4.22: The intervals of the Control Flow Graph of Figure 4.14

For some interval $I(h)$ (where h is the interval header), there is a loop headed by h if there is an edge from some node $l \in I(h)$ to h. These edges will invariably be a back edge and l will be the *potential* latching node for the loop. Figure 4.22 shows the intervals determined for the graph of Figure 4.14. The interval I_4 is the only interval so far that has a loop in it. This loop, $9 \circlearrowright 5$ results from the back edge from 5 to 9.

To detect the loop $12 \circlearrowright 2$, we make use of the derived sequence of graphs. To find the next graph in the sequence, we collapse each interval into a single node and form the new graph from these nodes. Figure 4.23 shows the derived sequence of graphs for the original flow graph of Figure 4.22. In the

first derived graph, G^1, the loop 12 $\circlearrowright$ 2 is now detectable by the back edge from the node I_5 (which contains node 2) to the node I_3 (which contains node 12). The derived graph G^2 does not contain any back edges and therefore, there are no more loops in the graph.

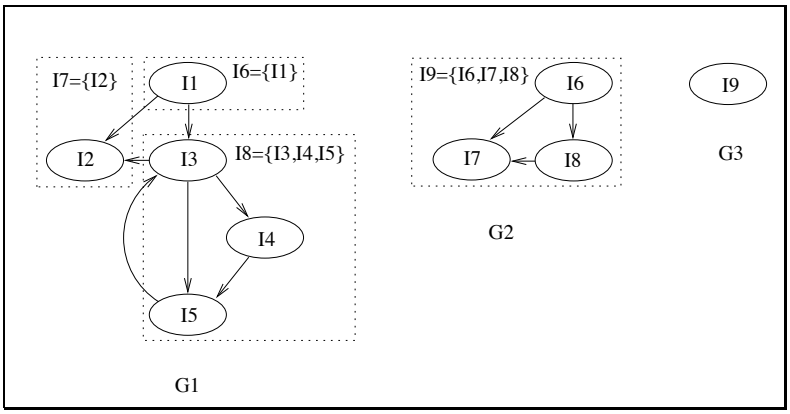


Figure 4.23: Derived Sequence of Graphs for Figure 4.22

The nesting of loop 9 $\circlearrowright$ 5 within loop 12 $\circlearrowright$ 2 is derived from the fact that the former is was detected earlier in the sequence of derived graphs.

We made a note earlier that a back edge from within an interval to the interval header is only a *potential* latching node. It may be the case that a more suitable latching node can be found. Consider the example in Figure 4.24. In the first graph, G^0, the interval I_1 contains a back edge to the interval header, node 1, from node 2 (which is within the same interval). This will result in the initial structuring of the loop 1 $\circlearrowright$ 2. When considering the next graph in the sequence, G^1, we are presented another possible latching node, node 4 (from interval I_2) for a loop headed by node 1. We therefore restructure the loop to be 1 $\circlearrowright$ 4 as it encloses more nodes. We generalise this decision and state that the latching node found latest in a sequence of derived graphs will always overide an alternative latching node for the same loop header found earlier in the sequence.

Given a control flow graph $G = G^0$ that has been annotated with its intervals, its derived sequence of graphs $G^0 \,..\, G^n$ and the set of intervals of these graphs, $\mathcal{I}^0 \,..\, \mathcal{I}^n$, we can use the following algorithm to structure loops: each header of an interval in G^0 is checked for having a back edge from a latching node that belongs to the same interval and is enclosed by the same nway

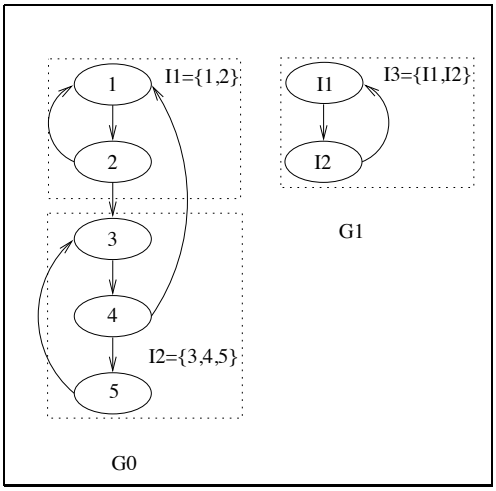


Figure 4.24: A loop header with two potential latching nodes.

conditional[2]. We choose the latching with the least post-order value if there is more than one. If a latching node was found, then we check to see if the header has already been structured as a loop header (from a derived graph earlier in the sequence). If so, then this latching node is set back to what is was determined to be (an **if..then..else** conditional header) before being detected as a latch node and all the nodes within the now defunct loop have their loop header marker set back to undefined. We now set the latch node to be the most recently found latch node and together with the header node, a loop has been found. We now determine its type (preTested, postTested or endless) and mark all the nodes belonging to it. Next, the intervals of $G^1, \mathcal{I}^1$ are checked for loops and the process is repeated until the intervals in $\mathcal{I}^n$ have been processed. Whenever there is a potential loop (i.e. an interval header that has a back edge into from a node within the same interval) such that the loop header and latch node are currently marked as belonging to different loops, the new loop is disregarded as it belongs to an unstructured loop. This back edge will always generate a **goto** during code generation. The complete algorithm is given in Figure 4.25. This algorithm correctly determines the appropriate nesting levels between the loops. Section A.3 describes the loopNodes attribute used in this and other loop structuring algorithms.

[2]See Section 4.4.3 for an explanation of why this check is made.

```
procedure StructLoops (G = (N, E, h))

* Pre: G^1 .. G^n and I^1 .. I^m have been constructed.
* Post: all nodes of G have been tagged with their most enclosing loop and all
* loop headers have been given their loop type, loop follow (if any) and latching node

   for (G^i := G^1 .. G^n)
     for (I^j with header h_j := each of the intervals in G^i)
       if (∃x ∈ nodesOf(I^j) •
             (x, h_j) ∈ E ∧ loopHead(x) is undefined ∧ caseHead(x) = caseHead(h_j) ∧
             ∄y ∈ nodesOf(I^j) • (y, h_j) ∈ E ∧ order(y) < order(x))
           if (h_j is already a loop header with latch l_j)
             for (n ∈ loopNodes(h_j ↻ l_j))
               loopHead(n) := undefined
             reset condType(l_j) if necessary
           structType(h_j) := loop
           FindLoopNodes(h_j, x, I^j)
           DetermineLoopType(h_j, x)
           FindLoopFollow(h_j, x)
end procedure
```

Figure 4.25: Loop Structuring Algorithm.

Determining the Nodes in a Loop

The graph in Figure 4.14 contains two loops that will be detected under the DSG algorithm; 12 $\circlearrowright$ 2 and 9 $\circlearrowright$ 5. It can be seen that for both of these loops, the set of nodes given by the following equality is an intuitively valid membership for each loop:

$$\mathit{loopNodes}(h \circlearrowright l) = \{n \in N \bullet \text{order}(l) \geqslant \text{order}(n) < \text{order}(h)\}$$

That is, the loop membership is determined to be all the nodes that have a post-order value in between the post-order values of the header and latch node, excluding the header node itself. Consider the graph in Figure 4.26. The loop membership for 5 $\circlearrowright$ 1 given by the above condition will be {5,4,3,2,1}. However, we can see that nodes 3 and 4 are not really appropriate nodes to include in the loop. If we restrict the above condition to include a test on interval membership, we will get the desired loop membership:

$$\mathit{loopNodes}(h \circlearrowright l) = \{n \in N \bullet \\ \text{order}(l) \leqslant \text{order}(n) < \text{order}(h) \wedge n \in I(h))\}$$

That is, a candidate node for loop membership has to also be within the

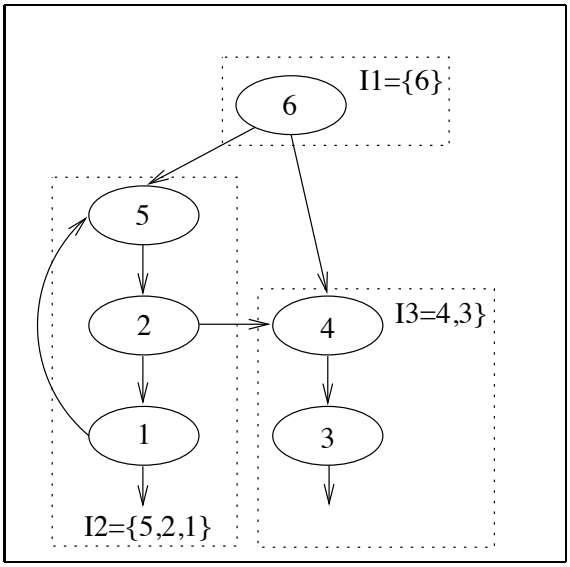


Figure 4.26: A loop that requires the use of intervals to derive the correct membership.

interval headed by the loop header. In the given example, only nodes {5,2,1} satisfy the new condition as they are all in the interval (I_2) headed by the loop header. They are also the only nodes we intuitively desire to determine as members of the loop.

The algorithm in Figure 4.27 is used to find all the nodes within a given loop. If a node found to be within a loop is not already part of another loop, then it is marked by setting its loopHead attribute to be the header of the loop. This ensures that all nodes are marked with the loop header of their most enclosing loop.

Determining the Type of a Loop

We rely upon the header and latch nodes of a loop to determine its type. The header of a preTested loop will be a 2way node that controls the loop iterations and will have a latching node that has a solitary edge back to the loop header. One of the edges from a preTested loop header will lead to follow node for the header when it was previously structured as a 2way conditional. A postTested may have either a 2way or one edge header node and a 2way latching node. It will have a 2way header node if the first statement within

```
procedure FindLoopNodes(G = (N, E, s), h, l, I)

* Pre: (h ↻ l) is a loop.
* Post: the loop nodes are marked as being within the loop and if it is their most
* enclosing loop, the corresponding loop loop header tag has been set.

   for (all m • order(h) > order(m) ⩾ order(l) in descending order)
     if (m ∈ baseNodesOf(I))
       loopNodes(h ↻ l) := loopNodes(hLoopl) ∪ {m}
       if (loopHead(m) = undefined)
         loopHead(m) := h
end procedure
```

Figure 4.27: Algorithm to tag the nodes within a loop.

the loop is a 2way conditional. An endless loop will have either a 2way or one edge header node for the same reasons as a postTested loop but it will only have a one edge latching node. The only difference between an endless loop with a 2way header and a preTested loop is whether or not one of the successors of the header is also its conditional follow node.

Using these heuristics to determine the type of a loop means that we will only ever find structured preTested loops but we can find both structured and unstructured versions of the other two types.

The types of the two loops in Figure 4.14 are as follows: 12 ↻ 2 is a preTested loop as firstly, it has a 2way header node, a successor of which, is also the follow node found when the header was initially structured as a 2way conditional. Secondly, it has a one edge latching node. The other loop, 9 ↻ 5, will be an endless loop as it has a one edge latching node and a 2way header node. Neither successor of the header is its conditional follow.

Figure 4.28 gives the algorithm to determine the type of a loop based upon its header and latching node and the conditional follow found for a 2way loop header when it was structured as a conditional. A single node loop will always be determined as either a postTested (2way latch) or an endless (one edge latch) loop. For both a postTested and endless loop, the conditional structuring information for the header will be retained if it is a 2way node.

```
procedure DetermineLoopType(G = (N, E, s), h, l)

* Pre: (h ↻ l) is a loop and set of nodes within the loop has been determined.
* Post: the type of the loop has been determined.

   if (nodeType(l) = 2way)
      loopType(h) := postTested
      if (nodeType(h) = 2way ∧ h ≠ l)
         structType(h) := loopCond
   else if (nodeType(h) = 2way)
      if (∃ x • x = succ(h) ∧ x = condFollow(h))
         loopType(h) := preTested
      else
         loopType(h) := endless
         structType(h) := loopCond
   else
      loopType(h) := endless
end procedure
```

Figure 4.28: Algorithm for determining the type of a loop.

Determining the Loop Follow

The loop follow node is the first node that is reached upon loop termination. For both structured preTested and postTested loops, there is only one candidate for the follow node but there may be multiple candidates in the unstructured forms of these two loops as well as for endless loops. When designing an algorithm to find the follow node, we aim to find the most appropriate one if there is more than one to choose from.

