A GOTO-ELIMINATION METHOD AND ITS IMPLEMENTATION FOR THE McCAT C COMPILER

by
Ana Maria Erosa

School of Computer Science
McGill University, Montreal

May 1995

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

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 thesis 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.

Our McCAT (McGill Compiler Architecture Testbed) optimizing/parallelizing C compiler is based on a compositional representation of the program, and hence does not support unrestricted use of `goto`s directly. We have implemented the method within the framework of the McCAT compiler. We present some experimental results and study the cost of structuring. The results show that applying a small number of simple transformations eliminates all the `goto` statements, usually with a minimal effect on the execution speed. Thus, we can exploit structured representations for desigining compilers, while paying a minimal penalty due to structuring.

Résumé

En créant des compilateurs optimisants et parallélisants, il est souvent plus simple et plus efficace de travailler avec des programmes possèdant un flot de contrôle structuré. Bien que la plupart des programmeurs programment naturellement d'une façon structurée, il subsiste plusieurs programmes et exemples importants contenant un nombre quelconque d'instructions `goto`, ceci résultant en des programmes entiers non-sructurés. De tels programmes ne peuvent être utilisés avec des compilateurs construits pour analyser et transformer des programmes structurés.

Dans cette thèse nous présentons un algorithme utilisable directement pour structurer les programmes en langage C en éliminant toutes les instructions `goto`. La méthode utilise directement un arbre de représentation abstraite de haut niveau des programmes, et pourrait facilement être intégrée à n'importe quel compilateur utilisant une représentation intermédiare basée sur un arbre de syntaxe abstraite. L'algorithme sous sa forme actuelle fonctionne en éliminant chaque `goto` en appliquant d'abord une séquence de transformations de *goto-movements* suivie par la transformation *goto-elimination* approprieé.

Notre compilateur McCAT (McGill Compiler Architecture Testbed) optimisant et parallélisant en langage C est basé sur une représentation compositionnelle du programme, et donc ne peut soutenir l'utilisation directe des `goto`s. Nous avons implémenté la méthode pour le compilateur McCAT. Nous présentons quelques résultats expérimentaux et étudions le coût de structuration. Les résultats montrent que d'appliquer un nombre réduit de transformations simples élimine toutes les instructions `goto`, avec un effet minimal sur la vitesse d'exécution dans la plupart des cas. Donc, nous pouvons exploiter des représentation structurées pour créer des compilateurs, tout en ne payant qu'un prix minimal pour la structuration.

To my mother, Myriam Etchebehere, and to my aunt, Graciela Etchebehere

Acknowledgments

First, I want to thanks my advisor Laurie Hendren for all the support she gave me during the course of my studies at McGill. From the beginning, she was helpful and encouraging. She is an excellent professor and organizer. I enjoyed doing my master's studies under her supervision.

I am thankful to all the members of the McCAT group, and especially to Bhama Shridharan, V.C Sreedhar, Maryam Emami, Chris Donawa, Justiani, Rakesh Ghiya and Luis Lozano. I appreciate Bhama's help in my first steps in understanding SIMPLE, Luis' help in those moments when I was stressed out, and Rakesh's and Chris' helpful advice.

But, for sure, the most important remark I can think of, is the nice atmosphere that we all contribute to create in the ACAPS lab, and the good friendship that evolved there and which I will always remember.

I wish to acknowledge the excellent work of the administrative staff of the Computer Science Department, specially of Lorraine Harper, Franca Cianci, and Lise Minogue. They are a perfect example of efficiency and service.

I would like to give special thanks to In.Co, the Institute of Computer Science of the University of Uruguay and CIDA (Canadian International Development Agency) for providing me the oportunity to pursue my master's studies in Canada.

Last, but not least, I would like to thank my Uruguayan friends in Montreal Graciela Piñeyro, Javier Pintos , Carlos Miranda and Juanjo Perez, for making me feel more at home, and my family and friends in Montevideo with whom I was always in touch in spite of the distance. I would like to especially emphasize the constant support and affection given by my brother Ramon and my friend Ines via e-mail, and the long phone conversations from Minneapolis with my brother Andres.

Contents

List of Figures

Chapter 1

Introduction and Motivation

The great advances in high-performance architectures demand a simultaneous development of sophisticated compilation techniques [HP90]. Thus, the design of optimizing and parallelizing compilers is a critical issue. These compilers perform optimizing transformation based on the information collected by various program analyses. From the pragmatic point of view there are many reasons why structured programs (i.e. programs without `goto`s) are simpler to handle in such compilers. One important consequence is that C programs without `goto`s are compositional, and therefore structured analyses techniques can be used to compute data flow information. For example, one can apply the efficient techniques available for structured data flow graphs [ASU88], or one can use 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 [CFR+91]; the structured transformations to ALPHA [HGS92], a family of intermediate representations designed to facilitate the development of specific analysis and transformations; the elegant formal system proposed by Hoare [Hoa69] to prove the correctness of structured and compositional programs; and the efficient construction of Program Dependence Graphs for structured programs [BM92]. 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.

In this thesis we are concerned about the automatic structuring of programs, by eliminating `goto` statements, in order to facilitate the construction of analyses and transformations required for optimizing and parallelizing C compilers.

1.1 Goto elimination and the McCAT compiler

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 of `goto`s from a software engineering or program understandability point of view [Dij68, Knu74, Weg76]. This discussion has led to the relatively infrequent use of `goto`s in typical C programs [BM92]. 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 fact, if we study some of the well known benchmarks such as the SPEC benchmarks, we find that many important benchmarks use some `goto`s. Thus, if a compiler is restricted to programs without `goto`s, it is a significant handicap.

The McGill Compiler Architecture Testbed(McCAT) [HDE+92] was designed to test different compilation techniques on different architecture testbeds. Two main objectives were pursued: (1) build a compiler that supports both high level and intermediate representations that facilitate analyses and transformations and related low-level transformations that are suitable for code generation; and (2) build architecture simulator tools to process the output of the compiler to produce different performance results. The source language chosen for our compiler was C. This decision was made taking into consideration the fact that the C language is widely used and powerful, as it supports a variety of features.

One of the unique features of our compiler is that it is based on a family of intermediate representations. Three structured intermediate representations are built and each of them fulfil a specific role in the compiler. The first, called FIRST, is a high-level abstract representation that accurately captures the original program. The main purpose of FIRST is to cleanly separate the front-end processing of parsing and type checking from the back-end phase of analysis, transformations and code generation. Then, a series of transformations are performed on FIRST, to create SIMPLE, an AST suitable for high-level analyses like points-to (alias) [Ema93, EGH94] and dependence analysis [Jus94, JH94]. Finally LAST, a low-level AST is obtained from SIMPLE, on which low-level optimizations such as register allocation and instruction scheduling [Don94] take place. This representation can be used to generate code for a variety of high-performance architectures. It should be noted, that each intermediate representation is related to the next so that one can use the results of analyses performed at higher-level representations at lower-level representations. Figure 1.1 presents an overview of the McCAT compiler and its principal components.

Thus, the McCAT compiler is based on a compositional representation of the program. It does not support unrestricted use of `goto`s directly. In order to have no restriction on the benchmarks used to test our compiler, a structuring phase that eliminates `goto`s automatically is required. In this thesis we present a goto-elimination method and its implementation for the McCAT compiler. Once all benchmarks pass through this structuring phase, structured intermediate representations are created, and all further analyses and transformations have to deal only with structured control flow.

1.2 Thesis Contributions

This thesis concentrates on the design of a general algorithm for elimintating any number of arbitrary `goto` statements from a C program. Our approach to eliminating `goto`s is based on a set of simple transformations that operate on SIMPLE, the second high-level structured intermediate representation built in McCAT. These transformations come in two categories: *goto-movements* and *goto-eliminations*. 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 algorithm proceeds by eliminating one `goto` at a time, applying a sequence of *goto-movement* transformations followed by the *goto-elimination* transformation until the `goto` is eliminated. It is a straight-forward algorithm that works directly on a

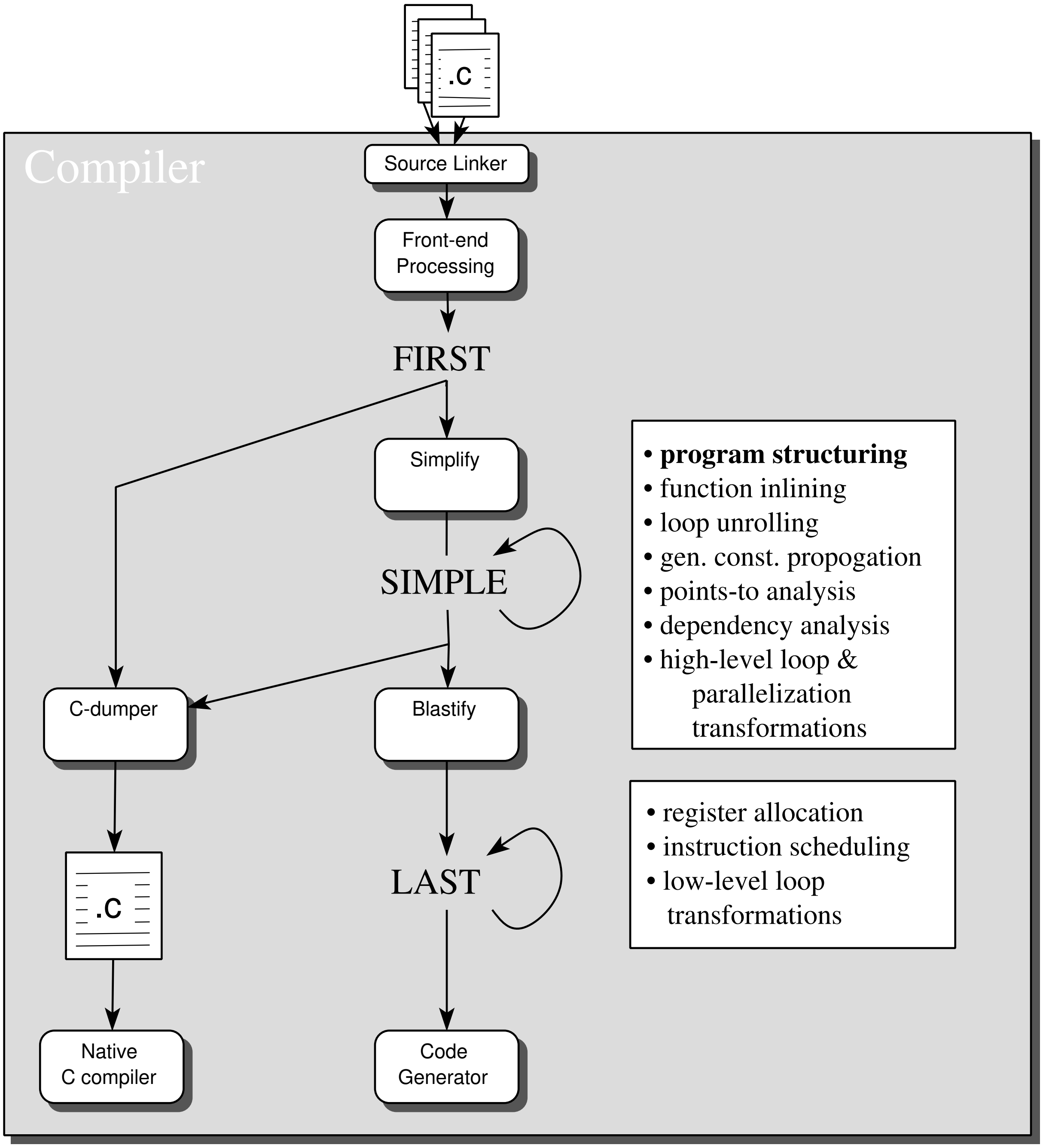


Figure 1.1: The McCAT Compiler

high-level abstract syntax tree representation of the program and can easily be integrated in any compiler that uses an AST-based intermediate representation. We have implemented this method for the McCAT compiler.

The main contributions of this thesis include:

- The design of a straight-forward and general method to eliminate `goto` statements from a C program.
- The implementation of this method for the McCAT compiler.
- The design and implementation of further optimizations to this method.
- The presentation of experimental results and discussion of the cost of structuring that show that this method is both efficient and effective.

1.3 Thesis Organization

The remainder of this thesis is structured as follows. In Chapter 2 we present the goto-elimination and goto-movement transformations. First we show how they can be applied to remove a single `goto` statement from a C program and second we present a high-level algorithm for eliminating *all* `goto`s from a C program, thus producing a semantically equivalent structured C program. Third we show how some optimizations to our method can improve the resulting code. In Chapter 3 we give a brief description of SIMPLE and in Chapter 4 we provide an overview of the important aspects of the implementation using SIMPLE. We have completely implemented the method and in Chapter 5 we give experimental results for both the unoptimized and optimized methods and discuss the cost of structuring. Finally, in Chapter 6 we compare our method with related methods, and in Chapter 7 we conclude and discuss further work.

Chapter 2

Description of the Goto-Elimination Method

In this chapter the description of the goto-elimination method is presented. We first explain the process of eliminating one arbitrary `goto` from a C program and then any number of `goto`s. Next, we study some optimizations that can be applied to the method.

2.1 Eliminating an arbitrary goto statement from a C program

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. Even though these statements represent a form of control-flow similar to `goto`s, they can be easily handled by our structured data flow analysis methods [Sri92, Ema93]. `Break` and `continue` statements do not change the control-flow of the programs outside the scope of the closest enclosing loop structure. In this sense the program remains compositional, the meaning of the program structure is given by the meaning of its components. Thus, for our purposes there is no benefit in eliminating the `break` and `continue` statements. However, we could easily modify our method to eliminate them if required.

Furthermore, we assume that each labelled statement of the form `Li: stmt` is really represented as a sequence of two statements, the empty statment `Li:;` and the actual statement `stmt`. Thus, when we refer to a *label statement*, we are referring to the empty statement containing the label.

2.1.1 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 2.1, 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.[1]

Goto statement is after label statement: if the `goto` statement is after the label statement, then the `goto` statement is eliminated by embedding the statements between the label and the `goto` in a `do-while` loop. The example program in Figure 2.2 illustrates this case.

[1]Note that we present each goto transformation as a rewriting of a general statement sequence. Thus rules like those in Figure 2.1 represent a general pattern for the transformation with each `stmt_i` standing for any SIMPLE C statement including assignment statements, procedure calls, and compositional statements such as conditionals and loops.

```
      {                                        {
        .......                                  .......
        stmt_i;                                  stmt_i;
        if ( cond ) goto L_i;                    if ( !cond )
        stmt_j;                   =>                {
        .......                                       stmt_j;
L_i:    stmt_k;                                       .......
        .......                                     }
      }                                 L_i:     stmt_k;
                                                 .......
                                               }
```

Figure 2.1: Eliminating a `goto` with a conditional

```
      {                                        {
        .......                                  .......
        stmt_i;                                  stmt_i;
L_i:    stmt_j;                                  do
        .......                                    {
        stmt_k;                   =>    L_i:         stmt_j;
        if ( cond ) goto L_i;                        .......
        .......                                      stmt_k;
      }                                            }
                                                 while ( cond );
                                                 .......
                                               }
```

Figure 2.2: Eliminating a `goto` with a loop

These two goto-elimination transformations are obvious, and it is unlikely that a programmer would use a `goto` in these situations where a conditional or loop is a much more reasonable construct. 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-elimination transformations. In fact, these transformations are just the inverse of standard code generation strategies for conditionals and loops.

2.1.2 Goto-movement Transformations

In order to categorize the goto-movement transformations precisely, we introduce notions of *offset*, *level*, *sibling statements*, *directly-related statements* and *indirectly-related statements*.

Definition 2.1.1 *The* offset *of a* `goto` *or label statement is* n *if, relative to the beginning of the program, the statement is the* n*th statement which is either a* `goto` *or a label statement. Offsets may be computed by traversing the source program from top to bottom and incrementing the offset counter each time a* `goto` *or label statement is encountered.*[2]

Definition 2.1.2 *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 2.1.3 *A label statement and a* `goto` *statement are* siblings *if there exists some statement sequence,* `stmt_1; ...stmt_i; ...stmt_j; ...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 2.1.4 *A label statement and a* `goto` *statement are* directly-related *if there exists some statement sequence,* `stmt_1; ...stmt_i; ...stmt_j; ...stmt_n;`, *such that either the label or the* `goto` *statement corresponds to some* `stmt_i` *and the matching* `goto` *or label statement is nested inside some* `stmt_j (stmt_i <> stmt_j)` *in the statement sequence.*

[2]Offsets are used to determine if a label statement occurs before or after the matching `goto` statement.

Definition 2.1.5 *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 2.1 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 2.2 is applied when the offset of the `goto` statement is greater than the offset of the target label statement.

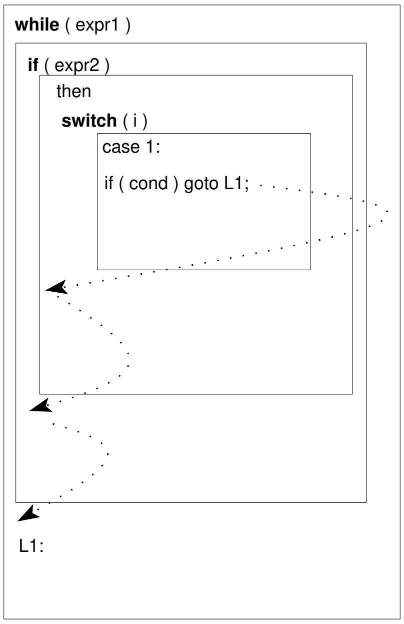
We can now restate our overall strategy as follows. Given any `goto`-label pair, we can eliminate the `goto` by first moving the `goto` until it becomes a sibling of the label, and then applying the appropriate goto-elimination transformation. Figure 2.3 illustrates the four situations that may occur.

