A Control-Flow Normalization Algorithm and Its Complexity*

Zahira Ammarguellat
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois, 61801

July 17, 1992

Abstract

We present a simple method for normalizing the control-flow of programs to facilitate program transformations, program analysis, and automatic parallelization. While previous methods result in programs whose control flowgraphs are reducible, programs normalized by this technique satisfy a stronger condition than reducibility and are therefore simpler in their syntax and structure than with previous methods. In particular, all control-flow cycles are normalized into single-entry, single-exit while loops, and all `goto`'s are eliminated. Furthermore, the method avoids problems of code replication that are characteristic of node-splitting techniques. This restructuring obviates the control dependence graph, since afterwards control dependence relations are manifest in the syntax tree of the program. In this paper we present transformations that effect this normalization, and study the complexity of the method.

Index Terms: Continuations, control-flow, elimination algorithms, normalization, program transformations, reducibility, structured programs.

1 Introduction

The problem we are considering here is the normalization of the control-flow of programs with the goal of facilitating program transformations, program

*This work was supported in part by the **National Science Foundation** under Grant No. NSF MIP-8410110, the **U.S. Department of Energy** under Grant No. DE-FG02-85ER25001, the **Office of Naval Research** under Grant No. ONR N00014-88-K-0686, the **U.S. Air Force Office of Scientific Research** under Grant No. AFOSR-F49620-86-C-0136, and by a donation from the **IBM Corporation**.

analysis, and automatic parallelization. There are several ways in which our technique makes these processes easier.

First, it reduces the number of syntactic constructions that must be treated by the system of analysis or transformation. This lessens the complexity of many compilation algorithms, which are often driven by the structure of the program. If the program structure is highly regular, the number of cases and conditions that must be considered to analyze or parallelize the program is simply decreased.

Second, it converts all control-flow cycles into single-entry, single-exit `while` loops and eliminates all branching instructions. These may have dramatic consequences on compilation. For example, loops that contain exit branches are difficult to parallelize since the number of iterations they will perform is unknown prior to their execution. Likewise, `goto`'s may be used to create cycles of control-flow. These hidden loops may cause a loss of parallelism, both because the loops they describe are missed by techniques of parallelization that apply only to `do` loops, and also because the control-flow of other loops and sometimes entire subroutines may be disturbed by such cycles. Similarly the use of `goto`'s makes program transformations and analyses more difficult and inefficient in general, for the reason that the simple compositionality of control-flow is lost. In terms of semantics, we may say that in the absence of `goto`'s, a direct semantics may be given, whereas in their presence, a continuation semantics is needed. The increased complexity of a continuation semantics is translated to increased complexity in any program analysis that models the semantics, e.g., abstract interpretation or dataflow analysis.

Third, our technique obviates the control dependence graph. Much effort is spent in traditional parallelizing compilers in the treatment of *control dependences* [3]. We will argue that when the control-flow of a program is properly normalized, control dependence relations are so manifest in the syntax tree of the normalized program that there is no need for a separate representation of such dependences. Similarly, the method makes it unnecessary to perform interval analysis as part of dataflow analysis, because after normalization the internal structure of a program is obvious and trivial. It may therefore be used to simplify the implementation of dataflow analysis since pathological flowgraphs [38] will not exist.

Unnormalized programs may arise in several ways. First, unstructured programs can be written in languages such as Common Lisp, Fortran, Pascal or C. Second, the compiler itself may produce unstructured code when it applies classical program transformations such as *tail recursion elimination* [14]. For an example see figures 1 and 2. Other transformations such as *recursion splitting* [22] result in even more complex output than the *tail recursion elimination.* If we wish the compiler to work always with a normalized program, we may apply normalization following such transformations.

A lot of work has been done in the normalization of the control-flow of programs [10, 47, 12, 5]. All of these techniques result in programs with *reducible*

```
(defun f
  (lambda (x y)
    (begin
      (cond ((null? x) y)
            ((atom? x) y)
            (t (f (cdr x)
                  (f (car x) y)))))))
```

Figure 1: A sample of recursive Lisp program

```
(defun f
  (lambda (x y)
    (begin
      I1 (cond ((null? x) (set! r y) (go I2))
               ((atom? x) (set! r y) (go I2))
               (t (set! y (f (car x) y))
                  (set! x (cdr x))
                  (go I1)))
      I2 (return r))))
```

Figure 2: After tail recursion elimination

control flowgraphs. However, the method of normalization presented here results in programs which are more highly regular than those produced by these methods. In other words, a condition much stronger than reducibility is satisfied by the control flowgraphs that result from our method. Furthermore, while some previous methods of control-flow normalization result in excessive code replication, we will show below that code replication is necessary under our method only for the elimination of irreducibility, a condition that is rare even in unstructured programs.

2 Presentation of the Normalization Method

Our normalization works by transforming a program into a system of simultaneous equations, whose unknowns represent the *continuations* associated with program labels. The solution of this system of equations is the normalized form of the program. The effect of this is to detect all of the loops and to eliminate all pathological syntactic constructions that the program contains. Williams and Ossher [47] have proved that such an elimination is necessary and sufficient to obtain a structured form of the program. A theorem of Böhm and Jacopini [13] says that we may transform any program into another one where only the following three control structures are used:

- Assignment

- Conditional
- Iteration

However, their theorem is not constructive in that it does not give a method for deriving such a program. The technique presented below does exactly this in a simple and efficient way. Normalizing the control-flow of a program consists of transforming this program into another equivalent one where only the three foregoing control structures are used.

The input language we treat has a Lisp-like syntax and includes branching instructions (`goto`'s) and labels at the top-level of procedures. We applied this method to Fortran 77 and Le-Lisp [16] [6], and are applying it to Common Lisp, and C [25].

The output language contains single-entry, single-exit **`while`** loops but neither `goto`'s nor labels, and represents the normalized form of the program.

2.1 Denotational Semantics

The semantics of a programming language is a precise mathematical specification of the meaning of programs in the language [42, 45, 40]. The idea of this approach is to define functions which map syntactic constructs into algebraic ones. This method is based on Scott and Strachey's work and has been used to define languages like ALGOL 60 [31], PASCAL [45], and CLU [39]. Our input language contains branching instructions. Simple jumps make the semantics of programs more complex because we lose the compositionality of the semantics of commands. We must then switch from a *direct* semantics to a semantics with *continuations*. The theory of *continuations* was developed by C. Wadsworth and L. Morris independently. This notion originated from the "tail function" of Mazurkiewicz [30].

2.1.1 Continuations

Continuations are a powerful tool because they allow us to give a straightforward meaning to branches, exceptions and errors, and because they allow us to regain a degree of the compositionality we lose by having branches. The continuation of an instruction I is a function which, when applied to a store, gives the final result of the program when its execution begins with this instruction I. The result is a new store (state of memory). Usually *continuations* are associated with each point of a program. In our approach a point will represent a program label.

The author provides, for the interested reader, a denotational semantics with *continuations* for the primitive expressions of our language [7, 8]. The language is described in the following section.

2.2 Abstract Syntax

Cste	$\in$	**Cste**	*Constants*
Ide	$\in$	**Ide**	*Identifiers (variables)*
Cmd	$\in$	**Cmd**	*Commands or expressions*
lambda	$\in$	**Proc**	*Lambda expressions*

$$\begin{array}{lll}
\mathbf{Cmd} & : & Cste \\
 & \mid & Ide \\
 & \mid & (unop\ Cmd_1) \\
 & \mid & (biop\ Cmd_1\ Cmd_2) \\
 & \mid & (set!\ Ide\ Cmd) \\
 & \mid & (begin\ Cmd_1\ Cmd_2\ \ldots\ Cmd_n) \\
 & \mid & (while\ Cmd) \\
 & \mid & (if\ Cmd_1\ Cmd_2\ [Cmd_3]) \\
 & \mid & (go\ Ide) \\
 & \mid & (defun\ Ide\ (lambda\ (Ide_2\ \ldots\ Ide_n)\ Cmd)) \\
 & \mid & (Cmd_1\ Cmd_2\ \ldots\ Cmd_n)
\end{array}$$

where *unop* is a unary operator such as $-$, *not*, ... and *biop* is a binary operator such as `+`,`-`,`*`,`/`,`and`, `or`, ... For the purpose of simplicity, the grammar includes only scalar variables. The normalization method we are presenting has been applied to Fortran and Lisp programs [6, 24], and is by no means restricted to the treatment of scalar data; nor does it require restrictions upon side effects and aliasing.