In a preTested loop, the follow node is determined to the non-conditional follow node of the loop header. It will always be the case that exactly one of the header's successors will be the non-conditional follow as we used this condition to structure the loop as a preTested loop in the first place. For postTested loop, the follow node will be the successor of the latch node that is not the loop header. For an endless loop, there may be zero or more jumps out of the loop. Since the follow node is the first node that is reached upon loop termination, we aim to find the node that is reached from the loop after an exit such that it is higher in the graph than all other nodes that are reached from the loop after an exit. This will be the candidate node with the highest post order value. This property of the chosen follow node means that all the other candidate nodes can be reached from it.

```
procedure FindLoopFollow(G = (N, E, s), h, l)

* Pre: (h ↻ l) is a loop and its nodes and type have been determined
* Post: the loop's follow (if any) has been determined

   if (loopType(h) = preTested)
     if (succ(h, 1) = condFollow(h))
       loopFollow(h) := succ(h, 1)
     else
       loopFollow(h) := succ(h, 2)
   else if (loopType(h) = postTested)
     if (succ(l, 1) = h)
       loopFollow(h) := succ(h, 2)
     else
       loopFollow(h) := succ(h, 1)
   else          /* endless loop */
     foll := undefined
     for (all m ∈ loopNodes(h ↻ l) • nodeType(m) = 2way)
       for (all x • x ∈ succ(m) ∧ x ≠ h)
         if (x ∉ loopNodes(h ↻ l) ∧ (order(x) > order(foll) ∨
               foll = undefined))
             foll := x
     if (foll ≠ undefined)
       loopFollow(h) := foll
end procedure
```

Figure 4.29: Algorithm for determining the follow node of a loop.

In Figure 4.14, 12 ↻ 2 has node 1 as it follow node and 9 ↻ 5 has node 2 as its follow node. Figure 4.29 gives the algorithm to determine the follow node of a loop based on the type of the loop detremined in the algorithm of Figure 4.28 and the set of nodes within the loop determined in the algorithm of Figure 4.27.

4.5.6 Loop Structuring using Parenthesis Theory

This section describes the loop structuring algorithms that have been developed as a more efficient alternative to those given in Section 4.5.5. Under the parenthesis theory (PT), we need to alter our definition of how loops are represented within a control flow graph. Without the use of the extra data structures defined for the DSG algorithm, we simply define a loop to *potentially* occur where there is a back edge. That is, we initially structure loops only by detecting cycles within the graph. We have to then rely of the

parentheses of the nodes to further constrain the loop structuring.

The cycles are found during a traversal of the nodes in ascending order. For a graph with only structured loops, this will ensure that outer loops are discovered before inner loops. We retain this assumption and use extra analysis in the face of unstructuredness.

Having found a back edge (l, h), we initially check the nodes l and h for meeting three conditions before structuring them as the latch and header of a new loop. Together, they must satisfy the following conditions:

1. Both h and l must currently belong to the same loop (i.e. have the same loop header). This may be undefined if neither is enclosed by a loop. This ensures that we do not structure overlapping loops.

2. Both h and l must currently belong to the same nway conditional (i.e. have the same nway header). This may be undefined if neither is enclosed by an nway conditional[3].

3. There must be no other back edge (l_2, h) such that l_2 is lower than l in the graph. This ensures that the most enclosing loop (i.e. the loop with the largest membership) is found.

Having found a new loop, $h \circlearrowright l$, we then determine its membership, its type and its follow node. The membership information is used when looking for the next loop. The algorithm shown in Figure 4.30 performs this loops structuring.

The graph in Figure 4.14 will have both the back edges, (2,12) and (5,9), analysed by this algorithm as potential loop inducing back edges. The edge (2,12) will be analysed first as node 2 is lower in the graph than node 5. This demonstrates that the outer loops will be structured first. This edge will successfully induce the new loop, $12 \circlearrowright 2$, as it meets the three necessary conditions; neither of them belong to a loop, neither belong to an nway conditional and node 2 is the only (and therefore lowest) node with a back edge to node 12. We delay discussion of whether or not the edge (5,9) induces a new loop until we have presented the method used to decide which nodes are in a loop.

[3]See Section 4.4.3 for an explanation of why this check is made.

```
procedure StructLoops (G = (N, E, h))

* Pre: G is a graph and the parenthesis pair for each node have been determined.
* Post: all nodes of G have been tagged with their most enclosing loop. All loop
* headers have been given their loop type, loop follow (if any) and latching node.

  for (all h ∈ N in ascending order)
    if (∃ x ∈ pred(h) •
        loopHead(x) = loopHead(h) ∧
        caseHead(x) = caseHead(h) ∧
        isBackEdge(x, h) ∧
        ∄y ∈ pred(h) • order(y) < order(x))
      if (x ≠ h)
        structType(x) = seq
      structType(h) = loop
      FindLoopNodes(G, h, x)
      DetermineLoopFollow(G, h, x) end procedure
```

Figure 4.30: Loop Structuring Algorithm.

Determining the Nodes in a Loop

Given the loop $h \circlearrowright l$, we make use of the parenthesis theory introduced in Section 4.3.2 to determine which nodes are in the loop. We determine a node n to be within the loop if $n \succ h$ and $l \succ n$. The latching node is also considered to be within the loop body but the header is not. Using both parentheses for the node ensures that we detect as many nodes as possible using this method.

Figure 4.31 gives a graphical representation of the true parenthesis in (a) and false parenthesis in (b) for each node in the graph of Figure 4.14. For the loop $12 \circlearrowright 2$, we detect the nodes 11 and 3 using the true parenthesis and nodes 10, 9, 8 and 6 using the false parenthesis to be nodes of the loop. This can be explained visually br referring to the example graph and its nodes' parentheses. We see that the line in Figure 4.31 representing the false parenthesis of node 2 (the latch node) lies within the line representing the false parenthesis of 10 which in turn lies within the line representing the false parenthesis of 12 (the header node). Therefore, node 10 is within the loop. The same relationship between the lines of the other nodes determined to be within the loop can also be observed. As with the DSG algorithm, we only need to consider the nodes that lie within the post-order range of the loop header and latch. The following equation defines the set of loop nodes that

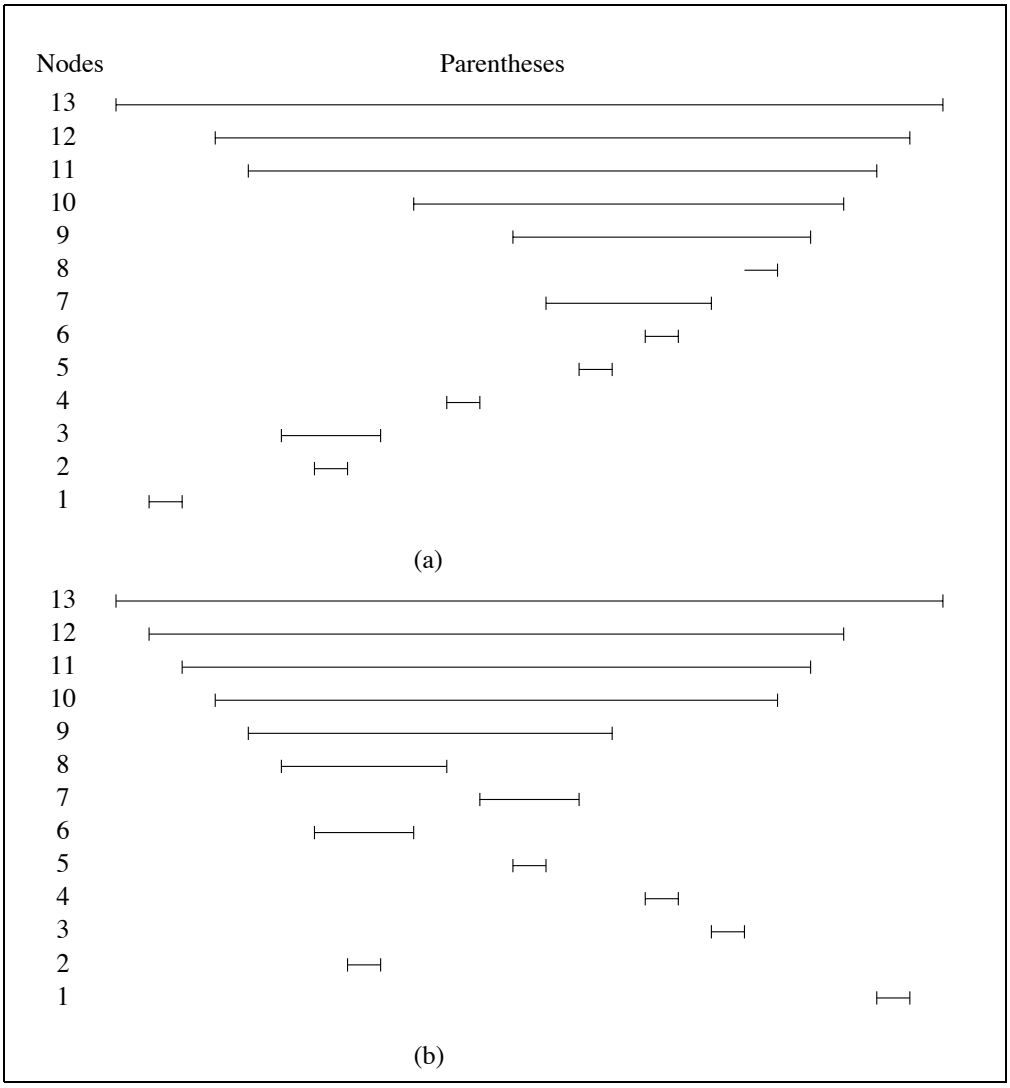


Figure 4.31: The (a) true and (b) false parenthesis for each node in the graph of Figure 4.14.

will be found:

$$loopNodes(h \circlearrowright l) = \{n \in N \bullet \text{order}(l) \geqslant \text{order}(n) < \text{order}(h) \wedge n \in I(h))\}$$

The number of nodes detected to be within the loop is considerably less than was detected by the DSG algorithm. However it is the most that can be achieved using only information provided by a DFS tree perspective of the

graph (which is essentially what the parenthesis theory gives us).

An immediate effect of having determined node 9 to be in the loop but not node 5 is that when we analyse the back edge (5,9) as a potential loop inducing back edge, the first condition for it to induce a loop will not be met. That is, node 5 and 9 will not both belong to the same loop.

Figure 4.32 give the algorithm for finding the nodes in a loop given the loop header and latch node as well as each node's pair of parenthesis.

```
procedure FindLoopNodes(G = (N, E, s), h, l, I)

* Pre: (h ↻ l) is a loop.
* Post: the loop nodes are marked as being within h ↻ l and if it is their most
* enclosing loop, the corresponding loop header tag has been set.

   for (all m • order(h) > order(m) ⩾ order(l) in descending order)
     if (m ≻ h) ∧ (l ≻ m ∨ m = l))
       loopNodes(h ↻ l) := loopNodes(h ↻ l) ∪ {m}
       loopHead(m) := h
end procedure
```

Figure 4.32: Algorithm to tag the nodes within a loop.

Determining the Type of a Loop

As with the DSG loop type determining algorithm, we are given the header and latch nodes of a loop, from which we must determine the type of the loop. As we aim to preserve as much similarity between the functionality of the two variations of loop structuring algorithms as possible, we use the same algorithm for this component of loop structuring as was given for the DSG algorithm in Figure 4.28. The type determined for the loop 12 ↻ 2 in our example graph is therefore the same as if detected under the DSG algorithm; node 1.

Determining the Loop Follow

As with the procedure followed to determine the type of a loop, we once again borrow heavily from the DSG algorithm when designing the algorithm to determine the follow node of a loop. The only change we have to make to

the algorithm shown in Figure 4.29 is to the subcomponent that determines the follow node for an endless loop.

