Reverse Compilation Techniques

by

Cristina Cifuentes

Bc.App.Sc – Computing Honours, QUT (1990)
Bc.AppSc – Computing, QUT (1989)

Submitted to the School of Computing Science
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

at the

QUEENSLAND UNIVERSITY OF TECHNOLOGY

July 1994

Chapter 6

Control Flow Analysis

The control flow graph constructed by the front-end has no information on high-level language control structures, such as `if..then..else`s and `while()` loops. Such a graph can be converted into a structured high-level language graph by means of a *structuring* algorithm. High-level control structures are detected in the graph, and subgraphs of control structures are tagged in the graph. The relation of this phase with the data flow analysis phase and the back-end is shown in Figure 6-1.

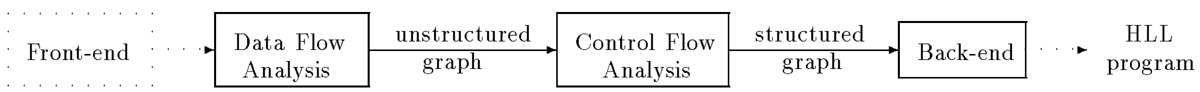

Figure 6-1: Context of the Control Flow Analysis Phase

A generic set of high-level control structures is used to structure the graph. This set should be general enough to cater for different control structures available in commonly used languages such as C, Pascal, Modula-2, and Fortran. Such structures should include different types of loops and conditionals. Since the underlying structure of the graph is not modified, functional and semantical equivalence is preserved by this method.

6.1 Previous Work

Most structuring algorithms have concentrated on the removal of `goto` statements from control flow graphs at the expense of introduction of new Boolean variables, code replication, the use of multilevel exit loops, or the use of a set of high-level structures not available in commonly used languages. A graph transformation system has also been presented, it aims at the recognition of the underlying control structures without the removal of all `goto` statements. The following sections summarize the work done in this area.

6.1.1 Introduction of Boolean Variables

Böhm and Jacopini[BJ66] proved that any program flowgraph can be represented by another flowgraph which is decomposable into π (sequence of nodes), ϕ (post-tested loop), and $\triangle$ (2-way conditional node) with the introduction of new Boolean variables and assignments to these variables. Cooper[Coo67] pointed out that if new variables may be introduced to the original program, any program can be represented in one node with at most one ϕ; therefore, from a practical point of view, the theorem is meaningless[Knu74].

Ashcroft and Manna[AM71] demonstrated that `goto` programs cannot be converted into `while()` programs without the introduction of new variables, and presented an algorithm for the conversion of these programs with the introduction of new Boolean variables. The conversion preserves the topology of the original flowchart program, but performs computations in different order.

Williams and Ossher[WO78] presented an iterative algorithm to convert a multiexit loop into a single exit loop, with the introduction of one Boolean variable and a counter integer variable for each loop.

Baker and Zweben[BZ80] reported on the structuring of multiexit loops with the introduction of new Boolean variables. The structuring of multiple exit loops is considered a control flow complexity issue, and is measured in this paper.

Williams and Chen[WG85] presented transformations to eliminate `goto` statements from Pascal programs. `Goto`s were classified according to the positioning of the target label: at the same level as the corresponding label, branch out of a structure, transferral of a label out of a structure, and abnormal exits from subroutines. All these transformations required the introduction of one or more Boolean variables, along with the necessary assignment and test statements to check on the value of a Boolean. The algorithm was implemented in Prolog on a PDP11/34.

Erosa and Hendren[EH93] present an algorithm to remove all `goto` statements from C programs. The method makes use of `goto`-elimination and `goto`-movement transformations, and introduces one new Boolean variable per `goto`. On average, three new instructions are introduced to test for each new Boolean, and different loop and if conditionals are modified to include the new Boolean. This method was implemented as part of the McCAT parallelizing decompiler.

The introduction of new (Boolean) variables modifies the semantics of the underlying program, as these variables do not form part of the original program. The resultant program is functionally equivalent to the original program, thus it produces the same results.