2.3 Continuation Equations

For a procedure P we may give a syntactic representation to the continuation associated with each program label I_i. We will call it x_i.

Figure 3 illustrates a sample of a Lisp program. We obtain from this program the system of continuations equations of figure 4. x_0 is the source continuation (it contains the solution of the system after its resolution) and x_1, x_2, x_3, and x_4 are the continuations associated with the labels I_1, I_2, I_3, and I_4 respectively. x_0, x_1, x_2 x_3, and x_4 represent the unknowns of the system.

The only control structures used in these systems are the three proposed by Böhm and Jacopini [13]. The solution to this system will represent the normalized form of the procedure. In the next section we will present the method we choose to solve this system.

2.4 Gaussian Elimination-Like Resolution

A straightforward Gaussian elimination-like resolution method may be used to solve the system of continuation equations. This method yields a solution with

```
(defun h
  (lambda (i j k)
    (begin
     (set! i 1)
     I2 (if (> i 10) (go I3))
     I1 (set! j 1)
     (set! k 1)
     I4 (set! i (+ j k))
     (go I2)
     I3 i)))
```

Figure 3: Sample of Lisp program

$$\left\{\begin{array}{lcl} x_0 & = & (begin\ (set!\ i\ 1)\ x_2) \\ x_2 & = & (begin\ (if\ (>\ i\ 10)\ x_3\ x_1)) \\ x_1 & = & (begin\ (set!\ j\ 1)\ (set!\ k\ 1)\ x_4) \\ x_4 & = & (begin\ (set!\ i\ (+\ j\ k))\ x_2) \\ x_3 & = & (begin\ i) \end{array}\right.$$

Figure 4: Continuation equations of the program in figure 3

$O(n^3)$ complexity, where n is the number of equations (or unknowns). But the solution we are presenting is a refinement of this method. We will see later in this paper that this refinement improves the complexity considerably. The Gaussian elimination method consists of the substitution and the elimination of each unknown, where it appears in the system. We want to satisfy two criteria. The first is to *minimize* the code size; this is accomplished by *factorization.* The second is to convert every control-flow cycle into a *while* loop, and this is accomplished by *derecursivation.* Solving this system in the style of Gaussian elimination and performing these transformations along the way gives the normalized form of the program.

2.5 Transformations

If we look at the continuation equations we have built, we may remark that they contain no branches and have a regular structure. In general an equation has the following form:

$$x_i\ =\ (begin\ Cmd\ (if\ Exp_1\ (begin\ Cmd_1\ x_j)\ (begin\ Cmd_2\ x_k)))$$

The true and false parts of the `if` statement may contain other `if` statements, or the `if` statement may be absent altogether. Let us present below the transformations we use to solve the system.

2.5.1 Pre-calculation

This is in effect a sub-transformation useful for the transformations presented in the rest of the paper. It puts the boolean expression of a conditional statement into a new temporary variable. The temporary is defined once and only once in the program. For example, the following statement

$$x_i \;=\; (begin\ Cmd\ (if\ Exp\ (begin\ Cmd_1\ x_j)\ (begin\ Cmd_2\ x_k)))$$

will become, after pre-calculation,

$$\begin{aligned} x_i = (&begin\ Cmd\ (set!\ pred_1\ Exp)\\ &\quad (if\ pred_1\ (begin\ Cmd_1\ x_j)\\ &\qquad\qquad (begin\ Cmd_2\ x_k))) \end{aligned}$$

2.5.2 If Distribution

Like pre-calculation, this sub-transformation is useful for the transformations presented in the rest of the paper. It rewrites an `if` statement containing continuation variables into a sequence of two `if` statements. The first contains the condition that is captured in a temporary variable (by pre-calculation), and the second one contains only the continuation variables controlled by the temporary variable. The `if` condition is saved in a temporary variable, not only to avoid multiple computations of it, but also to guarantee a correct result, since its arguments may be affected by Cmd_i. As an example let us consider the equation x_i

$$x_i \;=\; (begin\ Cmd\ (if\ Exp\ (begin\ Cmd_1\ x_j)\ (begin\ Cmd_2\ x_k)))$$

after `if` distribution, the equation is

$$\begin{aligned} x_i = (&begin\ Cmd\ (set!\ pred_1\ Exp)\\ &\quad (if\ pred_1\ (begin\ Cmd_1)\\ &\qquad\qquad (begin\ Cmd_2))\\ &\quad (if\ pred_1\ x_j\ x_k)) \end{aligned}$$

2.5.3 Factorization

One unknown may appear several times in a single equation. To avoid increasing the code size during the elimination process, we factor this equation. Two different factorization may be used to factor an unknown: factorization with boolean expressions, and factorization with selector variables.

Factorization with Boolean Expressions

This consists of assembling all the conditionals (boolean expressions) governing one unknown (using `if` distribution) and replacing the equation by one in which the unknown appears only once. Consider a continuation equation whose form is:

$$x_i \;=\; (begin\ Cmd\ (if\ Exp\ (begin\ Cmd_1\ x_j)\ (begin\ Cmd_2\ x_j)))$$

x_j is guarded by the boolean expression $(or\ Exp\ (not\ Exp)) = true$. Then we can rewrite the equation:

$$\begin{aligned} x_i \;=\; & (begin\ Cmd\ (set!\ pred_1\ Exp)\ (if\ pred_1\ Cmd_1\ Cmd_2) \\ & \qquad\qquad\qquad\qquad (if\ (or\ pred_1\ (not\ pred_1))\ x_j\ x_j)). \end{aligned}$$

This is equivalent to

$$x_i \;=\; (begin\ Cmd\ (set!\ pred_1\ Exp)\ (if\ pred_1\ Cmd_1\ Cmd_2)\ x_j).$$

Now if we add another nesting level of `if` and another unknown x_k

$$\begin{aligned} x_i = (begin\ Cmd\ (if\ & Exp_1 \\ & (begin\ Cmd_1\ x_j) \\ & (begin\ (if\ Exp_2 \\ & \qquad\qquad (begin\ Cmd_2\ x_k) \\ & \qquad\qquad (begin\ Cmd_3\ x_j))))) \end{aligned}$$

x_j is guarded by $(or\ Exp_1\ (and\ (not\ Exp_1)\ (not\ Exp_2)))$ and x_k is guarded by $(and\ (not\ Exp_1)\ Exp_2)$. We can then rewrite the equation as:

$$\begin{aligned} x_i = \ (begin\ Cmd\ & (set!\ pred_1\ Exp_1) \\ & (if\ pred_1 \\ & \quad (begin\ Cmd_1) \\ & \quad (begin\ (set!\ pred_2\ Exp_2)\ (if\ pred_2 \\ & \qquad\qquad\qquad\qquad\qquad (begin\ Cmd_2) \\ & \qquad\qquad\qquad\qquad\qquad (begin\ Cmd_3)))) \\ & (if\ (and\ (not\ pred_1)\ pred_2)\ x_k\ x_j)) \end{aligned}$$

To build the selection tree

$$(if\ (and\ (not\ pred_1)\ pred_2)\ x_k\ x_j)$$

we order the unknowns appearing inside x_i in increasing order of number of appearances. In this case x_k appears once inside x_i but x_j appears twice. Thus x_k will be treated first in the selection tree. In case the number of appearances of all unknowns are the same, they are taken in order of their appearance inside the equation. This heuristic order allows us to create boolean expressions of manageable size in the case of several `if` nesting levels.