If a conditional node, n, and its follow, f, are both detected as being in a loop, there is no guarantee (in contrast to the DSG algorithm where there is a guarantee) that both of h's successors will also be in the loop. For example, consider the endless loop 8 $\circlearrowright$ 1 in Figure 4.33. The shaded nodes indicate those determined to be within the loop. If we were to apply the same method for determining the loop follow of an endless as was used in the corresponding DSG algorithm, then node 4 will be considered as a potential loop follow. Ihis is obviously a bad choice as the only path from node 4 leads directly to a node within the loop. The modification we make to the algorithm to ensure that this type of node is not considered as a potential endless loop follow is as follows: the follow of an endless loop (if any) is determined to be the node with the highest post-order value such that it is not within the loop and it is the successor of a 2way conditional header that *is* in the loop but whose conditional follow is not in the loop.

The algorithm used to find the follow is given in Figure 4.34. If whilst searching for a 2way conditional that has a follow outside the loop, we come across one, $h \diamond f$, that has its follow inside the loop, we do one of two things depending on whether f is lower in the graph than h (i.e. has a lesser post-order value):

1. If f is lower in the graph than h, we continue the traversal from it as all the nodes whose post order value is in the range `order`(f).. `order`(h) will be within the body of the conditional and will therefore also really be inside the loop (even though it may not be tagged as such). This will happen when analysing node 3 in Figure 4.33.

   OR

2. If f is not lower in the graph than h then this the result of an unstructured backwards jump. This will hold for any conditionals further down the graph within the loop and so we just break out of the traversal of the nodes within the loop.

If after completing this traversal a suitable follow has been found, then it is set to be the follow of the loop.

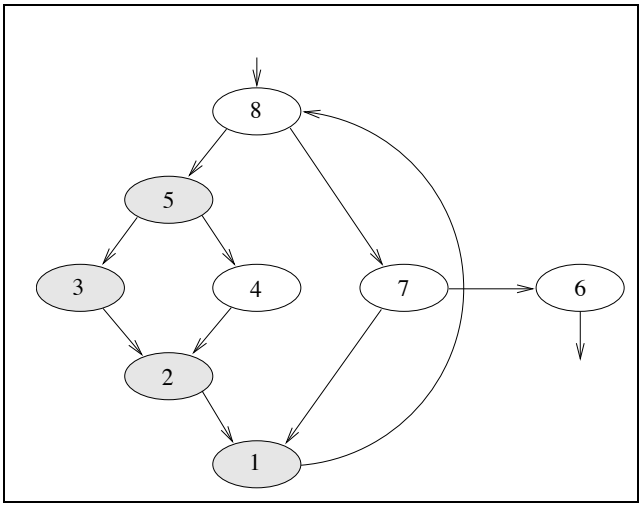


Figure 4.33: An Endless loop.

4.6 Restructuring 2way Conditionals

To improve the code generated when a 2way conditional overlaps with a loop or nway conditional, we make use of the nodes' nway header tag and their loop header tag. For a 2way conditional, $h \diamond f$, that represent an unstructured jump into or out of a loop, h's loop header tag will be different to f's loop header tag. Similarly for an unstructured jump into or out of an nway conditional — h and f's respective nway header tags will be different. We use this information to modify the type of a 2way conditional and its follow where doing so will make for generation of better HLL code.

We justify the need for restructuring through an example. The example includes some generated HLL code. There is no great need to understand how this code was generated, just to observe the overlook of it. Consider nodes 7 and 9 of the graph in Figure 4.14. Node 7 is the header of the 2way conditional $7 \diamond 2$ which has been determined as an ifThenElse type of conditional and 9 is the header of the conditional $9 \diamond 2$ (as well as being the header of the loop $9 \circlearrowright 5$) which also has the type ifThenElse. If we were to leave the 2way conditionals as they were originally structured and use a code generation algorithm similar to that given by Cifuentes in [5] then, the subgraph entered by 9 and exited at 2 would generate the code shown in Figure 4.35.

```
procedure FindLoopFollow(G = (N, E, s), h, l)

* Pre: (h ↻ l) is a loop and its nodes and type have been determined
* Post: the loop's follow (if any) has been determined

   if (loopType(h) = preTested)
      if (succ(h, 1) = condFollow(h))
         loopFollow(h) := succ(h, 1)
      else
         loopFollow(h) := succ(h, 2)
   else if (loopType(h) = postTested)
      if (succ(l, 1) = h)
         loopFollow(h) := succ(h, 2)
      else
         loopFollow(h) := succ(h, 1)
   else          /* endless loop */
      foll := undefined
      for (all m ∈ loopNodes(h ↻ l) in descending order •
            nodeType(m) = 2way ∧
            condFollow(m) is defined)
         if (condFollow(m) ∈ loopNodes(h ↻ l))
            if (order(m) > order(condFollow(m)))
               m := condFollow(m)
            else
               break
         else
            if (∃x ∈ succ(m) •
                  x ∉ loopNodes(h ↻ f) ∧
                  (foll = undefined ∨ order(foll) < order(x)))
               foll := x
      if (foll ≠ undefined)
         loopFollow(h) = foll
end procedure
```

Figure 4.34: Algorithm for determining the follow node of a loop.

We can improve on this by recognising that one of the branches of each of the two conditionals, 9◇2 and 7◇2, leads along a path that does not go through the latch node of the enclosing loop, 9 ↻ 5. Therefore, we can view both of these branches as leading to an unstructured exit of the loop and the other branches as staying within the loop. Consider the code shown in Figure 4.36 that we can generate using the code generation algorithms presented in Chapter 5 if we were to restructure both conditionals as ifElse's, resetting their follows to be the true successor of the header.

This is a marked improvement over the previously generated code as we

```
\begin{verbatim}
     do {
        BB9;
        if (cond_BB9) {
           BB7;
           if (cond_BB7) {
              goto L5;
           }
           else
           {
L6:           BB6;
              goto L2;
           }
           goto L2;
        }
        else
        {
           BB8;
           goto L6;
        }
L5:     BB5;
     } while (cond_BB5);
L2:  BB2;
```

Figure 4.35: HLL code generated by an existing code generation algorithm.

have halved the number of **goto**s generated. We now explain how such restructuring is performed.

The restructuring relies upon the use of the nodes' parentheses. As mentioned in Cormen [7], the parenthesis can be used to determine ancestor relationships between the nodes in an underlying DFS tree of a graph. In terms of the definitions we gave in Section 4.3.2, a node n is ancestor on a node m if $m \succ n$. This also implies that there is a path from m to n in the graph.

As mentioned earlier, we attempt to restructure a 2way conditional header if its follow is not in the same loop of nway conditional as its header. The algorithm to do this is given in Figure 4.37. This algorithm looks for 2way headers with a branch that leads to an unstructured entry or exit to or from a loop or nway. We also restructure 2ways nodes that have a successor reached by a back edge.

```
     do {
        BB9;
        if (!cond_BB9) {
           BB8;
L6:        BB6;
           goto L2;
        }
        else
        BB7;
        if (!cond_BB7) {
           goto L6;
        }
        BB5;
     } while (cond_BB5);
L2:  BB2;
```

Figure 4.36: HLL code generated as a result of 2way conditional restructuring.

To detect which branch leads to the 'offending' entry or exit, we make use of the ancestor relationship defined by the nodes' parentheses. For example, when restructuring a 2way conditional, $h \diamond f$, in which one branch makes an unstructured exit from a loop we will find it if both the following conditions are true:

1. h has a different loop header than f.
2. $\exists_1 s \in \text{succ}(h) \bullet$ there is a path from s to the latch node of the loop enclosing h (s may be the latch node).

If both hold, then s is determined to be the new follow for the conditional headed by h. If h is the true successor of h, then the type of the conditional is changed to be ifElse otherwise it is set to ifThen. For example, node 7 in Figure 4.14 is a candidate for restructuring as it is inside the loop $9 \circlearrowright 5$ but its follow, 2, is outside the loop. One of the successors of 7, node 5, happens to be the latch node of the loop enclosing 7 and so it is set to be the new follow for the conditional headed by node 7. Also, the type of this conditional is changed from ifThenElse to ifElse as it is the else clause that leads to the unstructured loop exit.

A similar restructuring is performed for unstructured entries into an nway.

We can not have an unstructured forward exits from an nway as the any conditional within an nway will have the same follow as the nway.

Lastly, we restructure any 2way nodes that have a successor, s, reached by a back edge that was not structured as the latch node for a loop. We set the new follow of the 2way to be the successor not reached by the back edge. If s is the true successor of the 2way, it type is changed to be ifThen otherwise it is set to ifElse.

```
procedure Restructure2ways (G = (N, E, s))

* Pre: loop and conditional structuring has been performed
* Post: 2way conditionals may have had their condType and condFollow changed

  for (all n ∈ N • nodeType(n) = 2way ∧ n is the header of n◇f)
    if (loopHead(n) ≠ loopHead(condFollow(n)))
      /* check for jump out of a loop */
      h_n ⟳ l_n := the loop in which n is nested
      h_f ⟳ l_f := the loop in which condFollow(n) is nested
      if (∃ x ∈ succ(n) • x = l_n ∨ l_n ≻ x
        if (x = succ(n,true))
          condType(n) := ifElse
          condFollow(n) := succ(n,true)
        else
          condType(n) := ifThen
          condFollow(n) := succ(n,false)
      /* check for a jump into a loop */
      else if (∃ x ∈ succ(n) • x = h_f ∨ h_f ≻ x)
        if (x = succ(n,true))
          condType(n) := ifElse
          condFollow(n) := succ(n,true)
        else
          condType(n) := ifThen
          condFollow(n) := succ(n,false)
    /* check for jump into an nway */
    else if (n_f ≠ condFollow(caseHead(n) ∧
            ∃ x ∈ succ(n) • caseHead(n) ≠ caseHead(x))
      if (x = succ(n,true)
        condType(n) := ifThen
        condFollow(n) := succ(n,false)
      else
        condType(n) := ifElse
        condFollow(n) := succ(n,true)
    /* check for a back edge from a 2way header */
    else if (condFollow(n) is undefined)
      if (isBackEdge(n,succ(n,true)))
        condType(n) := ifThen
        condFollow(n) := succ(n,false)
      else
        condType(n) := ifElse
        condFollow(n) := succ(n,true)
```

Figure 4.37: Algorithm to Restructure 2way Conditionals

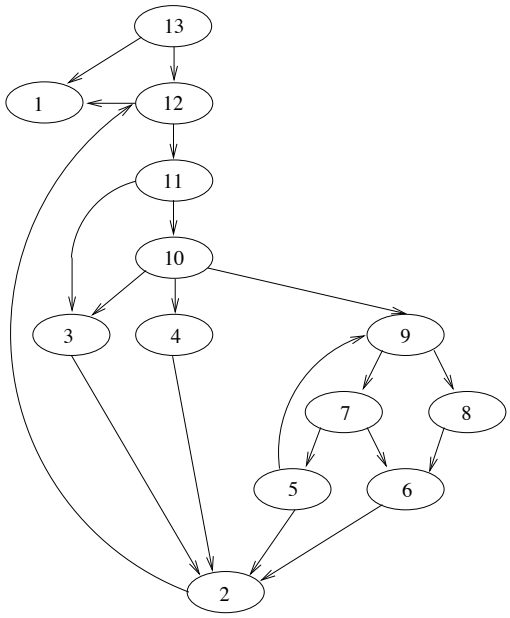
Chapter 5

Code Generation

The information summarised for a control flow graph is used in code generation to generate high level language (HLL) control flow structures. To ensure that we generate code at the appropriate nesting levels (represented by indentation in the output), the task of generating code is tackled as follows: we generate code for the root of a graph, n and then recurse on the successors of n that are within the structure rooted at n. Each level of recursion also represents a successive nesting level and so the code generated within the recursive call is indented one more indent space (which can be arbitrarily set) than the code for the node making the recursive call. Upon termination of the recursion, code is then generated for the follow node of the structure rooted at n. This follow node was determined to be the first node reached from a structure once execution of that structure had completed. A follow was only determined during the structuring phase for *compound* structures such as loops and conditionals. The follow for the other structures such as sequences and calls is simply the successor of these nodes as they are the only place to which they can transfer control. A return node does not transfer control anywhere and hence will generate code for itself and then simply terminate the sequence of recursive code generation call of which it is at the end.

The algorithms used to generate the code from a CFG are independent of the algorithms used to structure the CFG to a large extent. As long as both have the same context with respect to a set of HLL control structures, then it does not matter how these structures were determined. For this reason, we can use the algorithms presented in this chapter to generate HLL code for a control flow graph that was structured using either the derived sequence of graphs

(Section 4.3.1) or the parenthesis theory (Section 4.3.2) method to structure loops. However, the code generated will depend on what structures were determined by the respective algorithms. Figure 5.1 reproduces the CFG example used in Chapter 4 and shows the HLL structure determined for each node in the graph by the derived sequence of graphs (DSG) algorithms.