6.1.2 Code Replication

Knuth and Floyd[KF71] presented different methods to avoid the use of `goto` statements without the introductions of new variables. Four methods were given: the introduction of recursion, the introduction of new procedures, node splitting, and the use of a `repeat..until()` construct. The use of the node splitting technique replicates code in the final program. It is also proved that there exist programs whose `goto` statements cannot be eliminated without the introduction of new procedure calls.

Williams[Wil77] presents five subgraphs which lead to unstructured graphs: abnormal selection paths, multiple exit loops, multiple entry loops, overlapping loops, and parallel loops. In order to transform these subgraphs into structured graphs, code duplication is performed.

Williams and Ossher[WO78] presented an algorithm to replace multiple entry loops by single entry `while()` loop. The method made use of code duplication of all nodes that could be reached from abnormal entries into the loop.

Baker and Zweben[BZ80] reported on the use of the node splitting technique to generate executionally equivalent flowgraphs by replicating one or more nodes of the graph. Node splitting was considered a control flow complexity issue, and was measured.

Oulsnam[Oul82] presented transformations to convert six types of unstructured graphs to structured equivalent graphs. The methodology made use of node duplication, but no function duplication. It was demonstrated that the time overhead produced by the duplication of nodes was an increased time factor of 3 for at least one path.

Code replication modifies the original program/graph by replicating code/node one or more times, therefore, the final program/graph is functionally equivalent to the original program/graph, but its semantics and structure have been modified.

6.1.3 Multilevel Exit Loops and Other Structures

Baker[Bak77] presented an algorithm to structure flowgraphs into equivalent flowgraphs that made use of the following control structures: `if..then..else`, multilevel `break`, multilevel `next`, and endless loops. `Goto`s were used whenever the graph could not be structured using the previous structures. The algorithm was extended to irreducible graphs as well. It was demonstrated that the algorithm generated well-formed and properly nested programs, and that any `goto` statements in the final graph jumped forward. This algorithm was implemented in the `struct` program on a PDP11/54 running under Unix. It was used to rewrite Fortran programs into Ratfor, an extended Fortran language that made use of control structures. The `struct` program was later used by J.Reuter in the `decomp` decompiler to structure graphs built from object files with symbol information.

Sharir[Sha80] presented an algorithm to find the underlying control structures in a flow graph. This algorithm detected normal conditional and looping constructs, but also detected proper and improper strongly-connected intervals, and proper and improper outermost intervals. The final flow graph was represented by a hierarchical flow structure.

Ramshaw[Ram88] presented a method to eliminate all `goto` statements from programs, by means of forward and backward elimination rules. The resultant program was a structurally equivalent program that made use of multilevel exits from endless-type, named loops. This algorithm was used to port the Pascal version of Knuth's TEX compiler into the PARC/CSL, which uses Mesa. Both these languages allow the use of `goto` statements, but outward `goto`s are not allowed in Mesa.

The use of multilevel exits or high-level constructs not available in most languages restricts the generality of the structuring method and the number of languages in which the structured version of the program can be written. Currently, most 3rd generation languages (e.g. Pascal, Modula-2, C) do not make use of multilevel exits; only Ada allows them.

6.1.4 Graph Transformation System

Lichtblau[Lic85] presented a series of transformation rules to transform a control flow graph into a trivial graph by identifying subgraphs that represent high-level control structures; such as 2-way conditionals, sequence, loops, and multiexit loops. Whenever no rules were applicable to the graph, an edge was removed from the graph and a `goto` was generated in its place. This transformation system was proved to be finite Church-Rosser, thus the transformations could be applied in any order and the same final answer is reached.

Lichtblau formalized the transformation system by introducing context-free flowgraph grammars, which are context-free grammars defined by production rules that transform one graph into another[Lic91]. He proved that given a rooted context-free flowgraph grammar GG, it is possible to determine whether a flowgraph g can be derived from GG. He provided an algorithm to solve this problem in polynomial time complexity.