Factorization with Selector Expressions

This consists of replacing each unknown by assignment to a selector variable. A single, integer value is associated with each unknown and a `case` statement is added that sends control to the appropriate continuation. Consider the same continuation equation than above:

$$x_i \;=\; (begin\ Cmd\ (if\ Exp\ (begin\ Cmd_1\ x_j)\ (begin\ Cmd_2\ x_j)))$$

we add a variable *selector* and assign it a value for each unknown in the equation. x_j is replaced by $(set!\ selector\ 1)$. Then we can rewrite the equation:

$$\begin{aligned}x_i = (begin\ Cmd\ &(if\ Exp\ (begin\ Cmd_1\ (set!\ selector\ 1))\\ &\qquad\qquad (begin\ Cmd_2\ (set!\ selector\ 1)))\\ &(if\ (=selector\ 1)\ x_j)).\end{aligned}$$

Now if we add another nesting level of `if` and another unknown x_k

$$\begin{aligned}x_i = (begin\ Cmd\ (if\ &Exp_1\\ &(begin\ Cmd_1\ x_j)\\ &(begin\ (if\ Exp_2\\ &\qquad\qquad (begin\ Cmd_2\ x_k)\\ &\qquad\qquad (begin\ Cmd_3\ x_j)))))\end{aligned}$$

We rewrite the equation using the selector variable as follows:

$$\begin{aligned}x_i = (begin\ Cmd\ &(if\ Exp_1\ (begin\ Cmd_1\ (set!\ selector\ 1))\\ &\qquad (begin\ (if\ Exp_2\ (begin\ Cmd_2\ (set!\ selector\ 2))\\ &\qquad\qquad\qquad (begin\ Cmd_3\ (set!\ selector\ 1)))))\\ &(if\ (=selector 1)\ x_j\ (if\ (=\ selector\ 2) x_k)))\end{aligned}$$

Of course, the order in which we place the unknowns in the selection tree is irrelevant, since the guard is in any case simply a test for the value of the selector. To compare these two methods and to measure their respective cost, we have run an experiment that normalizes the scientific computations of the *Perfect Club* [34]. See figures 24 and 25 in section 5.

2.5.4 Derecursivation

Derecursivation, like T_1 Hecht's transformation [20], consists of making a loop explicit. A self-recursive equation has this form:

$$x_i \;=\; (begin\ Cmd\ (if\ Exp\ (begin\ Cmd_1\ x_i)\ (begin\ Cmd_2\ x_j)))$$

Intuitively this equation is a loop whose entry is the instruction labeled by x_i and whose condition of iteration is the condition that leads to x_i. In this case we iterate the loop until *Exp* becomes false, in which case we perform the begin form associated with the false part of the `if`.

The fixed point of the above equation is:

```
x_i = (begin (while
          (begin Cmd (set! pred_1 Exp)
                 (if pred_1 Cmd_1 Cmd_2)
                 pred_1))
          x_j)
```

The semantics of the `while` statement is to iterate the `begin` form contained inside the `while` form until $pred_1$ is false. It has the semantics of a `repeat`; therefore, it supposes that the loop body is performed at least once.

Consider an equation with another level of `if` nesting

```
x_i = (begin Cmd_1 (if Exp_1
                       (begin Cmd_2 x_j)
                       (begin (if Exp_2
                                  (begin Cmd_3 x_k)
                                  (begin Cmd_4 x_i)))))
```

Again here we must study the boolean expressions guarding each continuation inside the equation. x_j is guarded by Exp_1, x_k is guarded by $(and\ (not\ Exp_1)\ Exp_2)$ and finally x_i is guarded by $(and\ (not\ Exp_1)\ (not\ Exp_2))$. The latter condition represents the exit condition of the *while* loop induced by the continuation x_i:

```
x_i = (begin (while (begin Cmd_1 (set! pred_1 Exp_1)
                    (if pred_1 Cmd_2
                               (begin (set! pred_2 Exp_2)
                                      (if pred_2 Cmd_3 Cmd_4)))
                    (and (not pred_1) (not pred_2))))
          (if pred_1 x_j (begin (if pred_2 x_k))))
```

2.5.5 Substitution and Elimination (T_2')

T_2', like T_2 [20], consists of substituting an unknown in the continuations system and of eliminating its equation from this system. However, Hecht [20] performs T_2 on unknowns that have only a single use in the system. The T_2' transformation is not constrained by this condition since it replaces an unknown anywhere it appears in the system (i.e., an unknown may be replaced in several different equations). T_2' is performed after we have checked that the unknown we are eliminating is non-recursive (which in this case should be derecursivated first) and that the equations where it appears are factorized (to avoid several substitutions of the same unknown in a single equation). If our system is

$$\begin{cases} x_i &= (begin\ Cmd_1\ (if\ Exp_1\ (begin\ Cmd_2\ x_j)\ (begin\ Cmd_3\ x_k))) \\ x_j &= (begin\ Cmd_4\ (if\ Exp_2\ (begin\ Cmd_5\ x_k)\ (begin\ Cmd_6\ x_l))) \\ \ldots \end{cases}$$

$$\begin{cases} x_0 &= (begin\ (set!\ i\ 1)\ (if\ (>\ i\ 10)\ x_3\ x_1)) \\ x_1 &= (begin\ (set!\ j\ 1)\ (set!\ k\ 1)\ x_4) \\ x_4 &= (begin\ (set!\ i\ (+\ j\ k))\ (if\ (>\ i\ 10)\ x_3\ x_1)) \\ x_3 &= (begin\ i) \end{cases}$$

Figure 5: Elimination of x_2 from the system in figure 4

then after the substitution of x_j in x_i and its elimination from the system, we will obtain the following equivalent system.

$$\begin{cases} x_i = (begin\ Cmd_1 \\ \qquad (if\ Exp_1 \\ \qquad\quad (begin\ Cmd_2\ Cmd_4\ (if\ Exp_2\ (begin\ Cmd_5\ x_k) \\ \qquad\qquad\qquad\qquad\qquad\qquad (begin\ Cmd_6\ x_l))) \\ \qquad\quad (begin\ Cmd_3\ x_k))) \\ \ldots . \end{cases}$$

Each application of the T_2' transformation reduces the number n of unknowns in the system, into $n-1$.

2.6 Example of Resolution

The system of continuation equations resulting from the program in figure 3 is presented in figure 4. To solve this system we use the transformations presented above. We may begin the resolution by eliminating x_2, which appears in the equation of x_0 and x_4. The equivalent system after its elimination is in figure 5. The next step may be to eliminate x_1 from the resulting system. The equivalent system is in figure 6. Since x_4 is recursive, we want to apply the derecursivation transformation to create the corresponding `while` loop and the elimination transformation to replace and eliminate the unknown from the system. The resultant systems are in figure 7 and figure 8, respectively. At this point, x_3 appears twice in x_0; we then need to factor x_0. The resultant system is shown in figure 9. And finally, after the substitution and elimination of x_3, we obtain the normalized form in figure 10.