Node	Struct	Foll
1	seq	-
2	seq	12
3	seq	2
4	seq	2
5	seq	2
6	seq	2
7	ifThenElse	2
8	seq	6
9	postTest	2
10	nway	2
11	ifThenElse	2
12	preTest	1
13	ifElse	1

Figure 5.1: The HLL structure type determined for each node in the reproduced control flow graph of Figure 4.14 with respect to the DSG structuring algorithms. For 2way nodes, the true branch is the left one.

This chapter describes the different algorithms used to generate code for the respective structures and then demonstrates how these are combined to make the one recursive code generation procedure, `CodeGen()`. As we are only performing control flow analysis to the program, only HLL control flow statements will be generated. The rest will remain in assembly-like form.

The following sets are used to assist with code generation:

followSet - all the follow nodes of the control structures in which the current node is nested

gotoSet - the nodes which the current node must not generated code for

The semantics of these sets will be made clearer throughout the following sections.

5.1 Notation and Conventions Used

The pseudocode used for the algorithms in this chapter uses the same format as that used in Chapter 4 with some additions for describing output. We use the procedure `write` for output and adopt C style format strings. For example:

```
write("\%s do \{\CNewline")
```

outputs the following:

```
do {
```

followed by a new line to the screen (or file).

Several other minor functions will be used that we can define here:

- Indent(i) - return a string of i tabstops.
- AsmCond(n) - return the assembly opcode for the conditional branch instruction that delimits the basic block represented by the node n.
- WriteBB(n, i) - write out the sequence of non-transfer instructions in the basic block represented by n, one per line. Each line will be indented by i tabstops.
- EmitGotoAndLabel(n, i) - generates a **goto** jump to the label for n (represented by `Ln`) and places the label at the position where the code has been or will be generated for n. As we have chosen C as our target language, this procedure actually does a little bit of analysis to see if it is possible to generate a **continue** or **break** instead. The former can be generated if n is the loop header of the most nested loop enclosing the node generating the goto and the latter if n if the follow node of the same loop.

When generating code for any node in a structure that involves a conditional expression, we make use of the opcode of the underlying conditional assembly instruction. For example, when generating code for a **while** loop header node representing the following basic block:

```
...
bge .LL3
nop
```

we will generate the expression `while (cond_BBn) ...` where n is the node representing the above basic block. If necessary, we negate the condition by prepending an exclamation mark to the opcode as `while (!cond_BBn) ...` . The opcode to be used will be retrieved using the asmCond() function define above. In the HLL code samples given, we use the above convention of denoting the assembly condition for some basic block n as `cond_BBn`.

As we are only interested in demonstrating the HLL control structures generated, in our examples we represent the output of WriteBB for some node n as `BBn`.

5.2 Generating Code for Loops

The subgraph for a loop will have its code generated when the loop header is traversed during code generation. This header node, h, will have the details of the loops body, type and follow node. The code for any type of loop has the same structure: the loop header, loop body and the loop trailer. The condition controlling the loop iterations can be either in the loop header as for a preTested loop or in the trailer as for a postTested loop. For an endless loop, depending on what high level constructs are available and the type of code we wish to generate, the condition could be in either the loop header or trailer.

Consider the loops $12 \circlearrowright 2$ and $9 \circlearrowright 5$ in the graph of Figure 5.1. For the loop $12 \circlearrowright 2$, the body of the loop is executed if the underlying conditional branch delimiting the basic block represented by node 12 evaluates to false. This is because it is the false edge from node 12 that leads into the loop body.

The algorithm in Figure 5.2 generates the code for a loop. The structure of the algorithm consists of four phases, one for generating code for each of

the following; loop header, body, trailer and follow. We discuss these in the following sections. All explanations are given in terms of the loop $h \circlearrowright l$.

5.2.1 Generating Code for a Loop Header

Generating code for a header, h, of a preTested loop involves generating the code for the basic block represented h followed by a **while** statement with the appropriate loop condition. For a postTested loop, we need to output the header of a **do** loop followed by the code for the basic represented by h. This latter code needs to be generated at the next indentation level as it is within the loop body. An endless loop is a mixture of the two. It has a similar header to a preTested loop except that the condition is a constant — true. Its represented basic block is generated at the next indentation level.

In the given example, this phase will generate the following loop header for node 12:

```
BB12
while (cond_BB12) {
```

5.2.2 Generating Code for a Loop Body

When generating code for the body of the loop $h \circlearrowright l$, we want to ensure two things:

1. The code for l's basic block is generated at the same indentation level as the first node within the loop.
2. Code is not generated for the loop follow node from a node within the loop

To ensure both of these conditions, we update the parameters *latch* and *followSet* that are passed to the `CodeGen` procedure so that the former is now l and the latter includes the follow of the current loop.

The body of a loop is contained in the subgraph headed by one of h's successors or possible by h itself in the case where it is also a conditional header. It

is at a nesting level one greater than the current nesting level and therefore its code is generated at an indentation level one greater than the current indentation level.

In the case of a preTested loop, the loop body is headed by the successor that is not the loop follow node. To generate code for the loop body in this case, we perform a call to the `CodeGen` procedure on the appropriate successor.

For a postTested or endless loop that is also a conditional header, we initially reset the necessary attributes for h so that code can be generated for the conditional it heads (which also generates the code for the body of the loop) and another call is made to `CodeGen` for h. Code will be generated for h's basic block during this call. If the loop is not a conditional header, then we generate code for h's basic block at the next indentation level as the first node of the loop body and then generate code for the rest of the body with a call to `CodeGen` on h's only successor.

For any type of loop, we make sure that the code for l's basic block is generated if it hasn't already been generated.

5.2.3 Generating Code for a Loop Trailer

Generating the loop trailer for an endless loop only involves generating the closing bracket. The same applies for a preTested loop except that we must firstly rewrite the code for h's basic block. The reason for this is that the assembly code that determines the value of the condition used by the conditional branch at the head of the loop comes before this instruction in the basic block. So, to ensure that the condition is set correctly on each iteration, we must repeat the condition setting instructions. This does not disrupt the functionality of the original code and typically only involves a minimal amount of code duplication.

The loop trailer for a postTested loop is similar to the loop header for a preTested loop. We need to generate the conditional expression controlling the loop iterations. In this case however, this **while** expression will come after the closing brace for the loop body.

5.2.4 Generating Code for a Loop Follow

Continuing to generate code for the graph rooted at h involves generating code for the graph rooted at the loop's follow. This applies no matter what type of loop it is. In the case of an unstructured control flow graph, there is the possibility that the loop's follow node may have already been generated. If this is the case, we emit a **goto** label at the position where this code has been generated as well as generating a **goto** at the current position to jump to this label.

5.3 Generating Code for 2way Conditionals

The task of generating code for a 2way conditional, $h \diamond f$ involves similar steps as used when generating code for loops. We generate code for the conditional header which will mean writing an **if..then** or **if..then..else** with the appropriate expression controlling which conditional clause is executed. We use an **if..then** and a negated expression to generate code for an ifThen type of conditional. We use the AsmCond() function to find the appropriate expression.

Following this, code is generated for each of the (non-empty) clauses headed by a successor of h along with the appropriate clause-delimiting structures (i.e. **case** labels or the **else** statement). We take into account the fact that a 2way may have been restructured. In this case, we update the gotoSet, if h was determined to be the header of an unstructured entry or exit to/from a loop, to include the original conditional follow, the latch node (if any) of the loop enclosing f (if any) and the loop header node (if any) of the loop enclosing h. This prevents the code for any of these nodes being generated from within this conditional as they will all be reached by some other more appropriate path.

Finally, the closing bracket and code for the conditional follow is generated if it has not already been generated. If it has, a **goto** is inserted at the correct indentation level and a label is placed at the destination of the

The algorithm for generating code for the graph rooted at a 2way conditional is given in Figure 5.4. **goto**.

```
procedure WriteLoop (n, i, latch, followSet, gotoSet)

* Pre: n is the header of the loop n ↻ l, i is the current indentation level,
* latch is the latching node of the current enclosing loop (if any), followSet is
* the set of follow nodes for enclosing structures and gotoSet is the set of nodes
* for which code must not be generated by this call
* Post: code for the graph rooted at n is generated

   traversed(n) := true
   followSet := followSet + {loopFollow(n)}

            /* Write loop header */
   switch (loopType(n))
      preTested:
         writeBB(n, i)
         if (succ(n,false) = loopFollow(n))
            write("%s while (%s) {\n",Indent(i),asmCond(n))
         else
            write("%s while (!%s) {\n",Indent(i),asmCond(n))
      postTested:
         write ("%s do {\n",Indent(i))
      endless:
         write ("%s while (true) {\n",Indent(i))

            /* Write loop body */
   switch (loopType(n))
      preTested:
         if (succ(n,true) = loopFollow(n))
            CodeGen(succ(n,false),i, l, followSet, gotoSet)
         else
            CodeGen(succ(n,true),i, l, followSet, gotoSet)
      postTested ∨ endless:
         if (structType(n) = loopCond)
            traversed(n) := false
            structType(n) := cond
            CodeGen(n,i + 1, l, followSet, gotoSet)
         else
            WriteBB(n, i + 1)
            CodeGen(succ(n,1),i + 1, l, followSet, gotoSet)
   if (traversed(l) = false)
      WriteBB(l, i + 1)
(cont)
```

Figure 5.2: Algorithm to Generate Code for a Loop Header Rooted Graph

```
(cont)

            /* Write loop trailer */
    switch (loopType(n))
      preTested:
        WriteBB(n, i + 1)
        write ("%s }\n",Indent(i))
      postTested:
        if (succ(l,false) = loopFollow(n))
          write("%s } while (%s) \n",Indent(i),asmCond(l))
        else
          write("%s } while (!%s) \n",Indent(i),asmCond(l))
      endless:
        write ("%s }\n",Indent(i))

            /* Continue with loop follow */
    followSet := followSet - {loopFollow(f)}
    if (traversed(loopFollow(f)) = false)
      CodeGen(loopFollow(n),i, latch, followSet, gotoSet)
    else
      EmitGotoAndLabel(loopFollow(n),i)
end procedure
```

Figure 5.3: Algorithm to Generate Code for a Loop Header Rooted Graph — (cont)

5.4 Generating Code for Nway Conditionals

The procedure for generating code for the subgraph rooted at an nway node is as follows: the code for the nway basic block is generated followed by a **switch** statement representing the indexed jump. The expression used in the switch statement is simply the register upon which the jump was performed. This is retrieved from the basic block represented by the nway node by the function NwayReg(). Once again, data flow analysis is required to provide a HLL expression instead. Code is then generated for the subgraphs rooted at each successor of the nway. The case label for each clause of the nway is simply the position of the edge to that clause within the sequence of out edges from the nway (given by the function Index()). Then the nway trailer is generated as is the code for the graph rooted at the conditional follow of the nway. A **goto** and the appropriate label is emitted whenever a node, whose code has already been generated, is reached during any of the above phase of nway code generation.

```
procedure Write2way(n, i, latch, followSet, gotoSet)

* Pre: n is the header of the 2way conditional, n◇f, i is the current indentation level,
* latch is the latching node of the current enclosing loop (if any), followSet is
* the set of follow nodes for enclosing structures and gotoSet is the set of nodes
* for which code must not be generated by this call
* Post: code for the graph rooted at n is generated
   if (n was not restructured)
      followSet := followSet + {f}
   else
      if (n heads a jump into/out of a loop)
         gotoSet := gotoSet + {iPDom(n), latch,loopHead(f)}
      else if (n heads a jump into an nway)
         followSet := followSet + {f}
      /* write conditional header */
   WriteBB(n, i)
   if (condType(n) = ifElse)
      write ("%s if (!%s) {\n",Indent(i),asmCond(n))
   else
      write ("%s if (%s) {\n",Indent(i),asmCond(n))
      /* write body of 'then' clause */
   if (condType(n) = ifElse)
      c := succ(n,false)
   else
      c := succ(n,true)
   if (traversed(c) ∨ c =loopFollow(loopHead(n)))
      EmitGotoAndLabel(c, i + 1)
   else
      CodeGen(c, i + 1, latch, followSet, gotoSet)
      /* write body of 'else' clause */
   if (nodeType(n) = ifThenElse)
      write ("%s } else \n%s{",Indent(i),Indent(i))
      if (traversed(succ(n,false)))
         EmitGotoAndLabel(succ(n,false),i + 1)
      else
         CodeGen(succ(n,false),i + 1, latch, followSet, gotoSet)
      /* write tailer and continue with follow */
   write("%s }\n",Indent(i)
   remove all nodes added to followSet and gotoSet
      that weren't in them on entry to this procedure
   if (f is defined)
      if (traversed(f))
         EmitGotoAndLabel(f,i)
      else
         CodeGen(f, i, latch, followSet, gotoSet)
end procedure
```

