Taming Control Flow: A Structured Approach to Eliminating Goto Statements

Ana M. Erosa and Laurie J. Hendren
School of Computer Science
McGill University
Montréal, Québec H3A 2A7
{erosa,hendren}@cs.mcgill.ca

Abstract

In designing optimizing and parallelizing compilers, it is often simpler and more efficient to deal with programs that have structured control flow. Although most programmers naturally program in a structured fashion, there remain many important programs and benchmarks that include some number of `goto` statements, thus rendering the entire program unstructured. Such unstructured programs cannot be handled with compilers built with analyses and transformations for structured programs.

In this paper we present a straight-forward algorithm to structure C programs by eliminating all `goto` statements. The method works directly on a high-level abstract syntax tree (AST) representation of the program and could easily be integrated into any compiler that uses an AST-based intermediate representation. The actual algorithm proceeds by eliminating each `goto` by first applying a sequence of *goto-movement* transformations followed by the appropriate *goto-elimination* transformation.

We have implemented the method in the McCAT (McGill Compiler Architecture Testbed) optimizing/parallelizing C compiler and we present experimental results that demonstrate that the method is both efficient and effective.

1 Introduction and Motivation

Over the years there has been substantial discussion about the use of explicit `goto`s in high-level programs and there have been many arguments against the frequent use `goto`s from a software engineering or program understandability point of view [9, 13, 15]. This discussion has led to the relatively infrequent use of `goto`s in typical C programs [5]. However, in languages like C, there are still special occasions where programmers like to use `goto`s. These include: (1) using `goto`s to exit from deeply nested conditionals or loops; (2) using `goto`s to branch to a common piece of code that is shared among several branches of a switch statement; (3) using `goto`s in automatically generated code such as the code produced by `lex`; and (4) using `goto`s to handle exceptions.

In this paper we are concerned about automatically eliminating explicit `goto`s in order to facilitate the construction of analyses and transformations required for optimizing and parallelizing C compilers. That is, given C source programs that may contain some `goto`s, we wish to automatically transform them into equivalent structured or compositional programs that do not any use `goto`s. We have implemented this method in our McCAT (McGill Compiler Architecture Testbed) parallelizing/optimizing compiler, and thus our compiler can assume fully structured programs for all intermediate forms, analyses and transformations [11].

From the pragmatic point of view there are many reasons why programs without `goto`s are simpler to handle in such compilers. One important consequence is that C programs without `goto`s are compositional, structured analyses techniques can be used to compute data flow information. For example, one can use the efficient techniques available for structured data flow graphs [1], or simple abstract interpretation techniques that need not consider continuation-based semantics. From the program transformation standpoint, compositional programs also lend themselves to simpler and often more efficient algorithms. Consider, for example, the efficient creation of the Static Single Assignment (SSA) form for structured programs consisting of straight-line code, `if` statements, and `while` statements [8], the structured transformations to the ALPHA dependence representation [12], and the efficient construction of Program Dependence Graphs for structured programs [5]. Finally, compositional programs are naturally represented as trees, and intermediate representations based on compositional representations can be manipulated and transformed using a wide variety of strategies including the use of attribute grammars.

Our approach to eliminating `goto`s is based on a set of straight-forward transformations that operate on a high-level structured intermediate representation of the original program. These transformations come in two categories: *goto-eliminations* and *goto-movements*. Intuitively, the method relies on the following observations: (1) when the `goto` statement and target label are in the same statement sequence, a *goto-elimination* transformation can be directly applied to eliminate the `goto`; and (2) if the `goto` statement is in a different statement sequence from the target label, we can use one or more *goto-movement* transformations to move the `goto` to the same statement sequence as the target label and then apply the appropriate *goto-elimination* transformation.

The remainder of this paper is structured as follows. In Section 2 we present the goto-elimination and goto-movement transformations. We first show how they can be applied to remove a single `goto` statement from a C program and then we present a high-level algorithm for eliminating all `goto`s from a C program, thus producing an equivalent structured C program. In Section 3 we show how some optimizations to our method can improve the resulting code. We have completely implemented the method and in Section 4 we give experimental results for both the unoptimized and optimized methods. Finally, in Section 5 we compare our method with related methods, and in Section 6 we give some conclusions and discuss further work.

2 Eliminating Goto Statements

In this section we first present the goto-elimination transformations, and then we present the goto-movement transformations and show how to apply successive goto-movements in order to reach a point where a goto-elimination can be applied. To simplify the explanation of the method, we assume that a `goto` statement is always a conditional `goto` in the form `if (condition) goto Li`. Thus, we assume that any unconditional `goto` of the form `goto Li` is transformed into an equivalent conditional statement of the form `if (true) goto Li`.

Another important point is that we have chosen to directly support `break` and `continue` statements. That is, even though these are a form of control-flow similar to `goto`s, they are already quite tame in the sense that they are compositional, and we can easily handle them in our compiler framework. Thus, there is no benefit in eliminating these statements. Our method could easily be modified to eliminate `break` and `continue` statements, if this was required.

2.1 Goto-elimination Transformations

When both the `goto` statement and the label are in the same statement sequence, we can directly eliminate the `goto` statement. There are two possibilities: the `goto` statement occurs in the program before the `label` statement, or after the `label` statement. In the first case, the `goto` is eliminated and replaced by a conditional, while in the second case the `goto` is eliminated and replaced by a loop.

Goto statement is before label statement:

If the `goto` statement is before the `label` statement, there is an obvious transformation to a conditional statement. As illustrated in Figure 1, the `goto` is eliminated and the statements between the `goto` statement and the label are embedded into a conditional statement guarded by the negation of the condition of the original `goto` statement.

```
    stmt_1;                            stmt_1;
    if (cond) goto L_i;                if (!cond) {
    stmt_2;               ⇒               stmt_2;
    ...                                   ...
L_i:stmt_n;                            }
                                   L_i:stmt_n;
```

Figure 1: Eliminating a `goto` with a conditional

Goto statement is after label statement:

If the `goto` statement is after the `label` statement, then the `goto` is eliminated by embedding the statements between the `label` and the `goto` in a `do-while` loop. The example program in Figure 2 illustrates this case.

```
    stmt_1;                            stmt_1;
L_i:stmt_2;                            do {
    ...                   ⇒        L_i: stmt_2;
    stmt_n;                              ...
    if (cond) goto L_i;                  stmt_n;
                                       }while (cond);
```

Figure 2: Eliminating a `goto` with a loop

These two goto-elimination transformations are obvious, and it is unlikely that a programmer would have used a `goto` in these situations where a conditional or loop are much more reasonable constructs. However, a tool that generates C code could very easily produce such programs. Furthermore, these goto-elimination transformations provide the backbone for the complete method. As described in the next section, we can always eliminate a `goto` by moving the `goto` to the appropriate place and then applying one of these two goto-eliminations.