Figure 2.3(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 2.3(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.

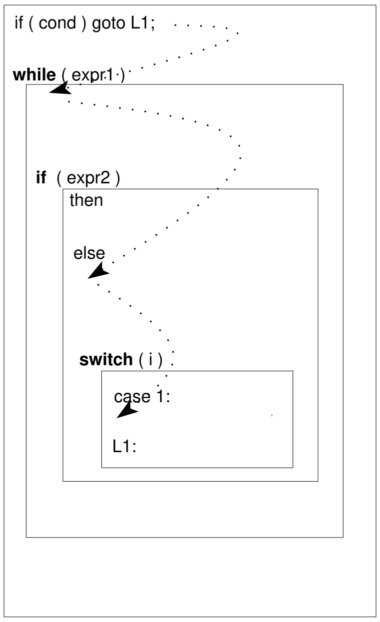
Figures 2.3(c) and 2.3(d) illustrate more complicated situations where the `goto` and label are indirectly-related. When the label and `goto` are in entirely different statements (Figure 2.3(c)), the `goto` is first moved using outward-movements until it becomes directly-related to the label, and then 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 2.3(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.

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. The next paragraphs present these transformations for each of the statements that need to be considered: loops (i.e. `for, do` and `while`), `if` and `switch` statements.

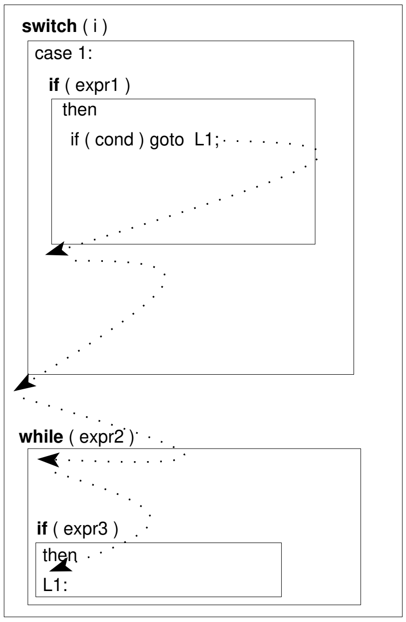
Outward-movement Transformations



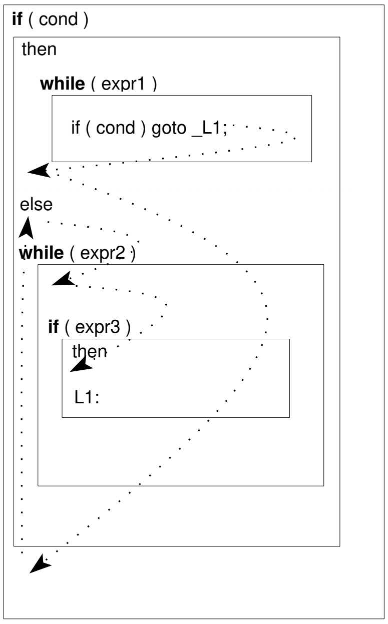
(a) Directly-related
($level(goto) > level(label)$)



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



(c) Indirectly-related
(*different statements*)



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

Figure 2.3: The four situations for goto/label relationships

The outward-movement transformations are very straight-forward. There are 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 does not make use of the `break` statement, if so desired. The complete transformation is illustrated in Figure 2.4. Note that a new variable is introduced to store the value of the conditional at the point at which the `goto` is encountered. This value is then reused in the `goto` statement that is introduced at the exit of the `switch`/loop. To be safe, to preserve the semantic correctness of the program, the `goto` variable must be reinitialized to false at the point of the label.

- **Moving a `goto` out of an `if` statement:**

  In this case the `break` statement cannot be used. Instead a new conditional is introduced as illustrated in Figure 2.5.

Inward-movement Transformations

In the previous subsection we presented the relatively simple outward-movement transformations. The inward-movement transformations are slightly more complicated. Firstly, we cannot 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 2.1.2) 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 2.6. 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 reinitialization to false of the `goto`

```
    {
      .......
      switch(i):
      {
        case 1:
           .......
           stmt_i;
           if ( cond ) goto L1;
           stmt_j;
           .......
           stmt_k;
           break;
        case 2:
           .......
        default:
           .......
      }
      .......
L1:   stmt_l;
      .......
    }
```

$\Rightarrow$

```
    {
      .......
      switch(i):
      {
        case 1:
           .......
           stmt_i;
           goto_L1=cond;
           if ( goto_L1 ) break;
           stmt_j;
           .......
           stmt_k;
           break;
         case 2:
           .......
         default:
           .......
      }
      if ( goto_L1 ) goto L1;
      .......
L1:   goto_L1=0;
      stmt_l;
      .......
    }
```

Figure 2.4: Moving a `goto` out of a `switch`

```
    {
      .......
      if ( expr )
         {
           .......
           stmt_i;
           if ( cond ) goto L1;
           .......
           stmt_j;
         }
      .......
L1:   stmt_k;
      .......
    }
```

$\Longrightarrow$

```
    {
      .......
      if ( expr )
         {
           .......
           stmt_i;
           goto_L1=cond;
           if ( !goto_L1 )
              {
                .......
                stmt_j;
              }
         }
      if ( goto_L1 ) goto L1;
      .......
L1:   goto_L1=0;
      stmt_k;
      .......
    }
```

Figure 2.5: Moving a `goto` out of an `if`

```
      {
        .......
        if ( cond ) goto L1;
        stmt_i;
        .......
        stmt_j;
        while ( expr )
          {
            .......
            stmt_k;
L1:         .......
            stmt_n;
          }
        .......
      }
```

$\Longrightarrow$

```
      {
        .......
        goto_L1=cond;
        if ( !goto_L1 )
          {
           stmt_i;
           .......
           stmt_j;
          }
        while ( goto_L1 || expr )
          { if ( goto_L1 ) goto L1;
           .......
           stmt_k;
L1:        goto_L1 = 0;
           .......
           stmt_n;
          }
        .......
      }
```

Figure 2.6: Moving a `goto` into a `loop`

variable at the point of the label preserves the correct behavior of the loop in succeeding iterations (i.e. force evaluation of the loop expression).

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 would be 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 2.7 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

```
     {
       .......
       if ( cond ) goto L1;
       stmt_i;
       .......
       stmt_j;
       if ( expr )
         { .......
           stmt_k;
           .......
         }
       else
         { stmt_l;
L1:        stmt_n;
           .......
         }
       .......
     }
```

⇒

```
     {
       .......
       goto_L1=cond;
       if ( !goto_L1 )
          { stmt_i;
            .......
            stmt_j;
          }
       if ( !goto_L1 && expr )
          { .......
            stmt_k;
            .......
          }
       else
          { if ( goto_L1) goto L1;
            stmt_l;
L1:         goto_L1=0;
            stmt_n;
            .......
          }
        .......
     }
```

Figure 2.7: Moving a `goto` into an `if`

introduced that initializes the new variable to the constant expression of the case in question when the condition of the `goto` is true and to the switch expression when the condition of the `goto` is false. If the label occurs in the `default` statement, the new variable is set to a default value. Figure 2.8 illustrates this case.

```
 {
   .......
   if ( cond ) goto L1;
   stmt_i;
   .......
   stmt_j;
   switch (i)
    {
      case 1:
       {
         stmt_k
         .......
L1:      stmt_l;
         .......
       }
      default:
        .......
    }
    .......
 }
```

⇒

```
 {
   .......
   goto_L1=cond;
   if ( !goto_L1 )
      { stmt_i;
        .......
        stmt_j;
        t_switch=i;
      }
   else t_switch = 1;
   switch (t_switch)
    {
      case 1:
       { if (goto_L1) goto L1;
         stmt_k;
         .......
L1:      goto_L1=0;
         stmt_l;
         .......
       }
      default:
        .......
    }
   .......
 }
```

Figure 2.8: Moving a `goto` into a `switch`

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.

Figure 2.9 illustrates the goto-lifting transformation. Let `stmt_label` be the statement that contains label `L1`, and let the matching `goto` statement be below `stmt_label` in the statement sequence. 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`.

```
{
  .......
  stmt_i;
  .......
  stmt_j;
  .......
  stmt_label; /* contains L1 */
  .......
  if ( cond ) goto L1;
  .......
  stmt_k;
  .......
}
```

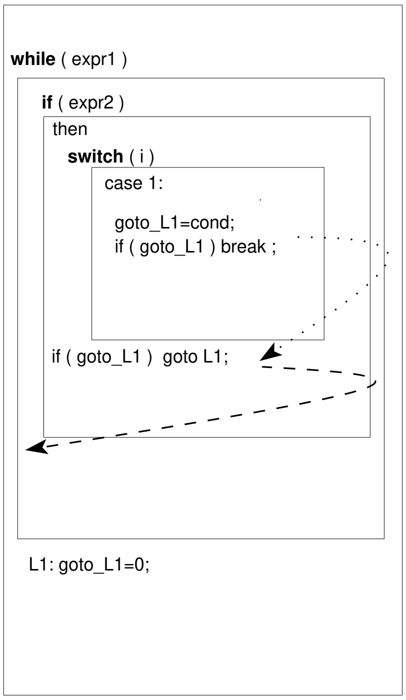
$\Longrightarrow$

```
{
  int goto_L1 = 0;
  .......
  stmt_i;
  .......
  stmt_j;
  .......
  do
    { if (goto_L1) goto L1;
      stmt_label; /* contains L1 */
      .......
      goto_L1 = cond;
    }
  while (goto_L1);
  .......
  stmt_k;
  .......
}
```

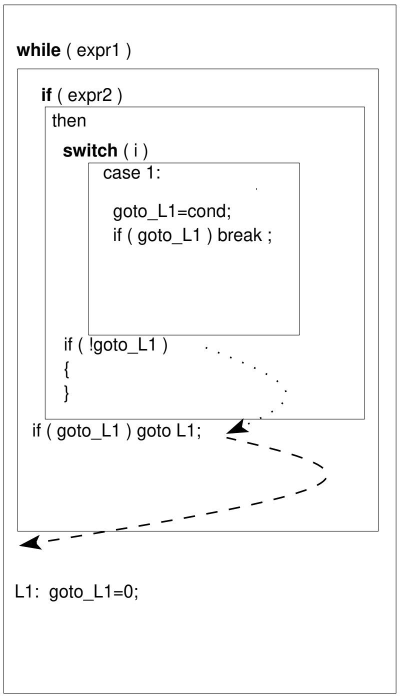
Figure 2.9: Lifting a `goto` above the statement containing the label

2.1.3 Examples of Inward and Outward Transformations

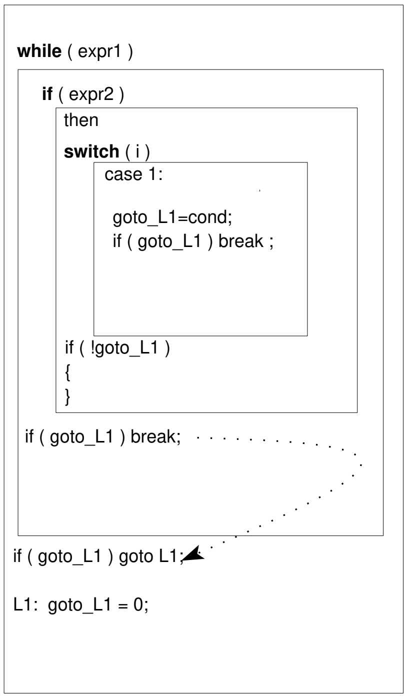
Figure 2.10 illustrates a series of outward-movement transformations followed by a goto-elimination transformation, performed to eliminate the `goto` in Figure 2.3(a). Figure 2.11 illustrates a series of inward-movement transformations followed by a goto-elimination transformation, performed to eliminate the `goto` in Figure 2.3(b). Figure 2.12 illustrates a series of outward-movement transformations, followed by a goto-lifting transformation, a series of inward-movement transformations, and a goto-elimination transformation, performed to eliminate the `goto` in Figure 2.3(d). Note that the dotted arrows indicate the movement just applied, while the dashed arrows indicate the next movement.



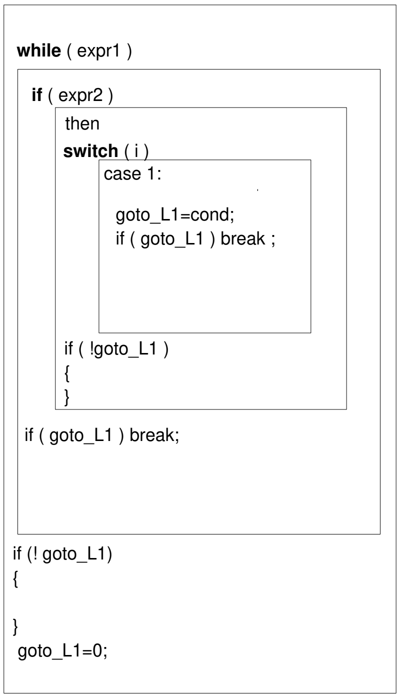
(a) outward-movement from `switch`



(b) outward-movement from `if`

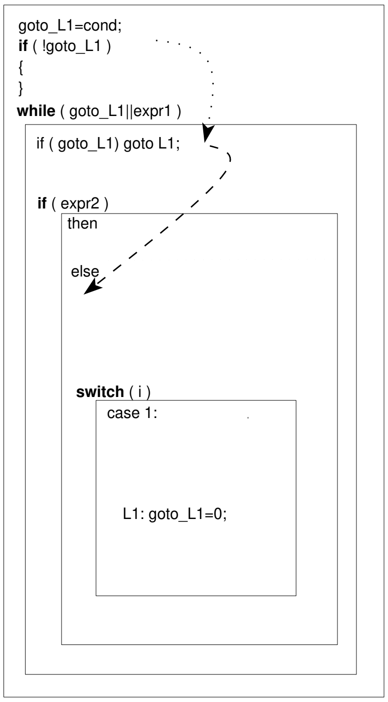


(c) outward-movement from `while`

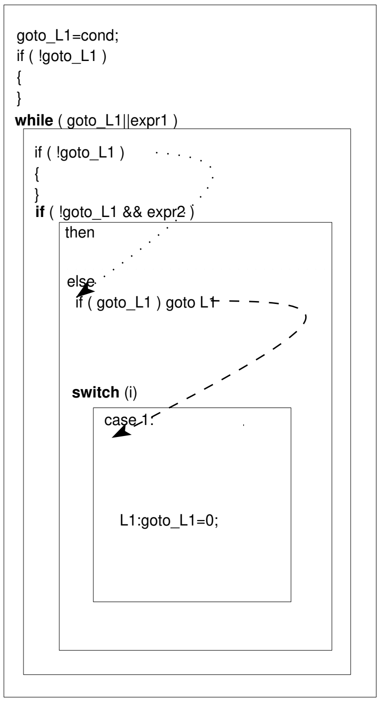


(d) application of goto-elimination

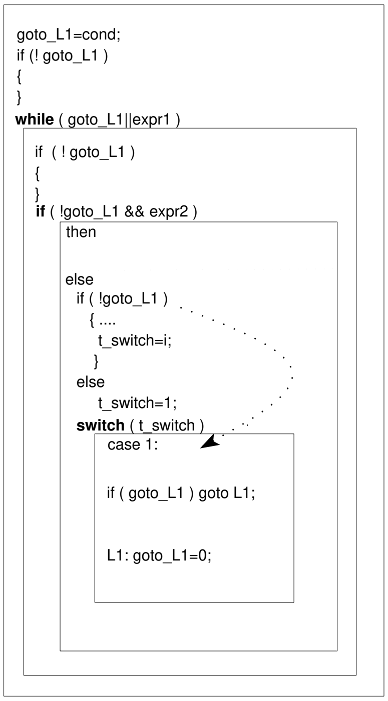
Figure 2.10: Outward-movements followed by goto-elimination