Figure 5.4: Algorithm to Generate Code for for 2way Rooted Graph

The algorithm in Figure 5.5 performs the code generation for a graph rooted at an nway node.

```
procedure WriteNway(n, i, latch, followSet, gotoSet)

* Pre: n is the header of the nway conditional, n◇f, i is the current indentation level,
* latch is the latching node of the current enclosing loop (if any), followSet is
* the set of follow nodes for enclosing structures and gotoSet is the set of nodes
* for which code must not be generated by this call
* Post: code for the graph rooted at n is generated
   WriteBB(n, i)
   write ("%s switch (%s) { \n", Indent(i), NwayReg(n))
   for (all s ∈ succ(n))
      write ("%s case %s: \n",Indent(i + 1),Index(s))
      if (traversed(s) = false)
         CodeGen(s, i + 2, latch, followSet, gotoSet)
      else
         EmitGotoAndLabel(s, i + 2)
      write ("%s break; \n", Indent(i+2))
   remove f from followSet if it was not in it upon entry to this procedure
   if (traversed(f) = false)
      CodeGen(f, i, latch, followSet, gotoSet)
   else
      EmitGotoAndLabel(f, i)
end procedure
```

Figure 5.5: Algorithm to Generate Code for a Nway Rooted Graph

5.5 Generating Code for Non-Compound Structures

The algorithm for generating the code of n, a one edge, fall through, return or call node, is as follows: the code for the basic block represented by n is generated by WriteBB. If n is a return node, then a return statement is generated after the call to WriteBB.

We generating code for the follow of n (i.e. its sole successor in the graph), we the use the loop header and nway header information to generate a **goto** if the successor is in a different structure or has already been generated. The only time we don't generate a **goto** under these conditions is when the successor is only reached from n. Neglecting this case would otherwise mean that the successor would never be generated.

The algorithm in Figure 5.6 performs the code generation for a graph rooted at an non-compound node.

```
procedure Write1way(n, i, latch, followSet, gotoSet)

* Pre: n is a non-compound (i.e. 1way) node, i is the current indentation
* level, latch is the latching node of the current enclosing loop (if any), followSet
* is the set of follow nodes for enclosing structures and gotoSet is the set of nodes
* for which code must not be generated by this call
* Post: code has been generated for the graph rooted at n

   WriteBB(n, i)
   if (nodeType(n) = ret)
      write("%s return; \n",Indent(i))
      return
      s := succ(n, 1)
   if (traversed(s) ∨ (loopHead(s) ≠ loopHead(n) ∧
         (∃ p ∈ pred(s) • traversed(p) = false ∨ s ∈ followSet)) ∨
         s = loopFollow(loopHead(latch)) ∨
         (caseHead(s) ≠ caseHead(n) ∧ s ≠ condFollow(caseHead(n)))
      EmitGotoAndLabel(n, i)
   else
      CodeGen(n, i, latch, followSet, gotoSet)
end procedure
```

Figure 5.6: Algorithm to Generate Code for a Non-compound Rooted Graph

5.6 The Complete Algorithm

We can now define the complete algorithm for generating the code for a control flow graph that has been annotated with structuring information. The procedure takes as arguments a pointer to a node/basic block n, the indentation level to be used, the latching node of any enclosing loop (will be the null pointer if there no enclosing loop), the set of follow nodes for any enclosing conditionals or loops and a set of nodes for which code must not be generated by the current call to this procedure. Initially, n is set to be the root of a control flow graph, the indentation level is 1, the latching node pointer is null and the two sets are empty. If n is within *gotoSet* we will generate a **goto** to it if items 1 and (2 or 3) from the following are met:

1. n is not a latch node. When a latch node is put into the gotoSet, it code will be generated by its corresponding loop header instead to ensure

that it is generated at the same indentation level at the loop head.

2. it is the loop follow of the currently enclosing loop. Being in the gotoSet means that code for this follow will be generated else where.

3. there exists some predecessor of n that has not yet been traversed during code generation. This means a **goto** to n will not be generated if this is the last time it will be traversed.

If a **goto** has not been generated for n yet, then if it is in the followSet but not the most recently added node in this set, a **goto** will be generated otherwise the CodeGen procedure returns as code for the n will be generated by the structure that added the latest node to the followSet. The procedure also returns at this point if a call to CodeGen on n has been made previously and reached the point where its traversal flag was set.

Finally, a call is made to the appropriate code writing algorithm for n based on its type determined during structuring. The complete algorithm is given in Figure 5.7.

```
procedure CodeGen(n, i, latch, followSet, gotoSet)

* Pre: n is a node in a CFG that has been structured, i is the current indentation
* level, latch is the latching node of the current enclosing loop (if any), followSet
* is the set of follow nodes for enclosing structures and gotoSet is the set of nodes
* for which code must not be generated by this call
* Post: code for the graph rooted at n is generated

   recentFoll := the most recently added member of followSet
   if (n ∈ gotoSet ∧ isLatchNode(n) =false∧
         (n = loopFollow(loopHead(latch)) ∨ ∃ p ∈ pred(n) • traversed(p) = false))
      EmitGotoAndLabel(n, i)
      return
   else if (n ∈ followSet)
      if (n ≠ recentFoll)
         EmitGotoAndLabel(n, i)
      return
   if (traversed(n) = true)
      return
   traversed(n) = true
   if (isLatchNode(n))
      if (i = the indentation level at which loopHead(n) was generated)
         WriteBB(n, i)
         return
      else
      traversed(n) := false
         EmitGotoAndLabel(n, i)
         return
   switch (structType(n))
      case loop:
      case loopCond:
         WriteLoop(n, i, latch, followSet, gotoSet)
      case cond:
         if (condType(n) = nway)
            WriteNway(n, i, latch, followSet, gotoSet)
         else
            Write2way(n, i, latch, followSet, gotoSet)
      case seq:
         WriteSeq(n, i, latch, followSet, gotoSet)
end procedure
```

Figure 5.7: Algorithm to Generate Code for a Control Flow Graph.

Chapter 6

Implementation and Results

This chapter describes **ast**, a prototype implementation of the assembly structuring process presented in Chapters 3, 4 and 5. Two variations were implemented; one for the algorithms that make use of the parenthesis theory described in section 4.3.2 for loop structuring and one for the algorithms that make use of the derived sequence of graphs described in Section 4.3.1 for loop structuring. Both implementations were run on a set of benchmarks programs for the purpose of making comparisons between the two in terms of both functionality (the quality of of the high level code generated) and resource efficiency (memory usage and execution speed).

Section 6.1 describes the specific aspects of the given implementations such as limitations imposed upon the type of input it can handle. Following this, Section 6.2 presents the tests that were performed on the implementations and the conclusions drawn as a result of these tests.

6.1 ast

6.1.1 Implementation Parameters

Before considering the coding of the **ast**, a number of other decisions had to be made. Namely, the target architecture had to be chosen as well as the language used for the implementation. The first decision came down to a choice between the Intel and SPARC architectures as machines implementing

both were readily available. The SPARC was chosen for two reasons. Firstly, as part of related research performed by the author, a subset of the **ast**'s functionality had already been implemented for the SPARC architecture. Secondly, being a RISC machine, the instruction set of the SPARC is smaller than that of a CISC machine such as the Intel family. This meant that less time would have to be spent of the architecture dependent parts of the implementation. Any means of minimising the time spent on these parts was desirable as the emphasis was on the machine independent structuring algorithms.

Given the potential for **ast** to be integrated with other reverse engineering tools, a criteria for the choice of implementation language was that it supported code reuse to some extent. Object oriented languages are proclaimed to enable better code reuse and so C++ was chosen, being one of the more popular object oriented languages currently available within the development environment.

As explained previously, **ast** works with SPARC8 assembly programs. However, it does not accept all the instructions defined for this architecture. The reason for this is twofold. Firstly, we restricted the domain of handled programs to be only applications programs. That is, only programs using unprivileged instructions[1] are accepted. Secondly, the scope has been narrowed further to realise development time resources available for the tool. To this end, we neglect all trap instructions. This is of little consequence in reality as the majority of programs never include these instructions.

6.1.2 Delayed Instructions

The concept of a delayed instruction is a feature of SPARC's pipelined architecture which means that the next instruction to be executed is fetched during the execution of the current instruction. For just about all unconditional control transfer instructions (CTIs), the instruction in their delay slot (i.e. at the address one greater than the program counter) is always executed before the transfer of control takes place. For the conditional CTIs however, the decision of whether or not to execute the delayed instructions depends upon the value of an *annul* flag within the instruction. Pages 50-51 of [9] explain the semantics of the delayed instruction further. In our implementation, we have not strictly adhered to these semantics. We make the

[1]Privileged instructions are those that can only be executed by the super user.

assumption that the instruction in the delay slot of a CTI, will always be executed before the transfer of control takes place. This is done once again in an attempt to place some constraint of the time required to develop **ast**. It also makes reasoning about the output HLL in terms of the input assembly significantly easier. Importantly, it does not influence the development of the structuring algorithms presented in Chapter 4.

The relaxing of our adherence to the semantics of the delayed instruction was covered earlier in Section 3.2.1. We briefly detail here the modification that would have to be made to ensure that the proper semantics are not violated when generating the control flow graph. The control flow graphs generated from the given definition of a basic block does not correctly capture the possible paths of control flow in all situations. An example is shown in Figure 6.1. The `,a` appended to instruction 2 (`bne,a .LL1`) indicates that the annul flag has been set for the succeeding delayed instruction. This means that if the conditional branch is taken (the right branch in the graph) then the sequence of instruction execution will be 1,2,3,5. However, if the condition code set in instruction 1 evaluates to false the conditional branch will not be taken and the flow of control will follow the left branch in the graph. In this case the execution sequence will be 1,2,4. A tempting solution to this problem is to move the delayed instruction into the block C. However, this may conflict with the original flow of control if there is another edge into C. The execution path including this edge will not originally have included instruction 3. The solution is to put instruction 3 in its own block which would lie in between block A and C as demonstrated in Figure 6.2.

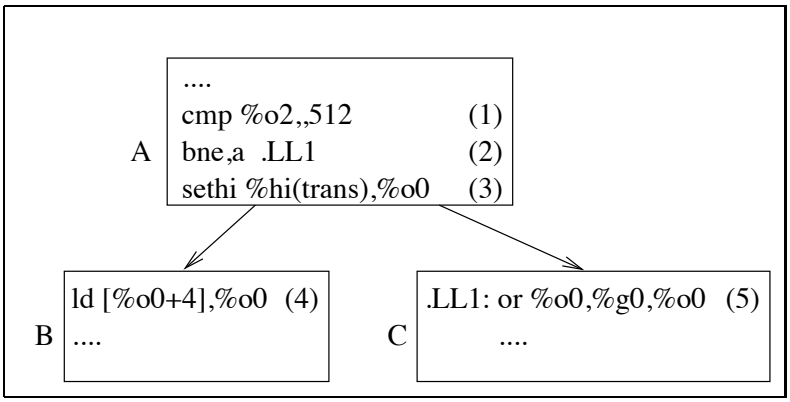


Figure 6.1: CFG demonstrating the semantics of a delayed instruction in the context of the annul flag.

