Structuring Decompiled Graphs

Cristina Cifuentes*

Department of Computer Science, University of Tasmania
GPO Box 252C, Hobart, Tas 7001, Australia
Email: C.N.Cifuentes@cs.utas.edu.au

Abstract. A structuring algorithm for arbitrary control flow graphs is presented. Graphs are structured into functional, semantical and structural equivalent graphs, without code replication or introduction of new variables. The algorithm makes use of a set of generic high-level language structures that includes different types of loops and conditionals. `Gotos` are used only when the graph cannot be structured with the structures in the generic set.
This algorithm is adequate for the control flow analysis required when decompiling programs, given that a pure binary program does not contain information on the high-level structures used by the initial high-level language program (i.e. before compilation). The algorithm has been implemented as part of the *dcc* decompiler, an i80286 decompiler of DOS binary programs, and has proved successful in its aim of structuring decompiled graphs.

1 Introduction

A decompiler is a software tool that reverses the compilation process by translating a pure binary input program to an equivalent high-level language (HLL) target program. The input program does not have symbolic information within it, and the HLL used to compile this binary program need not be the same as the target HLL produced by the decompiler.

Although decompilers have not been greatly studied in the literature, there are a variety of applications that could benefit from them, including the obvious maintenance of old code and recovery of lost source code, but also the debugging of binary programs, migration of applications to a new hardware environment [26], verification of generated code by the compiler [23], and translation of code written in an obsolete language.

When binary programs are decompiled, the control flow graph of the program is constructed and analyzed for data and control flow. Data flow analysis transforms the intermediate representation of the binary program into a higher level representation that resembles a high-level language. Control flow analysis determines the underlying structure of the program; that is, the high-level control structures used in each subroutine.

* This work was done while with the Queensland University of Technology in Brisbane, Australia. This research was partly funded by Australian Research Council (ARC) grant No. A49130261.

A structured control flow graph is a graph that can be decomposed into subgraphs that represent control structures of a high-level language. These subgraphs always have one entry point and one exit point. Unstructured graphs are generated by the use of `goto` statements in a program, such that the transfer of control leads to subgraphs with two or more entry points, division of the subgraphs representing a control structure due to an entry into the middle of the structure, or tail-recursive calls in languages such as Lisp [21] and Scheme [16]. Unstructured graphs can also be introduced by the optimizer phase of the compiler, when code motion is used. It is not hard to demonstrate that structured high-level languages that do not make use of the `goto` statement generate reducible graphs [18, 14].

During decompilation, a generic set of high-level control structures needs to be defined first in order to decompose a program's control flow graph into these fixed set of structures. The generic set of high-level structures should be general enough to cater for different control structures available in commonly used languages such as C, Pascal, Modula-2 and Fortran. Generic control structures will always include loops and conditionals. We distinguish different types of loops: pre-test loop (`while()`), post-test loop (`repeat..until()`), and infinite loop (endless `loop`). The `for` loop is a special case of the `while()` loop, so it is not considered a generic structure. Different types of conditionals must also be identified: 2-way conditionals are represented by `if..then` and `if..then..else` structures, and n-way conditionals are represented by `case` control structures. `Goto` statements are used only when the graph cannot be structured using the previous set of generic structures. This means that multiexit loops are structured as a loop with one real exit, and all other exits will make use of `goto` exits. As a general rule of thumb, it is always desirable to structure abnormal loops as multiexit loops rather than multientry loops. This is due to the fact that multientry loops are harder to understand and can produce irreducible graphs.

The rest of this paper is presented in the following way: §2 gives a brief introduction to the intermediate representation used in the *dcc* decompiler, §3 describes the order of application of the loop and conditional algorithms, §4 describes an algorithm to structure loops, §5 describes an algorithm to structure 2-way conditionals, §6 briefly mentions how code generation is done based on structured graphs, §7 mentions previous work done in the area and compares this approach to others, and §8 gives the summary and conclusions of this work.

2 Intermediate Representation of dcc

The intermediate representation of a binary program in *dcc* is in the form of a call graph with a control flow graph for each subroutine, and an intermediate language which has two levels: a low-level stage which is used initially when the program has not been analyzed, and a high-level stage that resembles a HLL [11]. The data flow analysis phase of the decompiler analyzes intermediate instructions to remove all references to low-level concepts such as condition codes and registers, and re-introduces high-level concepts such as expressions and parameter passing. The high-level instructions that are generated by this analysis are: `asgn` (assign), `jcond` (conditional jump), `jmp` (unconditional jump), `call` (subroutine call), and `ret` (subroutine return). No control structure instructions are restored by this phase. This phase preceeds the control flow analysis phase and is described in [10]. This paper concentrates on the control flow analysis phase.