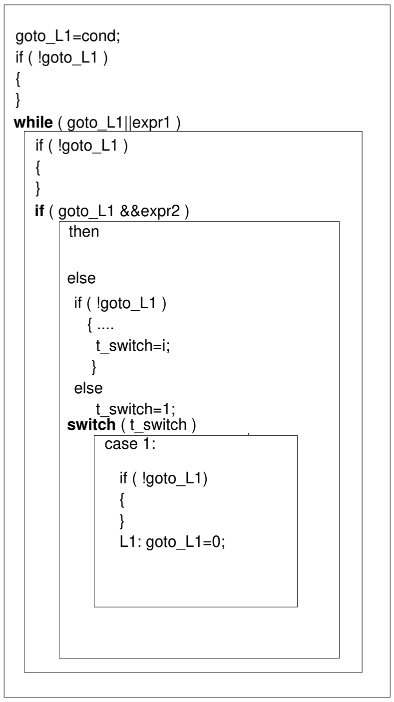
(a) inward-movement into a `while`



(b) inward-movement into an `if`

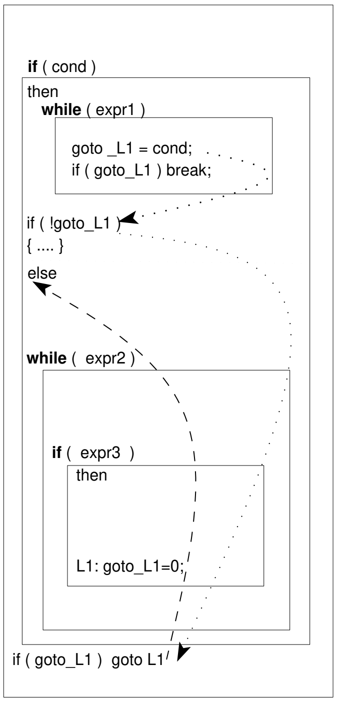


(c) inward-movement into a `switch`

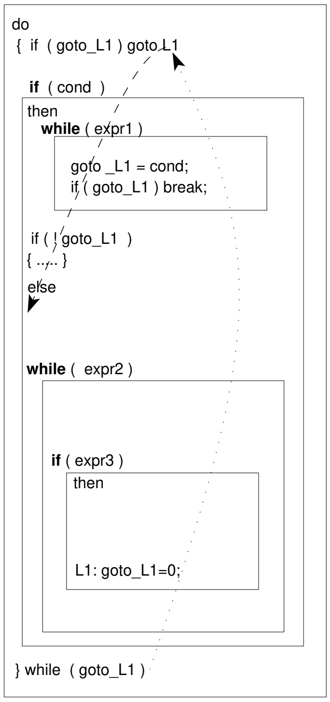


(d) application of goto-elimination

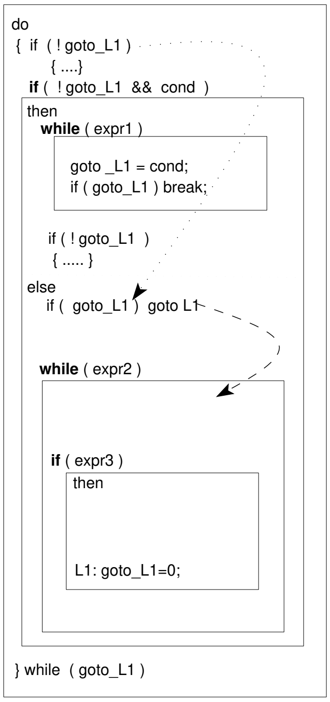
Figure 2.11: Inward-movements followed by goto-elimination



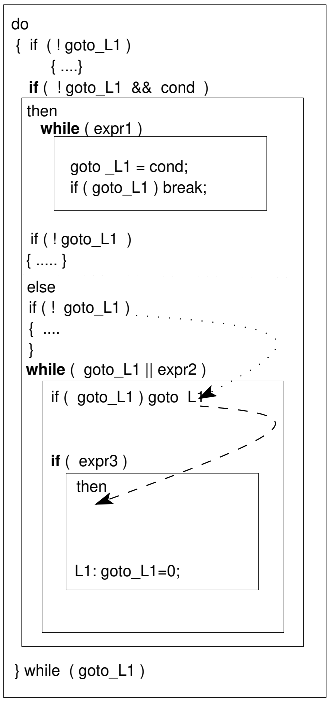
(a) outward-movements



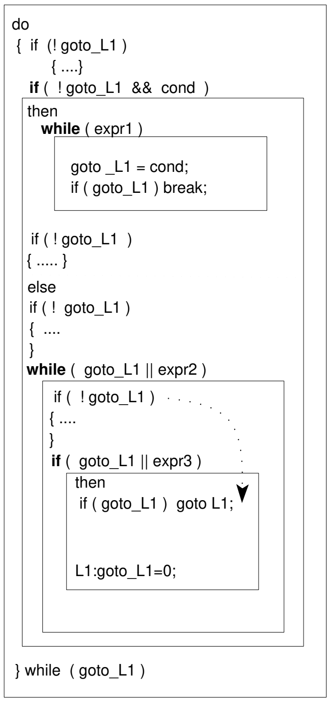
(b) goto-lifting



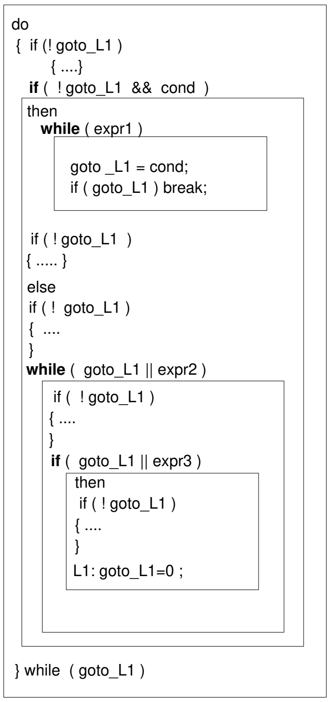
(c) inward-mov. into `if`



(d) inward-mov. into `while`



(e) inward-mov. into `if`



(f) goto-elimination

Figure 2.12: Outward- and Inward-movements followed by goto-elimination

2.1.4 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 2.1.1) and goto-lifting (Section 2.1.2) transformations that introduce new `do` loops. Although these transformations 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 2.13(a) and the incorrect capturing of a `break` statement in Figure 2.13(b). In order to avoid this situation, we must add one further transformation for each captured `break` or `continue`. As illustrated in Figure 2.13(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 is used, except that at the exit of the introduced loop, if the variable is true, we issue the `continue` instead of the `break` for the enclosing loop. Figure 2.14 illustrates this case.

2.2 Eliminating all goto statements from a C program

Based on the goto-elimination, goto-movement and goto-lifting transformations, we can now state the complete algorithm for removing *all* `goto` statements from a C program. The complete algorithm is presented in Figure 2.15.

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. 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. For each `goto`, the matching label is located. Once the `goto`-label pair

```
{ .......
  while (1)
   { ......
Li:  stmt_i;
     .......
     stmt_j;
     if (exp1) break;
     .......
     if (exp2) goto Li;
     .......
     stmt_k;
   }
}
```

(a) original program

```
{ .......
  while (1)
   { .......
     do{
Li:     goto_Li=0;
        stmt_i;
        .......
        stmt_j;
        if (exp1) break;
        .......
        goto_Li = exp2;
       } while (goto_Li);
     .......
     stmt_k;
    }
}
```

(b) incorrect capture of `break`

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

(c) correct treatment of captured `break`

Figure 2.13: Avoiding capture of `break` statements

```
{.......
 while (1)
  { .......
Li: stmt_i;
    .......
    stmt_j;
    if (exp1) continue;
    .......
    if (exp2) goto Li;
    .......
    stmt_k;
  }
}
```

(a) original program

```
{.......
 while (1)
  { .......
    do{
Li:    goto_Li=0;
       stmt_i;
       .......
       stmt_j;
       if (exp1) continue;
       .......
       goto_Li = exp2;
      } while (goto_Li);
    .......
    stmt_k;
  }
}
```

(b) incorrect capture of `continue`

```
{ int do_cont = 0;
  .......
  while (1)
   { .......
     do{
Li:     goto_Li=0;
        stmt_i;
        .......
        stmt_j;
        if (exp1)
          {do_cont=1;
           break;}
        .......
        goto_Li = exp2;
       } while (goto_Li);
     if (do_cont)
        {do_cont=0;
         continue;}
     .......
     stmt_k;
   }
}
```

(c) correct treatment of captured `continue`

Figure 2.14: Avoiding capture of `continue` statements

```
for each procedure p do

  {  /* get the list of labels and gotos for this 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 variable goto_Li initialized to false
         introduce a stmt just after the label Li that resets goto_Li to false
       }
     /* change all 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 transformations
             until it becomes directly related to l
           else /* different branches of the same if or switch */
             move g out using outward-movement transformations
             until it becomes directly related to the 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 transf.
               move g in to level(l) using inward-movement transformations
             }

         /* 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 2.15: High-level algorithm for removing all `gotos`

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

2.3 Optimizations

In this section we present some optimizations to the goto-elimination method. First, some simple optimizations that can be made as the goto-elimination and goto-movement transformations are applied are presented. Then an optimization for a particular situation in which `goto`s can occur is described.

2.3.1 Simple Optimizations

While applying the goto-movement and goto-elimination transformations by following the rules straight away, many unnecessary conditional `if` statements (i.e. with null bodies) can be introduced. There are three situations where we can avoid generating these statements.

Goto statement is next to the label statement: Figure 2.16 illustrates this case in which the `goto` statement is immediately next to the label statement. This situation may occur after several movement transformations, and clearly in this case we may just eliminate the `goto` statement.

```
     { .......                          { .......
       stmt_i;                            stmt_i;
       .......                            .......
       if (cond) goto L1;     =>   L1:    stmt_n;
 L1:   stmt_n;                            .......
       .......                          }
     }
```

Figure 2.16: `Goto` is next to the label

```
{   if ( expr )                          {   if ( expr )
       { .......                                 { .......
         stmt_i;                                   stmt_i;
         .......                                   .......
         if (cond) goto L1;    ⇒                   goto_L1=cond;
       }                                         }
    .......                                  if (goto_L1) goto L1;
L1: stmt_n;                                  .......
}                                        L1: goto_L1=0;
                                             stmt_n;
                                         }
```

Figure 2.17: `Goto` at the end of an `if` block

Goto is at the end of an if block: Figure 2.17 illustrates this case in which 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).

Goto is before a loop that contains the label: Figure 2.18 illustrates this case in which the `goto` is immediately before a loop. We can avoid introducing a conditional statement before this loop.

```
 {                                   {
  if (cond) goto L1;                  goto_L1=cond;
  while ( expr )                      while ( goto_L1 || expr )
        { .......                           { if ( goto_L1 ) goto L1;
          stmt_i;              ⇒              .......
          .......                             stmt_i;
 L1:      stmt_n;                             .......
        }                            L1:      goto_L1=0;
 }                                            stmt_n;
                                            }
                                     }
```

Figure 2.18: Goto immediately before a loop

2.3.2 A special case optimization

Another common situation that can be optimized occurs when there is more than one `goto` associated with a label inside the same `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 `if`, `switch` or loop. This conditional check introduced could be: (i) the conditional `if` introduced to guard a sequence of statements; (ii) the `do-while` introduced to create a cycle of control-flow; and (iii) the conditional `if` containing a `break` statement introduced to exit from a loop or `switch` statement. It is clear that when there is more than one `goto` statement to the same label, it would be preferable to insert only one of these conditional checks per label. For example, we would like the transformation given in Figure 2.19 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.

```
{ .......
  switch(x)
    { case 1:
           .......
           break;
      case 2:
           .......
           goto error;
      case 3:
           .......
           break;
      case 4:
           .......
           goto error;
    }
  .......
error:
}
```

$\Rightarrow$

```
{ .......
  switch(x)
    {  case 1:
            .......
            break;
       case 2:
            .......
            goto_error = 1;
            break;
       case 3:
            .......
            break;
       case 4:
            .......
            goto_error = 1;
            break;
    }
  if (goto_error) goto error;
  .......
error: goto_error = 0
}
```

Figure 2.19: Optimizing multiple `goto`s from the same `switch`

Chapter 3

An overview of SIMPLE

In this section an overview of SIMPLE is presented. As its name suggests, SIMPLE is a simplified version of the first intermediate representation FIRST, where complex program constructs are translated to a simpler form. SIMPLE is based on a simple grammar that is powerful enough to represent all constructs of C.

During the simplify process, complex expressions and statements are broken down to simpler forms, complicated variable names are split whenever possible and all loops, switches and conditionals are modified to adhere to the restricted SIMPLE format.

A complete description of SIMPLE is out of the scope of this thesis, but we will refer to some of the important features through examples which would be helpful in understanding some of the later sections. A detailed description of SIMPLE can be found in [Sri92].

For our purposes, the relevant features include: the different types of statements (statement nodes), along with the most relevant aspects of their tree representation, and the tree representation for sequences of statements and compound statements.

3.1 Basic statements

In SIMPLE a set of fifteen basic expression statements are identified and any other complex expression statement in C can be broken into a sequence of these statements. Figure 3.1 illustrates three examples of statements (two assignment statements and a function call) broken down into a series of simpler statements. The tree node related to the basic statement in SIMPLE is the EXPR_STMT node. Every statement node has a parent node called a TREE_LIST node. This TREE_LIST node is also used to link sequences of statements. Figure 3.2 illustrates a high-level representation of the SIMPLE-AST for the first example in Figure 3.1. The triangles in the figures represent subtrees that will not be described in detail as they are irrelevant to our work.

```
a = b * c + (*d) / e;     ⇒     temp_1 = b * c;
                                temp_2 = *d;
                                temp_3 = temp_2 / e;
                                a = temp_1 + temp_3;
```

```
f ( 3, &a, *b );          ⇒     temp_0 = &a;
                                temp_1 = *b;
                                f( 3, temp_0, temp_1);
```

```
a = b = c;                ⇒     b = c;
                                a = b;
```

Figure 3.1: Example of FIRST to SIMPLE transformations

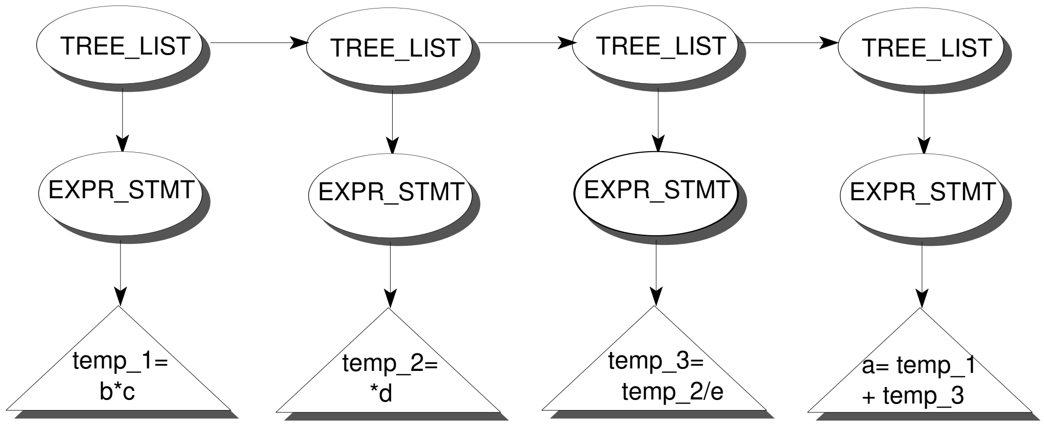


Figure 3.2: SIMPLE AST representation

3.2 Compositional Control Statements

The compositional control statement forms supported by SIMPLE are restricted (simplified) versions of statement sequences, `for`-loops, `while`-loops, `do`-loops, `switch` / `case` statements, and `if-else` statements. In addition, `return` is supported for exiting a procedure or function, `break` and `continue` are supported for exiting a loop and `break` is supported for exiting `switch`/`case` statements.

Figure 3.3 illustrates the syntax of some of the control statements in SIMPLE and the corresponding AST representation. For a complete description of the SIMPLE grammar rules, refer to [Sri92].

The condition expressions for loops and conditionals are reduced to equivalent simple expressions that are free from side-effects. Figure 3.4 shows an example of simplifying the condition of a `while` loop, while Figure 3.5 illustrates the handling of a typical short-circuit condition. In the later case the short-circuit is expressed directly by introducing the appropriate conditional statement.

It should be noted that `switch` and `case` statements need special attention since in C, the body of a `case` statement can be shared partially by different case statements and the compositionality of the control flow is then lost. Each `case` statement is forced to begin with a `case` and end with a `break`, `return` or `continue`, and to replicate shared code. Figure 3.6 illustrates an example of simplifying a `switch` statement.

In order to make the format of the SIMPLE-AST uniform, even when the body of a compositional control statement contains a single statement, it is treated as a compound statement and the statement is therefore put within braces.

Compound statements are represented by a TREE_LIST node being the parent of the sequence of statements that formed this compound statement. Figure 3.7 illustrates a sequence of statements in a compound statement and its corresponding tree. Thus each new compound statement is identified by the presence of two levels of TREE_LIST nodes in the AST.

SIMPLE does not allow variables to be defined inside compound statements. A process called unnesting [Sre92], removes them by lifting variables to the function level, renaming them if necessary.

Thus, the programs represented in SIMPLE, have a regular and simple grammar, where complex statements and expressions are simplified, and compound expressions are broken into simple ones. This is the most convenient point to insert our structuring phase. All analyses and transformations after this phase can assume structured programs.

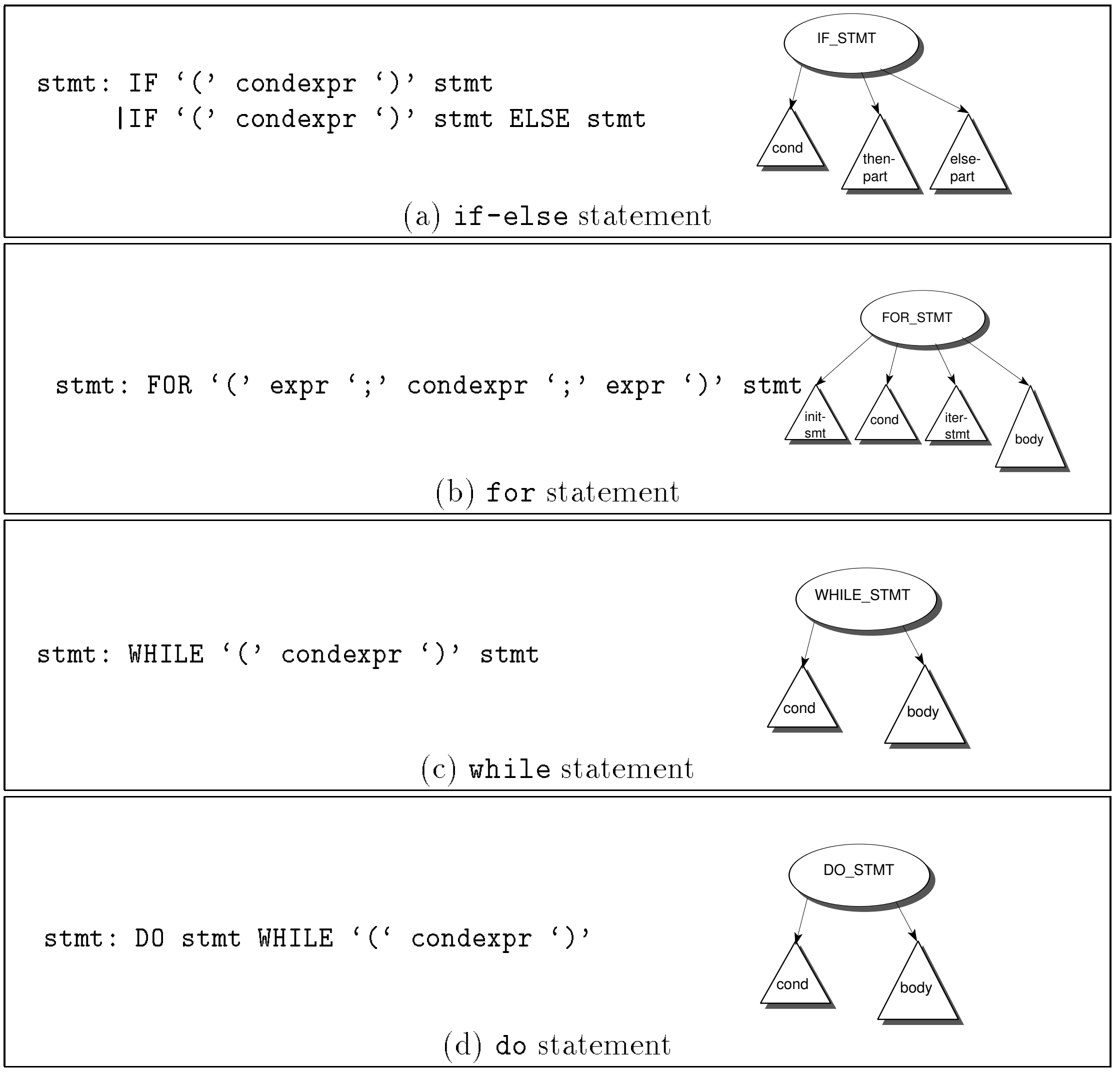


Figure 3.3: SIMPLE grammar and tree nodes for some compound statements