2.7 Structure of the Original Program

When we build the continuation equations, we attempt to preserve as much as possible the original structure of the program. That is to say, if the input procedure contains control constructs that are already normalized (by normalized we mean that there are neither branching instructions leaving this construct, nor branching instructions entering it), the construct is preserved and not renormalized. In figure 11, the `while` loop is already normalized. The associated

$$\left\{\begin{array}{l}
x_0 = (\mathit{begin}\ (\mathit{set!}\ i\ 1)\ (\mathit{if}\ (>\ i\ 10)\ x_3\ (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)\ x_4)))\\
x_4 = (\mathit{begin}\ (\mathit{set!}\ i\ (+\ j\ k))\\
\qquad\qquad (\mathit{if}\ (>\ i\ 10)\ x_3\\
\qquad\qquad\quad (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)\ x_4)))\\
x_3 = (\mathit{begin}\ i)
\end{array}\right.$$

Figure 6: Elimination of x_1 from the system in figure 5

$$\left\{\begin{array}{lcl}
x_0 & = & (\mathit{begin}\ (\mathit{set!}\ i\ 1)\ (\mathit{if}\ (>\ i\ 10)\ x_3\ (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)\ x_4)))\\
x_4 & = & (\mathit{begin}\ (\mathit{while}\ (\mathit{begin}\ (\mathit{set!}\ i\ (+\ j\ k))\ (\mathit{set!}\ \mathit{pred}_1\ (>\ i\ 10))\\
 & & \qquad\qquad\qquad\qquad (\mathit{if}\ (\mathit{not}\ \mathit{pred}_1)\ (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)))\\
 & & \qquad\qquad\qquad\qquad\ (\mathit{not}\ \mathit{pred}_1)))\ x_3)\\
x_3 & = & (\mathit{begin}\ i)
\end{array}\right.$$

Figure 7: Derecursivation of x_4 from the system in figure 6

$$\left\{\begin{array}{lcl}
x_0 & = & (\mathit{begin}\ (\mathit{set!}\ i\ 1)\\
 & & \qquad\quad (\mathit{if}\ (>\ i\ 10)\ x_3\\
 & & \qquad\quad (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)\\
 & & \qquad\quad (\mathit{begin}\ (\mathit{while}\ (\mathit{begin}\ (\mathit{set!}\ i\ (+\ j\ k))\ (\mathit{set!}\ \mathit{pred}_1\ (>\ i\ 10))\\
 & & \qquad\qquad\qquad\qquad\quad (\mathit{if}\ (\mathit{not}\ \mathit{pred}_1)\ (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)))\\
 & & \qquad\qquad\qquad\qquad\quad (\mathit{not}\ \mathit{pred}_1)))\ x_3))))\\
x_3 & = & (\mathit{begin}\ i)
\end{array}\right.$$

Figure 8: Elimination of x_4 from the system in figure 7

$$\left\{\begin{array}{lcl}
x_0 & = & (\mathit{begin}\ (\mathit{set!}\ i\ 1)\ (\mathit{set!}\ \mathit{pred}_2\ (>\ i\ 10))\\
 & & \qquad\quad (\mathit{if}\ (\mathit{not}\ \mathit{pred}_2)\\
 & & \qquad\qquad (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)\\
 & & \qquad\qquad\quad (\mathit{while}\ (\mathit{begin}\ (\mathit{set!}\ i\ (+\ j\ k))\ (\mathit{set!}\ \mathit{pred}_1\ (>\ i\ 10))\\
 & & \qquad\qquad\qquad (\mathit{if}\ (\mathit{not}\ \mathit{pred}_1)\ (\mathit{begin}\ (\mathit{set!}\ j\ 1)\ (\mathit{set!}\ k\ 1)))\ (\mathit{not}\ \mathit{pred}_1)))))\\
 & & x_3)\\
x_3 & = & (\mathit{begin}\ i)
\end{array}\right.$$

Figure 9: Factorization of x_0 from the system in figure 8

```
(defun h
  (lambda (i j k)
    (begin
     (set! i 1)
     (set! pred₂ (> i 10))
     (if (not pred₂)
     (begin
      (set! j 1)
      (set! k 1)
      (while
          (begin
           (set! i (+ j k))
           (set! pred₁ (> i 10))
           (if (not pred₁)
           (begin (set! j 1) (set! k 1)))
           (not pred₁)))
      i)))))
```

Figure 10: Normalized form of the program in figure 3

continuation equations system is in figure 12. The `while` is treated as if it were an assignment statement. Now, if we look at the program in figure 13, the `while` loop has an exit from inside it. In this case, the `while` loop is first rewritten in terms of `goto`'s and `if`'s as shown in figure 14, and then converted into a normalized form.

3 Order of Resolution

In the system of figure 4 above, we eliminated the unknowns in an arbitrary order. But it is easy to see that the quality of the normalized form of the program, in terms of code size, depends upon the order in which the unknowns are eliminated from the system. To give an idea of the importance of this order, let us take the previous system of equations in figure 4 and try to eliminate the unknowns in a different order. Let us choose the following order of resolution: (x_1, x_4, x_2, x_3). The system after elimination of x_1 is in figure 15. The variable x_3, and x_4 now occur in the equation for x_2. The elimination of x_4 leads to the equivalent system in figure 16. x_2 is recursive; after its derecursivation and substitution we obtain the system in figure 17. And finally after the elimination of x_3, we obtain the normalized program in figure 18.

Of course the programs in figures 10 and 18 are semantically equivalent, but the second one does not contain any replicated code and the body of the `while` loop is restrained strictly to the code dependent upon it; i.e., no code is inside the loop body that does not need to be.

```
(defun f
  (lambda (j i a b n)
    (begin
     (set! j 0)
     I₁ (set! i 1)
     (while (begin
         (set! a (+ b j))
         (set! j (+ j 2))
         (set! i (+ i 1))
         (< i n)))
     (set! b (+ j 1))
     (go I₁))))
```

Figure 11: Program containing a normalized control structure

$$\left\{\begin{array}{lcl} x_0 & = & (begin\ (set!\ j\ 0)\ x_1) \\ x_1 & = & (begin\ (set!\ i\ 1)\ (while\ (begin\ (set!\ a\ (+\ b\ j))\ (set!\ j\ (+\ j\ 2)) \\ & & \qquad (set!\ i\ (+\ i\ 1))\ (<\ i\ n)))\ (set!\ b\ (+\ j\ 1))\ x_1) \end{array}\right.$$

Figure 12: Continuation equations system of the program in figure 11

```
(defun g
  (lambda (j i a b n)
    (begin
     (set! j 0)
     I₁ (set! i 1)
     (while (begin
         (set! a (+ b j))
         (set! j (+ j 2))
         (if (< j 0) (go I₂))
         (set! j (+ j 1))
         (< i n)))
     I₂ (set! b (+ j 1))
     (go I₁))))
```

Figure 13: Program containing a non-normalized control structure

```
(defun g
  (lambda (j i a b n)
    (begin
     (set! j 0)
     I1 (set! i 1)
     (set! g 1)
     G1 (set! a (+ b j))
     (set! j (+ j 2))
     (if (< j 0) (go I2))
     (set! j (+ j 1))
     (set! g (+ g 1))
     (if (< g n) (go G1))
     I2 (set! b (+ j 1))
     (go I1))))
```

Figure 14: Equivalent form of the program in figure 13

$$\begin{cases} x_0 &= (begin\ (set!\ i\ 1)\ x_2) \\ x_2 &= (begin\ (if\ (>\ i\ 10)\ x_3\ (begin\ (set!\ j\ 1)\ (set!\ k\ 1)\ x_4))) \\ x_4 &= (begin\ (set!\ i\ (+\ j\ k))\ x_2) \\ x_3 &= (begin\ i) \end{cases}$$

Figure 15: Elimination of x_1 from the system in figure 4

$$\begin{cases} x_0 &= (begin\ (set!\ i\ 1)\ x_2) \\ x_2 &= (begin\ (if\ (>\ i\ 10)\ x_3 \\ & \qquad (begin\ (set!\ j\ 1)\ (set!\ k\ 1)\ (set!\ i\ (+\ j\ k))\ x_2))) \\ x_3 &= (begin\ i) \end{cases}$$

Figure 16: Elimination of x_4 from the system in figure 15

$$\begin{cases} x_0 &= (begin\ (set!\ i\ 1)\ (begin\ (while\ (begin\ (set!\ pred_1\ (>\ i\ 10)) \\ & \qquad\qquad (if\ (not\ pred_1)\ (begin\ (set!\ j\ 1)\ (set!\ k\ 1)\ (set!\ i\ (+\ j\ k)))) \\ & \qquad\qquad (not\ pred_1)))\ x_3)) \\ x_3 &= (begin\ i) \end{cases}$$

Figure 17: Elimination of x_2 from the system in figure 16

```
(defun h
  (lambda (i j k)
    (begin
     (set! i 1)
     (while (begin
         (set! $pred_1$ (> i 10))
         (if (not $pred_1$)
         (begin (set! j 1)
            (set! k 1)
            (set! i (+ j k))))
         (not $pred_1$)))
     i)))
```

Figure 18: Second normalized form of the program in figure 3

We observe from this that the order of resolution has an important impact on the running time of the resolution process and on the code replication. The order of resolution we propose is as follows: first, sort in an extended *topological* order the nodes of the graph associated with the equations, and second, move the loop headers in this order so that they appear after the unknowns representing their bodies. Note that our algorithm is slightly different from the classical method of interval analysis. The normalization method makes use of the strongly connected component, and of the topological sort of the acyclic continuations flowgraph to solve the system of continuations. In the next sections we describe the elimination order of the unknowns.