We present a sample control flow graph in Fig. 1 with intermediate instruction information. The intermediate code has been analyzed by the data flow analysis phase, and without loss of generality, all variables have been given names for ease of understanding (names are normally given by the code generator). From observation of Fig. 1, there are several control structures: there is a nested `repeat` loop inside a `while()` loop, several 2-way conditionals (some belong to loops, others stand alone), and a short-circuit evaluated expression. This graph is used throughout the paper to illustrate the structuring algorithms. The convention used in all graphs with 2-way conditional nodes is the following: the right arrow is the true branch and the left arrow is the false branch.

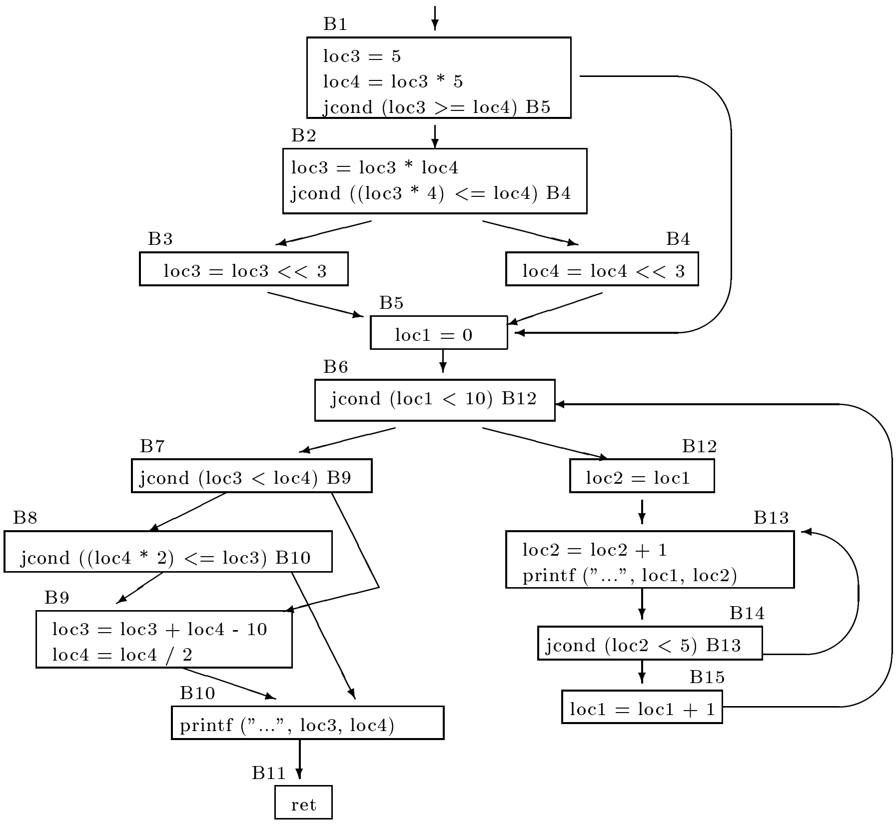


Fig. 1. Sample Control Flow Graph

3 Application Order

The structuring algorithms presented in the next sections determine the header, follow and latching nodes of subgraphs that represent high-level loops and 2-way structures. The header node is the entry node of a structure. The follow node is the first node that is executed after a possibly nested structure has finished. In the case of non-properly nested structures, the follow node is the one after the last execution of a nested structure. The latching node is the last node in a loop; the one that takes as immediate successor the header of a loop.

The algorithms presented in the next sections cannot be applied in a random order since they do not form a finite Church-Rosser system. For example, if node B6 in Fig. 1 is structured first as the header of a 2-way conditional, an `if..then..else` structure would be flagged for this node, and node B15 would have to use a `goto` jump to node B6 as it cannot be part of a loop (the header node of this loop would already belong to another structure, and hence it cannot belong to two different structures at the same nesting level). Therefore, loops are structured before 2-way conditionals to ensure the boolean condition that forms part of pre-tested or post-tested loops is part of the loop, rather than the header of a 2-way conditional subgraph. Once a 2-way conditional has been marked as being in the header or latching node of a loop, it is not considered for further structuring.