The detection of control structures by means of graph transformations does not modify the semantics or functionality of the underlying program, thus a transformation system provides a method to generate a semantically equivalent graph. Lichtblau's method uses a series of graph transformations on the graph to convert/transform the graph into an equivalent structured graph (if possible). These transformations do not take into account graphs generated from short-circuit evaluation languages, where the operands of a compound Boolean condition are not all necessarily evaluated, and thus generate unstructured graphs according to this methodology.

In contrast, the structuring algorithms presented in this thesis transform an arbitrary control flow graph into a functional and semantical equivalent flow graph that is structured under a set of generic control structures available in most commonly used high-level languages, and that makes use of `goto` *jumps whenever the graph cannot be structured with the generic structures. These algorithms take into account graphs generated by short-circuit evaluation, and thus do not generate unnecessary* `goto` *jumps for these graphs.*

6.2 Graph Structuring

The structuring of a sample control flow graph is presented in an informal way. The algorithms used to structure graphs are explained in Section 6.6. The control flow graph of Figure 6-2 is a sample program that contains several control structures. The intermediate code has been analyzed by the data flow analysis phase, and all variables have been given names.

The aim of a structuring algorithm for decompilation is to determine all underlying control structures of a control flow graph, based upon a predetermined set of high-level control structures. If the graph cannot be structured with the predefined set of structures, `goto` jumps are used. These conditions ensure functional and semantical equivalence between the original and final graph.