2.2 Goto-movement Transformations

In order to categorize the goto-movement transformations, we require a precise notion of *offset*, *level*, *sibling statements*, *directly-related statements* and *indirectly-related statements*.

Definition 1 *The* offset *of a* `goto` *or label statement is* n *if the statement is the* n*th* `goto` *or label statement that appears in the source program relative to the beginning of the program.*

Definition 2 *The* level *of a label or a* `goto` *statement is* m *if the label or the* `goto` *statement is nested inside exactly* m `loop`*,* `switch`*, or* `if/else` *statements.*

Definition 3 *A label statement and a* `goto` *statement are* siblings *if there exists some statement sequence,* `stmt_1; ... ; stmt_n`*, such that the label statement corresponds to some* `stmt_i` *and the* `goto` *statement corresponds to some* `stmt_j` *in the statement sequence.*

Definition 4 *A label statement and a* `goto` *statement are* directly-related *if there exists some statement sequence,* `stmt_1; ... ; stmt_n`*, such that either the label or* `goto` *statements corresponds to some* `stmt_i` *and the matching* `goto` *or label statement is nested inside some* `stmt_j` *in the statement sequence.*

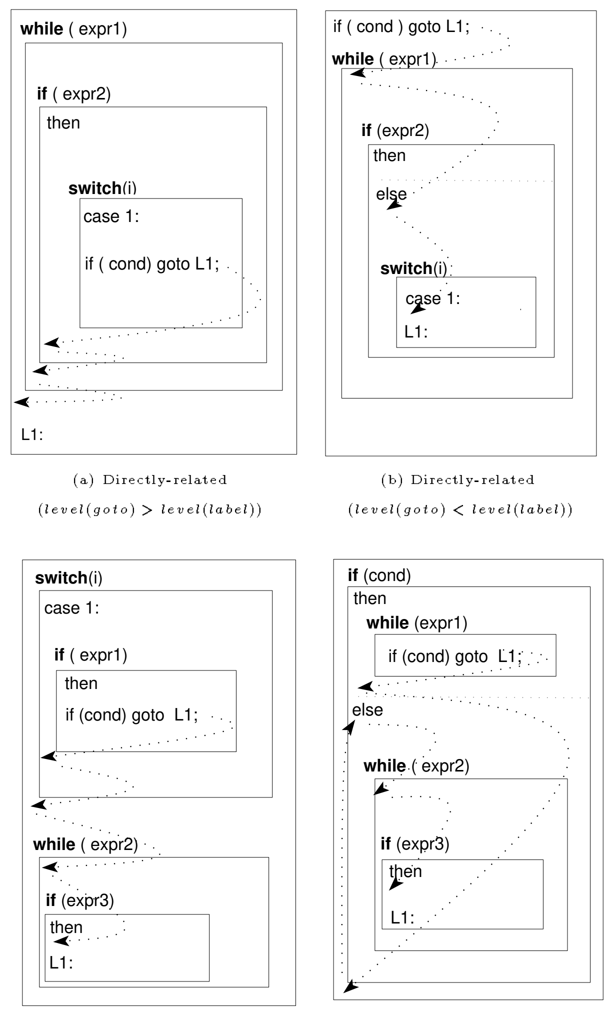
Definition 5 *A label statement and a* `goto` *statement are* indirectly-related *if they appear in the same procedure body, but they are neither* siblings *nor* directly-related*.*

Given these definitions, it is clear that the goto-elimination transformations presented in the previous subsection are applied exactly when the `goto` statement and target label statements are siblings. The goto-elimination transformation given in Figure 1 is used when the offset of the `goto` statement is less than the offset of the target label statement, while the goto-elimination transformation given in Figure 2 is applied when the offset of the `goto` statement is greater than the offset of the target label statement.

We can now restate our overall strategy as follows. Given any goto/label pair, we can eliminate the `goto` by first moving the `goto` until it becomes a sibling of the label, and then applying the appropriate goto-elimination transformation. Figure 3 illustrates the four situations that may occur.

Figure 3(a) illustrates the case when the label and `goto` are directly-related, and the level of the `goto` is greater than the level of the target label. The objective is to move the `goto` to the same level as the label. In this case we apply *outward-movement* transformations, where each transformation moves the `goto` out one level. Figure 3(b) illustrates the case where the label and `goto` are directly-related, and the level of the `goto` is less than the level of the label. In this case we apply *inward-movement* transformations, where each transformation moves the `goto` in one level.

Figures 3(c) and 3(d) illustrate the more complicated situations where the `goto` and label are indirectly-related. When the label and `goto` are in entirely different statements (Figure 3(c)), the `goto` is first moved using outward-movements until it becomes directly-related to the label, and then inward-movements are used to move the `goto` to the same level as the label. When the label and `goto` are in different branches of the same `if` or `switch` statements (Figure 3(d)), then the `goto` is first moved using outward-movements until it becomes directly-related to the enclosing `if` or `switch`, and then inward-movements are used to move it to the same level as the label.



(a) Directly-related ($level(goto) > level(label)$)

(b) Directly-related ($level(goto) < level(label)$)

(c) Indirectly-related (different statements)

(d) Indirectly-related (different branches of the same if/switch)

Figure 3: The four cases for goto/label relationships.

Given that all situations may be handled by inward or outward goto-movements, the only remaining problem is to define both outward- and inward-movement transformations for each kind of construct (loop, conditional, switch). These transformations are presented in the next subsections.

2.2.1 Outward-movement Transformations

The outward movement transformations are very straight-forward. There are basically two cases, moving a `goto` out of a `loop` or `switch` statement, and moving a goto out of an if statement.