4 Structuring Loops

In order to structure loops, a loop needs to be defined in terms of a graph representation. This representation must be able to not only determine the extent of a loop, but also provide a nesting order for the loops. As pointed out by Hecht [14], the representation of a loop by means of cycles is too fine a representation since loops are not necessarily properly nested or disjoint. On the other hand, the use of strongly connected components as loops is too coarse a representation as there is no nesting order. Also, the use of strongly connected regions does not provide a unique coverage of the graph, and does not cover the entire graph.

Interval[2] theory and the derived sequence of graphs[3] $G^1 \dots G^n$ was formulated by F.Allen and J.Cocke in the early 1970's [12, 1, 4]. Interval theory was

[2] An interval $I(h)$ is the maximal, single-entry subgraph in which h is the only entry node and in which all closed paths contain h. The unique interval node h is called the header node. By selecting the proper set of header nodes, graph G can be partitioned into a unique set of disjoint intervals $\mathcal{I} = \{I(h_1), I(h_2), \dots, I(h_n)\}$, for some $n \geq 1$.

[3] The derived sequence of graphs, $G^1 \dots G^n$ is based on the intervals of graph G. The first order graph, G^1, is G. The second order graph, G^2, is derived from G^1 by collapsing each interval in G^1 into a node. The immediate predecessors of the collapsed node are the immediate predecessors of the original header node which are not part of the interval. The immediate successors are all the immediate, non-interval successors of the original exit nodes. Intervals for G^2 are found and the process is repeated until a limit flow graph G^n is found. G^n has the property of being a single node or an irreducible graph.

used as an optimization technique for data flow analysis [2, 3, 5]. However, the use of intervals does provide a representation that satisfies the abovementioned conditions for loops: one loop per interval, and a nesting order provided by the derived sequence of graphs. This was first pointed out by Housel [15].

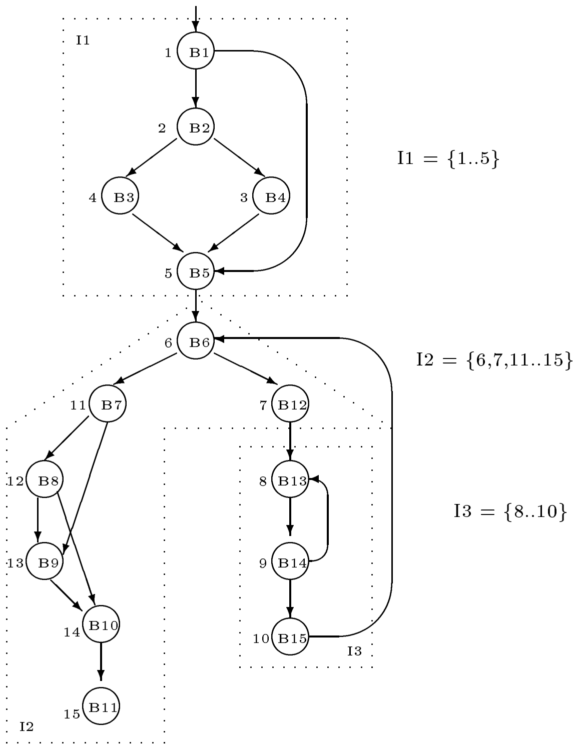


Fig. 2. Intervals of the Control Flow Graph of Fig. 1

Given an interval $I(h_j)$ with header h_j, there is a loop rooted at h_j if there is a back-edge to the header node h_j from a latching node $n_k \in I(h_j)$. Consider the graph in Fig. 2, which is the same graph from Fig. 1 without intermediate instruction information, and with intervals delimitered by dotted lines; nodes are numbered in reverse postorder. There are 3 intervals: I_1 rooted at basic block B1, I_2 rooted at node B6, and I_3 rooted at node B13.

In this graph, interval I_3 contains the loop (B14,B13) in its entirety, and interval I_2 contains the header of the loop (B15,B6), but its latching node is in interval I_3. If each of the intervals are collapsed into individual nodes, and the intervals of that new graph are found, the loop that was between intervals I_3 and I_2 must now belong to the same interval. Consider the derived sequence of graphs $G^2 \ldots G^4$ in Fig. 3. In graph G^2, the loop between nodes I_3 and I_2 is in interval I_5 in its entirety. This loop represents the corresponding loop of nodes (B15,B6) in the initial graph. It is also noted that there are no more loops in these graphs, and that the initial graph is reducible since the trivial graph G^4 was derived by this process. It is noted that the length of the derived sequence is proportional to the maximum depth of nested loops in the initial graph.

