A Structuring Algorithm for Decompilation [*]

Cristina Cifuentes
cifuente@fitmail.qut.edu.au

School of Computing Science
Queensland University of Technology
GPO Box 2434, Brisbane, QLD 4001, Australia

Abstract

This paper presents a structuring algorithm for arbitrary reducible, unstructured graphs. Graphs are structured into semantically equivalent graphs, without the need of code replication or introduction of new variables. The algorithm makes use of structures such as, `if..then..else`*s,* `while`*,* `repeat` *and* `loop` *loops, and* `case` *statements.* `Goto`*s are only used when the graph cannot be structured with any of the above constructs.*

This algorithm is adequate for the analysis needed in the decompilation of programs, given that a binary program does not contain information as to the language and compiler used to compile the original source program. And given that unstructuredness is introduced by the use of `goto`*s (still widely available in today's compilers) and optimizations produced by the compiler, we have to assume an unstructured graph for our decompilation analysis. This algorithm has been implemented as part of the* dcc *decompiler, currently under development at the Queensland University of Technology.*

*[***Key words:** *structuring algorithm, decompiler, intervals, graph theory]*

1 Introduction

A control flow graph (cfg) is a directed graph that represents the flow of control of a program. Each node represents a basic block and each directed arc represents flow of control from one node to another. Cfgs are normally built by compilers for analysis, code generation, and optimization purposes, but a lot of compilers nowadays produce code on the fly and therefore do not need to construct such cfg. In the case of decompilers, cfgs have proven to be a necessity due to the lack of information about the program constructs.

The control structures of high level languages such as Modula 2 produce structured control flow graphs. Unstructuredness is introduced by the use of `goto`s or optimizations done by the compiler. Even more, the optimizer might even produce irreducible graphs. It is not hard to demonstrate that structured high level languages which do not make use of the `goto` are reducible[1, 2].

A structuring algorithm is concerned with arbitrary graphs, that is, graphs produced by a structured or unstructured high level language, and that in most cases have been optimized and therefore transformed into unstructured or even irreducible graphs. The aim of a structuring algorithm is to transform an arbitrary graph into a semantically equivalent structured graph; whenever this is not possible, `goto`s are used. The final structured graph will be composed of high level language constructs selected from a predetermined set of constructs that suit the problem in hand.

1.1 Previous Work

Structuring algorithms have been used to structure flowgraphs produced by unstructured languages such as Fortran. Most of these algorithms make use of node splitting techniques (i.e. code replication) and introduce new Boolean variables[3, 4], or choose a set of high level language constructs not available in commonly used languages[5].

[0]appeared in *XIX Conferencia Latinoamericana de Informática*, Buenos Aires, Argentina, 2-6 August 1993, pp. 267-276.

In Fortran, the lack of control structures available in the language leads to very unstructured programs. A structuring algorithm that structured Fortran programs into Ratfor (a structured version of Fortran) was develop in 1977[6]. This algorithm made use of **if..then..else**s, loops, multilevel exit and continue, and **goto**s. It proved to be very useful, as the structured version of a program was easier to understand.

In the case of Pascal, **goto**s can transfer control not only within the procedure, but also from one procedure to another. Different methods have been proposed; most of them introduce new local and global variables[3], or make use of multilevel exits[7].

Only one method was directly related to decompilation, and it explained only how to structure loops by means of intervals; other constructs were not mentioned[8]. Since this decompiler took as input only programs compiled in the Mixal language, it did not have to consider complex control structures such as **case** statements, multiexit, or multientry loops.

The rest of this paper is structured in the following way: an explanation of the needs and requirements of a structuring algorithm for decompilation, basic notions from graph theory, a description of each major structuring stage, followed by a summary and conclusions.

2 The need for another structuring algorithm

The structuring algorithms available in the literature are not specifically targeted at decompilation and cannot be easily accommodated to this problem. As mentioned earlier, the control flow graph of executable programs is likely to be unstructured and therefore we need to handle this property. Also, the set of desired high level language constructs should be wide enough to cater for different control structures available in commonly used languages such as C, Pascal, Modula-2, and Fortran.