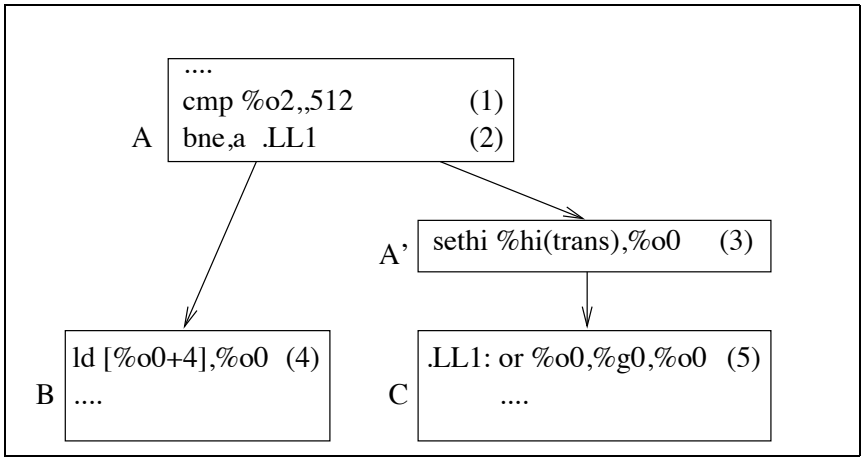


Figure 6.2: Solution to the delayed instruction dilemma.

6.1.3 Preprocessor

Knowledge of the patterns generated by the compiler used to generate the test cases meant that certain preprocessing could be performed on the input that would otherwise involve data flow analysis. As **ast** does not attempt any data flow analysis, this preprocessing allows for a wider and more interesting range of inputs to be tested against. In particular, any program containing an indexed jump can be included within the valid input domain of **ast** as the preprocessing can use the compiler patterns to determine the actual targets of these CTIs. This type of instruction is most often the implementation of a **case** or **switch** statement. An example of the preprocessor functionality is shown in Figure 6.3. Both the input (shown in the left column) and the output (shown in the right column) are have their instructions numbered for easier reference. Lines 14 - 21 of the input give an example of the pattern generated by the Gnu compiler for an indexed jump. The value of the register `%o0` will be the address of one of the lines 18 - 21. As can be seen in the output, the preprocessor have moved the label from each of these lines into the position previously occupied by the `%o0` register.

Secondly, the preprocessor has also removed all lines that are not one of the following:

instruction - an assembly instruction. The regular expression used to match this type of line is `^\t[a-z]`. The line `<tabstop>save %sp,-1136,%sp`

will be matched by this pattern for example.

branch label - these are the labels that will also be the operand of a branch instruction. The pattern used to match these lines is `^\.LL[0-9]`. The line `.LL3:` will be matched by this pattern.

routine label - these are the labels at the entrance to a routine. They will be matched by the two regular expressions `^[a-zA-Z_]` and `:$`. The line `bf:` will be detected as a procedure label.

```
1      .file "bf.c"
2   gcc2_compiled.:
3   .section ".text"
4      .align 4
5      .global bf
6      .type  bf,#function
7      .proc 04
8   bf:
9      !#PROLOGUE# 0
10     save %sp,-1136,%sp
11     !#PROLOGUE# 1
12     sethi %hi(trans+4),%o0
13     ...
14     jmp %o0
15     nop
16     .align 4
17  .LL32:
18     .word .LL30
19     .word .LL4
20     .word .LL5
21     .word .LL6
22  .LL4:
23     ...
```

```
1   bf:
2      save %sp,-1136,%sp
3      sethi %hi(trans+4),%o0
4      ...
5      jmp .LL30, .LL4, .LL5, .LL6
6      nop
7   .LL4:
8       ...
```

Figure 6.3: Assembly program before (left) and after (right) preprocessing. Each instruction has been given a numeric label for easier reference.

The annul flag discussed in Section 6.1.2 also presents one more problem that can be fixed by the preprocessor. The unconditional branch instructions (e.g. `ba`) also have the option of setting the annul bit. If this bit is set, then the delayed instruction is never executed after the branch is taken. For this reason, it (the delayed instruction) is invariably the target of another CTI (otherwise it would be an unreachable instruction). In order to allow these unconditional branch instructions to be treated in the same manner

as the conditional branch instructions, a `nop` is inserted after them by the preprocessor. In context of the proposed treatment of delayed instructions, inserting the `nop` does not affect the semantics of the original program.

6.1.4 Data Representation

An important issue in the implementation of **ast** is the efficiency of its resource usage. In this respect, a memory efficient representation of both the instructions and the CFGs is to be achieved. The use of reference mechanisms (e.g. pointers) available in languages such as C++ enables the desired memory efficiency to be attained. The instructions are stored as an array. The number of basic blocks that partition these instructions is unknown which suggests the use of a linked list when doing the partitioning. The graph edges are represented as pointers between the nodes. This is allows for optimal memory use efficiency as each block is only represented once in memory. The same applies for the sequence of instructions within each block. Pointers are used to index the subsection of the instruction array containing the instructions for the relevant block.

6.2 Testing and Results

This section describes the tests performed as well as providing information on the test programs used as input to generate the results.

6.2.1 Test Inputs

For the purpose of testing both variations of **ast**, the assembly input files used were generated automatically through use of the Gnu C compiler (version 2.7.2.1) with the -S and -O2 options which generates optimised assembly code. The C source files were taken from the SPEC group of benchmark programs[2]. These benchmarks are widely used in the compiler community. Testing on this set of input programs ensures that **ast** was tested on non-trivial inputs.

[2]See the url `http://www.specbench.org/` for further details on the SPEC group.

The programs chosen for testing were from the CINT95 suite and are shown in Figure 6.4 along with the relevant details of each. The figures for each program are totals of all the assembly files for that program.

Program	Assembly Ins.	Graph Nodes
gcc	411477	77668
go	108995	12934
ijpeg	59203	8570
li	16697	3703
m88ksim	43075	7032
perl	85473	16695
vortex	160761	25839

Figure 6.4: The programs used for testing along with the instruction count and graph node count.

6.2.2 Comparisons for Quality of Code Generated

This section presents the results of tests that give some kind of indication as to the type of code generated by each variation of **ast**. For simplicity, which we refer to the version of **ast** implementing the derived sequence of graphs algorithm as $\mathbf{ast_{DSG}}$ and the other version as $\mathbf{ast_{PT}}$. The metrics used to judge the quality of the code produced by $\mathbf{ast_{DSG}}$ and $\mathbf{ast_{PT}}$ are based on the number of HLL control structures and **goto**s generated. The other method that could be used is to just read the code produced. However, this is hampered by both the size of the code produced. However, we present the code generated for the control flow graph of Figure 5.1 by each algorithm as it is relatively small. It also gives some idea of how unstructuredness can produce a variation in the quality of code produced, depending on what algorithms are used to perform the structuring.

Figure 6.5 shows the code generated when using the parenthesis theory algorithms for loop structuring. The limitations of the parenthesis theory are well demonstrated here. Recall that the back edge (5,9) did not induce a loop as the two nodes, 5 and 9, were determined to be in two different loops. This is a direct result of relying upon a the parenthesis pair of each node which describes a relationship between the two in terms of the underlying DFS tree. As this example shows, extending the usage of the parenthesis to

imply a similar relationship but in terms of a graph does not always give us the result we desire. By not determining node 9 to be a loop header, the resulting code in this case involves twice as many **goto**s as the generated code shown in Figure 6.6. This latter code was given by the derived sequence of graphs based algorithm.

The DSG based algorithm does have one disadvantage however in comparison to the PT algorithm. This arises in the case where there is a forward jump into a loop. The forward edge representing this jump will cause the graph to be irreducible. What this means is that the two nodes participating in the back edge that would otherwise induce a loop, will never be in the same interval. Recall from Section 4.5.5 that this is one of the conditions necessary for the edge to induce a loop. The PT algorithm is not faced with this limitation.

Total number of HLL structures generated

Having seen that the DSG algorithm is more likely to generate less **goto**s than the PT algorithm, we present some test results that support this. The bar graphs in Figures 6.7, 6.8 and 6.9 show the number of 2way conditionals, nway conditionals and loops respectively generated by each implementation. The shaded bars indicate the results generated by $\mathbf{ast_{DSG}}$ and the clear bars represent the results for $\mathbf{ast_{PT}}$. These graphs show that while the DSG algorithm gives better results (i.e. more HLL structures) in each case, the difference is relatively small. As would be expected then, the $\mathbf{ast_{PT}}$ implementation will generate more **goto**s than $\mathbf{ast_{DSG}}$ and this is indeed the case as shown in Figure 6.10 which shows the number of **goto**s generated as a percent of all control transfer instructions in the generated code. That is, as a percent of the total number of **goto**, **while**, **if..then..else**, **if..then**, **switch**, **break** and **continue** statements generated. This last figure also shows that the proportion of **goto**s generated is still quite large (as high as 32% in one case). This goes to show that if the input assembly is reasonably unstructured (as the optimised assembly was in our tests), the resulting code will still retain a significant amount of unstructuredness. However, we have managed to transform up to 85% of the control transfer instructions into high level control structures. This is a significant improvement for anyone having to maintain assembly programs.

```
main()
{
   BB12;
   if (!bne) {
      BB11;
      while (bne)
      {
         BB10;
         if (bne) {
L2:
            BB2;
            goto L1;
         }
         BB9;
         switch (Reg0) {
         case cond_0:
            goto L2;
            break;
         case cond_1:
            BB3;
            goto L1;
            break;
         case cond_2:
L8:
            BB8;
            if (bne) {
               BB6;
               if (bne) {
                  BB4;
                  if (bne) {
                     goto L8;
                  }
               } else
               {
L5:
                  BB5;
               }
            } else
            {
               BB7;
               goto L5;
            }
            break;
         }
L1:
         BB1;
         BB11;
      }
   }
   BB0;
   return;
}
```

Figure 6.5: The C style high level code produced by $\mathbf{ast_{PT}}$

```
main()
{
   BB12;
   if (!bne) {
      BB11;
      while (bne)
      {
         BB10;
         if (bne) {
L2:
            BB2;
            goto L1;
         }
         BB9;
         switch (Reg0) {
         case cond_0:
            goto L2;
            break;
         case cond_1:
            BB3;
            break;
         case cond_2:
            do {
               BB8;
               if (!bne) {
                  BB7;
L5:
                  BB5;
                  break;
               }
               BB6;
               if (!bne) {
                  goto L5;
               }
               BB4;
            } while (bne);
            break;
         }
L1:
         BB1;
         BB11;
      }
   }
   BB0;
   return;
}
```

Figure 6.6: The C style high level code produced by $\mathbf{ast_{DSG}}$

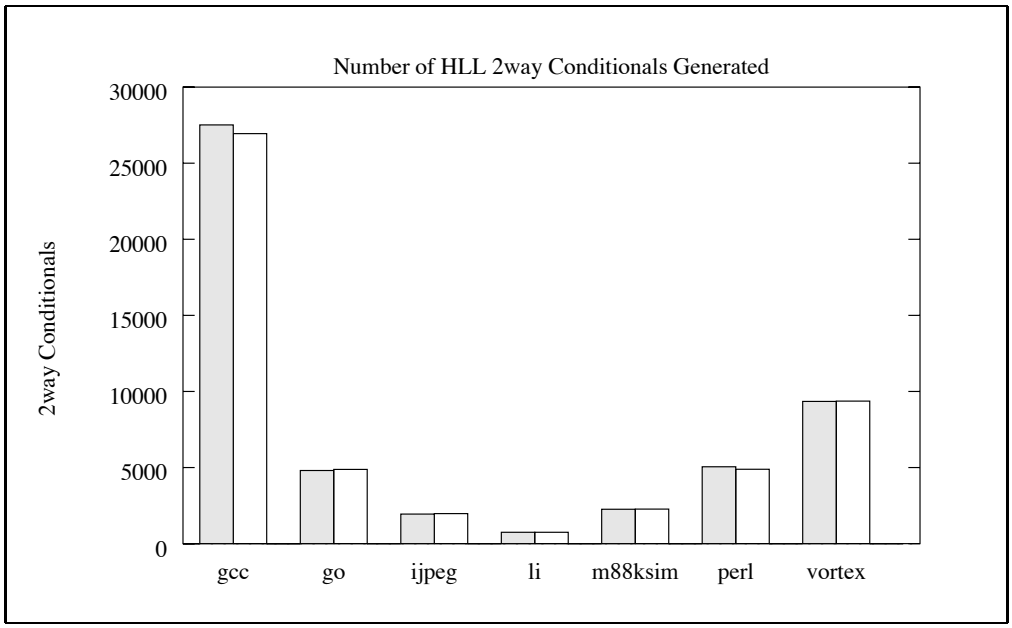


Figure 6.7: Number of HLL 2way conditionals generated.