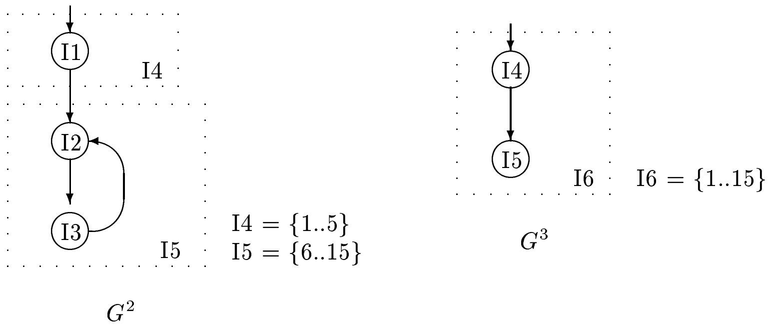


Fig. 3. Derived Sequence of Graphs $G^2 \dots G^4$

Once a loop has been found, the type of loop (e.g. pre-tested, post-tested, endless) is determined by the type of header and latching nodes of the loop. A `while()` loop is characterized by a 2-way header node and a 1-way latching node, a `repeat..until()` is characterized by a 2-way latching node and a non-conditional header node, and a endless `loop` loop is characterized by a 1-way latching node and a non-conditional header node. Heuristics are used when other combinations of header/latching nodes are used. In our example, the loop (B15,B6) has a 2-way header node and a 1-way latching node, hence the loop is equivalent to a `while()`. In a similar way, the loop (B14,B13) has a 1-way header node and a 2-way latching node, hence it is equivalent to a `repeat..until()`. The nodes that belong to the loop are also flagged as being in a loop, in order to prevent nodes from belonging to two different loops; such as in overlapping, or multientry loops. In the case of nested loops, a node will be marked as belonging to the most nested loop it belongs to.

Finally, the follow of the loop, that is, the first node that is reached from the exit of the loop is determined. In the case of a `while()`, the follow node is the target node of the header that does not form part of the loop. In a `repeat..until()`, the follow node is the target node of the latching node that is not a back-edge. And in an endless `loop`, there could be no follow node if there are no exits from the loop, otherwise, the follow is the closest node to the loop (i.e. the smallest node in reverse postorder numbering).

Given a control flow graph $G = G^1$ with interval information, the derived sequence of graphs $G^1, \dots, G^n$ of G, and the set of intervals of these graphs, $\mathcal{I}^1 \dots \mathcal{I}^n$, an algorithm to find loops is as follows: each header of an interval in G^1 is checked for having a back-edge from a latching node that belongs to the same interval. If this happens, a loop has been found, so its type is determined, and the nodes that belong to it are marked. Next, the intervals of G^2, $\mathcal{I}^2$ are checked for loops, and the process is repeated until intervals in $\mathcal{I}^n$ have been checked. Whenever there is a potential loop (i.e. a header of an interval that has a predecessor with a back-edge) that has its header or latching node marked as belonging to another loop, the loop is disregarded as it belongs to an unstructured loop. These loops always generate `goto` jumps during code generation. In this algorithm no `goto` jumps and target labels are determined as this is done during the traversal of the graph during code generation. The complete algorithm is given in Fig. 4. This algorithm finds the loops in the appropriate nesting level, from innermost to outermost loop. The loop follow node is the first node that is reached once the loop is terminated. This node is determined during the loop analysis and used during code generation to traverse the tree of structures.

5 Structuring 2-way Conditionals

Both a single branch conditional (i.e. `if..then`) and an `if..then..else` conditional subgraph have a common follow node that has the property of being immediately dominated by the 2-way header node. Whenever these subgraphs are nested, they can have different follow nodes or share the same common follow node. Consider the graph in Fig. 5, which is the same graph from Fig. 1 without intermediate instruction information, and with immediate dominator[4] information. The nodes are numbered in reverse postorder.