The main requirements of this algorithm are:

1. The need to structure loops, **if..then..else**s and **case** statements.
2. To differentiate between different type of loops, namely, **while**, **repeat**, and **loop**. Note that the **for** loop is a special case of **while** loops.
3. Multiexit loops should have one real exit, and all other exits should make use of **goto**s.
4. Multiexit loops are preferred over multientry loops, since multientry loops are not as easy to understand, and can produce irreducible graphs.
5. **Goto**s are only to be used when it is really necessary, i.e. the graph could not be structured in any other way.
6. Functional and structural equivalence are needed, therefore, the introduction of new variables or code replication are not allowed.

This algorithm will be concerned only with reducible flow graphs, given that any irreducible graph can be transformed into an equivalent reducible graph by means of node splitting. Different methods have been proposed in the literature[9, 2].

For decompilation, once the graph has been structured, an extra pass can be done to structure even further. This restructuring stage deals with specific control structures available in the target language. In the case of our decompiler, *dcc*, the target language is C, and several of C's structures were not considered in the structuring stage. One of such structures is the **continue** statement, which terminates the execution of the current loop iteration and starts the next iteration. This is equivalent to a multiexit loop with a **goto** to the start of the loop, and this can be easily checked for. Another example are **for** loops. They are equivalent to a **while** loop that makes use of an indexed variable. In this case there would be need to find this indexed variable via data flow analysis. It is not the purpose of this paper to explain any further about this restructuring stage, as it is target language dependent.

3 Basic Notions

A *directed graph* G is a tuple (V, E, h) where V is the set of nodes, E is the set of edges, and h is the root of the graph. An edge is a pair of nodes (v, w), with $v, w \in V$. A *path* from v_1 to v_n, represented $v_1 \rightarrow v_n$, is a sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$.

An *interval* is a graph theoretic construct first defined by J.Cocke[10], and widely used by F.Allen[11, 12]. An interval $I(h)$ is the maximal, single-entry subgraph in which h is the only entry node and in which

```
$\mathcal{I} := \{\}$.
$H := \{h\}$.
for (all unprocessed $n \in H$) do
  $I(n) := \{n\}$.
  repeat
      $I(n) := I(n) + \{m \in G :$
            $\forall p = \text{immedPred}(m), p \in I(n)\}$.
  until
      no more nodes can be added to $I(n)$.
  $H := H + \{m \in G : m \notin H$ and $m \notin I(n)$
      and $(\exists\, p = \text{immedPred}(m) : p \in I(n))\}$.
  $\mathcal{I} := \mathcal{I} + I(n)$.
endFor
```

Figure 1: Interval Algorithm

```
$G^1 = G$.
$\mathcal{I}^1 = \text{intervals}(G^1)$.
$i = 2$.
Repeat /* Construction of $G^i$ */
    Make each interval of $G^{i-1}$ a node in $G^i$.
    immedPreds($n$) $n \in G^i$ = immedPreds($h$) :
        immedPred($h$) $\notin I^{i-1}(h)$.
    $(a,b) \in G^i$ iff $\exists\, n \in I^{i-1}(h)$ and
        $m = \text{header}(I^{i-1}(m)) : (m,n) \in G^{i-1}$.
    $\mathcal{I}^i = \text{intervals}(G^i)$.
Until
    $G^i == G^{i-1}$.
```

Figure 2: Derived Sequence Algorithm

all closed paths contain h. The unique interval node h is called the ***header node***. By selecting the proper set of header nodes, G can be partitioned into a unique set of disjoint intervals $\mathcal{I} = \{I(h_1), I(h_2), \ldots, I(h_n)\}$, for some $n \geq 1$. The algorithm to find the unique set of intervals of a graph is described in ***Figure 1*** for a graph G with entry node h. This algorithm makes use of the variables H (set of header nodes), $I(i)$ (set of nodes of interval i), and $\mathcal{I}$ (list of intervals of the graph G), as well as the function immedPred(n) which returns the next immediate predecessor of n.