```
while (a + b > c)
   {
   ...
   }
```

$\Rightarrow$

```
temp_1 = a + b;
while (temp_1 > c)
   {
   ...
   temp_1 = a + b;
   }
```

Figure 3.4: Simplification of a `while`-loop condition expression

```
if  ( a > c && a > d)              temp_1 = a > c;
   {                               if ( temp_1 )
   ...                     ⇒          temp_1 = a > d ;
   }                               if ( temp_1 )
                                      {
                                      ...
                                      }
```

Figure 3.5: Simplification of a condition with a short-circuit

```
                                 switch(a)
                                 {
                                     case 1:
switch(a)                            case 2:
{                                        stmt_1 ;
    case 1:                              stmt_2 ;
    case 2:                              stmt_3 ;
        stmt_1 ;                          break ;
    case 3:                          case 3:
        stmt_2 ;          ⇒              stmt_2 ;
    default:                             stmt_3 ;
        stmt_3 ;                          break ;
        break ;                      case 4:
    case 4:                              stmt_4 ;
        stmt_4 ;                         break ;
}                                    default:
                                         stmt_3 ;
                                         break ;
                                 }
```

Figure 3.6: An example of `switch` statement in SIMPLE

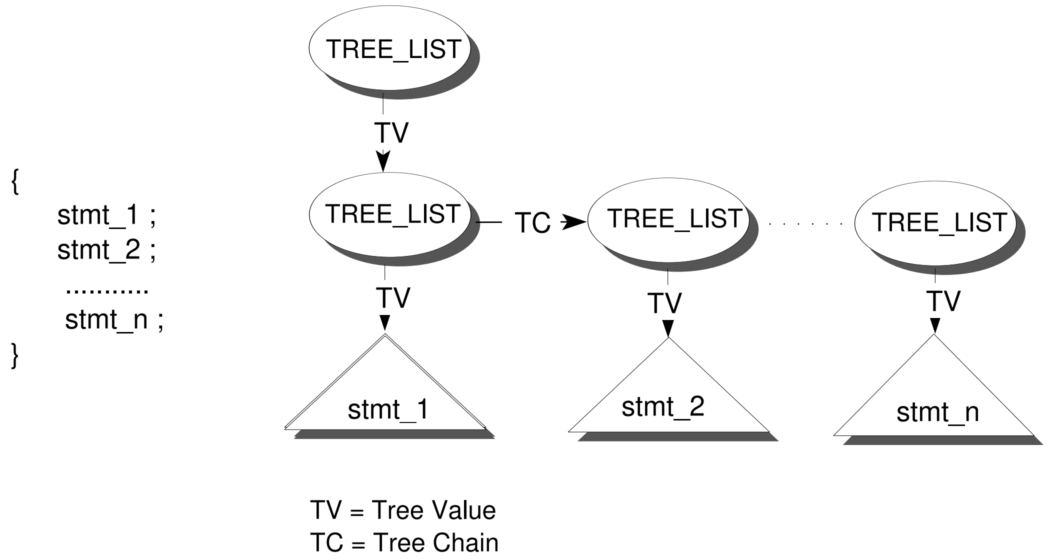


Figure 3.7: SIMPLE representation of statements in a compound statement

Chapter 4

Implementing Goto-elimination in the McCAT compiler

4.1 Overview

This chapter presents some implementation details of the algorithm given in Section 2.2.

First, the selected data structures are discussed. Data structures that efficiently support the operations to categorize a `goto`-label relationship are mandatory. All the information concerning the `goto` and label statements should be stored in them, to be able to implement the goto-elimination algorithm after.

The implementation of the goto-elimination method is divided into two subphases. The first subphase deals with the first three steps of the algorithm presented in Section 2.2, i.e. the initialization phases. They include: (i) the collection of all the label and `goto` statements information; (ii) the creation of the statements to define, initialize and reinitialize the `goto` variables; (iii) the conversion of unconditional `goto`s to conditional ones.

In the second subphase, the goto-elimination algorithm is implemented. The `goto`s are eliminated one by one, and at the end all the labels are removed.

An efficient method to determine if the given `goto` and label statements are siblings, directly-related, or indirectly-related is required. The information stored in the data structures and the SIMPLE-AST should be enough to be able to categorize the relationship of a `goto`-label pair and apply the required transformation.

4.2 Data Structures

Two data structures are used to handle the information related to the `goto`s and labels. The `goto`s are stored in a linked list. The labels are stored in a hash table. The simplest order to eliminate the `goto`s is in the order in which they occur in the linked list. However, as discussed in Chapter 5, there may be better orderings that can be considered. For each `goto` to be eliminated, the matching label is located. To do this efficiently we make use of the hash table of labels. Figure 4.1 presents an example program, and Figures 4.2 and 4.3 illustrate the label table and `goto` list contents for the given program. The next section describes in detail how the information is stored in these data structures.

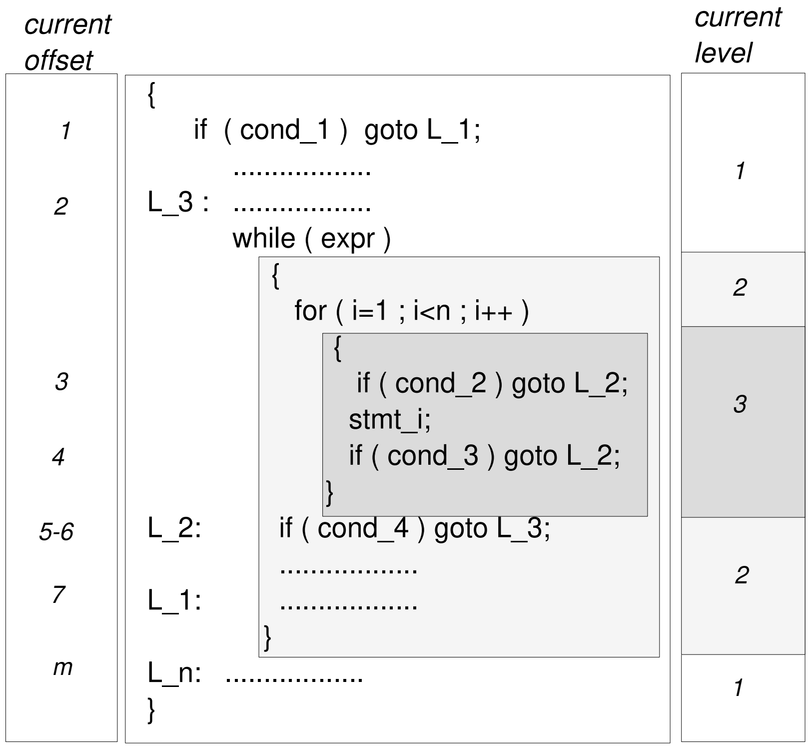


Figure 4.1: Example program

4.3 Initialization Phases

As mentioned before, the initialization phases are implemented in one subphase. There, the SIMPLE-AST of the current procedure is traversed in a recursive manner,

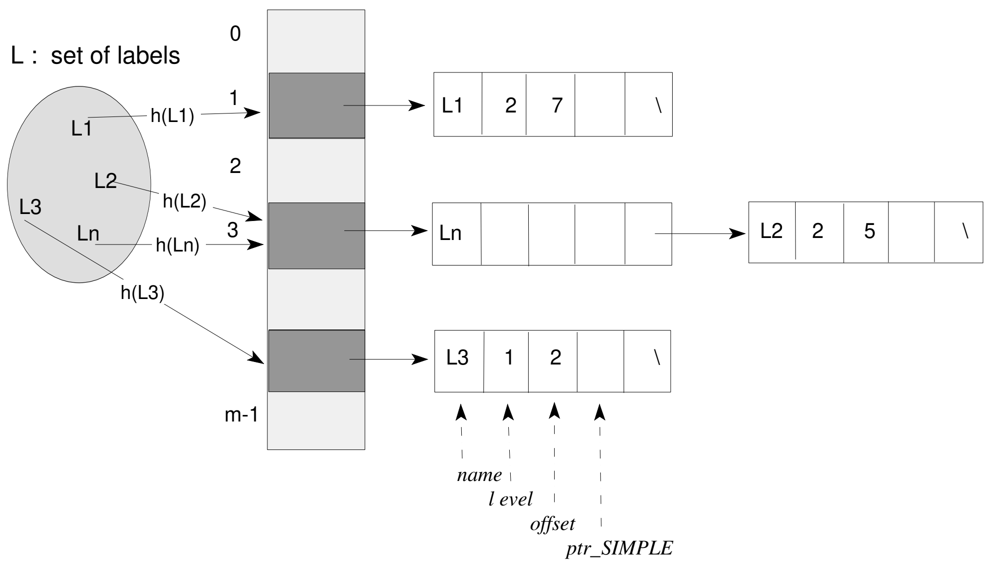


Figure 4.2: Label hash table

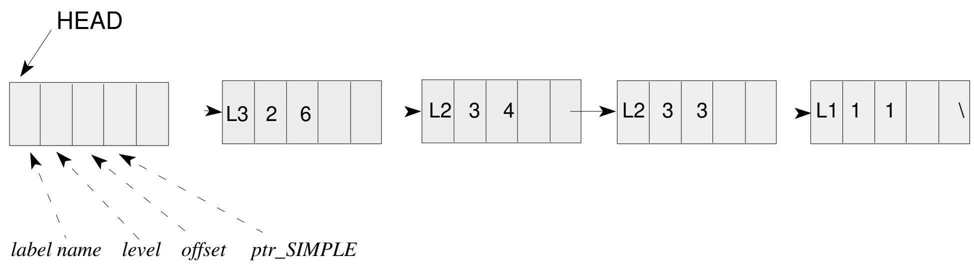


Figure 4.3: Goto linked list

collecting all the information for the `goto` list and the label table, creating the initialization statements, and converting the unconditional `goto`s to conditional `goto`s. More specifically, during the traversal the following actions are performed:

1. A global variable for *level* is maintained. It is incremented each time we enter a loop, `switch` or `if` statement, and decremented each time we exit these statements.

2. A global variable for *offset* is maintained, it is incremented each time we find a `goto` or a label statement.

3. Every TREE_LIST node is linked to its parent's TREE_LIST node and the current *level* is assigned to the level field of this node. That is, from every statement node (i.e., the TREE_LIST node associated with the statement) we can access the parent statement node (i.e. the TREE_LIST node associated with the parent statement) and we can know the statement level.

4. Every unconditional `goto` is transformed into a conditional one.

5. For each `goto` found:

   - the *offset* is incremented.

   - a new node in the `goto` list is created and the `goto` information is stored.

6. For each label statement found:

   - the *offset* is incremented.

   - a new variable associated with the label is created.

   - the statements to initialize and reset the new variable are created.

   - a new node in the hash table is created and the label information is stored.

From the Figures 4.2 and 4.3 we observe that a label node is inserted in the hash label table each time a label is found, using the label name as the hash function argument. The current *level* and *offset* (refer to Figure 4.1), the label name and a pointer to the SIMPLE-AST label node are stored.

For each `goto`, a node is inserted at the head of the `goto` list. The current *level* and *offset*, the label name associated with the `goto`, and a pointer to the SIMPLE-AST `goto` node are stored.

The nodes are inserted at the head of the `goto` list, and `goto`s are eliminated in the order they appear in the `goto` list. Thus, as it can be observed from Figure 4.3, the order of the goto-elimination is the reverse order to the order the `goto`s appear in the procedure.

It should be noted that we actually implement all the initialization steps during one pass through the SIMPLE-AST in the first subphase, and then no further passes through the SIMPLE-AST are required. Subsequent steps can be performed directly using the information collected during this first pass. That is, in the first subphase 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 in the second subphase. We create parent pointers in the SIMPLE tree to find common ancestors that can be used to efficiently determine the relationship between the `goto` and label. Thus, we are able to support efficient operations to get the level and offset of each label and `goto`(by accessing the data structures) , and determine if the `goto` and the label are indirectly-related, directly-related, or siblings.

4.4 Determining the relationships between gotos and labels.

Now we will show how to categorize a `goto`-label pair as *siblings, directly-related*, or *indirectly-related*.

4.4.1 Siblings

Definition 2.1.3 states that a label and a `goto` statement are *siblings* if there exists some statement sequence `stmt_1; ...stmt_i; ...stmt_j; ...stmt_n`; such that the label statement corresponds to some `stmt_i` and the `goto` to some `stmt_j`.

In the SIMPLE_AST a `goto`-label pair are *siblings* if both nodes have a common parent. As illustrated in the example in Figure 4.4 this means that the back-pointer of the TREE_LIST node associated with the `goto` and the backpointer of the TREE_LIST node associated with the label point to the *same* TREE_LIST node.

4.4.2 Directly-related

Definition 2.1.4 states that a label and a `goto` statement are *directly-related* if there exists a statement sequence `stmt_1; ...stmt_i; ...stmt_j; ...stmt_n`; such that

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

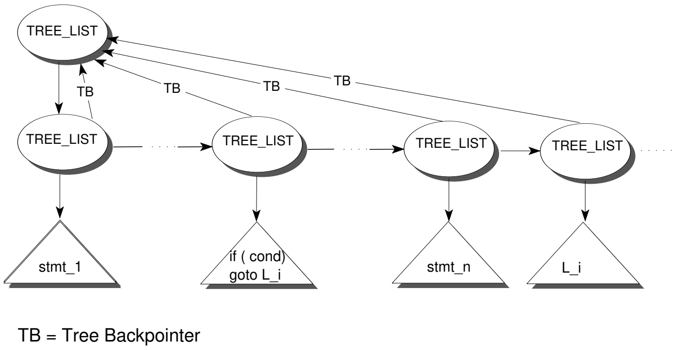


Figure 4.4: `goto` and label are siblings

either (i) the label corresponds to some `stmt_i` and the matching `goto` is nested inside some `stmt_j` in the statement sequence or (ii) the `goto` corresponds to some `stmt_i` and the matching label is nested inside some `stmt_j` in the statement sequence.