Moving a goto out of a loop or switch statement:

This transformation is very simple since we make use of the break statement to exit the switch or loop. We have made use of break since it is compositional and our compiler can handle it easily. However, note that it would also be possible to use a more complicated transformation that did not make use of the break statement, if this was desired. The complete transformation is illustrated in Figure 4. Note that a new variable is introduced to store the value of the conditional at the point at which the goto was encountered. This value is then reused in the goto statement that is introduced at the exit of the switch/loop.

```
 { ...                          { ...
   switch(i): {                   switch(i): {
    case 1:                        case 1:
      stmt_1;                        stmt_1;
      if (cond) goto L1;             goto_L1=cond;
      stmt_2;                        if (goto_L1) break;
      ...                            stmt_2;
      stmt_i;                        ...
      break;         =>              stmt_i;
    case 2:                          break;
      ...                          case 2:
    default:                         ...
      ...                          default:
   }                                 ...
   ...                            }
L1:stmt_n;                        if (goto_L1) goto L1;
 }                                ....
                               L1:stmt_n;
                                }
```

Figure 4: Moving a goto out of a switch

Moving a goto out of an if statement:

In this case the break statement cannot be used, and instead a new conditional is introduced as shown in Figure 5.

```
                                { ...
 { ...                            if (expr) {
   if (expr) {                     stmt_1;
    stmt_1;                        goto_L1=cond;
    if (cond) goto L1;             if (!goto_L1) {
    ...              =>              stmt_2;
    stmt_i;                          ...
   }                                 stmt_i;
   ...                             }
L1:stmt_n;                        }
 }                                if (goto_L1) goto L1;
                                  ...
                               L1:stmt_n;
                                }
```

Figure 5: Moving a goto out of an if

2.2.2 Inward-movement Transformations

In the previous section we presented the relatively simple outward-movement transformations. The inward-movement transformations are slightly more complicated. Firstly, we can not take advantage of the break statements, and secondly we must consider whether the goto appears before or after the target label. We describe the inward-movement transformations for the cases where the goto appears before the label, and then show how we can apply a *goto-lifting* transformation (see Section 2.2.3) that can always move the goto so that it appears before the label.

Moving a goto into a loop statement:

This transformation first introduces a conditional that: (1) embeds the statements that occur between the goto and the start of the loop; and (2) modifies the loop condition such that it will be entered either when the goto expression is true, or when the original loop expression is true. The transformation is illustrated in Figure 6. Note that the short-circuit evaluation in C will ensure that the original loop expression will not be evaluated if entry into the loop is due to the goto. Further, note that the goto variable must be set to false at the point of the label in order to preserve the correct behaviour of the loop in succeeding iterations (i.e. force evaluation of the loop expression).

```
                              { ...
                                goto_L1=cond;
 { ...                          if (!goto_L1) {
   if (cond) goto L1;             stmt_1;
   stmt_1;                        ...
   ...                            stmt_i;
   stmt_i;                      }
   while (expr) {      =>       while (goto_L1||expr) {
    stmt_j;                      if (goto_L1) goto L1;
    .....                        stmt_j;
L1:.....                         ...
    stmt_n;                   L1:goto_L1 = false;
   }                              ...
 }                                stmt_n;
                                }
                              }
```

Figure 6: Moving a goto into a loop

The transformation for do loops is similar, except that the condition of the loop does not need to be modified. To handle for loops that have labels in their body, one can simply transform it to the equivalent while or do loop and then apply the appropriate inward-movement transformation.

Moving a goto into an if statement:

In this case the transformation is similar to the loop transformation, except that the if condition is modified differently depending on whether the label is in the then or else part. If the label is in the then part, the modification of the condition is the same as for the while condition. If the label is in the else part the if condition is modified to lead to the else part, when the goto condition is true, or the if condition is false. Figure 7 illustrates this case.

Moving a goto into a switch statement:

In order to move a goto into a switch statement, one must first locate the case that contains the target label. In order to force control to enter this case, a new variable is defined to be used as the switch variable, and a conditional is introduced that initialize the new variable to the constant expression of the case in question when the condition of the goto is true and to

```
{ ...                          { ...
  if (cond) goto L1;             goto_L1=cond;
  stmt_1;                        if (!goto_L1) {
  ...                              stmt_1;
  stmt_i;                          ...
  if (expr) {                      stmt_i;
    ...                          }
    stmt_j;              =>      if (!goto_L1 && expr) {
    ...                            ...
  }                                stmt_j;
  else {                           ...
    stmt_k;                      }
L1: ...                          else {
    stmt_n;                        if (goto_L1) goto L1;
  }                                stmt_k;
}                              L1: ...
                                   stmt_n;
                                 }
                               }
```

Figure 7: Moving a `goto` into an `if`

the switch expression when the condition of the `goto` is false. This is illustrated in Figure 8.

```
{ ...                          { ...
  if (cond) goto L1;             goto_L1=cond;
  stmt_1;                        if ( !goto_L1 ) {
  ...                              stmt_1;
  stmt_i;                          ...
  switch (i) {                     stmt_i;
   case 1: {                       t_switch=i;
      stmt_j;            =>      }
L1:   ...                        else t_switch = 1;
      stmt_k;                    switch (t_switch) {
    }                              case 1: {
   default:                          if (goto_L1) goto L1;
      ...                            stmt_j;
  }                            L1:   ...
}                                    stmt_k;
                                   }
                                  default:
                                     ...
                                 }
                               }
```

Figure 8: Moving a `goto` into a `switch`

2.2.3 Goto-lifting Transformation

Each of the previous inward-movement transformations have moved a `goto` that appeared before the target label (i.e. *offset(goto)* < *offset(label)*).

However, there are also situations where the target label appears before the matching `goto`. In this case, one must first move the `goto` to just before the statement containing the target label using the goto-lifting transformation, and then apply the appropriate inward-movement transformation.