The ***derived sequence of graphs***, $G^1 \ldots G^n$, was described by F.Allen[11, 12] based on the intervals of graph G. The construction of graphs is an iterative method that collapses intervals. 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 we find a limit flow graph G^n. G^n has the property of being a single node or an irreducible graph. ***Figure 2*** describes such an algorithm.

Depth first search, DFS, is a traversal method that selects edges to traverse emanating from the most recently visited node which still has unvisited edges[13]. A ***depth first spanning tree*** (DFST) of a flow graph

```
procedure DFS($x$)
    $x$.visited = True
    for (all unprocessed $s \in$ successors($x$))
        if ($s$.visited == False)
            add $(x, s)$ to DFST
            DFS($s$)
        fi
    endFor
    $x$.revPostorder = $i$
    $i = i - 1$
end procedure
```

Figure 3: Depth first search

G is a directed, rooted, ordered spanning tree of G grown by a DFS algorithm[14, 15, 2]. A DFST T can partition the edges in G into three sets:

1. Back edges = $\{(v,w) : w \rightarrow v \in T\}$.
2. Forward edges = $\{(v,w) : v \rightarrow w \in T\}$.
3. Cross edges = $\{(v,w) : \nexists\, (v \rightarrow w$ or $w \rightarrow v)$ and $w \leq v$ in preorder$\}$.

A DFS algorithm also defines a partial ordering of the nodes of G. The ***reverse postorder*** is the numbering of nodes during their last visit; the numbering starts with the maximum number of nodes and finishes at 1. ***Figure 3*** describes a recursive DFS algorithm that constructs a DFST and numbers the nodes

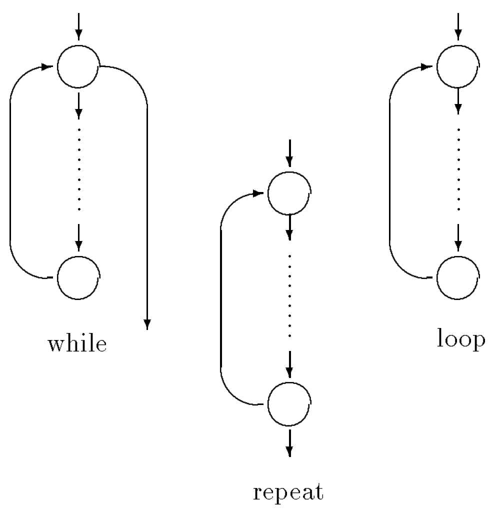

Figure 4: Proper loops

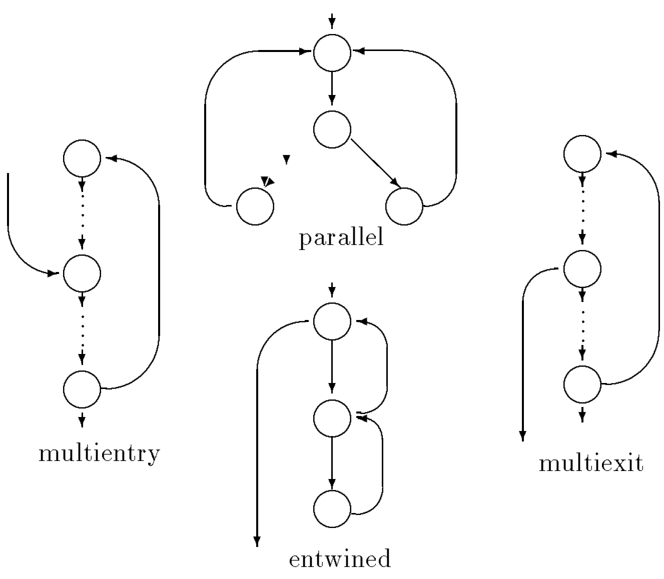

Figure 5: Improper loops

in reverse postorder. The variable i is initialized to the maximum number of nodes in G.