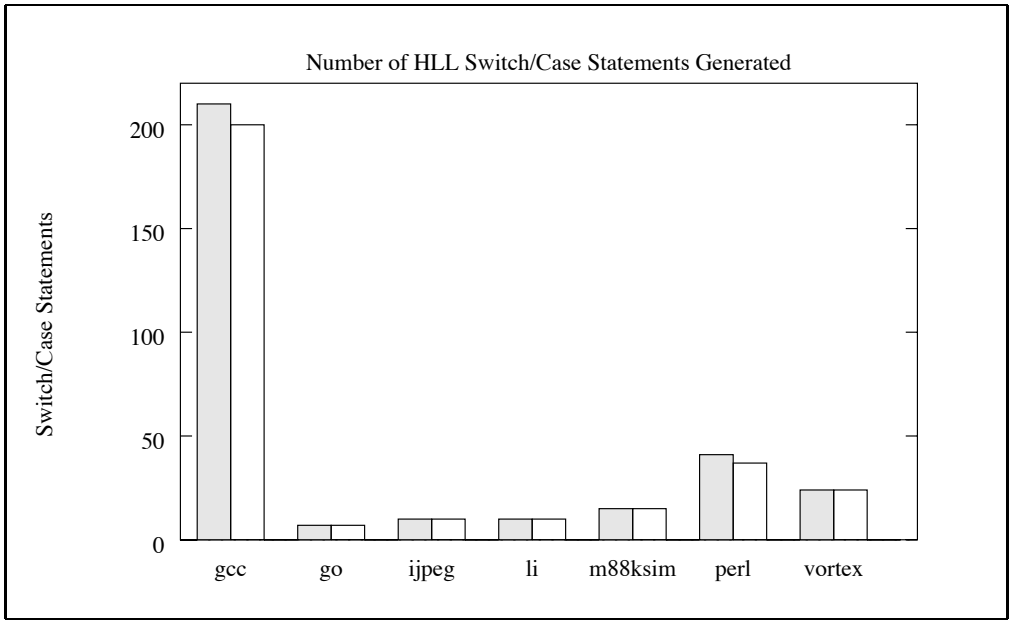


Figure 6.8: Number of HLL nway conditionals generated.

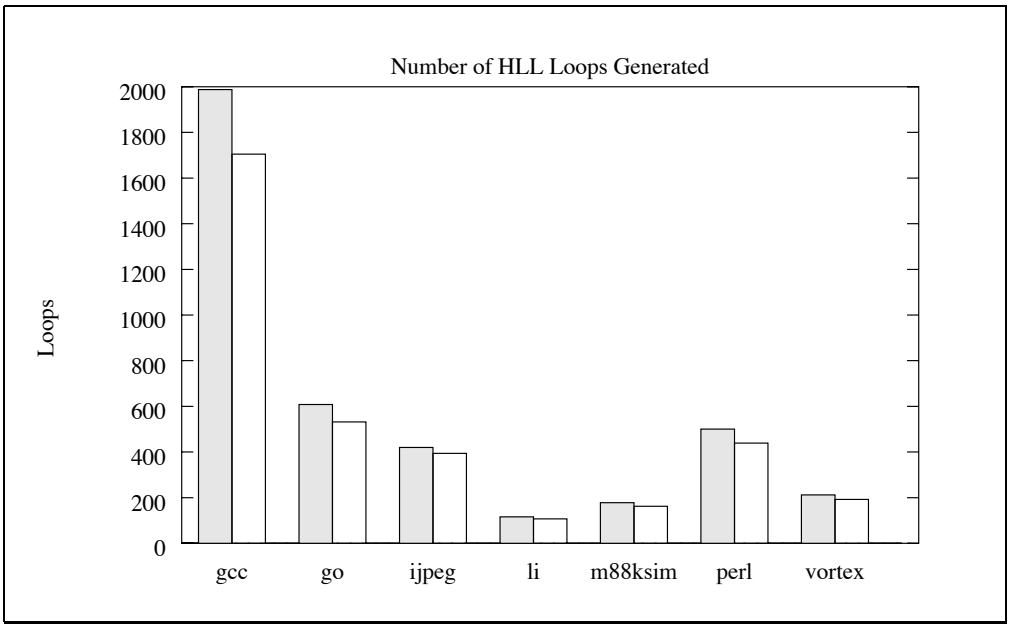


Figure 6.9: Number of HLL loops generated.

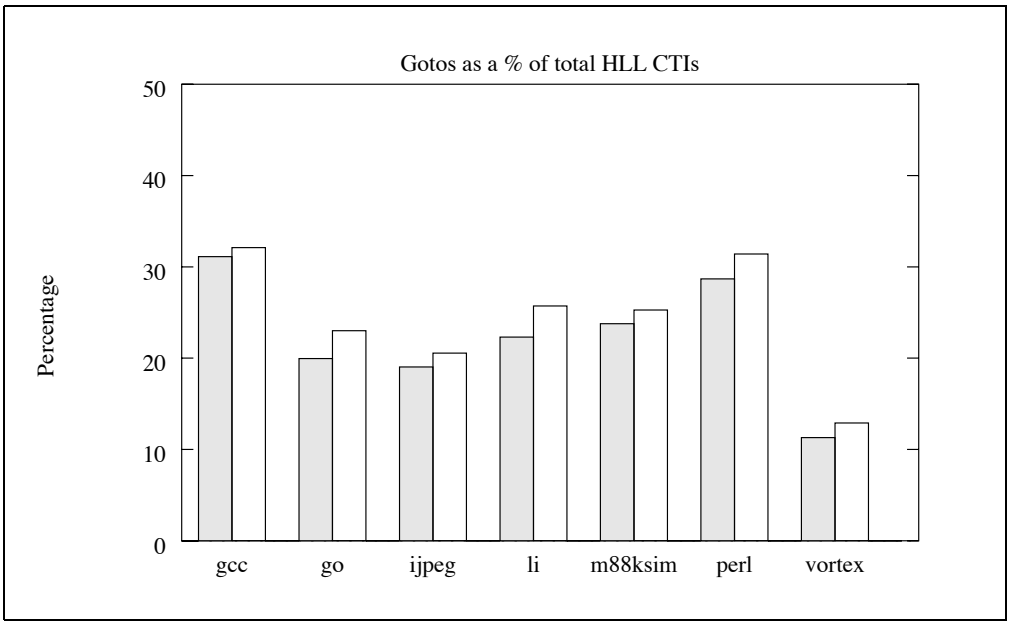


Figure 6.10: Number of **goto**s as a percentage of the total HLL control structures generated.

6.2.3 Resource Usage

The second groups of tests was performed only on the $\mathbf{ast_{DSG}}$ implementation to compare its overheads in terms of memory usage and execution efficiency with that of $\mathbf{ast_{PT}}$. The reason the data was only collected for $\mathbf{ast_{DSG}}$ is that apart from the construction and usage of the derived sequence of graphs, both implementations were identical. What we aim to determine with these tests is whether or not the parenthesis theory algorithm gives a significant enough improvement in runtime resource usage to offset its disadvantages with respect to the quality of code it generates.

The graph in Figures 6.11 shows the memory requirements of the derived sequence of graphs for each program. As can be seen, the maximum required by any program was about 2MB. This is to be expected as gcc was the largest test program used (411477 instructions). For any modern computer, this kind of memory usage does not impose a significant drain on the memory available, especially given that the program will not be used as frequently as other equally memory hungry programs. And, as Figure 6.12 shows, larger inputs (such as gcc) require relatively less memory for DSGs than smaller programs (such as ijpeg). Therefore, we can safely conclude that, in terms of memory requirements, the DSG algorithm does not perform badly enough to justify the use of $\mathbf{ast_{PT}}$ instead.

The same is true for the speed overhead incurred by $\mathbf{ast_{DSG}}$. Figure 6.13 shows, the time taken to build the DSGs as a percent of the total time taken to build the DSGs, perform the structuring and generate the code. It does not take into account the time required to do the parsing or control flow graph generation. Even without taking into account the extra time required for these phases, the DSG building only takes up on average 25 percent of the runtime. The mostly costly phase (which is common to both $\mathbf{ast_{DSG}}$ and $\mathbf{ast_{PT}}$) is the code generation and parsing phases as they require intensive disk reading and writing.

With regard to the DSG performance, the conclusion is that its extra resource requirements are not as significant a disadvantage the it advantage in generating better HLL code and should therefore remain the algorithm of choice when structuring assembly code.

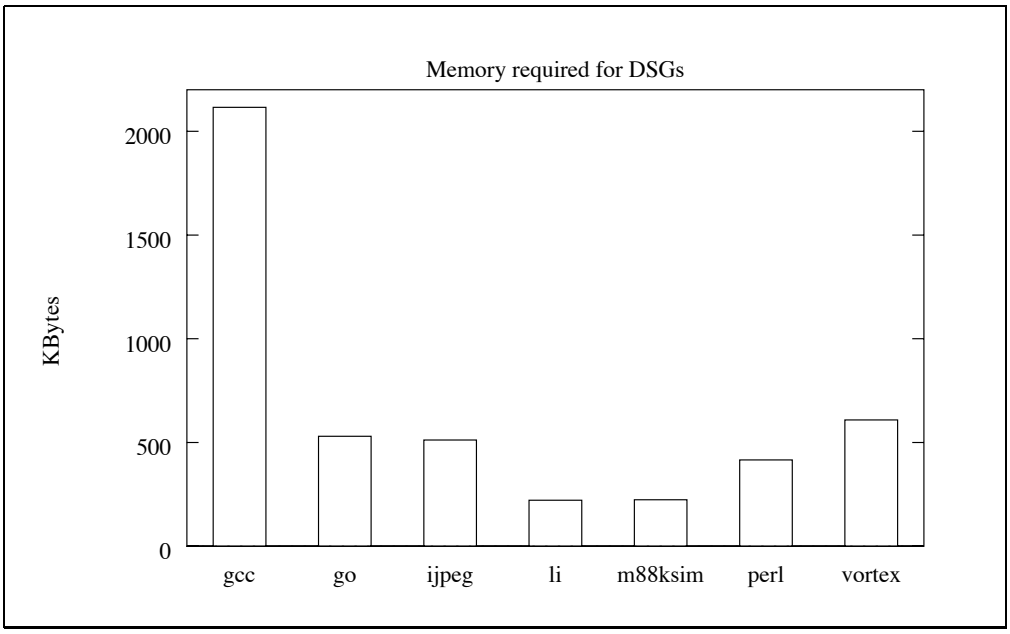


Figure 6.11: Memory cost of DSGs

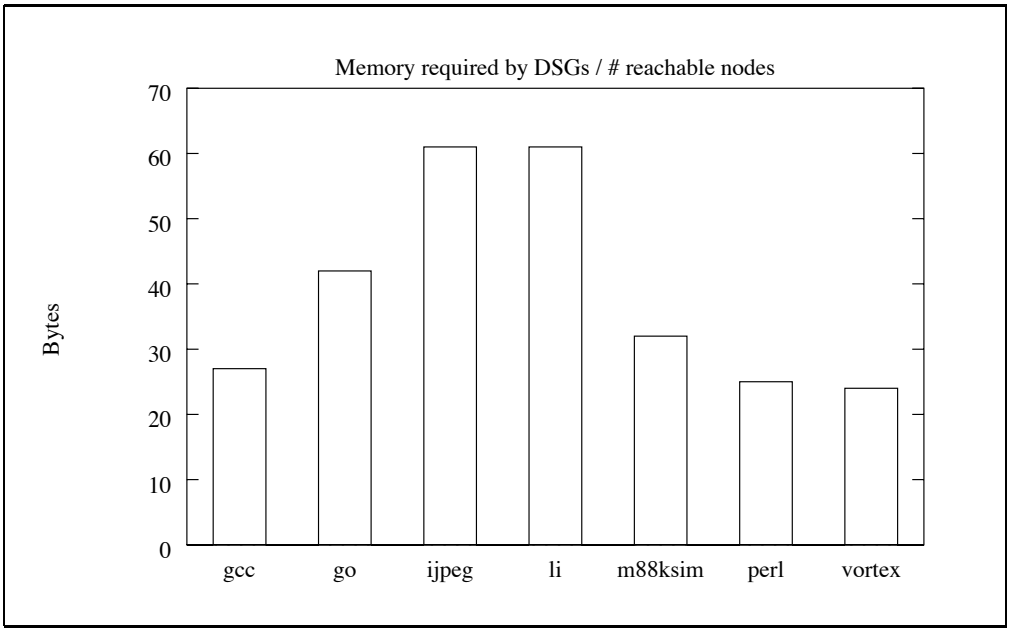


Figure 6.12: Memory cost of DSGs / number of reachable nodes.