The goto-lifting transformation is illustrated in Figure 9. In this example `stmt_label` is the statement that contains the label `L1`, and the `goto` statement is originally below `stmt_label`. We can lift the `goto` up above `stmt_label` by introducing a `do` loop that on the first iteration ignores the `goto` and on subsequent iterations uses the value of the conditional at the bottom of the loop. After the `goto` has been lifted, the inward-movement transformations can be used to move the `goto` inside `stmt_label`.

```
                                 { int goto_L1 = false;
                                    ...
                                    stmt_1;
                                    stmt_2;
{ ...                               ...
  stmt_1;                           do {
  stmt_2;                            if (goto_L1) goto L1;
  ...                      =>        stmt_lab;/*contains L1*/
  stmt_lab;/*contains L1*/           ...
  if (cond) goto L1;                 goto_L1 = cond;
  ...                               }
  stmt_n;                           while (goto_L1);
}                                   ...
                                    stmt_n;
                                 }
```

Figure 9: Lifting a `goto` above the statement containing the label

2.3 Examples of Inward and Outward Transformations

Figure 10 illustrates a series of outward transformations followed by a goto-elimination transformation, while Figure 11 illustrates a series of inward transformations followed by a goto-elimination transformation. Note that the dotted arrows indicate the movement just applied, while the dashed arrows indicate the next movement.

2.4 Avoiding the Capture of `break` and `continue` Statements

Since we are directly supporting `break` and `continue` statements, there is one twist that we must consider when applying the goto-elimination (Section 2.1) and goto-lifting (Section 2.2.3) transformations that introduce new `do` loops. Although these transformation seem quite simple and innocent at first, there is one subtle point that arises due to the presence of `break` and `continue` statements. The crucial point is that, on rare occasions, the `do` loop that we introduce captures a `break` or `continue` statement that belongs to an enclosing loop or `switch` statement. Consider, for example, the original program in Figure 12(a) and the incorrect capturing of a `break` statement in Figure 12(b). In order to avoid this situation, we must add one further transformation for each captured `break` or `continue`. As illustrated in Figure 12(c), we need to: (1) introduce one new logical variable for each loop that captures a `break`, (2) set these variables to false at the beginning of procedure, (3) set the appropriate variable to true at the point of the `break`, and (4) check the variable at the exit of the introduced loop: if it is true reset the logical variable to false and issue the proper `break` for the enclosing loop. A similar method for captured `continue` statements may be used.

2.5 Eliminating all goto statements

Based on the goto-elimination, goto-movement and goto-lifting transformations, we can now state the complete algorithm for removing all `goto` statements

```
{ ...
  while (true) {
L_i: stmt_i;
     ...
     stmt_j;
     if (exp1) break;
     ...
     if (exp2) goto L_i;
     ...
     stmt_n;
  }
}
```

(a) original program

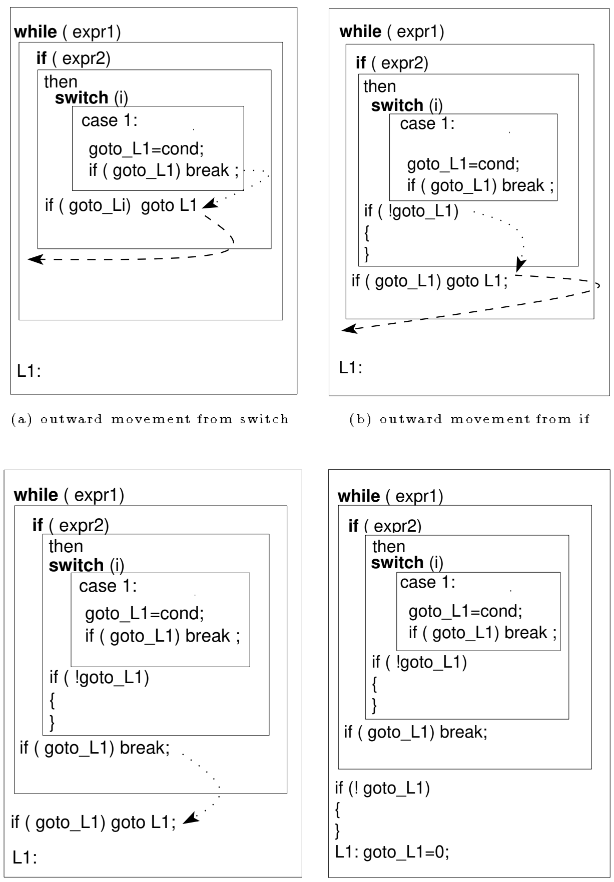
```
{ ...
  while (true) {
     do {
L_i:    stmt_i;
        ...
        stmt_j;
        if (exp1) break;
        ...
        goto_Li = exp2;
     } while (goto_Li);
        ...
        stmt_n;
  }
```

(b) incorrect capture of `break`

```
{ int do_brk = 0;
  ...
  while (true) {
     do {
L_i:    stmt_i;
        ...
        stmt_j;
        if (exp1)
          {do_brk=1; break;}
        ...
        goto_Li = exp2;
     } while (goto_Li);
     if (do_brk)
          {do_brk=0; break;}
     ...
     stmt_n;
  }
}
```

(c) correct treatment of captured `break`

Figure 12: Avoiding capture of `break` and `continue` statements



(a) outward movement from switch (b) outward movement from if

(c) outward movement from while (d) application of goto elimination

Figure 10: Outward movements followed by goto elimination.

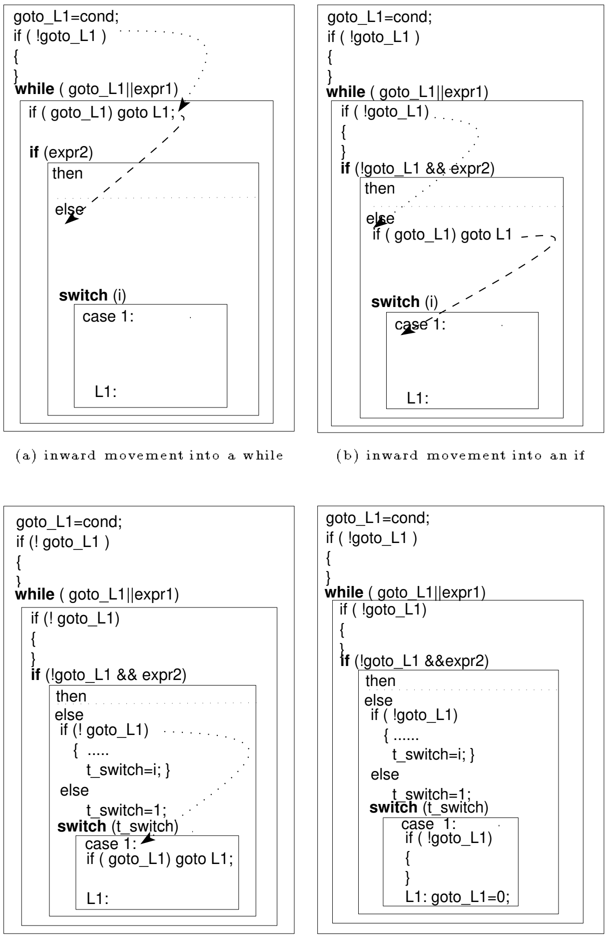
from a C program. The complete algorithm is presented in Figure 13.