A node v ***dominates*** node w if every path from the root to w contains v. The ***immediate dominator*** of w is the closest dominator node to w, other than itself.

4 Structuring Loops

A loop is a directed subgraph such that there is a path between any two nodes and there is one back edge to the head of the loop (i.e. a strongly connected region (SCR) with one back edge). Such a directed graph has a unique entry point. The back edge comes from a conditional or unconditional node, commonly referred to as the latching node.

4.1 Types of Loops

Loops have been classified into two main groups: proper and improper. A ***proper loop*** is one that fits into the definition of a loop. Different types of simple loops are determined by the header and latching nodes (see ***Figure 4***):

1. While: the header node is conditional, and the latching node is unconditional.
2. Repeat: the latching node is conditional.
3. Loop (endless loop): the latching node is an unconditional node.

Improper loops are based on the structure of proper loops, but have some added characteristics. The following definitions describe their nature (see ***Figure 5***):

1. Multiple entry loop: a loop with n different entries.
2. Multiple exit loop: a loop with m different exits.
3. Tangent loops: two or more loops found in the same strongly connected region. Two different types can be distinguished:
   (a) Parallel loops: loops with a common header node.
   (b) Entwined loops: two loops in different SCRs such that the header node of one SCR is the latching node of the other SCR.

Even though this is an exhaustive list of possible loops, some loops may have characteristics from two different categories, and therefore will need a heuristic method to determine the type of loop. For example, what happens when a loop has both header and latching nodes being conditional nodes? A heuristic method needs to decide whether this is a **`while`** or a **`repeat`** loop. In our case we say that such a loop is considered a **`while`** with an abnormal exit (the one

```
nodesInLoop := {revPost($h$)}
for ($i$ := revPost($h$)+1 ... revPost($a$)-1)
  if ((revPost(immedDom($i$)) $\in$ nodesInLoop) and
    ($i \in$ nodesInInterval))
    nodesInLoop := nodesInLoop + {$i$}
  fi
endFor
nodesInLoop := nodesInLoop + {revPost($a$)}
```

Figure 6: Finding nodes in a loop

from the latching node) if the successors of the header node belong to the loop; otherwise it is considered a **repeat** loop and there are no abnormal exits.

4.2 The method

The method for finding loops is based on interval theory and the derived sequence of graphs $G^1 \ldots G^n$. Intervals give us a way of finding loops, and the derived sequence gives us a way of finding nesting levels. We are interested in structuring proper loops, any other exits or entries the loop might have are going to be handled with labels and **goto**s.

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 $n_k \in I(h_j)$. This property is easily checked when the nodes are numbered in reverse postorder; the latching node will be a predecessor of the header node that will have a greater reverse postorder number than the header. If there is such a back edge, the nodes that belong to the loop have to be flagged. A linear pass (in reverse postorder numbering) over the subgraph rooted at h_j is all that is needed: a node belongs to the loop if its immediate dominator also belongs to the loop. ***Figure 6*** describes an algorithm that returns a list of nodes that belongs to a loop, given the back edge (a, h) that delimits it.

Our method for finding loops in a graph G is as follows, after constructing $\mathcal{I}^1 \ldots \mathcal{I}^n$, the intervals in $\mathcal{I}^1$ are checked for loops, as previously explained. The nodes that belong to loops are flagged so that they only belong to the one loop at any time. Next, the intervals in $\mathcal{I}^2$ are checked for loops, and the process is repeated until intervals in $\mathcal{I}^n$ are checked for. If there is a potential loop that takes a node that has already been flagged, such node cannot be considered and instead a label and a **goto** are to be used; the appropriate fields are flagged (see ***Figure 7***). This process gives us the loops in appropiate nesting levels, from innermost to outermost. This algorithm makes use of extra fields in the basic block node, namely, the loopType (will hold the type of loop, if any), the inLoop Boolean (will determine whether the node belongs to a loop or not), and the label Boolean (determines whether there is need of a label).

```
for ($G^i := G^1 \ldots G^n$)
  for ($I^i(h_j) := I^1(h_1) \ldots I^m(h_m)$)
    if ($h_j$ has a back edge, $(x, h_j)$)
      if ($x$.inLoop == False)
        find all nodes in loop
        flag inLoop for all these nodes
      else
        flag $h_j$.label
      fi
    fi
  endFor
endFor
```