Let *level(stmt_i)* represent the level associated with statement `stmt_i`, and let *parent(stmt_i)* represent the parent pointer (backpointer) of the TREE_LIST node associated with `stmt_i`.

In the SIMPLE-AST the first case can be specified by the following two conditions:

1. $level(goto) > level(label)$

2. Let the `goto` be nested inside some `stmt_j`, which is a sibling of `stmt_i`, the label statement. Then *parent(label) = parent(stmt_j)* where `stmt_j` is the statement obtained by traversing $2{*}(level(goto) - level(label))$ backpointers from the `goto` node. (For each statement that contains the `goto` and not the label, two parent pointer levels should be followed since, as we specified in Chapter 3 a compound statement is represented by two levels of TREE_LIST nodes).

   Thus we can state the condition as:

   $parent(label) = parent^{2*(level(goto)-level(label))+1}(goto)$

The example in Figure 4.5 illustrates this case, where the `goto` is nested inside two statements (`if` and `while`), and the label is in the same statement sequence as the outermost of these statements. The label level is one and the `goto` level is three. From the TREE_LIST node associated with the `goto` statement, by following four backpointers of TREE_LIST nodes, we reach the TREE_LIST node associated with the `while` statement. The backpointer of this TREE_LIST node points to the same

TREE_LIST node as the backpointer of the TREE_LIST node associated with the label.

Similarly, the second case can be specified by these two conditions:

1. $level(label) > level(goto)$
2. Let the label be nested inside some `stmt_j`, which is a sibling of `stmt_i`, the `goto` statement. Then *parent(goto) = parent(stmt_j)* where `stmt_j` is the statement obtained by traversing $2*(level(label)-level(goto))$ backpointers from the label node.

   Thus we can state the condition as:

   $parent(goto) = parent^{2*(level(label)-level(goto))+1}(label)$

   The example in Figure 4.6 illustrates this case.

4.4.3 Indirectly-related

Definition 2.1.5 states that a label and a `goto` statement are *indirectly-related* if they appear in the same procedure body but they are neither *siblings* nor *directly-related.*

Thus, *indirectly-related* `goto`-label pairs include the cases when the `goto` and label are in entirely different statements and the special cases when the `goto` and label are in different branches of the same `if` or `switch` statement.

To categorize a `goto` and label statements as *indirectly-related*, first it is checked they are not *siblings* (*parent(goto)* <> *parent(label)*).

Then the *level* of the `goto` and the *level* of the label is compared. According to this result there are two possibilities:

- if *level(goto) = level(label)* then they are not *directly-related*, so it can be stated they are *indirectly-related.*
- if *level(goto)* <> *level(label)* the following steps are performed:

  (i) Select the statement (`goto` or label) that has greater *level*. Let this statement be `greater_level_stmt` and let the other statement be `smaller_level_stmt`.

  (ii) From the `greater_level_stmt` node, pairs of parent pointers are traversed, until the *level* of `smaller_level_stmt` is reached. If the parent pointer of

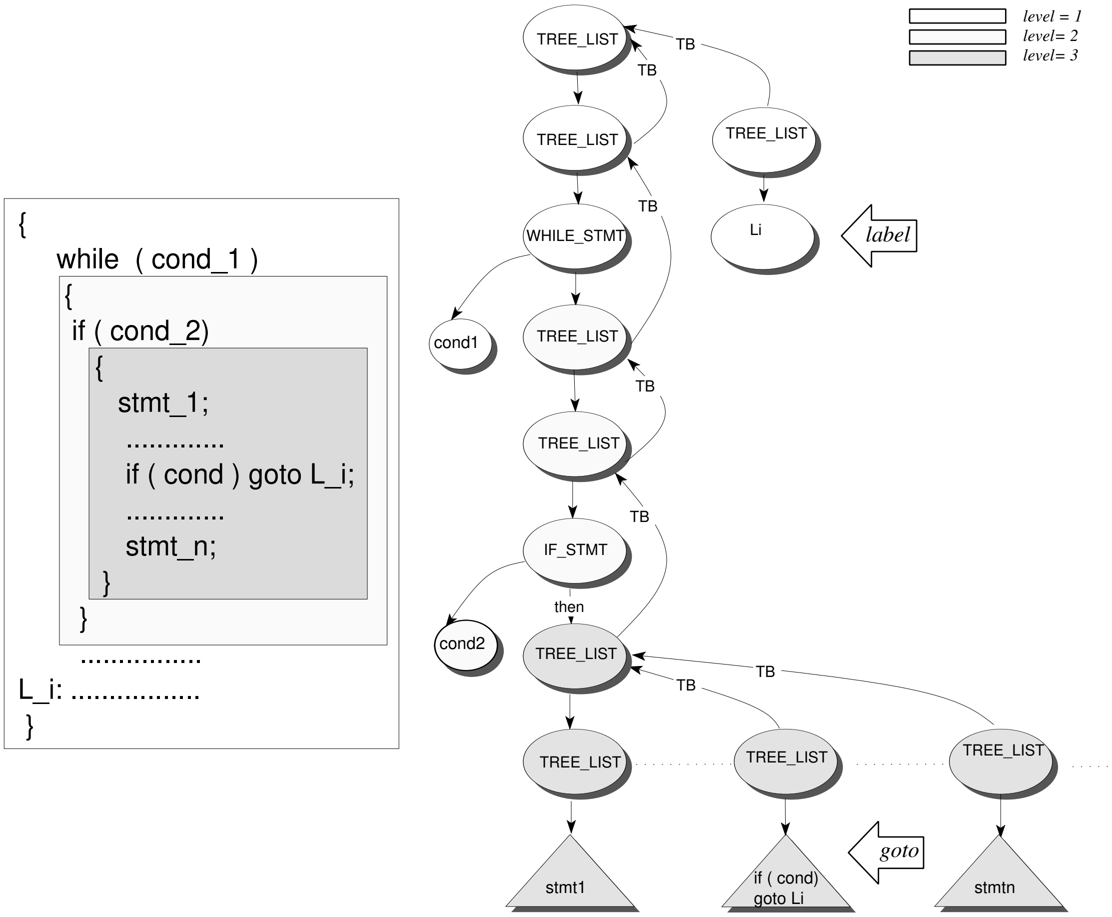


Figure 4.5: Determining a directly-related `goto`-label pair

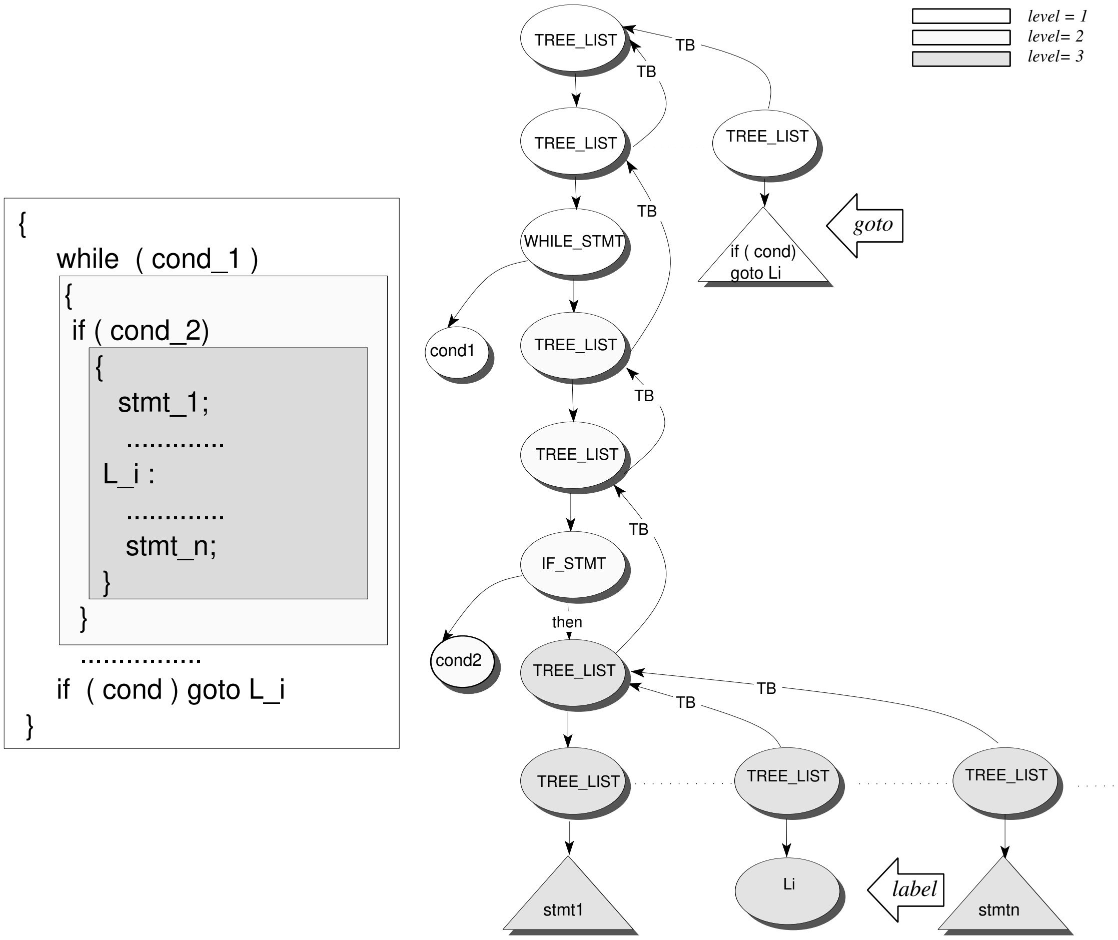


Figure 4.6: Determining a directly-related `goto`-label pair

the statement node that was reached is the same as the parent pointer of `smaller_level_stmt`, then they are *directly-related*, otherwise they are *indirectly-related*.

Once a `goto`-label pair has been categorized as *indirectly-related* then the conditions for the special cases are checked.

General case: goto and label are entirely in different statements

In this case, there exists a statement sequence `stmt_1; ...; stmt_i; ...stmt_j; ...stmt_n;` where the `goto` is nested inside some `stmt_i` and the label is nested inside some `stmt_j` (`stmt_i` $<>$ `stmt_j`).

In the SIMPLE-AST, from the `goto` node by traversing pairs of parent pointers, and from the label node by traversing pairs of parent pointers, `stmt_i` and `stmt_j` nodes are reached such that *parent(stmt_i) = parent(stmt_j)*.[1]

Figure 4.7 illustrates this case. The `goto` statement is nested inside an `if` and a `while` statement, so its *level* is 3. The label statement is inside an `if` statement so its *level* is 2. From the TREE_LIST node associated with the `goto`, by following four backpointers we reach the TREE_LIST node associated with the `while` statement that contains the `goto`, and from the TREE_LIST node associated with the label, by following two backpointers we reach the `if` statement that contains the label. The backpointers of these `while` and `if` statements point to the same TREE_LIST node.

Special cases: goto and label are nested in different branches of the same if or switch

In this case the `goto` and label are nested in zero or more statements inside different branches of the same `if` or `switch`.

In the SIMPLE-AST, from the `goto` node by traversing pairs of parent pointers, and from the label node by traversing pairs of parent pointers, `stmt_i` and `stmt_j` nodes are reached such that *parent(stmt_i)* $<>$ *parent(stmt_j)* and either:

(i) *parent(parent(stmt_i)) = parent(parent(stmt_j))*

for the case the `goto` and label are in different branches of the same `if`

[1] For each statement where the `goto` (label) is nested inside two parent pointer levels should be followed.

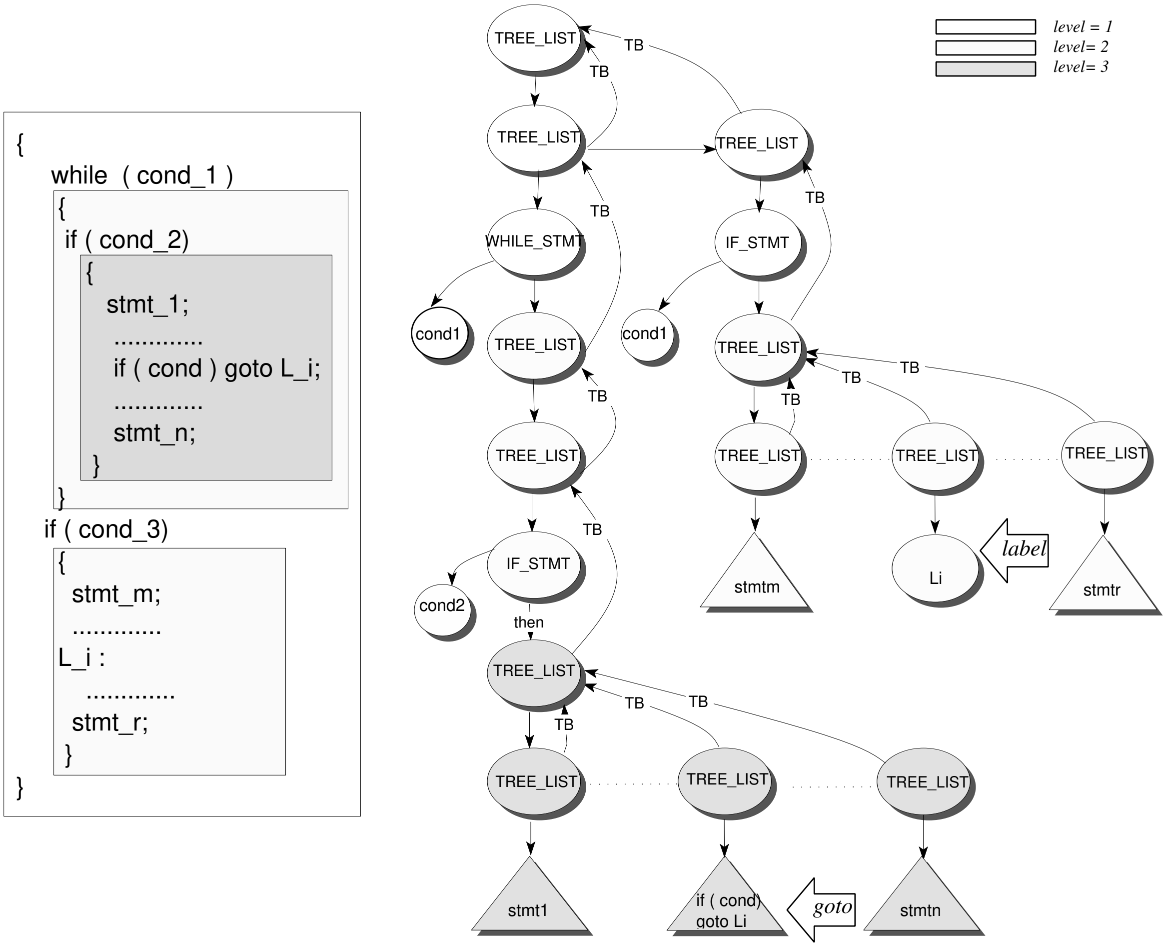


Figure 4.7: Determining an indirectly-related goto-label pair

or

(ii) *parent(parent(parent(stmt_i))) = parent(parent(parent(stmt_j)))*

parent(parent(stmt_i)) and *parent(parent(stmt_j))* are `case` statements for the case the `goto` and label are in different branches of the same `switch`.

In the case of the `if` statement, the fact that this statement has *two children* that are TREE_LIST nodes (one that points to the compound statement of the `then` body and the other that points to the compound statement of the `else` body) is a unique condition in the format of the SIMPLE-AST. Thus, the backpointers of these TREE_LIST nodes point to the *same* TREE_LIST node (the one corresponding to the `if` statement).

In the case of the `switch` statement, there is no special condition regarding the format of the SIMPLE-AST itself. We specially check if the label and the `goto` are in different `case` of the same `switch`.

Figure 4.8 illustrates the case of a label and a matching `goto` that belong to different parts of the same `if`. From the TREE_LIST node of the `goto` by following two backpointers and from the TREE_LIST node of the label by following two backpointers the same TREE_LIST node is reached. Figure 4.9 illustrates the case of a label and a matching `goto` that belong to different `case` statements of the same `switch` statement. From the TREE_LIST node of the `goto` by following two backpointers the TREE_LIST node associated with a `case` statement is reached, and from the TREE_LIST node of the label by following two backpointers the TREE_LIST node associated with a different `case` but the same `switch` is reached.

These tests to categorize the general and special cases are done after the adjustement of the `greater_level_stmt` to be equal to `smaller_level_stmt`.

4.5 Elimination Phase

The elimination phase is implemented in the second subphase, as explained before. In this subphase, the `goto` list is traversed sequentially, eliminating one by one the `goto`s associated with each of the nodes in the list.

For each `goto` the matching label is searched in the hash table. Both the `goto` list and the hash table contain pointers to the TREE_LIST nodes associated with the corresponding `goto` and label statements. Making use of these pointers, and the backpointers we can:

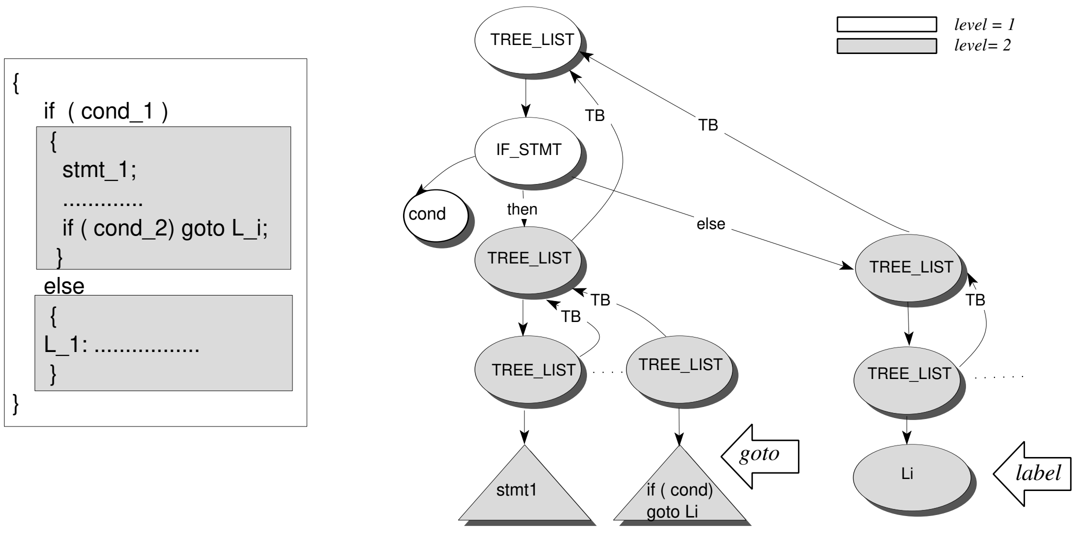


Figure 4.8: Determining an indirectly-related `goto`-label pair in same `if`

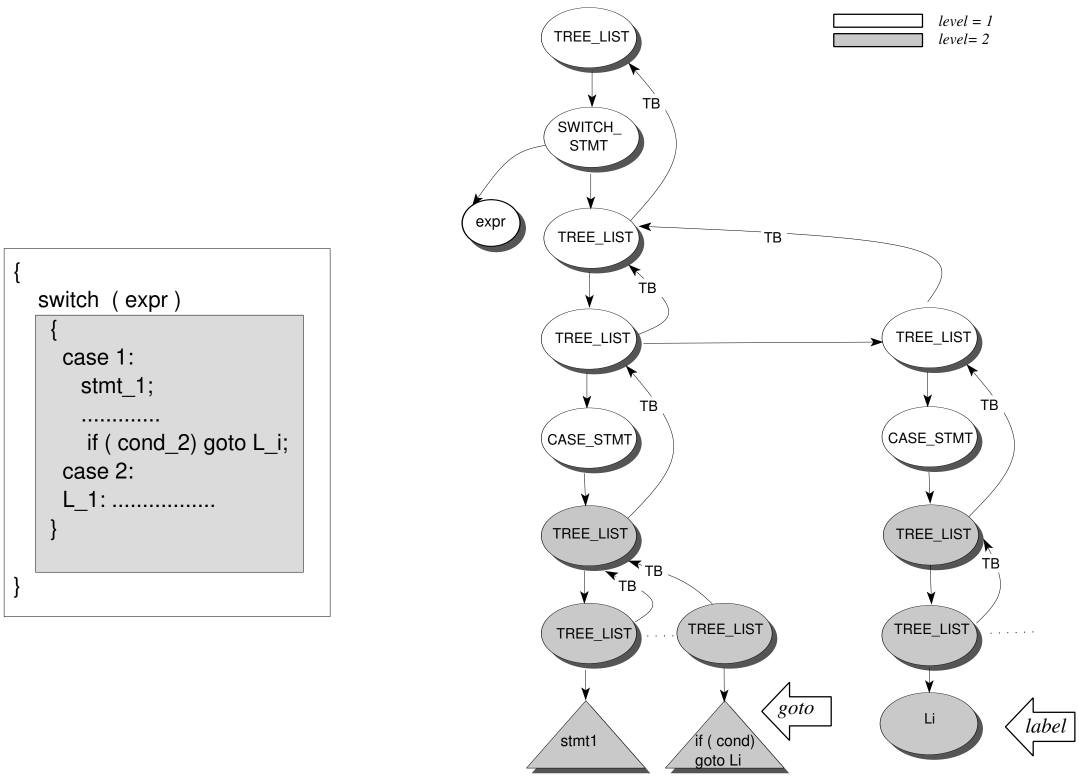


Figure 4.9: Determining an indirectly-related `goto`-label pair in same `switch`

- determine if the `goto` and the label are siblings, directly-related or indirectly-related, as explained in the previous section.
- apply the required goto-movement or goto-elimination transformations according to the previous classification.
- perform the required changes to the fields of the statement nodes (*level* and *backpointers*) that need to be modified after applying a goto-movement transformation.

A high level description of the implementation of the transformations applied to eliminate one `goto` is given in Figures 4.10, 4.11 and 4.12.