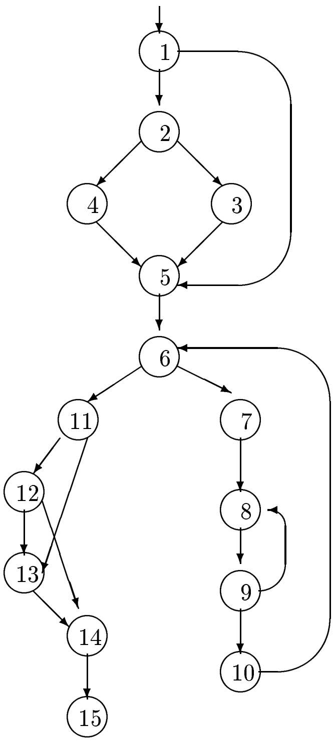
In this graph there are six 2-way conditional nodes; namely, nodes 1, 2, 6, 9, 11, and 12. As seen during loop structuring (§4), a 2-way node that belongs to either the header or the latching node of a loop is marked as being part of the loop, and must therefore not be processed during 2-way conditional structuring. Hence, nodes 6 and 9 in Fig. 5 are not considered in this analysis. Whenever two or more conditionals are nested, it is always desirable to analyze the innermost nested conditional first, and then the outer ones. In the case of the conditionals at nodes 1 and 2, node 2 must be analyzed first than node 1 since it is nested in the subgraph headed by 1; in other words, the node that has a greater reverse postorder numbering needs to be analyzed first since it was last visited first in the depth first search traversal that numbered the nodes. In this example, both subgraphs share the common follow node 5; therefore, there is no node that is immediately dominated by node 2 (i.e. the inner conditional), but 5 is immediately dominated by 1 (i.e. the outer conditional), and this node is the follow node for both conditionals. Once the follow node has been determined, the type of the conditional can be known by checking whether one of the branches of the 2-way header node is the follow node; in which case, the subgraph is a single branching conditional, otherwise it is an `if..then..else`. In the case of nodes

[4] A node n_i dominates n_k if n_i is on every path from the header of the graph to n_k. It is said that n_i immediately dominates n_k if it is the closest dominator to n_k.

```
procedure loopStruct (G = (N, E, h))
/* Pre:  G^1 ... G^n has been constructed.
 *       I^1 ... I^n has been determined.
 *       ∀j ∈ {1 ... n} • I^i = {I_1^i(h_j1), ..., I_m^i(h_jm)}
 *       ∀i, j • I_j^i(h_ij) is the jth interval of G^i with header h_ij.
 * Post: all nodes of G that belong to a loop are marked.
 *       all loop header nodes have information on the type of loop and the
 *       latching node. */

 for (G^i := G^1 ... G^n)
     for (all I_j^i(h_ij) ∈ I^i)
         /* find latching node x */
         if ((∃x ∈ N^i • (x, h_ij) ∈ E^i) ∧ (inLoop(x) == False))
             for (all n ∈ loop (x, h_ij))
                 inLoop(n) = True
             end for
             /* determine loop type */
             if (nodeType(x) == 2-way)
                 if (nodeType(h_ij) == 1-way)
                     loopType(h_ij) = Post-tested
                 else /* 2-way header node */
                     use heuristics to determine best type of loop
             else /* 1-way latching node */
                 if (nodeType(h_ij) == 2-way)
                     loopType(h_ij) = Pre-Tested
                 else
                     loopType(h_ij) = Endless
             end if
             /* determine loop follow */
             case (loopType(h_ij))
             Pre-Tested:
                 if (inLoop(successor(h_ij,1)))
                     loopFollow(h_ij) = successor(h_ij,2)
                 else
                     loopFollow(h_ij) = successor(h_ij,1)
             Post-Tested:
                 if (inLoop(successor(x,1)))
                     loopFollow(h_ij) = successor(x,2)
                 else
                     loopFollow(h_ij) = successor(x,1)
             Endless:
                 determine follow node (if any) by traversing all nodes in the loop
             end case
         end if
     end for
 end for
end procedure
```

Fig. 4. Loop Structuring Algorithm



Node	Immediate Dominator
1	-
2	1
3	2
4	2
5	1
6	5
7	6
8	7
9	8
10	9
11	6
12	11
13	11
14	11
15	14

Fig. 5. Control Flow Graph with Immediate Dominator Information

11 and 12, node 12 is analyzed first and no follow node is determined since no node takes it as immediate dominator. This node is left in a list of unresolved nodes, because it can be nested in another conditional structure (whether fully nested or not). When node 11 is analyzed, nodes 12, 13, and 14 are possible candidates for follow node, since nodes 12 and 13 reach node 14; this last node is taken as the follow (i.e. the node that encloses the most number of nodes in a subgraph; the largest node, given that the subgraphs at nodes 11 and 12 are not properly nested). Node 12, that is in the list of unresolved follow nodes, is also marked as having a follow node of 14. It is seen from the graph that these two conditionals are not properly nested, and a `goto` jump can be used during code generation. A generalization of this example provides the algorithm to structure conditionals, and it is shown in Fig. 6.