Figure 7: Loop structuring algorithm

It is interesting to see how improper loops are structured by this algorithm, and whether the result is appropriate to the type of loop. The following is the description of each case:

1. ***Parallel loops***: parallel loops belong to the same interval since there is only one header node. Given that parallel loops correspond to **continue**s or optimization of the last jump (i.e. the original graph only had one loop), it is appropriate to consider them as one loop with multiple exits back to the header node.

2. ***Entwined loops***: entwined loops represent nested while loops in most cases. Our method will consider the most nested loop a **while** loop, the second most nested will not be considered to be loop and therefore will make use of a label and a **goto** statement, the third most nested loop will be a **while** loop, and so on.

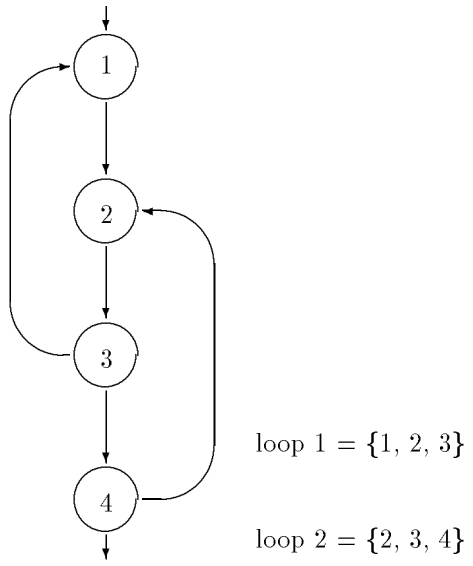

Figure 8: Overlapping loops

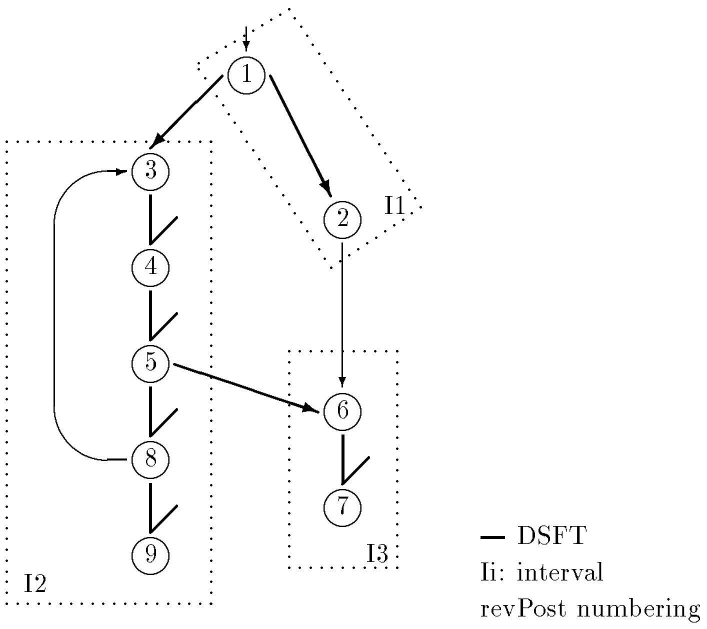

Figure 9: Abnormal tree-edge exit

3. ***Multiple exit loops***: a multiexit loop is contained in the one interval most of the time. When all the exits have the same target node, the loop is certainly contained in the one interval. When there are different target exit nodes, one will be selected to be the real exit and all others will be considered abnormal exits. The type of abnormal exit in the underlying DFST provides some interesting cases:

   (a) Back edge: this case produces ***overlapping loops***. Overlapping loops can be seen from two different perspectives: one sees a multientry loop with an in-edge from the latching node of the other loop, and the second sees a multiexit loop with an out-edge being a back edge for the other loop (see ***Figure 8***). Our method will decide in favour of a multiexit loop rather than a multientry loop.