For each procedure, the algorithm proceeds in five steps. The first two steps are simple initializations. The first step collects a list of all label and `goto` statements in the procedure. In our implementation, we store the `goto`s in a list in the order in which they appear, and we store the labels in a hash table. The second step introduces one logical variable for each label, initializes the variable to false, and inserts a reinitialization to false at the point of the label. These initializations and reinitializations are required to make sure that the value of the logical variable is false on all paths except the path coming from the point at which the appropriate conditional test evaluated to true. The third step converts all unconditional `goto`s to conditional `goto`s.

The fourth step is the heart of the algorithm where each `goto` is eliminated one at a time. The simplest order to eliminate them is in the order in which they occur in the `goto_list`. However, as we point out in Section 6, there may be better orderings that can be considered. For each `goto`, the matching label is located. For our implementation we make use of the hash table of labels to do this efficiently. Once the `goto`/label pair has been located, it is simply a matter of applying goto-movement transformations until the `goto`/label pair become siblings and then applying the appropriate goto-elimination transformation. Any implementation of this algorithm should be able to support efficient operations to get the level and offset of each label and `goto`, and an efficient means to determine if the `goto` and label are indirectly-related, directly-related, or siblings. In our implementation we store the level and offsets in the goto_list and label hash table, and we make use of parent pointers in the SIMPLE tree to find common ancestors that can be used to efficiently determine the relationship between the `goto` and label.

The fifth step is the elimination of all the labels (since all `goto`s to these labels have now been eliminated).

It should be noted we actually implement all of the initialization steps during one pass through the AST and no further passes are required as all subsequent



(a) inward movement into a while (b) inward movement into an if

(c) inward movement into a switch (d) application of goto elimination

Figure 11: Inward movements followed by goto elimination

steps can be done directly using the information collected in this first pass. That is, we store enough information about the location of `goto` and label statements so as to allow direct manipulation of the required parts of the AST.

3 Optimizations

There are several simple optimizations that can be made as the goto-elimination and goto-movement transformations are applied. Figure 14(a) illustrates the case where a `goto` is immediately next to the label. This situation may occur after several movement transformations, and clearly in this case we may just eliminate the `goto`. Figure 14(b) illustrates the situations where the `goto` is at the end of a statement sequence and is being moved out of an `if`. In this case we can avoid introducing a conditional statement at the end of the block (there are no statements after the `goto` that must be guarded). Similarly, Figures 14(c) and (d) illustrate that when the `goto` is immediately

```
for each procedure p do
{
 /*get list of labels and gotos for procedure*/
   label_list := all labels in procedure p
   goto_list := all gotos in procedure p

 /*introduce and initialize the logical variables*/
   for each label Li in label_list do
    { introduce a var. goto_Li initialized to false
      introduce a stmt. just after the label Li
      that resets goto_Li to false
    }

 /*change unconditional gotos to conditional gotos*/
   for each unconditional goto g in goto_list do
       change g to a conditional goto

 /*eliminate gotos*/
   while not empty(goto_list) do
    { /*select the next goto/label pair*/
        g := select a goto from goto_list
        l := label matching g

        /*force g and l to be directly related*/
        if indirectly_related(g,l) then
          if different_statements(g,l) then
            move g out using outward-movement transf.
            until it becomes directly related to l
          else
            /*diff. branches of same if or switch*/
            move g out using outward-movement transf.
            until it becomes directly related to
            an if or switch containing l

        /*force g and l to be siblings*/
        if directly_related(g,l) then
          if level(g) > level(l) then
            move g out to level(l) using
             outward-movement transformations
          else  /* level(g) < level(l) */
            { if offset(g) > offset(l) then
                lift g to above stmt containing l
                using goto-lifting transformations
              move g in to level(l) using
              inward-movement transf.
            }

        /*g and l are guaranteed to be siblings,
          eliminate g*/
        if offset(g) < offset(l) then
          eliminate g with a conditional
        else
          eliminate g with a do-loop
      }

    /* eliminate labels */
   for each label Li in label_list do
      eliminate Li
 }
```

Figure 13: High-level algorithm for removing all `goto`s

before a loop or `switch` we can avoid introducing a conditional statement before the loop or `switch`.

Another common situation that can be optimized occurs when there is more than one goto associated with a label inside the same `while`, `if`, `switch` or loop statement. If we were to apply the transformations blindly, we would introduce, for each goto, a conditional check at the exit of the `while`, `if`, `switch` or loop. It is clear that when there is more than one `goto` statement to the same label, it would be preferable to insert only one conditional check per label. For ex-

```
{
   stmt_1;
   ...
   if (cond) goto L1;      ⇒     {
L1:stmt_n;                          stmt_1;
}                                   ...
                                 L1:stmt_n;
                                 }
```

(a) goto next to label

```
{                                {
 if (expr) {                      if (expr) {
  stmt_1;                          stmt_1;
  ...                              ...
  if (cond) goto L1;     ⇒         goto_L1=cond;
 }                                }
 ...                              if (goto_L1) goto L1;
L1:stmt_n;                        .....
}                                L1:stmt_n;
                                  }
```

(b) goto at the end of an if block

```
{                                {
 if (cond) goto L1;               goto_L1=cond;
 while (expr) {                   while (goto_L1||expr) {
   stmt_1;                         if (goto_L1) goto L1;
   ...                   ⇒         stmt_1;
L1:stmt_n;                         ...
 }                               L1:stmt_n;
}                                 }
                                 }
```

(c) goto immediately before a loop

```
{                                {
 if (cond) goto L1;               goto_L1=cond;
 switch (i) {                     if(goto_L1)i=1;
   case 1: {                      switch (i) {
     stmt_1;                       case 1: {
     ...                             if (goto_L1) goto L1;
L1:  stmt_n;             ⇒           stmt_1;
   }                                 ...
   default:                      L1: stmt_n;
    .....                          }
 }                                 default:
}                                     ...
                                  }
                                 }
```

(d) goto immediately before a switch

Figure 14: Simple Optimizations

ample, the transformation given in Figure 15 for the case where there are multiple `goto`s to the same label from a `switch`. We implement this optimization by first checking to see if the appropriate conditional has already been inserted, and avoiding duplicating the code if it is already there.