N-way conditionals are structured in a similar way to 2-way conditionals. Nodes are traversed from bottom to top of the graph in order to find nested n-way conditionals first, followed by the outer ones. For each n-way node, a follow node is determined. This node will optimally have n in-edges coming from a path from the n succesor nodes of the n-way header node, and be immediately

```
procedure struct2Way (G=(N,E,h))
/* Pre: G is a graph numbered in reverse postorder.
 * Post: 2-way conditionals are marked in G.
 *       the follow node for all 2-way conditionals is determined. */

    unresolved = {}
    for (all nodes m ∈ N in descending order)
        if ((nodeType(m) == 2-way) ∧ (¬ isLoopHeader(m)) ∧
            (¬ isLoopLatchingNode(m)))
            if (∃ n • n = max{i | immedDom(i) = m ∧ #inEdges(i) ≥ 2})
                follow(m) = n
                for (all x ∈ unresolved)
                    follow(x) = n
                    unresolved = unresolved - {x}
                end for
            else
                unresolved = unresolved ∪ {m}
            end if
        end if
    end for
end procedure
```

Fig. 6. 2-way Conditional Structuring Algorithm for Graph G

dominated by such header node. The method is slightly more complex given the existance of more nodes in the structure. For more information refer to [9].

5.1 Compound Conditions

When structuring graphs in decompilation, not only the structure of the underlying structures is to be considered, but also the underlying intermediate instructions information. Most high-level languages allow for short-circuit evaluation of compound boolean conditions. In these languages, the generated control flow graphs for these conditional expressions become unstructured since an exit can be performed as soon as enough conditions have been checked and determined the expression is true or false as a whole. For example, if the expression `x and y` is compiled with short-circuit evaluation, and expression `x` is false, the whole expression becomes false as soon as `x` is evaluated and therefore the expression `y` is not evaluated. In a similar way, an `x or y` expression is partially evaluated if the expression `x` is true. Figure 7 shows the four different subgraph sets that arise from compound conditions during compilation. The top graphs represent the logical condition that is under consideration, and the bottom graphs represent the short-circuit evaluated graphs for each compound condition. In these graphs, t stands for the *then* node and e stands for the *else* node.

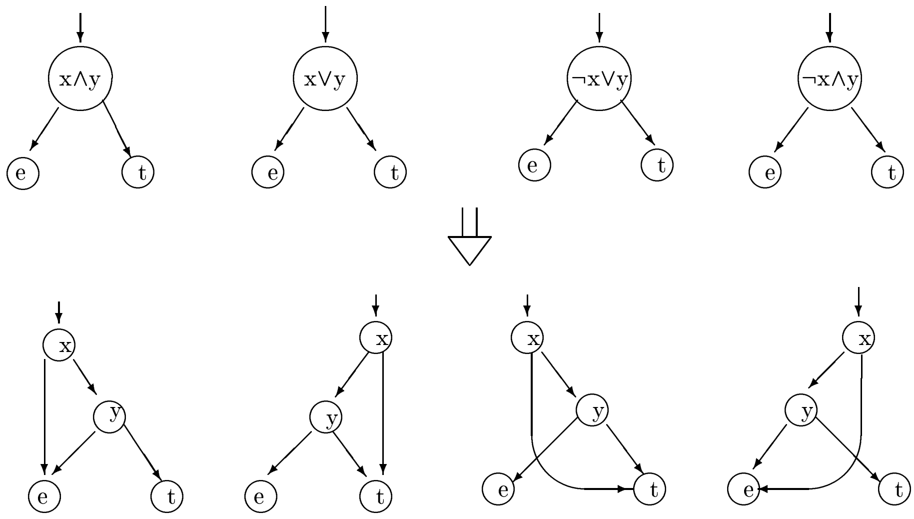


Fig. 7. Compound Conditional Graphs

During decompilation, whenever a subgraph of the form of the short-circuit evaluated graphs is found, it is checked for the following properties:

1. Nodes `x` and `y` are 2-way nodes.
2. Node `y` has only 1 in-edge.
3. Node `y` has a unique instruction; a conditional jump (`jcond`) high-level instruction.
4. Nodes `x` and `y` must branch to a common `t` or `e` node.