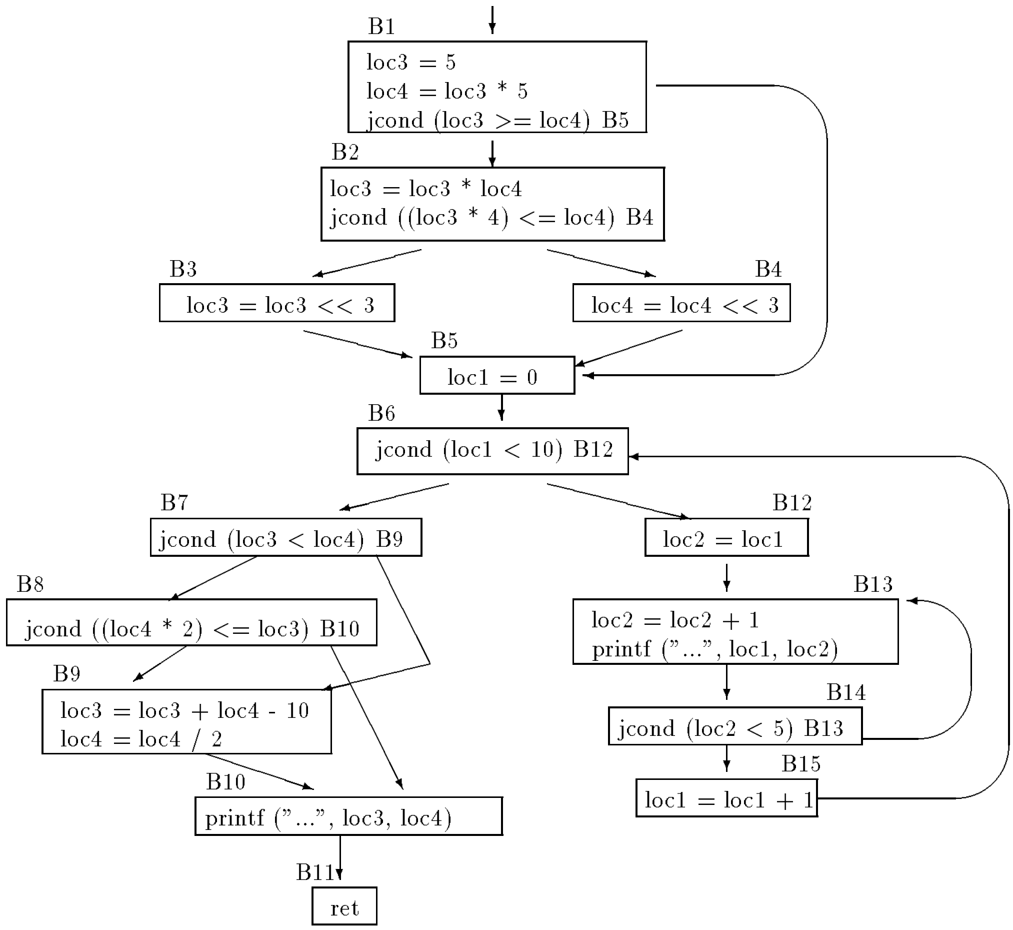

Figure 6-2: Sample Control Flow Graph

6.2.1 Structuring Loops

In graphs, loops are detected by the presence of a back-edge; that is, an edge from a "lower" node to a "higher" node. The notion of lower and higher are not formally defined yet, but can be thought as the nodes that are lower and higher up in the diagram (for a graph that is drawn starting at the top). In the graph of Figure 6-2 there are 2 back-edges: (B14,B13), and (B15,B6). These back-edges represent the extent of 2 different loops.

The type of the loop is detected by checking the header and the last node of the loop. The loop (B14,B13) has no conditional check on its header node, but the last node of the loop tests whether the loop should be executed again or not; thus, this is a post-tested loop, such as a `do..while()` in C, or a `repeat..until()` in Modula-2. The subgraph that represents this loop can be logically transformed into the subgraph of Figure 6-3, where the loop subgraph was replaced by one node that holds all the intermediate code instructions, as well as information on the type of loop.

The loop (B15,B6) has a conditional header node that determines whether the loop is executed or not. The last node of this loop is a 1-way node that transfers control back

B14,B15

```
do /* start of loop */
    loc2 = loc2 + 1
    printf ("...", loc1, loc2)
while (loc2 < 5)
```

Figure 6-3: Post-tested Loop

to the header of the loop. This loop is clearly a pre-tested loop, such as a `while()` loop in a variety of languages. The subgraph of this loop can be logically transformed into the subgraph of Figure 6-4, where the loop subgraph has been replaced with the one node that holds all information on the loop and its instructions.

B6,B13..B16

```
while (loc1 < 10)
   loc2 = loc1
   do /* start of loop */
       loc2 = loc2 + 1
       printf ("...", loc1, loc2)
   while (loc2 < 5)
   loc1 = loc1 + 1
end while
```

Figure 6-4: Pre-tested Loop

6.2.2 Structuring Conditionals

The 2-way conditional node B2 branches control to node B4 if the condition `(loc3 * 2) <= loc4` is true, otherwise it branches to node B3. Both these nodes are followed by the node B5, in other words, the conditional branch that started at node B2 is finished at node B5. This graph is clearly an `if()then..else` structure, and can be logically transformed into the subgraph of Figure 6-5, where the node represents basic blocks B2, B3, and B4. Note that all instructions before the conditional jump that belong to the same basic block are not modified.

The 2-way conditional node B1 transfers control to node B5 if the condition `loc3 >= loc4` is true, otherwise it transfers control to node B2. From out previous example, node B2 has been merged with nodes B3 and B4, and transformed into an equivalent node with an out-edge to node B5; thus, there is a path from node B2 $\rightarrow$ B5. Since B5 is one of the target branch nodes of the conditional at node B1, and it is reached by the other branch of the conditional, this 2-way node represents a single branch conditional (i.e. an `if()then`). This subgraph can be transformed into the node of Figure 6-6, where the condition at node B1 has been negated since the false branch is the single branch that forms part of the `if`.