4 Experimental Results

In this section we give some experimental results using our implementation of the algorithm presented in this paper.

4.1 McCAT Compiler

As shown in Figure 16, the McCAT compiler is based on a family of three intermediate representations that range from a high-level abstract representation, FIRST, to a low-level representation, LAST [11]. The design of each intermediate representation is driven by the requirements of the analysis and trans-

```
{ ...                            { ...
  switch(x) {                      switch(x) {
    case 1:                          case 1:
     ...                              ...
     break;                           break;
    case 2:                          case 2:
     ...                              ...
     goto error;                      goto_error=true;
    case 3:                           break;
     ...                 ⇒           case 3:  ...
     break;                           break;
    case 4:                          case 4:  ...
     ...                              goto_error=true;
     goto error;                      break;
  }                                }
  ...                              if (goto_error) goto error;
error:                             ...
}                                error:
                                 }
```

Figure 15: Optimizing multiple `goto`s from the same `switch`

formations that we consider most important at that level. Note that we have implemented our goto-elimination restructuring phase at the SIMPLE level. This is the most convenient place to insert the restructuring since all statements and conditional expressions have been simplified at this point. Note that from the SIMPLE representation we can either dump out a C program (using McCAT as a source-to-source compiler), or continue with the backend phases of McCAT. Also note that all analyses and transformations that are done after the restructuring (goto-elimination) can assume structured programs.

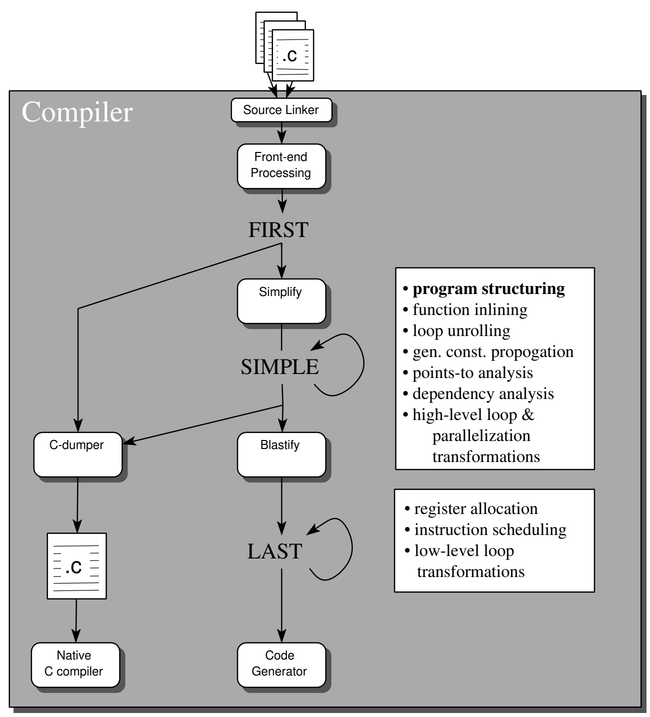


Figure 16: The McCAT Compiler

program	description
asuite	Test for C vectorizing compilers
nrcode2	Test for C vectorizing compilers
compress	File compression
tomcatv	Mesh generation
FSM	Implementation of finite state machine
lex	Output program generated by lex
cq	Tests on a C compiler
par	Program filter
whetstone	Synthetic benchmark
frac	Finds rational aproximation to FP value

Table 1: Benchmark Description

4.2 Benchmarks

In order to test our restructuring method we collected a set of 10 benchmarks that contain `goto` statements. Although in practical terms, our restructurer is required for programs that contain even one `goto`, we wanted to test the effect and complexity of our approach on at least some benchmarks that contained a significant number of `goto` statements. The benchmarks are described in Table 1.[1].

4.3 Experimental Method

In order to measure the effectiveness of our restructuring phase, we performed the following experiment. For each benchmark we used our McCAT compiler as a source-to-source compiler and we produced the following three semantically equivalent versions of the benchmark:

SIMPLE version: This is a C program that is dumped after conversion into our high-level SIMPLE intermediate representation. All `goto` statements remain.

GTE version: This is a C program that is dumped after the SIMPLE representation has been restructured using the transformation rules presented in Section 2. No optimizations of the transformation rules were used.

GTE(opt) version: This is a C program that is dumped after the SIMPLE program has been restructured using the transformation rules presented in Section 2, and the optimizations presented in Section 3.

Note that in the two GTE versions we eliminated the `goto` statements in the reverse order from how they appeared in the source code.

Given the three versions of each program, we then compiled each version using the GNU C `gcc, version 2.4.5`, with the `-O option`, and timed the resulting executables using the UNIX `time` command on a Sparcstation 10. We have reported the user time from these experiments.

[1]The benchmarks we used for these experiments can be obtained by contacting `benadmin@bluebeard.cs.mcgill.ca`. We would also appreciate receiving other benchmarks that contain numerous `goto` statements

4.4 Results and Discussion

Table 2 gives the experimental measurements for each benchmark. In the first section of columns we give the name of the benchmark, the number of `goto`s in the program, and the number of lines of source code. To provide a fair comparison for the number of lines of source code we ran a script that strips comments, and formats the programs into a standard form.

The second section of columns gives the execution times for each version of the benchmark (times collected as described in the previous section).

The third section of columns gives a count of the total number of transformations applied, and the number of statements inserted for the GTE and GTE(opt) versions of the programs. Note that the number of extra statements in the GTE versions is really the number of statements inserted minus the number of `goto` statements eliminated. To count the statements inserted in the GTE versions we counted all new statements including the added `if` and `do-while` statements, the initialization of the logical variable with the goto condition and the re-initialization of the logical variables.

First, let us consider the effect of restructuring on execution time. As expected, restructuring programs with very few `goto` statements have very little impact on execution time. For example, this is true for *nrcode2* and *cq* with only two and one `goto` respectively. This is an important observation since many programs have only a few `goto` statements, and our method allows us to handle them with a structure-based compiler at low cost.