   (b) Tree edge: this case represents the importance of checking each possible node that belongs to the loop, as being part of the same interval (see ***Figure 9***).

4. ***Multiple entry loops***: a loop with n entries will belong to n different intervals, but there will always be one interval that will hold the back edge

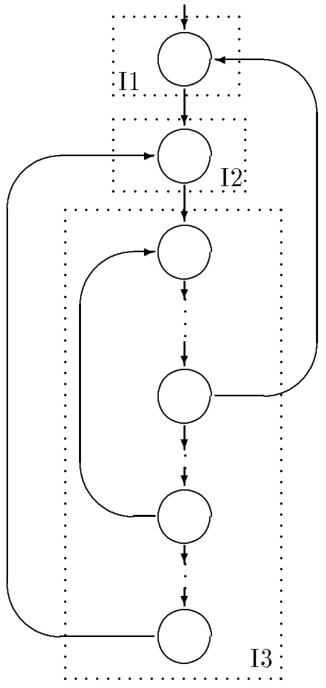

Figure 10: Intervals of a multiple entry loop

(see *Figure 10*). Two cases can be distinguished by the type of in-edge from the underlying DFST:

(a) Backward edge: produces an overlapping loop, as for the case of multiexit loops with an abnormal back edge exit (refer to item 3a).

(b) Tree, cross, and forward edges: all of these cases produce irreducible graphs. Since we are dealing with reducible graphs, this case will never occur.

It is worth to mention that the interval method is not widely used for flow analysis due to its overhead. In our case, ***dcc*** makes use of intervals to check for reducibility of a graph, and this means that the structure already exists and is being used for other parts of the analysis of graphs.

5 Structuring Case Statements

A `case` statement is an n-way conditional branch which has a common end node (i.e. all paths from the n branches should reach this common end node). The end node has the property of being immediately dominated by the case header node; the only other nodes being immediately dominated by the header are the n immediate successors of this header node.

5.1 Abnormal case's

An abnormal `case` is defined as an n-way conditional branch that has more than one end node (i.e. abnormal exits), or has more than one entry point (i.e. abnormal entries). Abnormal entries and exits make it difficult to determine the extent of the `case`, and therefore heuristics are needed for such cases. The following cases demonstrate the heuristics methods used in our algorithm when dealing with different types of abnormal edges of the underlying DFST:

1. Abnormal forward exit: this case has the problem of not knowing which is the real `case` exit. Our heuristic method opted for the node with the greater number of in-edges from paths coming from the n branches to be taken as the real exit.

2. Abnormal backward exit: the problem introduced by this exit is that the back edge should not be considered to form a loop, as it is an abnormal backward exit. The solution is to flag the target nodes as requiring a label, and a `goto` will be used to get to this node.

```
for (all nodes $x$ in reverse revPostorder)
  if (nodeType($x$) == caseNode)
    caseHead = $x$
    endNode = $y$ : (immedPred($y$) = $x$) and ($\forall z$:
      immedPred($z$) = $x$, #inEdges($y$) $\geq$ #inEdges($z$)
    caseNodes := {caseHead}
    for (all successors(caseHead), $s$)
      flagCaseNodes (caseNodes,caseHead,endNode,$s$)
    endFor
    caseNodes := {}
  fi
endFor
```

Figure 11: Case structuring algorithm

3. Abnormal cross-edge or tree-edge exit: the target exit node is not dominated by the `case` header node, so therefore this exit is considered abnormal and the target node is labelled.

4. Abnormal backward in-edge: the latching node is dominated by the end node of the `case`, so all is needed is to label the target node and use a `goto` when jumping to this node.

5. Abnormal forward in-edge: this case is really pathological as it makes a subgraph of the `case` be dominated by a node that does not belong to the `case`; therefore the `case` end node is not dominated by the `case` header node. No solution has been proposed to this case yet.