```
for each goto g in the goto-list do
   {
     /* for g find the matching label l */
     l := label matching g in the hash label table
     if parent(l) <> parent(g) then
        /* g and l are not siblings, move g to be a sibling of l*/
        g := Goto-movement-transformations(g,l)
     /* g and l are siblings, apply one of the goto-elimination
        transformations*/
     if offset(g) < offset(l) then
       apply goto-elimination transformation for g before l
     else
       apply goto-elimination transformation for g after l
   }
```

Figure 4.10: Implementation of the goto-elimination phase

Note that the variables `stmt_has_l` and `stmt_has_g` in Figures 4.11 and 4.12 correspond to what we call `greater_level_stmt` and `smaller_level_stmt` in the previous section, depending on the value of the `goto` and label levels.

```
Goto-movement-transformations(g,l)
{ if level(g) > level(l) then
  { /* determine whether g is directly or indirectly related to l*/
    stmt_has_g := g
    while  level(stmt_has_g) > level(l) do
           stmt_has_g := parent(parent(stmt_has_g))
    if  parent(stmt_has_g) = parent(l)  then
         /* g and l are directly-related => move g out using outward
            movements until it becomes a sibling of l */
         while  level(g) > level(l)
               g := apply outward-movement transformation to g
    else
      {  /* g and l are indirectly-related => move g out using outward
            movements and then move g in using inward movements */
        stmt_has_l := l
        g := Indirectly-related-transformations(g,l,stmt_has_g,stmt_has_l)
      }
   }
 else
  {  if level(l) > level(g) then
     {  /* determine whether g is directly or indirectly related to l*/
       stmt_has_l := l
       while  level(stmt_has_l) > level(g) do
              stmt_has_l := parent(parent(stmt_has_l))
       if  parent(stmt_has_l) = parent(g) then
           /* g and l are directly-related => move g in using inward
              movements until it becomes a sibling of l */
           while  level(g) < level(l)
               if offset(g) > offset(l) then
                  g := apply goto-lifting transformations to lift g above l
               g := apply inward-movement transformation to g
       else
         {   /* g and l are indirectly-related  => move g out using outward
                movements and then move g in using inward movements */
          stmt_has_g := g
          g := Indirectly-related-transformations(g,l,stmt_has_g,stmt_has_l)
         }
     }
    else
      {  /* g and l are indirectly-related => move g out using outward
            movements and then move g in using inward movements */
       stmt_has_l := l; stmt_has_g := g
       g := Indirectly-related-transformations(g,l,stmt_has_g,stmt_has_l)
      }
 return g
 }
```

Figure 4.11: Implementation of the goto-movement transformations

```
Indirectly-related-transformations( g,l,stmt_has_g,stmt_has_l)
{
  /* g and l are either in entirely different statements or in different
     branches of the same if or switch statement */
  while  parent(stmt_has_g) <> parent(stmt_has_l) do
        {
         stmt_has_g := parent(parent(stmt_has_g))
         stmt_has_l := parent(parent(stmt_has_l))
        }
  while  level(g) > level(stmt_has_l) do
         if  stmt_has_l = parent(parent(g))
             /* g and l are in <> branches of the same if statement */
             g :=  apply outward-movement transformations to g for the
                   case g and l in <> branches of an IF stmt
         else
          if  parent(parent(parent(g))) =  parent(stmt_has_l)
              and parent(parent(g)) and stmt_has_l are CASE stmts
              /* g and l are in <> branches of the same switch statement */
               g :=  apply outward-movement transformations to g for the
                     case g and l in <> branches of a SWITCH stmt
          else
              /* g and l are in entirely different statements */
              g := apply outward-movement transformation to g
  while  level(g) <  level(l) do
         if offset(g) > offset(l) then
            g := apply goto-lifting transformations to lift g above l
         g :=  apply inward-movement transformation to g
  return g
}
```

Figure 4.12: Implementation of the transformations for indirectly-related `goto`-label pairs

Chapter 5

Experimental Results

In this chapter we give some experimental results using our implementation of the goto-elimination method for the McCAT compiler. First a description of the selected benchmarks is presented. Then, the experimental method is described, and finally the results are discussed.

5.1 Benchmarks

In order to test our structuring method we collected a set of 11 benchmarks that contain `goto` statements. Although in practical terms, our structurer 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.

5.1.1 Benchmark description

Here a brief description of each of the benchmarks is presented.

asuite and nrcode2 : These programs are part of the kernels designed by Lauren Smith to test C vectorizing compilers and their ability to recognize vector structures [Smi91]. For `asuite` we work with a subset of its functions, the ones that contain `goto`s.

compress : A file compression program of the style similar to the one described in IEEE Computer, March 1992. This version of the program was written by Spencer W. Thomas et al.

cq : This program performs a series of tests on a C compiler, based on information in *The C Programming Language* by Kerningham and Ritchie [KR78].

frac : This program finds a rational approximation for a floating point value. It was written by Robert J. Craig at AT&T Bell Laboratories, Naperville.

FSM : This is a program that implements a finite state machine with an irreducible loop. The program is presented in Appendix A and was provided by David Chase.

indent : This program is the GNU's indentation/formatting program, version 1.8.

tomcatv : A C version of the FORTRAN program `tomcatv`, a highly vectorizable double precision floating point mesh generating benchmark. The FORTRAN version is part of the SPEC benchmark suite.

lex.yy : This is the output of a program generated by `lex`. The input `lex` specification is given in Appendix B.

par : This program is a filter which copies its input to its output, changing all white characters (except newlines) to spaces, and reformatting each paragraph. It was written by Adam M. Costello, 1993.

whetstone : This is a C version of the FORTRAN synthetic benchmark `whetstone`.

5.1.2 Benchmark characteristics

The benchmark characteristics relevant to our work are presented in Table 5.1. For each benchmark the number of `goto`s, the number of labels, the number of lines of source code and a general characterization of the types of `goto`s used are presented. To provide a fair comparison of the number of lines of source code we ran a script that strips comments, eliminates blank lines and formats the programs into a standard form.

To help the discussion of the results, depending on the complexity of the transformations required to eliminate a `goto`, we classify the usage of `goto`s into two categories: *simple goto-usage* and *complex goto-usage*. *Simple goto-usage* includes the cases when the `goto`s are siblings to their labels or used as outward branches from control constructs. *Complex goto-usage* includes the cases when the `goto`s branch to a label that is within a control construct, or when the `goto`s belong to a `goto`-label pair that overlaps with other `goto`-label pairs (i.e. a `goto` jumps to a label within a region spanned by another `goto`-label pair).

Based on the above classification and depending on the frequency of the simple and complex `goto`-usages in the benchmark, we have divided the set of benchmarks into two sets: *simple goto-usage benchmarks* (with a majority of simple goto-usages) and *complex goto-usage benchmarks* (with a majority of complex goto-usages).

We will refer to the specific characteristics of each benchmark throughout the discussion of the results.

name of benchmark	# of `gotos`	# of labels	# of stmts	goto-usage
cq	1	1	5760	simple
nrcode2	2	2	106	simple
lex.yy	4	3	1681	complex
frac	6	5	58	simple
tomcatv	7	6	197	complex
compress	9	6	1331	complex
FSM	12	6	56	complex
asuite	22	21	244	simple
indent	28	10	3923	complex
whetstone	31	31	316	simple
par	59	11	1665	complex

Table 5.1: Benchmark characteristics

5.2 Experimental Method

As explained in the previous chapters, structuring takes place after the simplification process. From the SIMPLE intermediate representation, we can either dump out a C program (using McCAT as source-to-source compiler) or continue with the back-end phases of McCAT, as illustrated in Figure 1.1.

In order to measure the effectiveness of our structuring 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: [1] This is a C program that is dumped after the SIMPLE representation has been structured using the transformation rules presented in Section 2.1. No optimizations of the transformation rules are used.

GTE(opt) version: This is a C program that is dumped after the SIMPLE program has been structured using the transformation rules presented in Section 2.1, and the optimizations presented in Section 2.3.

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 system call `, getrusage` on a SPARCstation SLC. We have reported the user time from these experiments.

5.3 Results and Discussion

Next, the results of the experimental measurements are described using the following five tables:

(i) A *comparison of the number of transformations* applied for the GTE and GTE(opt) versions of the programs is presented in Table 5.2.

(ii) A *comparison of the number of new statements* created for the GTE and GTE(opt) versions of the programs is presented in Table 5.3.

(iii) Concentrating on the GTE(opt) version of the benchmarks, the *distribution of the different types of transformations* applied is presented in Table 5.4.

(iv) Concentrating on the GTE(opt) version of the benchmarks, the *distribution of the different types of new statements* is presented in Table 5.5.

(v) A *comparison of the execution times* (times collected as described in the previous section) for the GTE and GTE(opt) versions of the program is presented in Table 5.6.

[1] GTE stands for goto-elimination

5.3.1 Comparing transformations for GTE and GTE(opt)

In this subsection we evaluate our structurer depending on the number of transformations applied. Table 5.2 illustrates these results for our set of benchmarks. First note that, whereas for the GTE(opt) version around 2 or 3 transformations per `goto` eliminated are applied, for the GTE around 2 to 4 transformations are applied. The number of transformations that occur when we apply the GTE and the GTE(opt) versions varies for five of the benchmarks (`lex.yy, compress, FSM, indent` and `par`). These benchmarks are the ones that apply the last of the optimizations referred to in Section 2.3 and illustrated in Figure 2.19. In this case there is more than one `goto` associated with a label inside the same `if,switch` or loop statement. The first of these `goto`s is eliminated using the regular transformations. For the rest of the `goto`s, the remaining transformations are the same, once the common `if`, `switch` or loop statement is exited. Thus, we avoid duplicating the same code. `Indent` is a good example to illustrate the benefit obtained from this optimization, where the number of transformations is reduced by 33% by using the GTE(opt) instead of the GTE.

name of benchmark	GTE # of transf.	GTE # transf./ #`goto`	GTE(opt) # of transf.	GTE(opt) # transf./ #`goto`
cq	1	1	1	1
nrcode2	5	2.5	5	2.5
lex.yy	16	4	12	3
frac	7	1.2	7	1.2
tomcatv	14	2	14	2
compress	27	3	23	2.5
FSM	23	1.9	19	1.6
asuite	34	1.5	34	1.5
indent	108	3.9	74	2.6
whetstone	63	2	63	2
par	189	3.2	160	2.7

Table 5.2: Transformations for the GTE and the GTE(opt)

5.3.2 Comparing new statements for GTE and GTE(opt)

In this subsection, we evaluate our structurer depending on the number of new statements created. Table 5.3 illustrates these results for our set of benchmarks. Let us first discuss a special case, the case of `lex.yy`. Note the remarkable difference between the number of new statements per `goto` for `lex.yy` as compared to the other benchmarks for both the GTE and the GTE(opt) versions. The reason for this is that this benchmark contains 4 `break` statements inside a `for` statement that require the transformation to avoid their incorrect capture by a new `do-while` statement (this result is illustrated next, in Table 5.4). Thus, for this reason, 6 new statements are created, which relative to the number of `gotos` in the benchmarks (4) represents a significant increase. Next, consider the rest of the benchmarks. There is an important difference between the number of statements created by the GTE and the GTE(opt). For the GTE(opt) version between 2 to 4 statements are created whereas, for the GTE between 2 to 7 are created. The difference is due mostly to the number of new conditional `if` and `do-while` statements. The *plain* application of the goto-elimination method (i.e. without applying the simple optimizations described in Section 2.3) usually produces many `if` statements with null bodies.

	GTE		GTE(opt)	
name of benchmark	# of new stmt	# new stmt/ #`gotos`	# of new stmt	# new stmt/ #`gotos`
cq	2	2	2	2
nrcode2	8	4	5	2.5
lex.yy	30	7.5	16	4
frac	12	2	11	1.8
tomcatv	28	4	20	2.9
compress	43	4.8	30	3.3
FSM	50	4.2	33	2.8
asuite	78	3.5	68	3.1
indent	187	6.7	95	3.4
whetstone	107	3.5	62	2
par	262	4.4	164	2.8

Table 5.3: New statements for the GTE and the GTE(opt)

Since the GTE(opt) is clearly more efficient with respect to the number of transformations and the number of new statements created, we shall now concentrate on studying these two aspects in more detail for just the GTE(opt) version of the programs.

5.3.3 Distribution of transformations for GTE(opt)

Table 5.4 presents the distribution of the transformations performed for the GTE (opt).

The transformations can be classified as follows:

(i) goto-elimination transformations that includes the transformations when the `goto` and label are in the same compound statement (i.e. siblings), where the `goto` occurs either before or after the label;

(ii) goto-movement transformations that includes the outward-movement transformations (moving `goto`s out from control constructs), the inward-movement transformation (moving `goto`s into control constructs) and the goto-lifting transformations (performed before the inward-movement transformation, in order to move a `goto` that occurs after the label before the label);

(iii) the transformations that avoid the incorrect capture by a new `do-while` statement of a `break` or a `continue` statement enclosed in a loop or a `switch` statement.

Let us first consider the goto-elimination transformations. We observe that for each `goto` a goto-elimination is performed, except when the optimizations for the cases when more than one `goto` is associated with a label inside the same `if`, `switch` or loop is applied. We observe that forward branches occur six times more often than backward branches.

Now, let us consider the goto-movement transformations. Looking at the table, we observe that outward-movement transformations are applied almost five times more frequently than the inward-movement transformations. `Indent` and `lex.yy` are the ones which have the highest ratio of outward-movement transformations to number of `goto`s. In `Indent` there is a big `switch` statement where `goto`s are used to branch from different `case` statements, to a label in another `case` (all `case`s belong to the same `switch`). `Lex.yy` presents three overlapping `goto`-label pairs and with the `goto`s nested inside three levels of `if`s.

A study of all the benchmarks suggests that inward-movement transformations are rarely used; i.e., `goto`s are rarely use to jump into a control construct. Most of the inward-movement transformations performed are caused by labels that are in the scope of new statements introduced by a previous transformation of a `goto` previously eliminated. The number of goto-lifting transformations is insignificant.

Benchmarks such as `frac` where the number of goto-movement transformations per `goto` is very low are cases where the majority of the `goto`-label pairs are siblings that seldom overlap with other `goto`-label pairs and where almost all the labels are associated with a single `goto`.

The third group of transformations, the ones performed to avoid an incorrect capture of a `break` or a `continue` statement, as illustrated by the table, are performed an insignificant number of times. As we said in Section 2.1.4, these situations happen on rare occasions, but it is a subtle point that must be taken into account.

Finally, we can note that all these results are consistent with a study done by Ballance and Maccabe [BM92], which indicates 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 in the same compound statement or the `goto` is used to exit from a control structure.

name of benchmarks	#of gotos	Transformations for GTE(opt)					
		goto-elimination		goto-movement			capture
		goto-first	label-first	Outw.	Inw.	Lift.	break-cont.
cq	1	1	0	0	0	0	0
nrcode2	2	1	1	3	0	0	0
lex.yy	4	1	1	10	0	0	4
frac	6	5	1	0	1	0	0
tomcatv	7	6	1	5	2	0	0
compress	9	3	4	13	3	0	0
FSM	12	8	2	5	4	0	0
asuite	22	16	6	8	4	0	0
indent	28	10	1	56	5	2	0
whetstone	31	30	1	29	3	0	0
par	59	41	2	94	23	0	0
TOTAL :	181	122	20	215	45	2	4

Table 5.4: Detail of the transformations for GTE(opt)

5.3.4 Distribution of new statements for GTE(opt)

Table 5.5 illustrates the different types of new statements created for the GTE(opt).

`If` and `do-while` statements are the most expensive of all the added statements. Also, their introduction might increase the number of transformations for the elimination of subsequent `goto`s that overlap with the one that is being eliminated. From Table 5.5 we observe that usually 1, and at most 2 conditional `if` or `do-while` statements per `goto` are created. The worst cases occur, as expected, for the benchmarks included in the group of *complex-goto usage* benchmarks. However, note that for `whetstone` the number of `if`s and `do-while`s created is only roughly half the number of `goto`s contained. The reason is that this benchmark, translated from FORTRAN with `f2c`, has the same pattern repeated *13 times* in the program. Figure 5.1 illustrates this pattern. For one of the `goto`s (`goto L2`) the initialization, setting and reinitialization of the `goto` variables are the only new statements required for the GTE(opt). No conditional `if`s need be added. Thus, the GTE(opt) substantially reduces the total number of `if`s created for this benchmark, resulting in a very low ratio of `if` and `do-while` statements per `goto`.