On the other extreme, is the *FSM* benchmark which is an irreducible loop, has many nested `goto`s, and the ratio of `goto`s to total statements is very high. Furthermore, most of the execution time is spent in the part of the program that implements the finite state machine. Thus, we see that there is a significant performance impact, with even the optimized GTE version executing significantly slower. This result would indicate that we should explore some further optimizations of the restructurer for such `goto`-intensive programs. In experimenting with various orderings of the `goto` statements, we observed that the order of elimination is important for such programs, and this is one source of optimization that we will consider.

For *asuite*, although the ratio of `goto`s to the number of statements is also high, we observed that the transformations applied are very simple (almost all the `goto`s are siblings of their labels, or exit from a `for` loop).

name of benchmark	# of gotos	# of stats.	time for SIMPLE	time for GTE	time for GTE(opt)	GTE(opt)/ SIMPLE	transf GTE	new stat GTE	transf GTE(opt)	new stat GTE(opt)
asuite	21	91	4.2	4.5	4.3	1.02	34	81	34	74
nrcode2	2	132	3.2	3.3	3.2	1.00	5	9	5	6
compress	18	1052	2.3	2.6	2.5	1.09	27	42	24	31
tomcatv	7	317	1.1	1.2	1.1	1.00	14	29	14	21
FSM	12	53	2.5	3.3	3.0	1.20	23	45	19	34
lex	5	1670	0.42	0.50	0.46	1.10	22	16	16	11
cq	1	5760	3.0	3.2	3.1	1.03	1	3	1	3
par	59	1665	0.57	0.65	0.62	1.09	189	266	187	170
whetstone	31	433	10.8	11.3	11.1	1.03	63	128	63	83
frac	6	56	0.16	0.21	0.17	1.06	7	18	7	17

Table 2: Experimental Measurements

Next, consider the expense of the restructurer as measured by the number of transformation applied, and the number of new statements introduced. First, we note that we apply around 2 to 3 transformations per `goto` eliminated. This means that we apply 1 or 2 movement transformations per `goto`. Also, we see that we introduce about 3 statements per `goto` (giving a net increase of 2 statements per `goto`). These results are consistent with the results of a study done by Ballance and Maccabe [5], that indicated that only 2.9% of 119,000 functions examined use `goto`s. Of those `goto`s, 68% can be characterized as simple `goto`s: one target label per function, with one or more associated `goto`s, where the `goto` and `label` are siblings or the `goto` is an exit from a control structure.

We can summarize by stating that our results show that applying a small number of simple transformations eliminates all `goto` statements, and on most benchmarks the effect on execution speed is minimal. Thus, for the vast majority of programs, we can exploit structured representations for designing compilers while paying only a minimal penalty due to restructuring.

5 Related Work

One of the first approaches to restructuring was given by Bohm and Jacopini [6]. Their restructuring method was done in the context of normalizing flow graphs (where the flow graph represented mappings of a set onto itself). This result is mostly of historical and theoretical interest and does not give a complete algorithm, but rather presents a set of pattern matching rules and transformations.

There have been several approaches to restructuring program flowgraphs. Peterson *et al.* present a proof that every flowgraph can be transformed into an equivalent well-formed flowchart(loops and conditionals are properly nested and can only entered at the beginning) [14]. They present a graph algorithm to do such a transformation using a technique of node-splitting and they proved the transformation was correct. William and Osher also use node-splitting, but they present the problem as recognizing five basic unstructured sub-graphs, and show how to replace these sub-graphs with equivalent structured forms [17, 16]. Ashcroft and Manna tackled the problem of restructuring by presenting two algorithms for converting program schemas into while schemas. Rather than using node-splitting they use extra logical variables to achieve these transformations [10].

All of the previous methods were intended to restructure all flow charts. However, there have also been approaches suggested that are used to restructure programs in order to expose the natural structure of the program, leaving some `goto`s unstructured. The first such method was given by Baker as a method for restructuring Fortran programs [4] in order to make them more understandable. Since her goal was understandable Fortran programs, she only restructures in situations where there is a clear use of a structured construct and leaves some `gotos` in the program. This is of historical interest, but since she leaves some `goto`s in the program, her method is not applicable to the complete restructuring of programs for the purposes of compilation. More recently, Cifuentes has presented an algorithm for restructuring in the context of decompilation [7]. This work is similar in spirit to Baker's problem in that she only structures the parts of the program that correspond naturally to structured control constructs.

More relevant to our work are the structuring methods proposed by Allen et al for vectorizing compilers [2], and the work by Ammarguellat for parallelizing compilers [3].

Allen et al present the *IF conversion* method that converts control dependences into data dependences by eliminating `goto` statements and introducing logical variables to control the execution of the statements. The goal of this work is to vectorize statements in loops which contain conditional transfers in Fortran programs. But although the goal is not the same as ours, this method can also be applied for restructuring, and in fact has similar characteristics to ours. Their method can be divided in three different steps: First they categorize the branches into three types: *exit branches* (exits from a loop), *forward branches* and *backward branches*. Then, according to this branch classification , *IF conversion* uses two different transformations to eliminate the branches in the programs: *branch relocation* and *branch removal*. Branch relocation moves branches out of loops until the branch and its label are nested in the same number of `do loops`. This is accomplished by introducing guard expressions to enforce conditional execution of statements. Branch removal takes place after removing all the forward branches. They do not eliminate backward branches.

Their method is similar to ours in that both methods consist of step-by-step transformations applied to a structured intermediate representations of the program, that result after each transformation in a more structured code. The idea of the *branch relocation* and *branch removal* are somewhat similar to our *goto-movement* and *goto-elimination*. We both use logical variables to guard the execution of statements. Differences between the methods include the fact that we restructure C programs (and thus treating `break`, `continue`, and `switch` statements) rather than Fortran programs. Furthermore, we are interested in removing all gotos, not just those associated with forward branches. Another difference is in the way in which we introduce guards into the code. Since they were interested in vectorization they introduced a new conditional for each action statement whereas in our method it is preferable to introduce one conditional for each block of statements. A potential advantage of our approach is that we only have to make one pass through the program collecting information about gotos and labels, and then we can directly modify the intermediate representation of the program. Their approach requires several passes through the program for the different stages of branch categorization, branch relocation and branch removal.