3.1 Graph and Topological Sort

The graph $G = (N, E)$ associated with the system of equations represents the control flowgraph of the program and is defined by taking the nodes as the unknowns of the system and by creating an edge (x_i,x_j) when the unknown x_j appears inside the equation x_i. $|N| = n$ and $|E| = e$ are the number of nodes and edges. For example, the graph associated with the system of equations in figure 4 is in figure 19.

A *topological ordering* of the nodes of such a graph is a labeling of the nodes with integers 1,2,..., n [36]. Of course this graph must be acyclic for this ordering to be meaningful. Since, in general, our input graphs contain cycles, we order the nodes by first eliminating the back and cross edges of the graph and sorting the resulting graph in a topological order. We call this the *extended* topological sort of the graph. By visiting the graph nodes in a depth-first order, the complexity of the combined algorithm is kept to $O(n + e)$.

In figure 19, x_2 is a single loop header and its body consists of the unknowns x_1 and x_4. An extended topological order is (x_0,x_2,x_1,x_4,x_3). The next step

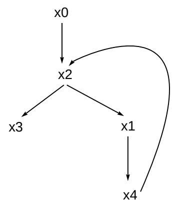

Figure 19: Graph associated with the system in figure 4

now is to reorder the loop headers. x_2 must appear after x_1 and x_4 (the two unknowns in its body). The order of resolution (x_0,x_1,x_4,x_2,x_3) we have used in the previous example follows this order.

3.2 Algorithm to Find the Order of Resolution

Input: A list H of loop headers
A list b_h of labels in the body of each loop header h
A list L of unknowns ordered in extended topological order

Output: Ordered list of unknowns.

```
For each unknown $x_i$ in $L$
If $x_i \in H$
Then
    • Delete $x_i$ from $L$ and insert it immediately after all the
    unknowns constituting the body of the loop designated by
    $x_i$. These unknowns are given by $b_{x_i}$.
```

3.3 Algorithm for Resolution of the System

Input: Ordered list of unknowns L

Output: Source label whose equation represents the normalized form of the program.

```
1. Select the first unknown $x_i$ from $L$
2. If $x_i$ is self-recursive
   Then
       • Derecursivate $x_i$ (apply $T_1$)
   For each $x_j$ such that $x_i$ is an element of the unknown set of
   $x_j$
       • Factorize the occurrences of $x_i$ in $x_j$
       • Substitute $x_i$ in $x_j$ (apply $T_2'$)
3. If $L$ non-empty goto 1.
```

3.4 Example

Let us consider now the irreducible program in figure 20. We do not give the details of all the transformations performed on the continuation equations system but present only the system in figure 21 and the associated continuation graph in figure ??. There are two nested loops in this graph whose headers are x_1 and x_2. The order of resolution is (x_0,x_3,x_g,x_2,x_1,x_4,x_5). The normalized form relative to this order is in figure 23.

```
(defun g
  (lambda (i j x)
    (begin
     (set! i 1)
     (set! j i)
     I₁ (if (> i 0) (go I₄))
     I₂ (set! x i)
     I₃ (set! i (+ i 1))
     (if (> x i)
     (begin (if (> x j) (go I₂))
        (go I₁))
       (go I₅))
     I₄ (set! j 1)
     I₅ (set! j 2))))
```

Figure 20: Irreducible program

$$\left\{\begin{array}{lcl}
x_0 & = & (begin\ (set!\ i\ 1)\ (set!\ j\ i)\ x_1) \\
x_1 & = & (begin\ (if\ (>\ i\ 0)\ x_4\ x_2)) \\
x_2 & = & (begin\ (set!\ x\ i)\ x_3) \\
x_3 & = & (begin\ (set!\ i\ (+\ i\ 1))\ (if\ (>\ x\ i)\ x_g\ x_5)) \\
x_g & = & (begin\ (if\ (>\ x\ j)\ x_2\ x_1)) \\
x_4 & = & (begin\ (set!\ j\ 1)\ x_5) \\
x_5 & = & (begin\ (set!\ j\ 2))
\end{array}\right.$$

Figure 21: Continuation equations of the program in figure 20

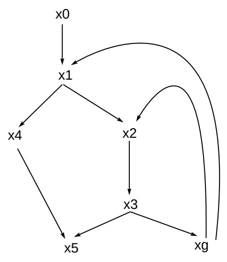

Figure 22: Continuation graph of the system in figure 21

```
(defun g
  (lambda (i j x)
    (begin
     (set! i 1)
     (set! j i)
     (while (begin
         (set! pred₁ (> i 0))
         (if (not pred₁)
         (while
             (begin
              (set! x i)
              (set! i (+ i 1))
              (set! pred₂ (> x i))
              (if pred₂ (begin (set! pred₃ (> x j))))
              (and pred₂ pred₃))))
         (and (not pred₁) (and pred₂ (not pred₃)))))
     (if (and pred₂ (or pred₁ (not pred₃)))
     (begin (set! j 1)))
     (set! j 2))))
```

Figure 23: Normalized form of the irreducible program in figure 20

4 Complexity

In this section, we will analyze the complexity of the normalization method by counting the number of transformations necessary to solve a system of continuations equations. A complete time - and space - complexity study would involve us in details of data structures that are beyond the scope of this paper. However, the analysis below reveals the most important aspects of the complexity of the normalization algorithm and can be extended to a complete one by considering the cost of applying individual transformations to a chosen representation of the program. Let S be a system of continuation equations and $G = (N, E)$ the graph associated with S. $i \in N$. Let $|N| = n$ and $|E| = e$.

In_i represents the set of nodes entering the node i or, alternatively, the set of unknowns in whose equation i appears.

$d(j, i)$ is the degree of multiplicity of the edge (j, i), i.e., the number of times i appears in the equation of j.

n_s is the total number of substitutions performed during the resolution process; n_f represents the number of factorizations, and n_d the number of derecursivations.

4.1 Number of Substitutions

We will show that the number of substitutions performed during resolution does not exceed the number of unknowns n in the system, in the case that G is reducible.

Theorem 1 *G reducible $\Rightarrow$ each equation is substituted only once in the system S.*

Proof: Let $In_i = \{k \ / \ (k,i) \in E\}$

This means that i appears (at least once) in the equation of k. We want to substitute the unknown i in the system. As we have seen in the algorithm, before any substitution of an equation of i into k, k is factorized so that i occurs only once in the equation of k. Several cases may appear:

- $|In_i| = 1$ Then i is substituted for only once in the system.
- $|In_i| > 1$

  In this case i appears in several equations of the system. It may be inside a loop whose header is h, or may itself represent the header of a loop[1], or finally may be neither of these two cases. Let s be the source node of the graph.