The first, second, and fourth properties are required in order to have an isomorphic subgraph to the bottom graphs given in Fig. 7, and the third property is required to determine that the graph represents a compound condition, rather than an abnormal conditional graph. Consider the subgraph of Fig. 1, in Fig. 8, with intermediate instruction information. Nodes B7 and B8 are 2-way nodes, node B8 has 1 in-edge, node B8 has a unique instruction (a `jcond`), and both the true branch of node B7 and the false branch of node B8 reach node B9; i.e. this subgraph is of the form $\neg x \wedge y$ in Fig. 7.

The algorithm to structure compound conditionals makes use of a traversal from top to bottom of the graph, as the first condition in a compound conditional expression is higher up in the graph (i.e. it is tested first). For all 2-way nodes that are not a header or latching node of a loop, the `then` and `else` nodes are checked for a 2-way condition. If either of these nodes represents one high-level conditional instruction (`jcond`), and the node has no other entries (i.e. the only in-edge to this node comes from the header 2-way node), and the node forms one of the bottom 4 subgraphs illustrated in Fig. 7, these two nodes are merged into

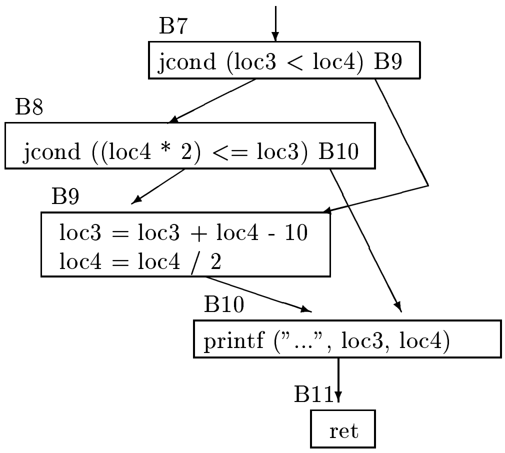


Fig. 8. Subgraph of Fig. 1 with Intermediate Instruction Information

a unique node that has the equivalent semantic meaning of the compound condition (i.e. depends on the structure of the subgraph), and the node is removed from the graph. This process is repeated until no more compound conditions are found (i.e. there could be 3 or more compound `and`s and `or`s, so the process is repeated with the same 2-way node until no more conditionals are found). The algorithm is omitted in this paper and is described in [9].

6 Code Generation from Structured Graphs

The code generation phase of a decompiler generates high-level language code from the intermediate representation of the program. Each procedure's graph is traversed according to the type of basic block and the control structures available in the graph. Basic blocks are checked for being the header of a control structure, in which case, appropriate code is generated for that structure, followed by a depth-first traversal of the nodes associated with the structure; that is, until the follow node is reached. Once code has been generated for the subgraph of a structure, the generation of code is continued with the follow node of the structure. Within a basic block, code is generated sequentially for each statement of the intermediate representation. Once code has been generated for a basic block, the block is marked as having been visited. If during code generation a marked node is reached again, the code for this node is not repeated (i.e. no code replication is used), but a unique label is placed on the first statement of this block, and a `goto` statement is used to reach it. In this way, `goto`s are generated only by the code generator, and no labels are placed on the graph on the earlier control flow analysis phase.

7 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 [8, 29, 22, 28, 6, 13], code replication [17, 27, 29], the use of multilevel exits [7, 24], or the use of a set of high-level structures not available in commonly used languages [25]. None of these methods are applicable to a decompiler because: the introduction of new boolean variables modifies the semantics of the underlying program, as these variables did not form part of the original program; code replication modifies the structure of the program, as code that was written only once gets replicated one or more times; and the use of multilevel exits or high-level structures that are not available in most languages restricts the generality of the method and the number of languages in which the structured version of the program can be written in.

Lichtblau [19] 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 can 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 [20]. 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. Although the detection of control structures by means of graph transformations does not modify the semantics or functionality of the underlying program, Lichtblau's 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 paper transform an arbitrary control flow graph into a functional, semantical and structural 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 subgraphs.