```
 if ( cond )
    goto L1;
 else
    goto L2;
L2: ....
    ....
L1: ....
```

$\Rightarrow$

```
int goto_L2=0;
int goto_L1=0;
.....
if ( cond )
   goto_L1 = 1;
else
   goto_L2 = 1;
if ( ! goto_L1 )
   {
    goto_L2=0;
    ....
   }
goto_L1=0;
.....
```

Figure 5.1: Repeated pattern in `whetstone`

We now consider the new basic statements created. They include (i) the initialization, reinitialization and setting with the `goto` condition of the `goto` variable; (ii) the initialization and setting of the temporary variables used by the method; and (iii) the `break` statements used to exit loops and `switch` statements. Since the `goto`s and label statements are eliminated, this number is subtracted from the total number of new statements.

Note that the new statements created by the structurer should be consistent with the SIMPLE grammar. When performing an inward-movement transformation into a `while` or `if` statement, the condition of these statements is modified into a compound condition, as explained in Chapter 3. In this case, to adhere to the SIMPLE format, the compound expression should be simplified by transforming it into an `if-then-else` statement. Figure 5.2 illustrates this case for the `if` statement. Three new statements should be created, and a temporary variable defined to store the condition. The same applies to the `while`, but these statements should be included at the end of the `while` body as well, to evaluate the condition once more. In order to distinguish the number of statements created strictly by the structurer method, from the ones created to be consistent with SIMPLE, we present these results in two different columns.

Finally, let us consider the number of new variables created. These include the `goto` variables and the following temporary variables: (i) the ones created to adhere to the SIMPLE format; (ii) the ones created to save the `switch` condition; and (iii) the ones created to avoid an incorrect capture of a `break` or `continue`. One single variable is created for each label, to store the `goto` condition, regardless of the number of `goto`s associated with a label. Thus, as expected, the number of new variables per `goto` decreases when the number of `goto`s associated with a label is high. Two good example benchmarks to illustrate this fact are `par` and `indent` which have the highest ratio of `goto`s per label. They have the lowest ratio (less than one half) of new variables per `goto`.

```
if ( goto_L1 || expr )          ⇒   int temp_1;
   {                                ....
    ....                            temp_1 = goto_L1;
                                    if ( !temp_1 )
                                       { temp_1=(expr!=0);}
                                    if ( temp_1 )
                                       {
                                       ....
```

Figure 5.2: Transforming the new compound condition to the SIMPLE format

5.3.5 Comparing execution times for SIMPLE, GTE and GTE(opt)

Finally, let us consider the effect of structuring on execution time. Figure 5.6 contains the execution times for the three different versions of the programs (SIMPLE, GTE

name of benchmark	# of `gotos`	New Statements for GTE(opt)			New Variables for GTE(opt)
		if/do-while	basic	SIMPLE	
cq	1	1	1	0	1
nrcode2	2	2	3	0	2
lex.yy	4	5	11	0	4
frac	6	6	5	0	5
tomcatv	7	6	6	8	8
compress	9	15	15	0	6
FSM	12	15	10	8	8
asuite	22	24	28	16	25
indent	28	44	40	11	12
whetstone	31	18	32	12	34
par	59	101	35	28	18

Table 5.5: Detail of the new statements and variables for GTE(opt)

and GTE(opt)).

As expected, structuring programs with very few `goto` statements has very little impact on execution time. This is true, for example, 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 hand, the *FSM* benchmark which is an irreducible loop, is the other extreme. This has many overlapping `goto`-label pairs, and the ratio of `goto`s to total of statements is very high (if we consider only the function that implements the finite state machine, which contains all the `goto`s, and where the program spends most of its time, the ratio of `goto`s to the number of statements is one to two). Thus, we see that there is a significant performance impact with even the optimized GTE version executing significantly slower. In the next section we show some further experiments, where we observe that the order of goto-elimination considerably influences in the run time for this example.

For the other benchmarks, the difference registered in the execution times is not considerable. `Compress` and `frac` are the ones that follow `FSM`, considering differences in execution time. The first one has complex goto-usages with almost all the `goto`s concentrated in the same function. The second one contains simple goto-usages but the program is a single small function, with a high ratio of `goto`s per lines of source code.

The power of the optimizations is demonstrated by benchmarks such as `indent` with a significant difference between GTE and GTE(opt) versions of the program.

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, we can exploit structured representations for designing compilers while paying only a minimal penalty due to restructuring.

name of benchmark	time for SIMPLE	time for GTE	time for GTE(opt)	GTE(opt)/ SIMPLE
cq	0.04	0.04	0.04	1.00
nrcode2	10.75	10.78	10.75	1.00
lex.yy	1.47	1.51	1.52	1.03
frac	0.51	0.54	0.54	1.06
tomcatv	7.8	7.99	7.97	1.02
FSM	7.46	9.02	9.00	1.20
compress	1.33	1.55	1.51	1.14
asuite	11.41	11.69	11.68	1.02
indent	3.78	4.88	3.85	1.02
whetstone	46.49	48.2	47.96	1.03
par	2.09	2.16	2.14	1.02

Table 5.6: Execution times for SIMPLE, GTE and GTE(opt)

5.4 Studying different orderings of goto-elimination

In some programs with complex goto-usages, and where the `goto` density is very high, the order in which `goto`s are eliminated can cause a significant difference in the number of new `if` and `do-while` statements introduced, and hence cause a significant change in the running time of the structured program.

`FSM` is a good example to illustrate this fact. Its principal function which implements the finite state machine, has 24 lines of code, and 12 `goto`-label pairs. These `goto`-label pairs are all siblings and many of them overlap.

In this section, the results of the experiments performed with three different goto-elimination orderings for `FSM` using the GTE(opt) are presented.

Figure 5.3 illustrates the `FSM` source code and the intervals created by the `goto`-label pairs. To distinguish among the different `goto` statements associated with a single label, subscripts are used. Consider the interval graph for these intervals , as illustrated in Figure 5.4. From the interval graph, the maximum independent set can be calculated [Gav72]. A maximum independent set is a subset of the set of vertices of the graph, of maximum cardinality, such that no edge is adjacent to two vertices in this subset. This maximum independent set represents, in our case, the maximum set of `goto`-label pairs that do not overlap[2] with each other. If we eliminate the `goto`s corresponding to these intervals first, for these `goto`s, the number of transformations applied and new statement created are minimal. All these `goto`-label pairs are siblings, thus only goto-elimination transformations would be applied, and none of the transformations for these `goto`s is affected by a new statement introduced by a previous `goto`-elimination. Hence the possibility of reducing the number of new `if` and `do-while` statements in the structured program is high comparing to other possible orderings.

The structuring was tested by eliminating the `goto`s using the following orderings:

(i) the regular ordering of goto-elimination (reverse order of how they appear in the program)

(ii) the inverse to the regular ordering of goto-elimination (same order in which they appear in the program)

(iii) eliminating first the `goto`s corresponding to the maximum independent set of the interval graph, and eliminating the rest of the `gotos` in the same order as they appear in the program.

elimination ordering	Transformations				New stmt/var		Run times
	goto-elim.	out-mov.	inw-mov.	goto-lif.	`if/do-while`	var	
regular	10	5	4	0	17	8	9.00
inverse	12	40	3	1	26	9	12.4
MIS-first	10	13	2	0	17	7	9.20

Table 5.7: Comparing different goto-elimination orderings for FSM

The results are presented in Table 5.7. First, we observe a remarkable difference between the results of the inverse-goto-elimination ordering and the other two. With

[2] We do not consider as overlapping the case where a `goto`-label pair is completely contained in another.

```
start:
  x = fgetc(in)
  if ( x=='a') goto a1
  if ( x=='b')  goto b1
  goto fail1
a:
  x = fgetc(in)
  if ( x=='b') goto b2
  if ( x=='c') goto c1
  goto fail2
b:
  x = fgetc(in)
  if ( x=='a') goto a2
  if ( x=='c') goto c2
  goto fail3
c:
  x = fgetc(in)
  if ( x=='d') goto start
  if ( x=='e') goto accept
  goto fail4

fail:
 return "FAIL"
accept:
 return "ACCEPT"
```

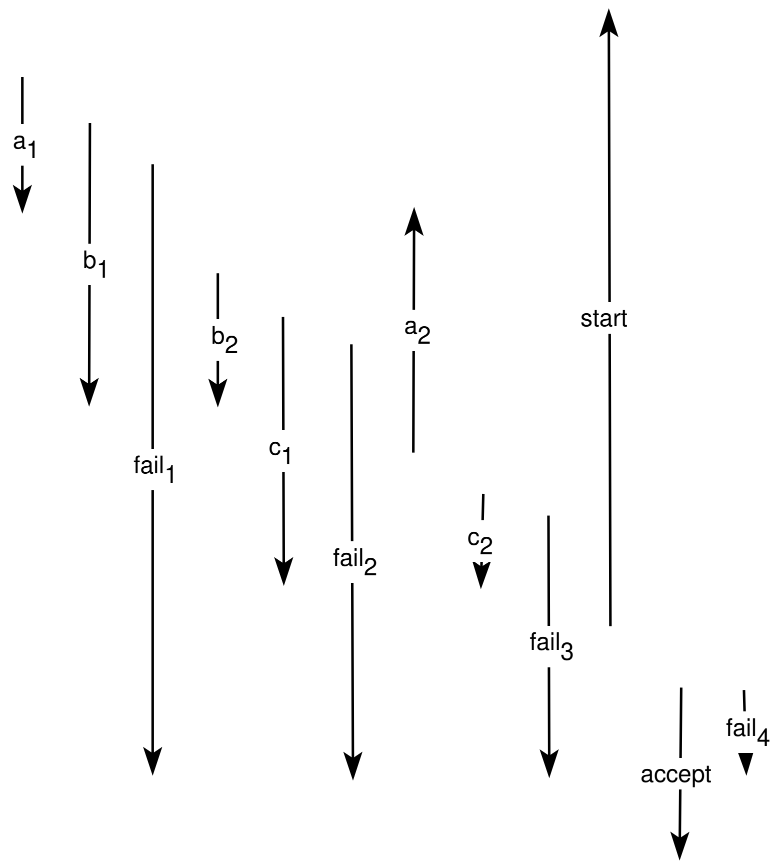


Figure 5.3: `FSM` and its `goto`-label intervals

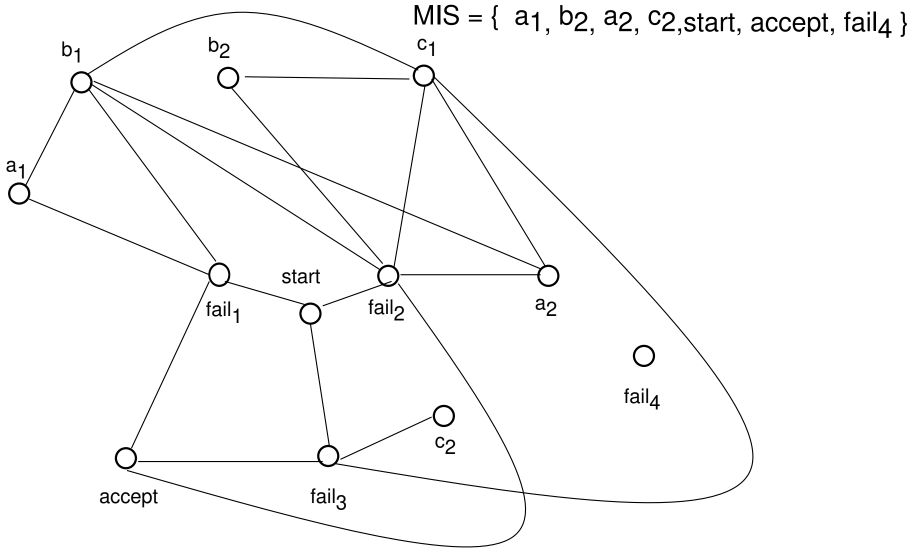


Figure 5.4: Interval graph and its Maximum Independent Set

this ordering 53% more `if` or `do-while` statements are created. Thus, the difference in execution time, as expected, is also significant.

For the other two versions, the results are similar. The number of transformations for the regular version is lower, however the number of new `if` and `do-while` statements is the same. For the regular version, more inward-movement transformations are performed, and then one more temporary variable is created. The difference between both versions in execution time is negligible.

We observe that the order of goto-elimination for the regular version happens to be very good. It eliminates almost all the `goto`s corresponding to the non overlapping `goto`-label pairs first, together with the `goto`s associated with the `fail` label. In this particular case, this ordering is a good choice. When the `goto` associated with the `start` label is eliminated, a `do-while` statement is introduced, and this captures three of the `goto fail` statements to be eliminated after. The `fail` label is the next statement after the new `do-while`. Thus, only a conditional `if` with the `break` statement is added to eliminate each of these `goto` statements associated with the `fail` label.

We performed the same experiment with a simpler finite state machine (with one state less) and the results of the maximum independent set ordering were much better compared to the results of the regular goto-elimination ordering.

It is difficult to state a general best goto-elimination ordering. It depends very much on the specific type of goto-usages in the program under consideration.

Regarding the FSM benchmark, eliminating the gotos in the inverse order in which they appear in the program seems a good choice. But that is not the case for other benchmarks. For example, for asuite, the pattern illustrated in Figure 5.5 is repeated several times in the program. In this case there are two goto-label pairs that are siblings. If the first goto in the program is eliminated first, an outward-movement transformation is required later. Whereas if the second goto is eliminated first, an inward-movement transformation is required later. Considering the number of if statements created, the inward-movement transformation is more expensive than the outward-movement transformation. The new compound expression for the if, created by the inward-movement transformation, should be simplified and therefore one extra conditional if is created. Thus in this case, to eliminate the gotos in the regular order is not the best choice.

However, it is difficult to calculate in advance the number and type of transformation to be applied, because after eliminating one goto the situation for the remaining gotos may change.

We conclude that it is hard to determine, in general, the best goto-elimination ordering. We believe it is an interesting problem to study.