  > **Case 1**: The simplest case is when i has several predecessors and is neither a loop header, nor inside a loop. In accordance with the resolution order we have chosen, the predecessors of i will be treated before i itself. When we come to substitute i, it will appear in the equation of its nearest *dominator* h [2]. After factorization of h, i will appear only once in h, and will be substituted for only once.
  >
  > **Case 2**: i is in a loop whose header is h but i and h are distinct. In this case, by the order we have chosen and the fact that G is reducible, all of the predecessors of i in the loop (except for h) will be treated before i itself. When we come to substitute i, it will appear only in the equation of h by the fact that h dominates i. And after the factorization of h, i will appear only once in the system and will therefore be replaced only once by substitution.
  >
  > **Case 3**: i is a loop header. According to the resolution order we have described earlier, we treat every node inside the loop before the header. Let h be the header of the innermost loop containing the i loop (or s, if there is no such loop). When we treat i, it will therefore be a self-recursive variable that appears

[1] Since G is reducible, for every control-flow cycle of G there is a unique loop header that dominates every node in the body of the loop.

also in the equation of h (or s). After derecursivation of i and factorization of h, i is substituted for only once in the system.

Since we have n equations in the system,

$$n_s = n.$$

□

In an irreducible graph, our method may replicate code. In the worst case, without any preliminary factorization, the number of substitutions for the unknown i will be bounded by the number of its predecessors.

Theorem 2 *G irreducible $\Rightarrow$ each unknown is substituted at most n times.*

Proof: In the first step of the elimination the first unknown is substituted in the worst case, $n-1$ times. The second time, it is substituted $n-2$ times, since after the first substitution one unknown is definitely eliminated from the system and so on, until we eliminate all the unknowns. The number of substitution is then

$$n_s \leq \sum_{i=1}^{n-1} n - i = n(n-1)/2 = O(n^2).$$

□

Note that this bound is very conservative, as it assumes that the control flowgraph of the program is a clique!

4.2 Number of Factorizations

If we look at the algorithm of resolution above in section 3.3, we see that a factorization is applied to each predecessor x_j of x_i before substituting x_i. The factorization is of course unnecessary if x_i occurs only once in an equation. It must be noted that applying one single step of factorization on each x_j has the effect of reducing the instances of all the occurrences in x_j to one (the degree of each edge between x_j and every occurrence of x_j is 1). The number of instances of a variable inside an equation is bounded by n, and a single factorization step is able to reduce these instances to one. In the worst case a factorization is necessary before every substitution; therefore $n_f \leq n_s$.

4.3 Number of Derecursivations

A derecursivation is performed every time a node has itself as an occurrence. The number of derecursivation depends then upon the number of loop headers in the entire graph. In the worst case, every node in the graph is a loop header; then

$$n_d \leq n.$$

Again this is a very conservative bound.

In summary, the number of transformations performed in the case of a reducible flowgraph is on the order of the number of continuation equations in the system. Of course, in this study, the complexity does not take into account the size of the continuation equations, nor the time necessary during substitution, to find within the equation the unknown for which to substitute, nor in factorization the cost of collecting and simplifying boolean conditions.

5 Application of the Normalization Process

The normalization method that we have presented has been put to several uses in projects with which the author has been involved: PAF [44] and MIPRAC [24].

PAF is an experimental Fortran parallelizer developed at the University of Paris 6 (France). In PAF the normalization method has two applications. The first is to convert every cycle (explicit or implicit) into `while` loops, which are themselves transformed into `do` loops whenever possible [9], and finally into `doall` loops when the dependences permit it. In other words, even when a programmer writes loops using `goto`'s, or writes unstructured `do` loops, they are made eligible for parallelization by normalization. The second purpose is vectorization. In order to vectorize statements that are conditionally executed in a loop, one must attach boolean variables (mode vectors) to the statements [29, 48, 17, 32]. By control-flow normalization we may accomplish this easily for any program.

MIPRAC is a multilingual compiler for shared memory machines being implemented at the University of Illinois. Its applications of the normalization method are much more ambitious.

First, it allows us to write a genuinely multilingual compiler. MIPRAC accepts programs in Common Lisp, C, Scheme, and Fortran. Together these four languages contain many control structures, such as `do`, `dolist`, `dotime`, `loop`, `cond`, `block`, `return`, `break`, `case`, `switch`, `exit`, `for`, `while`, `goto`, `continue`, `if`, and `pause`. After normalization only three control-structures remain: `begin`, `if`, and `while`. Moreover, the intermediate form is properly structured. In other words, multilinguality in MIPRAC means that programs from various languages all have a simple, structured representation in MIPRAC's intermediate form, and this is accomplished by normalization.

The second application is to simplify program analysis. In effect, normalization allows programs that require a continuation semantics (because of `goto`'s) to be converted into programs that may be given a direct semantics. For example, a program with `goto`'s might have as its meaning a function of type $(Store \rightarrow Store) \rightarrow (Store \rightarrow Store)$. After normalization the same program would have as its meaning a function of type $Store \rightarrow Store$. This simplification shows up when we write an analysis of the program as well. Whereas an abstract interpretation of the unnormalized program might be a function of type $(\hat{Store} \rightarrow \hat{Store}) \rightarrow (\hat{Store} \rightarrow \hat{Store})$ (where $\hat{Store}$ is an abstraction of memory), an abstract interpretation of the normalized program could be a function of type $\hat{Store} \rightarrow \hat{Store}$. This makes the analysis simpler to implement and more efficient. The reader may contrast the interprocedural analysis, where continuations are used [22], to that where direct semantics are applied to programs whose procedures bodies are normalized [23].

The third application is to simplify program transformations. When restructuring an expression, we do not need to be concerned with branches into the middle of, and out from the middle of the expression: control flows into and out of the expression in an orderly way. The reader may look at Harrison's work [21] to see the difficulties that we may encounter for program transformations such as *exit-loop parallelization* and *recursion splitting* [22], when they are performed on a code that is not so structured. By control-flow normalization, we effectively make `while-` and `do-` loop transformations applicable to all iterative structures by replacing arbitrary control-flow cycles by single-entry single-exit loops, and by reducing the number of different syntactic structures.

Program points after normalization are quite different than program points in the program text known to the user. This problem is fairly easily solved by observing that any expression in a normalized program comes from exactly one expression in the source, and then a map from the transformed program to the original source is well-defined and can be maintained.

As stated in section 2.5.3, we compared, by running experiments on the Perfect Club code, the factorization using boolean expressions and selector expressions. We measured the number of boolean expressions and selector expressions generated, and the code replication that results with and without factorization. See figures 24 and 25.

Column *EBN* gives the number of expressions present in the program before normalization. Column *EANF* gives the number of expressions in the program after normalization with factorization. Column *BEXP* gives the number of boolean expressions that are added during factorization. Column *SEXP* gives the number of selector expressions (a selector expression is an assignment of a selector variable) that are added during factorization. Column *BEXP/EANF* gives the ratio of these counts. Likewise for the ratio *SEXP/EANF*. Column *EAN* gives the number of expressions in the program after normalization without factorization. *GF = EANF - EBN* represents the growth of code when factorization is performed and *G = EAN - EBN* is the growth when the code is normalized with no factorization. Note that when factorization is not used, neither boolean nor selector expressions are added to the program by the normalization. The factorization with boolean expressions requires that every predicate guarding a conditional statement is stored in a temporary variable (see transformation of *if* distribution above). These temporary variables are not necessary when the factorization with selector expressions is performed. That is the reason why *EBN* for the same code has different value, when the factorization is performed using selector expressions versus when using boolean expressions.

The average ratio of boolean expressions *BEXP* created over the total number of expressions in the program *EANF* is 0.01. The average ratio of selector expressions *SEXP* created over the total number of expressions in the program *EANF* is 0.004. If we compare *GF* with *G*, we may see that *GF* is always smaller than the *G* (except when factorizing TRFD with selector expressions), and that the boolean expressions that are created have a reasonable size. For some codes, *GF* is negative; this is due to the fact that the normalization process simplifies the source code and produces a compact form of the code (for example all of the labels and `goto`'s in the source code are eliminated.