5.2 The method

The method for structuring `case` statements makes use of the reverse postorder numbering of the nodes in the graph, and a depth first search for each branch of the `case` in hand. This method also handles nested `case` statements, by performing a reverse traversal of the nodes. The algorithms produce a list of nodes that belong to a `case`, such list representing the nodes that are flagged in the current pass.

Given that nested `case`s are allowed in a graph, nodes are traversed in reverse reverse-postorder numbering, so that inner nested `case`s are analysed first than outer ones. Once a `case` header node has been determined, we traverse nodes in reverse-postorder to find any node(s) such that their immediate dominator is the `case` header node. Note that if there are several such nodes, the one with the most number of in-edges

```
flagCaseNodes (caseNodes, head, end, $s$)
  $s$.traversed = True
  if (($s \neq$ end) and (nodeType($s$) $\neq$ case) and
    (immedDom($s$) $\in$ caseNodes))
    caseNodes := caseNodes $\cup$ {$s$}
    for (all successors $r$ of $s$)
      if ($r$.traversed == False)
          flagCaseNodes (caseNodes,head,end,$r$)
      fi
    endFor
  fi
end procedure
```

Figure 12: Finding nodes of a case

is considered the end node (see *Figure 11*). Then, nodes that belong to the **case** need to be flagged, for this process, a depth first procedure is used. Basically, the path of each successor of the header to the end node is followed, and nodes are flagged during this procedure (see *Figure 12*).

6 Structuring Ifs

An **if** statement is a 1- or 2-way conditional branch, which has a common end node that is reached by all paths from the branches. This final end node is referred to as the follow node, and has the property of being immediately dominated by the **if** header node.

6.1 Types of ifs

Two types of **if**s are distinguished, proper and improper. Proper **if**s conform to the previous definition, and are widely known as:

1. **if..then**: 1-way conditional branch. The branch is taken if a condition is true, and the path along the **then** clause is followed. The follow node is the target node of the branch not taken (i.e. when the condition is false), and must be reached by the path along the **then** clause.

2. **if..then..else**: two-way conditional branch. A branch along the **then** clause is taken when the condition is true, otherwise the branch along the **else** clause is taken. Both paths should convey to a common follow node.

```
unresolved = {}
for (all nodes $m$ in reverse revPostorder)
  if (nodeType($m$) == ifNode)
    if ($\exists$ $n$: $n$ = min{$i$: immedDom($i$) = $m$
      and $\exists$ 2 paths from $m \rightarrow i$})
      $m$.endIf = $n$
      for (all $x \in$ unresolved)
        $x$.endIf = $n$
      endFor
    else
      unresolved = unresolved $\cup$ {$m$}
    fi
  fi
endFor
if (#(unresolved) $\geq$ 1)
  for (all $x \in$ unresolved)
    $x$.label = True
  endFor
fi
```

Figure 13: If structuring algorithm

Improper **if**s allow for abnormal entries into the **then** or **else** clauses. In these cases, the follow node will not be immediately dominated by the **if** header node.

6.2 The method

The method for structuring **if**s is based on the fact that the follow node of an **if**has as immediate dominator the head of such **if**. It is also important to notice that nested **if**s are possible, and therefore a follow node might end several **if**s. This method makes use of the reverse postorder numbering of the graph.

Nodes are traversed in reverse reverse-postorder so that inner nested **if**s are analysed before outer ones. A list of unresolved **if** follow nodes is kept throughout the process; it is initially empty. For each **if** node, we check for any other node that takes it as immediate dominator and is a junction node for at least two **if** paths. When there are several of such nodes, the closest node is chosen. The closest relation is given by the reverse-postorder relation; the one with the smallest number is the closest one. This node is selected as the follow of the **if** header node. If there are any nodes in the unresolved list, these nodes will also have the same follow as they are nested in the current **if** node. All nodes in the unresolved list get assigned this follow node, and are eliminated from the list. On the other hand, if there is no node that takes the **if** node as an immediate dominator and is the end of at least two paths, the **if** node is placed on the list of unresolved nodes. The process is repeated for all nodes. Once all nodes have been traversed, if there are any nodes left on the unresolved list, their target branch node is flagged as needing a label during code generation; given that they are not properly nested in another **if**. This procedure is illustrated in *Figure 13*. This algorithm is not optimal (i.e. it does not provide us with the maximum number of **if**s in the graph), but is simple, easy to implement, and covers most of the cases of structured and unstructured **if**s.