The method presented by Ammarguellat, which she calls control-flow normalization, is the closest work in terms of the goals of restructuring. That is, we both wish to fully restructure programs in order to facilitate program transformations, program analysis and automatic parallelization. However, the intermediate representations that we restructure are quite different. We are restructuring a high-level representation of C programs that directly supports `break` and `continue`, while Ammarguellat restructures a lisp-like intermediate representation and she requires that all loops have a single exit.

Ammarguellat's approach to the problem is very different from ours. She converts the program into a system of simultaneous equations whose unknowns represent the continuations associated with the programs labels. By transformations applied to the system of equations: *precalculation, if distribution, factorization , derecursivation* and *substitution* and *elimination* , she solves this system of continuation equations. The quality of the normalizing form of the program in terms of code duplication, code size and running time of resolution process depends on the order in which unknowns are eliminated. To study this order she has to consider the control flow of the program, eliminate the back and cross edges of the graph and sort the resulting graph in a topological order.

In Figure 17(a) we present an example of an irreducible loop. We can compare the result of Ammarguellat's control-flow normalization (Figure 17(b)) and our goto-elimination (Figure 17(c)). As illustrated by this example, the results are similar in that we both create new logical variables to store conditions and to guard the execution of the statements and we both create cycles of control flow when there is an implicit cycle. However, Ammarguellat replicates code in the case of irreducible loops and in the case she does not study the best order of the unknowns. In the cases of backward branches that do not imply cycles, we introduce a loop where Ammarguellat does not. However, this loop will not execute more times than the original program will, and it does not imply an increase in the execution time of the program.

Another distinction is that we do not require single-exit loops because our compiler analysis framework easily handles `continue` and `break` statements. However, we can easily modify our approach to force single-exit loops if this is required. It appears to us that our method is easier to explain and more straight-forward to implement as we only need a set of simple transformations, and we do not require the collection or solution of equations.

6 Conclusions and Future Work

In this paper we have presented a structured approach to eliminating all `goto` statements in C programs. The goal of this transformation is to provide a structured and compositional intermediate representation that is amenable to structured approaches to analysis, optimization and parallelization.

The method is straight forward and can be easily implemented directly on an abstract tree representation of C programs. The approach is built upon a set of *goto-elimination* and *goto-movement* transformations. Each `goto` statement is removed by using the *goto-movement* transformations to move the `goto` to the same statement sequence and then applying the appropriate *goto-elimination* transformation.

We completely implemented our method on the SIMPLE intermediate representation of the McCAT parallelizing/optimizing compiler, and we have presented experimental measurements for 10 benchmark programs using this implementation. It appears

```
{
    if (x) goto L2;
L1: stmt_1;
L2: stmt_2;
    if (y) goto L1;
 }
```

(a) an irreducible loop

```
{ pred_50=x;
  if ( pred_50 ) {
     stmt_2;
     pred_52=y;
  }
  if (!pred2_50 && (!pred_52)) {
    do {
      stmt_1;
      stmt_2;
      pred_52=y;
    } while ( pred_52==1)
  }
}
```

(b) control flow normalization

```
{ goto_L2=x;
  do {
    if (!goto_L2) {
      goto_L1=0;
      stmt_1;
    }
   goto_L2=0;
   stmt_2;
   goto_L1=y;
  } while(goto_L1)
}
```

(c) our method

Figure 17: A comparison of methods for an irreducible loop

that most C programs use `goto` statements relatively sparsely and on such programs the restructured programs have similar execution speeds as the original programs. Thus, for most programs, the restructuring does not have a performance penalty, while at the same time allowing us to use structured analysis and transformations in the latter phases of the compiler. For programs that are dense in goto statements (i.e. C programs produced by scanner-generator tools like `lex`), there is some performance penalty, and it may be worth studying further optimizations for the restructuring methods. For example, we may want to study the best order of eliminating `goto`s and look at the elimination of redundant conditional checks and initializations.

We feel that a major advantage of our approach is that restructuring method itself is straight-forward to integrate into any C compiler using a structured intermediate representation. Furthermore, as shown by our experimental results, the approach is very efficient, applying only a small number of simple transformations per `goto` statement. Finally, it has been our experience that the presence of a restructuring phase that can always eliminate `goto`s allows us to develop more efficient and simpler analyses and transformations in the remainder of the compiler.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley Pub. Co., corrected edition, 1988.

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Conf. Rec. of the POPL-10*, pages 177–189, Austin, Texas, Jan. 1983.

[3] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. on Software Eng.*, 18(3):237–250, 1992.

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

[5] Robert A. Balance and Arthur B. Maccabe. Program dependence graphs for the rest of us. Technical report, The University of New Mexico, Oct. 1992.

[6] C. Bohm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. of the ACM*, 9(5):366–371, May 1966.

[7] C. Cifuentes. A structuring algorithm for decompilation. In *XIX Conferencia Latinoamericana de Informática*, pages 267–276, Buenos Aires, Argentina, 2-6 August 1993. Centro Latinoamericano de Estudios en Informática.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[9] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11:147–148, March 1968.

[10] E.Ashcroft and Z. Manna. Translating programs schemas to while-shemas. *SIAM J. Comput*, 4(2):125–146, 1975.

[11] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Conf. Rec. of the 5th Work. on Languages and Compilers for Parallel Computing*, pages 261–275, New Haven, Conn., Aug. 1992. Dept. of Comp. Sci., Yale U. Also available as ACAPS 46, Sch. of Comp. Sci., , Montréal, Qué.

[12] L. J. Hendren, G. R. Gao, and V. C. Sreedhar. ALPHA: A dependence-based intermediate representation for an optimizing/parallelizing C compiler. ACAPS Tech. Memo 49, Sch. of Comp. Sci., McGill U., Montréal, Qué., Nov 1992.

[13] D. E. Knuth. Structured programming with go to statements. *Computing Surveys*, pages 261–302, Dec 1974.

[14] W.W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat and exit statements. *Comm. of the ACM*, 16(8):503–512, 1973.

[15] P. Wegner. Programming languages - the first 25 years. *IEEE Transactions on Computers*, pages 1207–1225, Dec 1976.

[16] M.H. Williams. Generating structured flow diagrams: The nature of unstructuredness. *Comput. J*, 20(1):45–50, 1977.

[17] M.H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *Comput. J*, 21(2):161–167, 1978.