```
{
 if (expr1) goto L1;
 ...
 goto L2;
L1: ...
L2: ...
}
```

(a) repeated pattern

```
{
 goto_L1=expr1;
 if (!goto_L1)
    {...
     goto_L2=1;
    }
 if (!goto_L2)
    {
     goto_L1=0;
     ...
    }
    goto_L2=0;
     ....
}
```

(b) eliminate goto L1 first

```
{
 goto_L1=expr1;
 if (!goto_L1)
    {...
     goto_L2=1;
    }
 if (goto_L1||!goto_L2)
    {
     goto_L1=0;
     ...
    }
L2:  goto_L2=0;
     ....
}
```

(c) eliminate goto L2 first

Figure 5.5: Different orderings of goto-elimination for asuite

Chapter 6

Related Work

One of the first approaches to structuring was given by Bohm and Jacopini [BJ66]. Their structuring method was done in the context of normalizing flowgraphs (where the flowgraph 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 structuring 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 be entered only at the beginning) [PKT73]. They present a graph algorithm to do such a transformation using a technique of node-splitting and they prove that the transformation is 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 [Wil77, WO78]. Ashcroft and Manna tackled the problem of structuring 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 [EM75].

All of the previous methods were intended to structure flowcharts. However, there have also been approaches suggested that are used to structure 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 structuring Fortran programs [Bak77] in order to make them more understandable. Since her goal was to obtain understandable Fortran programs, she only structures in situations where there is a clear possibility of the use of a structured construct and leaves some `goto`s 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 structuring of programs with a goal like ours. More recently, Cifuentes has presented an algorithm for program structuring in the context of decompilation [Cif93]. 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.

Mueller et. al. present a compiler back-end optimization method that attempts to eliminate unconditional branches, whether they originate from `goto`s or not [MW92]. The method eliminates almost all the unconditional branches by performing code duplication. It replaces each unconditional jump with the shortest possible sequence of instructions, to minimize growth in code size. It is implemented on a RTL intermediate representation in the early stages of the back-end phase, so that later optimizations can benefit of the simplified control flow. The results show the number of instructions executed is decreased, and also the total cache work is reduced (except for small caches), despite the increases in the code.

More relevant to our work are the structuring methods proposed by Allen et. al. for vectorizing compilers [AKPW83], and the work by Ammarguellat for parallelizing compilers [Amm92].

The first method was developed at Rice University [AKPW83] for a translator that converts Fortran sequential programs into equivalent Fortran vector programs. In vectorizing compilers, the dependence analysis performed is based on data dependence. Statement `S1` is dependant on statement `S2` if `S2` uses the value that `S1` has created. However, in the presence of complex control flow, data dependence is not sufficient to transform programs because of the introduction of control dependences. `S1` is control dependent on `S2`, when the outcome of a test in `S1` determines whether `S2` will be executed. Allen et. al. present the *IF-conversion* method that converts control dependences into data dependences by introducing logical variables to control the execution of the statements and eliminating `goto` statements. Figure 6.1 illustrates an example where a control dependence is transformed into a data dependence.

```
{
 for (i=1;i<100;i++)
  if ( a[i]<=0 )
        a[i]=b[i]+10;
}
```

$\Rightarrow$

```
{
 for (i=1;i<100;i++)
  { br1= a[i]<=0;
    if(br1)a[i]=b[i]+10;
  }
}
```

$\Rightarrow$

```
{
  br1[1:n]= a[1:n]<=0;
  if (br1[1:n])a[1:n]=b[1:n]+10;
}
```

Figure 6.1: Loop vectorization transformation via control dependence elimination

Although the goal of this work is not the same as ours, this method can also be used for structuring, and in fact has similar characteristics. Their method proceeds in three 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. Afterwards, branch removal takes place, removing all the forward branches. They do not eliminate backward branches.

Figures 6.2, 6.3 and 6.4 present three example programs in which *IF-conversion* and the *goto-elimination* methods are applied.

The first example program, Figure 6.2, illustrates an irreducible loop that presents a forward branch (`goto L2`), and a backward branch (`goto L1`). The *IF-conversion* method introduces two logical variables (`br1` and `br2`) to eliminate the forward branch. The branch flag, `br1`, is defined to be true if and only if the condition of the `goto` evaluates to true. This logical variable is used as a guard to all statements between the `goto` and the label, avoiding their execution when the guard is true. As the label associated with the `goto` is inside an iterative region (determined by `L1` and `goto L1`), the statements in this region are also executed when this backward jump is taken. An extra logical variable, a *branch back flag* (`bb1`) is needed to denote this case. The backward jump `goto L1` is not eliminated and the cycle of control-flow is not replaced by a structured loop. An extra transformation, not presented in the paper, should be applied to replace this backward branch by a `while` loop. Sometimes, as in the case of this example, a loop-carried dependence (on the variables `bb1` and `goto_L2`) might be introduced. If we intend to perform loop parallelization, this can be a negative factor, as the dependence may inhibit parallelization of the loop.

The second example program, Figure 6.3, presents two forward branches to be eliminated. Two logical variables `br1` and `br2` are created. For each of the statements between the `goto` and the matching label a guard is generated as disjoin or conjoin of these booleans variables, according to the conditions under which each statement should be executed. For this example we present two *goto-elimination* solutions: the first one eliminates the `goto`s in the regular order, while the second one in the order reverse to it. The similarities and differences between the second solution and the *IF-conversion* solution can be clearly observed. For *IF-conversion* a separate conditional is introduced to guard each of the first three statements. While the *goto-elimination* algorithm uses a single conditional to guard these three statements as a block. Further, the statements for which *IF-conversion* introduces compound conditionals, *goto-elimination* uses nested `if`s. However, for the last conditional created by

IF-conversion, the *simplification* of the compound conditional, would yield the same result as the *goto-elimination* method.

The third example program, Figure 6.4, illustrates an exit branch. In this case, all the statements in the loop, i.e. the statements both before and after the branch are affected by the guard. Thus, the do-loop, once entered, will run its full course, even when the exit flag is false and no real computation is being done. The authors expect that the speedup gained from vectorization will more than offset this inefficiency.

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

```
{
    br1=x;
    bb1=0;
L1: if (!br1 || (br1&&bb1))
       stmt_1;
L2: stmt_2;
    if (y)
       {
        bb1=1;
        goto L1;
       }
 }
```
(b) IF-conversion

```
{
 goto L2=x;
 do{
    if (!goto_L2)
        {
         goto_L1=0;
         stmt_1;
        }
    goto_L2=0;
    stmt_2;
    goto_L1=y;
   } while(goto_L1)
}
```
(c) goto-elimination

Figure 6.2: Irreducible loop example

The *IF-conversion* method is similar to ours in that both methods consist of step-by-step transformations applied to structured intermediate representations of the program, where each transformation produces a more structured code. The ideas of *branch relocation* and *branch removal* are somewhat similar to our concepts of *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 structure C programs (and thus treating `break`, `continue`, and `switch` statements) rather than just Fortran programs. Furthermore, we are interested in removing all gotos, not just those associated with forward branches. Another difference is the way 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 `goto`s and labels, and then we can directly modify the intermediate representation of the program. Their approach requires several passes through

```
for ( i=1; i<100 ;i++ )
{
   if (a[i]>10) goto L1;
   x=0;
   a[i]=a[i]+10;
   if (b[i]>10) goto L2;
   b[i]=x+10;
L1:a[i]=b[i]+a[i];
L2:b[i]=a[i]+5;
 }
```

(a) example program

```
for ( i=1; i<100 ; i++)
{
  br1=(a[i]>10);
  if (!br1)
       x=0;
  if (!br1)
       a[i]=a[i]+10;
  if (!br1)
       br2=b[i]>10;
  if ((!br1)&&(!br2))
       b[i]=x+10;
  if (br1||(!br1&&!br2))
       a[i]=b[i]+a[i];
  b[i]=a[i]+5;
}
```

(b) IF-conversion

```
for ( i=1; i<100; i++ )
{
 goto_L1=a[i]>10;
 if (!goto_L1)
    {x=0;
     a[i]=a[i]+10;
     goto_L2=b[i]>10;
    }
 if(!goto_L1||!goto_L2)
   {
    if(!goto_L1)
       b[i]=x+10;
    goto_L1=0;
    a[i]=b[i]+a[i];
   }
 goto_L2=0;
 b[i]=a[i]+5;
 }
```

goto-elimination
(c) eliminate `goto L1` first

```
for ( i=1; i<100; i++ )
{
 goto_L1=a[i]>10;
 if (!goto_L1)
    {x=0;
     a[i]=a[i]+10;
     goto_L2=b[i]>10;
     if(!gotoL2)
       {
         b[i]=x+10;
       }
    }
 if (!goto_L2)
    { goto_L1=0;
      a[i]=b[i]+a[i];
    }
 goto_L2=0;
 b[i]=a[i]+5;
 }
```

goto-elimination
(d) eliminate `goto L2` first

Figure 6.3: Forward branches example

```
{
   for (i=1;i<100;i++)
   {
    stmt_1;
    if x[i] goto L1;
    stmt_2;
   }
   stmt_3;
L1:stmt_4;
}
```

(a) exit branches

```
{
   ex1=1;
   for (i=1;i<100;i++)
    {
     if (ex1) stmt_1;
     if (ex1) ex1=!x[i];
     if (ex1) stmt_2;
    }
     br1= !ex1
     if (br1)
        stmt_3;
L1:stmt_4;
}
```

(b) IF-conversion

```
{
 for ( i=1;i<100;i++)
 {
  stmt_1;
  goto_L1=x[i];
  if (goto_L1) break;
  stmt_2;
  }
 if (!goto_L1)
    {
     stmt_3;
    }
 goto_L1=0;
 stmt_4;
}
```

(c) goto-elimination

Figure 6.4: Exit branches example

the program for the different stages of branch categorization, branch relocation and branch removal.

The method presented by Ammarguellat [Amm92], which she calls *control-flow normalization*, is the closest work in terms of the goals of structuring. That is, we both wish to fully structure programs in order to facilitate program analyses, program transformations and automatic parallelization. However, the intermediate representations that we structure are quite different. We structure a high-level representation of C programs that directly supports `break` and `continue`, while Ammarguellat structures 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 defines a continuation-based semantic language and transforms the syntactic constructs of the program into algebraic constructs. She converts the program into a system of simultaneous equations whose unknowns represent the continuations associated with the programs labels. A source continuation will contain the solution of the system after its resolution. By transformations applied to the system of equations: *precalculation, if distribution, factorization, derecursivation* and *substitution* and *elimination*, the system is solved. The quality of the normalized 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.

Figure 6.5 illustrates an example of an irreducible loop along with the result given by the *control-flow normalization* and our solution. Figure 6.6 illustrates another example program, and the results of applying the *goto-elimination*, the unoptimized and the optimized *control-flow normalization*. The comparison of these methods for these examples shows that the results are similar in that we both create new logical variables to store the 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 when 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 whereas Ammarguellat does not. Figure 6.7 illustrates an example. However, this loop will not execute the enclosed statements more times than in the original program, and it does not imply an increase in the execution time of the program.

(a) an irreducible loop

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

(b) control flow normalization

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

(c) goto-elimination

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

Figure 6.5: Irreducible loop example

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.

```
h(i,j,k)
{
   i=1;
L2:if (i>10) goto L3;
   j=1;
   k=1;
   i=j+k;
   goto L2;
L3:return(i);
 }
```

(a) example program

```
h(i,j,k)
 {
   i=1;
   do{
      goto_L2=0;
      goto_L3=(i>10);
      if (goto_L3) break;
      j=1;
      k=1;
      i=j+k;
      goto_L2=1;
     } while(goto_L2);
   goto_L3=0;
   return(i);
 }
```

(b) goto-elimination

```
h(i,j,k)
{
 i=1;
 pred2=(i>10);
 if(!pred2)
   {j=1;
    k=1;
    while(!pred1)
     {i=j+k;
      pred1=(i>10);
      if (!pred1)
         { j=1;
           k=1;
         }
     }
   }
  return(i);
}
```

(c.1) control-flow normalization

```
h(i,j,k)
{
 i=1;
 while(!pred1)
  { pred1=(i>10);
    if(!pred1)
      {j=1;
       k=1;
       i=j+k;
      }
  }
  return(i);
}
```

(c.2) optimized control-flow normalization

Figure 6.6: Example program

<table>
<tr>
<td>

```
{
    if (x) goto L2;
L1: stmt_1;
    goto L3;
L2: stmt_2;
    if (y)goto L1;
L3: stmt_3;
 }
```

</td>
<td>

```
{
  pred_162=x;
  if (pred_162)
   { stmt_2;
     pred_164=y;
   }
  if (!pred_162||pred_164)
      stmt_1;
  stmt_3;
 }
```

</td>
<td>

```
{
 goto_L2=x;
 do{
    if (!goto_L2)
     {goto_L1=0;
      stmt_1;
      goto_L3=1;
      if (goto_L3) break;
     }
    goto_L2=0;
    stmt_2;
    goto_L1=y;
   }while(goto_L1);
 goto_L3=0
 stmt_3;
 }
```

</td>
</tr>
<tr>
<td>(a) example program</td>
<td>(b) control-flow normalization</td>
<td>(c) goto-elimination</td>
</tr>
</table>

Figure 6.7: Transforming a program with no cycles

Chapter 7

Conclusions

In this thesis 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 present some optimizations to the method that avoid creating unnecessary new statements.

We completely implemented our method on the SIMPLE intermediate representation of the McCAT parallelizing/optimizing compiler, and we have presented experimental measurements for 11 benchmark programs using this implementation. It appears that most C programs use `goto` statements relatively sparsely and on such programs the structured programs have similar execution speeds as the original programs. Thus, the structuring does not have a performance penalty, while at the same time allows us to use structured analyses and transformations in the latter phases of the compiler. For programs that are very dense in `goto` statements there is some performance penalty. Experiments performed with different `goto` elimination orderings show a significant difference in the execution times of the resulting programs. It is hard to determine a general best goto-elimination ordering. It would be an interesting problem to study in the future.

We feel that a major advantage of our approach is that the structuring 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 structuring phase that can always eliminate `goto`s allows us to develop more efficient and simpler analyses and transformations in the remainder of the McCAT compiler.

Bibliography

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, Austin, Texas, January 24–26, 1983. ACM SIGACT and SIGPLAN.

[Amm92] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, corrected edition, 1988.

[Bak77] B. Baker. An algorithm for structuring flowgraphs. *Journal of the ACM*, 24(1):98–120, January 1977.

[BJ66] C. Bohm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[BM92] Robert A. Balance and Arthur B. Maccabe. Program dependence graphs for the rest of us. Technical report, The University of New Mexico, October 1992.

[CFR+91] 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, October 1991.

[Cif93] 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.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. New York: Academic Press, 1968.

[Don94] C. M. Donawa. A structured approach for the design and implementation of a backend for the McCAT C compiler. Master's thesis, McGill University, March 1994.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994. ACM SIGPLAN. *SIGPLAN Notices,* 29(6), June 1994.

[EM75] E.Ashcroft and Z. Manna. Translating programs schemas to while-shemas. *SIAM J. Comput*, 4(2):125–146, 1975.

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montréal, Québec, September 1993.

[Gav72] F. Gavril. Algorithms for minimal coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput*, 1(2):180–187, 1972.

[HDE+92] 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 Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[HGS92] L. J. Hendren, G. R. Gao, and V. C. Sreedhar. ALPHA: A dependence-based intermediate representation for an optimizing/parallelizing C compiler. ACAPS Technical Memo 49, School of Computer Science, McGill University, Montréal, Québec, November 1992.

[Hoa69] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12:576–583, 1969.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[JH94] Justiani and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In Peter A. Fritzson, editor, *Proceedings of the 5th International Conference on Compiler Construction, CC '94*, number 786 in Lecture Notes in Computer Science, pages 309–323, Edinburgh, Scotland, April 7–9, 1994. Springer-Verlag.

[Jus94] Justiani. An array dependence framework for the McCAT C compiler. Master's thesis, McGill University, expected December 1994.

[Knu74] D. E. Knuth. Structured programming with go to statements. *Computing Surveys*, pages 261–302, Dec 1974.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[MW92] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 322–330, San Francisco, California, June 17–19, 1992. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), July 1992.

[PKT73] W.W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat and exit statements. *Communications of the ACM*, 16(8):503–512, 1973.

[Smi91] P. D. Smith. Experiments with a very fast substring search algorithm. *Software — Practice and Experience*, 21(10):1065–1074, October 1991.

[Sre92] V. Sreedhar. Unnesting Nested Blocks in SIMPLE. McCAT ACAPS Design Note 7, McGill University, School of Computer Science, 1992.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.

[Weg76] P. Wegner. Programming languages - the first 25 years. *IEEE Transactions on Computers*, 19(12):1207–1225, December 1976.

[Wil77] M.H. Williams. Generating structured flow diagrams: The nature of unstructuredness. *Comput. J*, 20(1):45–50, 1977.

[WO78] M.H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *Comput. J*, 21(2):161–167, 1978.

Appendix A

Finite State Machine example program

This is the function that implements the Finite State Machine presented for our experiments.

```
fsm()
{
 int x;
 start:
   x = fgetc(in);
   if ( x=='a') goto a;
   if ( x=='b')  goto b;
a:

   x = fgetc(in);
   if ( x=='b') goto b;
   if ( x=='c') goto c;
   goto fail;
b:
   x = fgetc(in);
   if ( x=='a') goto a;
   if ( x=='c') goto c;
   goto fail;
c:
   x = fgetc(in);
```

```
   if ( x=='d') goto start;
   if ( x=='e') goto accept;
   goto fail;
fail:
 return "FAIL";
accept:
 return "ACCEPT";
 }
```

Appendix B

Lex Specifications

These are the regular expressions given in the `lex` specification, used to generate the output program `lex.yy.c`.

```
%%

"%"[^\n]* ;
"@unpublished{"[^@]* ;
"@book{"[^@]* ;
"@booklet{"[^@]* ;
"@inbook{"[^@]* ;
"@incollection{"[^@]* ;
"@manual{"[^@]* ;
"@phdthesis{"[^@]* ;
"@string{"[^@]* ;
"@techreport{"[^@]* ;
"@misc{"[^@]* ;
"@article" printf("%s***",yytext);
"@inproceedings"     { printf("%s***",yytext); }
[\n\t]     ;
. printf(yytext);
%%
```