8 Summary and Conclusions

This paper describes a set of structuring algorithms for transforming arbitrary graphs generated by any imperative programming language, into functional, semantical and structural equivalent graphs, without the introduction of new variables or code replication. The algorithm is adequate for the analysis needed in the control flow analysis phase of a decompiler, and has been implemented as part of the *dcc* decompiler; a decompiler for the i286 and the DOS operating system [9]. This set of algorithms are shown to be non Church-Rosser with a counter example.

Structured graphs contain high-level language control structures. Unstructured graphs are introduced by the use of `gotos`, tail-recursion calls and by optimizations produced by the compiler. In a binary program, it is unknown what type of language or compiler was used on the original source program. This means that we cannot determine whether the graph is structured or not, and therefore we assume the general case of unstructured graphs.

The generic control structures that are considered by this structuring algorithm are: `if..then`, `if..then..else`, `case`, `while()`, `repeat..until()` and endless `loop` loops. `Gotos` are used only when the graph cannot be structured with any of the above structures. All other structures available in high-level languages (e.g. `for`, multiexit loops) can be modelled by a second structuring stage that is targeted at the language-specific control structures.

Acknowledgements and Future Work

I would like to thank Vishv Malhotra and the anonymous referees for suggestions on improving the presentation of this paper.

The *dcc* decompiler is being ported to the SPARC architecture and is supported by Sun Microsystems Laboratories and the Centre for Software Maintenance at the University of Queensland. For information on the status of the *dcc* decompiler, check: `http://crg.cs.utas.edu.au/dcc.html`

References

1. F.E. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970.
2. F.E. Allen. A basis for program optimization. In *Proc. IFIP Congress*, pages 385–390, Amsterdam, Holland, 1972. North-Holland Pub.Co.
3. F.E. Allen. Interprocedural data flow analysis. In *Proc. IFIP Congress*, pages 398–402, Amsterdam, Holland, 1974. North-Holland Pub.Co.
4. F.E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report RC 3923 (No. 17789), IBM, Thomas J. Watson Research Center, Yorktown Heights, New York, July 1972.
5. F.E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.
6. Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.
7. B.S. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.
8. C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.
9. C. Cifuentes. *Reverse Compilation Techniques*. PhD dissertation, Queensland University of Technology, School of Computing Science, July 1994.

10. C. Cifuentes. Interprocedural dataflow decompilation. In print: Journal of Programming Languages, 1996.
11. C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.
12. J. Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–25, July 1970.
13. A.M. Erosa and L.J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the International Conference on Computer Languages*, Université Paul Sabatier, Toulouse, France, May 1994. IEEE Computer Society.
14. M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc, 52 Vanderbilt Avenue, New York, New York 10017, 1977.
15. B.C. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages*. PhD dissertation, Purdue University, Computer Science, August 1973.
16. G.L. Steele Jr. and G.J. Sussman. Design of a LISP-based microprocessor. *Communications of the ACM*, 23(11):628–645, November 1980.
17. D.E. Knuth and R.W. Floyd. Notes on avoiding go to statements. *Information Processing Letters*, 1(1):23–31, 1971.
18. S.R. Kosaraju. Analysis of structured programs. *Journal of Computer and System Sciences*, 9(3):232–255, 1974.
19. U. Lichtblau. Decompilation of control structures by means of graph transformations. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, Berlin, 1985.
20. U. Lichtblau. Recognizing rooted context-free flowgraph languages in polynomial time. In G. Rozenberg H. Ehrig, H.J. Kreowski, editor, *Graph Grammars and their application to Computer Science*, number 532 in Lecture Notes in Computer Science, pages 538–548. Springer-Verlag, 1991.
21. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.
22. G. Oulsnam. Unravelling unstructured programs. *The Computer Journal*, 25(3):379–387, 1982.
23. D.J. Pavey and L.A. Winsborrow. Demonstrating equivalence of source code and PROM contents. *The Computer Language*, 36(7):654–667, 1993.
24. L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, October 1988.
25. M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.
26. R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.
27. M.H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal*, 20(1):45–50, 1977.
28. M.H. Williams and G.Chen. Restructuring Pascal programs containing goto statements. *The Computer Journal*, 28(2):134–137, 1985.
29. M.H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured form. *The Computer Journal*, 21(2):161–167, 1978.

This article was processed using the LaTeX macro package with LLNCS style