N.B. In Miprac Fortran, C and CL front-ends translate every loop using `goto`'s and labels.[2] Therefore the column *EBN* includes all these labels and `goto`'s added by the front-ends. This accounts for much of the negative code growth.

It is clear when we look at these measures, that factorizing with selector expressions results in less code growth than factorizing with boolean expressions. Although in both cases the growth seems to be manageable. The main reason for the small number of boolean expressions is that we use an heuristic order that allows us to create boolean expressions of manageable size in the case of several `if` nesting levels. Miprac does not use a phase of boolean expression simplification; rather, factorization with selector expressions is used so that excessive boolean expressions are not created. It appears, however, that the boolean expressions generated by the normalization are small enough that simplification would not be a major expense.

6 Position of Our Work

Several techniques exist for structuring flowgraphs [13, 28, 33, 11, 12, 5]. Most of these techniques consist of modifications such eliminating `goto` statements, adding control-flow variables, copying code, creating and calling procedures and adding levels of iteration. Each of them may be appropriate for some cases; however the programs they produce are often less regular than those produced by our method and often contain more replicated code [27]. None presents a simple comprehensive algorithm for the normalization of all control flowgraphs. We present, below, a limited overview of each of these methods; however, we emphasize Kennedy's method [5], since it is the most recent and the closest to our own method.

Böhm and Jacopini [13] present two normalization methods of flow diagrams. They decompose the flow diagrams into base diagrams of three types or two types. These methods, like ours, add boolean variables, but they replicate code

[2]The motivations for that is first, a single representation of source code written in three different languages and, second a uniform representation of loops that have `goto`'s and `break`'s in and out of them.

Code	*EBN*	*EANF*	*BEXP*	*EAN*	*BEXP/EAN*	*GF*	*G*
ADM	69229	70378	1350	87557	0.02	1149	18328
ARCD2D	48598	47679	96	48842	0.002	-919	244
BDNA	52579	52657	567	61733	0.01	78	9154
DYFESM	30018	31618	1519	44213	0.05	1600	14195
FLO52Q	36027	35080	68	53167	0.001	-947	17140
MDG	13870	13648	43	19342	0.003	-222	5472
MIGRATION	43369	42716	154	43899	0.003	-653	530
OCEAN	25303	24466	17	138298	0.0006	-837	112995
QCD	24591	23926	1030	25376	0.04	-665	785
SPEC77	54162	52169	50	59186	0.0009	-1993	5024
TRACK	23025	15985	15	18268	0.0008	-7040	-4757
TRFD	5877	5552	6	5610	0.001	-325	-267
Total	426648	416968	4915	605491	0.01	-10774	178843

Figure 24: Factorization with Boolean Expressions

Code	*EBN*	*EANF*	*SEXP*	*EAN*	*BEXP/EAN*	*GF*	*G*
ADM	67561	66919	297	87157	0.004	-642	17596
ARCD2D	47530	46490	80	47499	0.001	-1040	-31
BDNA	51147	51105	282	59852	0.004	-42	8705
DYFESM	28678	27802	187	41634	0.007	-876	12956
FLO52Q	34959	34614	199	51654	0.005	-345	16695
MDG	13542	13499	74	18934	0.005	-43	5392
MIGRATION	42349	41912	178	42632	0.004	-437	283
OCEAN	24571	23990	100	136135	0.004	-581	111564
QCD	23803	23704	142	24401	0.006	-99	598
SPEC77	52214	50590	204	56691	0.004	-1624	4477
TRACK	22201	15939	106	17744	0.006	-6262	-4457
TRFD	5533	5208	19	5180	0.003	-325	-353
Total	414088	401772	1868	587513	0.004	-12316	173425

Figure 25: Factorization with Selector Expressions

even in normalizing reducible flowgraphs. Furthermore, the authors do not present a simple algorithm, but rather describe the method by pattern-matching of flowgraphs, which could be complex and costly to implement.

Knuth and Floyd [28] study program transformations that eliminate `goto` statements without introducing new variables or modifying the sequence of the program computations. The first possibility is to eliminate the `goto`'s by introducing procedures; this is sometimes quite a clean solution, except that the procedure-calling overhead may be important in programs that involve many loop iterations. The second possibility is to write a flowchart according to the BNF they have defined. Both methods replicate code in normalizing reducible flowgraphs. The authors declare that these methods do not suffice to eliminate `goto`'s in all programs.

Peterson, Kasami and Tokura [33] define a well-formed program as a program in which loops and conditional statements are properly nested and have a single entry. To obtain such a program they use a node splitting transformation that may replicate code, or procedure calls in case the code replication is too big. The method replicates code in normalizing reducible flowgraphs (no bound is given on the size of the resulting program) and the resulting programs have multiple-exit loops, and branches that exit several nested control structures.

Ashcroft and Manna [11] introduce two transformations to translate programs with `goto`'s into programs without. The first one adds temporary variables and the second adds logical variables to the program. The first method replicates code in normalizing reducible flowgraphs, and both methods result in loops with multiple exits.

Baker [12] concentrates on making programs more understandable rather than on eliminating the `goto` statements entirely. Some `goto` statements are generated when they give a clearer description of the control-flow. Some syntactic restrictions are imposed upon the input program as well. The algorithm is divided in two steps: locating the loops in the flowgraph and adding branching statements. The first step uses the classical notion of *dominators* [2] after building the depth-first spanning tree of the flowgraph. The second step of the algorithm adds branching statements to the basic form of the program generated in the first step. The algorithm can be extended to handle irreducible graphs. The shortcomings of Baker's method are first, that some `goto`'s remain, second, the loops may be left with multiple exits, and third that the number of control forms is greater than with our method, so that the resultant syntax is still fairly complex.

Allen and Kennedy's method for converting control dependences to data dependences is called *if conversion* [5]. The primary goal of this transformation is to transform programs for the purpose of vectorization. It takes every `do` loop of a program and transforms each of its statement into a guarded one. Beyond this goal, this transformation may be useful in applications such as code structuring and `goto` elimination. `If` conversion is performed in three steps. The first step analyzes the branches in the code, classifying them as

```
       IF X GOTO 200
100     S1
       GOTO 300
200     S2
       IF Y GOTO 100
300     S3
```

Figure 26: A Program with no cycles

either *exit branches*, *backward branches* or *forward branches*. The second step is *branch relocation* and the last step is *branch removal.* An exit branch is defined to be one that terminates a loop. A forward branch is defined to be one whose target is at the same loop nesting level, and which precedes the target (lexically). A backward branch is defined to be one whose target is at the same loop nesting level, and which follows the target (lexically). Branch relocation moves a branch out of a loop until the branch and its target are at the same loop nest level. Branch removal eliminates forward branches by attaching guard expressions to targets.

`If` conversion has goals similar to our normalization method. Its shortcomings are presented below along with differences between it and our work. The examples that are used below are taken from Allen and Kennedy's paper [5] and are written in a Fortran-like syntax. The normalized forms are also written in this same syntax in order to make the differences more apparent.

The first problem with `if` conversion is that backward branches are improperly identified as those whose targets precede them (lexically). Thus the program in figure 26 is treated in the reference paper [5] as a cycle; the `if` conversion algorithm detects `GOTO 100` as being a backward branch. But if we follow carefully the control-flow of the code, there is no loop in this program. The program obtained after `if` conversion is in figure 27. With our method, some factorization was performed since two different paths could be taken to arrive to label `300`. The final normalized form is shown in figure 28; it is written in a Fortran-like syntax to facilitate the comparison.