7 Summary and Conclusions

This paper presents a structuring algorithm for transforming arbitrary reducible graphs into semantically equivalent structured graphs. This algorithm is adequate for the analysis needed in decompilation, and has been implemented as part of the *dcc* decompiler, currently under development at the Queensland University of Technology.

Structured graphs contain high level language constructs. Unstructuredness is introduced by the use of **goto**s and by optimizations produced by the compiler. Given a binary program, we do not know what type of language and compiler was used on the original source program. This means that we cannot determine whether the graph is structured or not, and thus we suppose we have an unstructured graph (which is true for most graphs).

The major control constructs that are considered by this structuring algorithm are: **if..then..else**s, **while**, **repeat** and **loop** loops, and **case** statements. **Goto**s are only used when the graph cannot be structured with any of the above constructs. All other constructs available in high level languages can be modelled by a second structuring stage, that is targeted at the control structures specific to a language.

The structuring algorithm provides a method for transforming unstructured graphs into structured ones (whenever possible), without the introduction of new variables or code replication. This method makes the final target programs easier to follow and understand.

Acknowledgements

This research is partly funded by Australian Research Council (ARC) grant no.A49130261.

References

[1] S.R.Kosaraju, "Analysis of structured programs," *Journal of Computer and System Sciences*, vol. 9, no. 3, pp. 232–255, 1974.

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

[3] M.H.Williams and G.Chen, "Restructuring pascal programs containing goto statements," *The Computer Journal*, vol. 28, no. 2, pp. 134–137, 1985.

[4] M.H.Williams and H.L.Ossher, "Conversion of unstructured flow diagrams to structured form," *The Computer Journal*, vol. 21, no. 2, pp. 161–167, 1978.

[5] M.Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers," *Computer Languages*, vol. 5, pp. 141–153, 1980.

[6] B.S.Baker, "An algorithm for structuring flowgraphs," *J. ACM*, vol. 24, pp. 98–120, Jan. 1977.

[7] L.Ramshaw, "Eliminating go to's while preserving program structure," *Journal of the ACM*, vol. 35, pp. 893–920, Oct. 1988.

[8] B.C.Housel, *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages*. PhD dissertation, Purdue University, Computer Science, Aug. 1973.

[9] F.E.Allen and J.Cocke, "Graph theoretic constructs for program control flow analysis," Tech. Rep. RC 3923 (No. 17789), IBM, Thomas J. Watson Research Center, Yorktown Heights, New York, July 1972.

[10] J.Cocke, "Global common subexpression elimination," *SIGPLAN Notices*, vol. 5, pp. 20–25, July 1970.

[11] F.E.Allen, "Control flow analysis," *SIGPLAN Notices*, vol. 5, pp. 1–19, July 1970.

[12] F.E.Allen, "A basis for program optimization," in ***Proc. IFIP Congress***, (Amsterdam, Holland), pp. 385–390, North-Holland Pub.Co., 1972.

[13] R.E.Tarjan, "Depth-first search and linear graph algorithms," ***SIAM Journal of Computing***, vol. 1, pp. 146–160, June 1972.

[14] R.E.Tarjan, "Testing flow graph reducibility," ***Journal of Computer and System Sciences***, pp. 355–365, Sept. 1974.

[15] M.Hecht and J.Ullman, "A simple algorithm for global data flow analysis problems," ***SIAM Journal of Computing***, vol. 4, pp. 519–532, Dec. 1975.