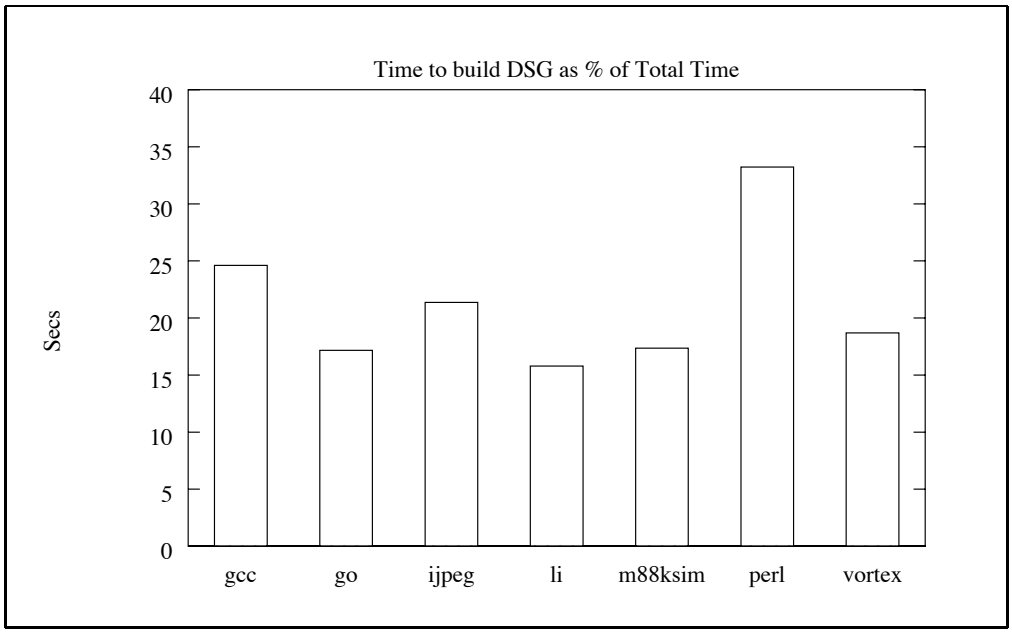


Figure 6.13: Time taken to build the DSGs as a percent of the total time to perform the structuring.

Chapter 7

Conclusion

This thesis has presented some the issues involved when translating assembly programs to a program in a higher level language. In particular we concentrated on deriving high level control flow constructs from the control flow graph representation of an assembly program. When presenting the algorithm that perform the analysis and transformations involved, we investigated two variations on the algorithms particular to loop structuring. The first was based upon work by Cifuentes in [5]. A comparison between the performance characteristics of each variation was a major motivation for this thesis. We also made a number of enhancements to some of the other algorithms presented by Cifuentes.

An assembly to high level language structuring process involves a number of phases; parsing, control flow graph generation, structuring and code generation. Parsing reads in the assembly source and performs a simple type of semantic analysis to categorise the instructions according to the control flow characteristics. The control flow graph generation phase does exactly that — constructs the control flow graph representing the control flow between the basic blocks of the program. The structuring phase annotates the nodes with information summarising the role they play in a high level control structure (if any). This information is then used in the code generation phase to produce a semi high level language program. We cannot produce a complete high level program as we do not perform any data flow analysis.

When dealing with unstructured control flow graphs, we made use of a canonical set of unstructured forms derived from the work by Williams and Oulsnam (see Section 2.1) as a guideline for the types of unstructured code we

would have to handle. However, this was only a basis from which to start. The myriad of combinations of these forms that presented themselves when analysing real assembly programs meant that a considerable amount of extra effort was required to handle all unstructured cases. In addition, the presence of nway nodes was not accounted for Williams and Oulsnam. These type of nodes add extra complexity to the analysis as we have to prioritise which high level structures will be generated when their representations within the control flow graph overlap each other. The main heuristic guiding a decision in these cases was to minimise the potential number of gotos that would be generated in HLL code.

Two other extensions and/or modifications made to existing structuring algorithms. Firstly, the method used to determine the follow node for a conditional was changed to rely on the post-dominator properties of the nodes as opposed to the forward dominator. The result of this change was that detecting whether or not an nway subgraph had an unstructured entry or exit was made considerably easier and more reliable. Secondly additional analysis was applied to 2way conditional headers so that in case where one of the branches included an unstructured entry or exit to another structure, the 2way was restructured in such a way that fewer **goto**s were generated in the final HLL code.

The final part of this thesis presented the results of comparison tests between the implementations of the two variations on the structuring process. The first variation was when the process involved using a derived sequence of graphs (see Section 4.3.2) to do loop structuring where as the second was when parenthesis theory (see Section 4.3.1) was used for loop structuring instead. The results showed that while the latter was slightly more efficient in terms of memory usage and execution speed, it was enough of an improvement to offset the higher quality of code generated by the former.

This thesis provides the impetus for two types of future work in this area. Firstly, additional development of **ast** can be undertaken to remove some of the limiting assumptions under which it was developed that restrict its functionality, or even, as in the case of how it handles delayed instructions, compromises its strict correctness.

Secondly, and of more import, would be further research into two of the algorithms described within this thesis. The first is the algorithm used to mark the nodes within an nway subgraph. Currently this uses a depth first search which incurs the overhead of recursion. Secondly, the algorithm for deter-

mining the immediate post-dominator of each node makes three passes over all the nodes in the graph. There is evidence to suggest that this can be done more efficiently by considering that the closely related concept of immediate forward-dominators has a faster algorithm to determine this property for a node.

Appendix A

Attribute definitions

A.1 Base Node Attributes

The following group of attributes are defined for the nodes in the control flow graphs. Additional attributes that are relevant under only one of the two variations of the structuring algorithm are defined in later sections.

nodeType - the type of the node. One of **ucond, jcond, nway, call, ret, fall**. Occasionally, **2way** is used instead of cond.

order - the post-ordering order of the node. This ordering is used when considering the nodes of a graph in ascending or descending order.

revOrder - the post-ordering of the node in the reverse graph.

pred - the immediate predecessors of the node. This can be used as either a set or a sequence. If used as a sequence, a second parameter can be given to indicate which predecessor to return. For example, pred(n) denotes the set of predecessors of n where as pred($n, 1$) denotes the first predecessor of n.

succ - the immediate successors of the node. The semantics of this attribute are similar to **pred** with one addition. For a 2way node, we can indicate which successor to return by indexing with a boolean value. This enables us to access the true false successors which correspond with the first and second successors respectively.

pred$_{rev}$ - the immediate predecessors of the node in the reverse graph. The membership of this set is identical to the **succ** set for the node.

iPDom - the immediate post-dominator of the node.

caseHead - the case header node for the most nested nway in which this node is a member. May be undefined.

loopHead - the loop header node for the most nested loop in which this node is a member. May be undefined.

condFollow - the follow node of the conditional headed by this node. This may be an nway or 2way conditional and the attribute is only defined for nodes that have been structured as conditional headers.

loopFollow - the follow node of the loop headed by this node. Only defined for loop headers.

structType - the high level control structure this node is determined to be the header of. Will be one of:

- **seq** - a sequential statement (default for all nodes before structuring)
- **loop** - node is a loop header
- **cond** - node is a conditional header
- **loopCond** - node is both a loop and conditional header

condType - the type of conditional this node heads. One of:

- **ifThen** - header of a 2way conditional where only the true branch is non-empty.
- **ifElse** - header of a 2way conditional where only the false branch is non-empty.
- **ifThenElse** - header of a 2way conditional where both clauses are non-empty.
- **case** - header of an nway conditional header.

loopType - the type of loop this node heads. One of:

- **preTested** - header of a loop where the loop condition is tested before the first iteration (e.g. **while** loops).

- **postTested** - header of a loop where the loop condition is tested after the first iteration.
- **endless** - header of a loop with no loop condition.

traversed - a boolean flag used indicate whether or not a node has been visited during a given graph traversal. The value of this flag is implicity set back to false after a complete traversal has been completed.

A.2 Predicates

This section describes the predicates used. While they are implemented as attributes of a node, their functionality differs in that they only return a boolean value which depends on the other attributes and cannot be explicitly set.

isBackEdge - given a pair of nodes, returns true if the pair represents a back edge in a graph.

isLatchNode - return true if the node is a latch node of a loop

A.3 Attributes for Other Objects

Certain other concepts used in the algorithms are more easily reasoned about when treated as objects with their own attributes. We describe these attributes here.

loopNodes - this attribute is defined for a loop $(h \circlearrowright l)$ and represents all the nodes that either have h marked as their loop header or some other loop headed, h_2, such that h_2 is nested within $(h \circlearrowright l)$. The recursive definition for a test of nesting is:

A node n is nested within $(h \circlearrowright l)$ if
(loopHead(n) $= h$ or loopHead(n) is nested within $(h \circlearrowright l)$)

A.4 Derived Sequence of Graphs Attributes

The following attribute is only defined for the nodes used in the derived graphs under the DSG algorithm. That is, it is defined for each node in G^i where $n \geqslant 2$. These nodes also inherit all of the above attributes.

nodesOf - the set of nodes from G^{i-1} in the interval this node represents.

baseNodesOf - the set of nodes from G^0 (i.e. the original CFG) in the interval this node represents.

Under the DSG algorithm where the derived sequence is $G^0, .., G^n$, the following attribute is defined for all nodes in graphs $G^0, .., G^{n-1}$ (which may be no graph if $n = 0$).

inInterval - the node in G^{i+1} which represents the interval in which this node is a member.

A.5 Parenthesis Theory Attributes

The following attributes are defined for control flow graph nodes when using the PT algorithm. As both these attributes are a tuple type, their individual elements can be accessed using the standard indexing mechanism.

parenthesis$_{true}$ - the parenthesis for this node defined during a DFS traversal of the graph where the true edges were traversed first.

parenthesis$_{false}$ - the parenthesis for this node defined during a DFS traversal of the graph where the false edges were traversed first.

Bibliography

[1] Ole Agesen. Design and Implementation of Pep, a Java^{TM} Just-In-Time Translator. *Theory and Practice of Object Systems*, 3(2):127–155, 1997.

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

[3] B.S.Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[4] C. Cifuentes. Structuring decompiled graphs. In T. Gyimóthy, editor, *Proceedings of the International Conference on Compiler Construction*, Lecture Notes in Computer Science 1060, pages 91–105, Linköping, Sweden, 24-26 April 1996. Springer Verlag.

[5] Cristina N. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[6] J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–25, July 1970.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 18, pages 365–367. The MIT Press and McGraw-Hill Book Company, 1989.

[8] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc, 52 Vanderbilt Ave, New York, New York 10017, 1977.

[9] Sparc International Inc. *The SPARC Architecture Manual*. Prentice-Hall, 1992.

[10] Ulrike Lichtblau. Decompilation of control structures by means of graph transformations. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 284–297, Berlin, 1985.

[11] M.H.Williams. Generating structured flow diagrams: The nature of unstructuredness. *Computer Journal*, 25:393–396, 1982.

[12] M.S.Hecht and J.Ullman. A simple algorithm for global data flow analysis problems. *SIAM Jounal of Computing*, 4(4):519–532, December 1975.

[13] G. Oulsam. Unravelling unstructured programs. *Computer Journal*, 25:379–387, 1982.

[14] Thomas Pittman and James Peters. *The art of compiler design : theory and practice*. Prentice Hall, Englewood Cliffs,NJ, 1992.

[15] Andrew S. Tanenbaum. *Structured Computer Organization*, chapter 7. Prentice-Hall International, 1990.