B2..B4

```
loc3 = loc3 * loc4
if ((loc3 * 2) <= loc4) then
   loc4 = loc4 << 3
else
   loc3 = loc3 << 3
end if
```

B5

Figure 6-5: 2-way Conditional Branching

B1..B4

```
loc3 = 5
loc4 = loc3 * 5
if (loc3 < loc4) then
   loc3 = loc3 * loc4
   if ((loc3 * 2) <= loc4) then
      loc4 = loc4 << 3
   else
      loc3 = loc3 << 3
   end if
end if
```

B5

Figure 6-6: Single Branch Conditional

The 2-way conditional nodes B7 and B8 are not trivially structured, since, if node B8 is considered the head of an `if..then..else` finishing at node B10, and node B7 is considered head of an `if..then`, we do not enter the subgraph headed by B8 at its entry point, but in one of the clauses of the conditional branch. If we structure node B7 first as an `if..then`, then node B8 branches out of the subgraph headed at B7 through another node other than the exit node B9; thus, the graph cannot be structured with the `if..then`, and `if..then..else` structures. But since both B7 and B8 only have a conditional branch instruction, these two conditions could be merged into a compound conditional in the following way: node B9 is reached whenever the condition in node B7 is true, or when the condition at B7 is false and the condition at B8 is false as well. Node B10 is reached whenever the condition at node B7 is false and the one at B8 is true, or by a path from node B9. This means that node B9 is reached whenever the condition at node B7 is true or the condition at node B8 is false, and the final end node is basic block B10. The final compound condition is shown in Figure 6-7, along with the transformed subgraph.

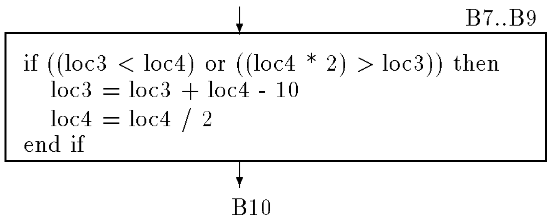

Figure 6-7: Compound Conditional Branch

6.3 Control Flow Analysis

Information on the control structures of a program is available through *control flow analysis* of the program's graph. Information is collected in the different nodes of the graph, whether they belong to a loop and/or conditional, or are not part of any structure. This section defines control flow terminology available in the literature; for more information refer to [All72, Tar72, Tar74, HU75, Hec77, ASU86b].

6.3.1 Control Flow Analysis Definitions

The following definitions define basic concepts used in control flow analysis. These definitions make use of a directed graph $G = (N, E, h)$.

Definition 44 *A* **path** *from n_1 to n_v; $n_1, n_v \in N$, represented $n_1 \rightarrow n_v$, is a sequence of edges $(n_1, n_2), (n_2, n_3), \ldots, (n_{v-1}, n_v)$ such that $(n_i, n_{i+1}) \in E, \forall\, 1 \leq i < v, v \geq 1$.*

Definition 45 *A* **closed path** *or* **cycle** *is a path $n_1 \rightarrow n_v$ where $n_1 = n_v$.*

Definition 46 *The* **successors** *of $n_i \in N$ are $\{n_j \in N \mid n_i \rightarrow n_j\}$ (i.e. all nodes reachable from n_i).*
The **immediate successors** *of $n_i \in N$ are $\{n_j \in N \mid (n_i, n_j) \in E\}$.*

Definition 47 *The* **predecessors** *of $n_j \in N$ are $\{n_i \in N \mid n_i \rightarrow n_j\}$ (i.e. all nodes that reach n_j).*
The **immediate predecessors** *of $n_j \in N$ are $\{n_i \in N \mid (n_i, n_j) \in E\}$.*

Definition 48 *A node $n_i \in N$* **back dominates** *or* **predominates** *a node $n_k \in N$ if n_i is on every path $h \rightarrow n_k$. It is said that n_i* **dominates** *n_k.*

Definition 49 *A node $n_i \in N$* **immediately back dominates** *$n_k \in N$ if $\nexists\, n_j \bullet n_j$ back dominates $n_k \wedge n_i$ back dominates n_j (i.e. n_i is the closest back dominator to n_k). It is said that n_i is the* **immediate dominator** *of n_k.*

Definition 50 *A* **strongly connected region** *(SCR) is a subgraph $S = (N_S, E_S, h_S)$ such that $\forall n_i, n_j \in N_S \bullet \exists\, n_i \rightarrow n_j \wedge n_j \rightarrow n_i$.*