The second problem with `if` conversion is its ad-hoc treatment of irreducible programs. Let us take an example. The program in figure 29 is transformed after `if` conversion into the program of figure 30. A boolean variable is used to record which branches are taken to reach statements in the loop body. The resulting program contains a `goto`, and the cycle of control-flow has not been replaced by a structured loop. A subsequent transformation (not described in the reference paper [5]) must be used to replace this backward branch by a `while` loop. By contrast, our method produces the program of figure 31, using a uniform treatment of reducible and irreducible flowgraphs (since the flowgraph is irreducible, there is some code replication). Notice that `if` conversion has introduced an additional loop-carried dependence (on the variable `BB1`) that is not

```
        BR1 = X
100     IF (.NOT BR1 .OR. BB1 .AND. BR1) S1
        /* GOTO 300 has been eliminated */
200     IF (.NOT. BB1 .AND. BR1) S2
        IF (.NOT. BB1 .AND. BR1 .AND. Y) THEN
           BB1 = .TRUE.
           GOTO 100
       ENDIF
300     S3
```

Figure 27: The Program of figure 26 after `if` conversion

```
    PRED-162 = X
    IF PRED-162 THEN
       S2
       PRED-164 = Y
    ENDIF
    IF (.NOT. (PRED-162 .AND. (.PRED-164. Y))) S1
    S3
```

Figure 28: Normalized form of the program in figure 26

present in the program of figure 31. This dependence may inhibit parallelization of the loop.

The third problem with `if` conversion is the extra generation of boolean expressions guarding each statement of the program. The program in figure 32 is equivalent after `if` conversion to the one in figure 33. Now, if we look at the normalized form in figure 34 corresponding to the the program in figure 32, we can see that in the latter form, a straightforward dataflow analysis would be more accurate for two reasons. First, the statement `B(I) = A(I) + 5` is not guarded by any condition, and therefore we may easily conclude that it is the only definition of the array `B` that reaches out of the program. Whereas the program in figure 33 necessitates a deeper analysis that accounts for boolean guards to arrive at the same conclusion. Second, it is very easy to see from the control structure of the program in figure 34 that the definition of the variable `X` in statement (2) is reached only by the (last) definition on the statement (1)

```
       IF X GOTO 200
100     S1
200     S2
       IF Y GOTO 100
```

Figure 29: Example of an irreducible code

```
        BR1 = X
        BB1 = .FALSE.
100     IF (.NOT. BR1 .OR. (BR1 .AND. BB1)) S1
200     S2
        IF (Y) THEN
           BB1 = .TRUE.
           GOTO 100
       ENDIF
```

Figure 30: `If` conversion of the program in figure 29

```
PRED-50 = X
IF PRED-50 THEN
 S2
 PRED-52 = Y
ENDIF
IF (.NOT. PRED-50 .AND. (.NOT. PRED-52))
REPEAT
  S1
  S2
  PRED-52 = Y
UNTIL PRED-52
ENDIF
```

Figure 31: Normalized form of the program in figure 29

```
    IF (A(I) > 10) GOTO 100
    X = 0
    A(I) = A(I) + 10
    IF (B(I) > 10) GOTO 200
    B(I) = X + 10
100     A(I) = B(I) + A(I)
200    B(I) = A(I) + 5
```

Figure 32: Reachable uses

```
    BR1= A(I) > 10
    IF (.NOT. BR1)
       X = 0
    IF (.NOT. BR1)
       A(I) = A(I) + 10
    IF (.NOT. BR1)
       BR2 = B(I) > 10
    IF ((.NOT. BR1) .AND. .(NOT. BR2))
       B(I) = X + 10
    IF (BR1 .OR. (.NOT. BR1 .AND. .NOT. BR2))
       A(I) = B(I) + A(I)
    IF (BR1 .OR. (.NOT. BR1 .AND. BR2)
            .OR. (.NOT. BR1 .AND. .NOT. BR2))
       B(I) = A(I) + 5
```

Figure 33: `If` conversion of the program in figure 32

and not by any previous definitions of `X`. Again, in the `if` converted code, a deeper analysis is necessary and perhaps a conservative decision would have to be taken.

7 Related Application of the Gaussian Elimination-like Method

There are several problems closely related to control-flow normalization that make use of Gaussian elimination resolution. These include global flow analysis [19, 20], shortest path problems [15, 18, 26] and conversion of finite automata to regular expressions [41]. The fundamental framework of these problems is to build a system of equations based on *regions* of a flowgraph and to solve this system using the Gaussian resolution. In the following we give an overview of some methods: Allen and Cocke's interval analysis, Hecht and Ullman's analysis, Tarjan's interval analysis, and finally Graham and Wegman's analysis. The latter three are improvements to the first one. In the literature these algorithms are

```
PRED-20 = A(I) > 10
IF (.NOT. PRED-20) THEN
   X = 0                         (1)
       A(I) = A(I) + 10
       PRED-22 = B(I) > 10
   IF (.NOT. PRED-22)
          B(I) = X + 10          (2)
ENDIF
IF (.NOT. ((.NOT. PRED-20) .AND.  PRED-22))
   A(I) = B(I) + A(I)
B(I) = A(I) + 5
```

Figure 34: Normalized form of the program in figure 32

called *elimination* algorithms. Ryder and Paull present a comparison of these four algorithms [38]. They are, in general, described for a specific implementation, and therefore it is difficult to see their common points and differences. Our goal is to give a presentation of these algorithms describing their complexity and performances.

We want to emphasize that the following four algorithms are not methods for program normalization, but rather, are presented to show the reader that the Gaussian elimination-like solution of systems is widely used in similar problems and that several improvements of its complexity have been studied. These algorithms are used for global dataflow analysis. e represents the number of edges in the flowgraph and is assumed to be in order $O(n)$, where n in the number of nodes of the flowgraph.

7.1 Allen and Cocke Method

This method, known as *Interval Analysis*, was introduced by Allen and Cocke [35]. It does not treat irreducible graphs, but it can be adjusted to handle them. The equations it uses are quite different from ours. They represent the dataflow equations of the program. They describe the *reaching definitions* of each variable of the program [4, 38]. The Allen and Cocke algorithm consists of the iteration of three phases: a partitioning algorithm that finds single entry regions in the dependency graph, elimination of the dataflow equations, and finally propagation.

The elimination process turns out to be the application of successive *substitution* and *loop-breaking* transformations. The latter transformation is equivalent to the *derecursivation* transformation we have described earlier in the paper. The unknowns of the system are eliminated in the natural order in which each node of the graph is added to an interval.

During propagation, back-substitutions are performed to propagate global dataflow side effects to the regions where they apply. It consists of finding variable correspondences and substituting interval head variables solutions into reduced equations. The process of reducing the system into a smaller one produces an $O(n^2)$ solution.

7.2 Hecht and Ullman Method

This algorithm is also applicable only to reducible graphs. It takes as input a system of equations analogous to the one described in the Allen and Cocke algorithm, and its dependency graph (i.e., the control flowgraph). The elimination process is directed by the region of the graph, much as the Allen and Cocke algorithm, and consists of applying the transformations T_1 and T_2, as described in the paper earlier, to these regions. The search for common factors in the reduced equations allows a saving in the calculation [1, 37, 46]. This improvement provides a complexity of $O(n \log n)$ rather than a $O(n^2)$ Allen and Cocke's complexity.

7.3 Tarjan Method

The Tarjan method uses a different notion of intervals from the above methods; in a sense they represent the loops in the control flowgraph. When calculating these intervals an implicit order arises, as in the Allen and Cocke method. This order will be used to calculate the reduced equations. It is clear that this order will follow one of the depth-first orders of the graph. The resolution like Hecht's method is based upon the T_1 and T_2 transformations as shown above, and T_3 transformation that is the composition of the two previous ones. Again this method is applied to reducible flowgraphs. For such a graph the algorithm requires a time of $O(n\alpha(n))$ where α is the inverse of *Ackerman*'s function. A simpler algorithm that runs in $O(n \log n)$ exists [43].

7.4 Graham and Wegman Method

This algorithm is close to Tarjan's interval analysis; but it handles irreducible graphs without the need for eliminating the irreducibility. The notion of intervals is called here *S-sets* [19]. They represent the loops in the flowgraph. The *S-sets* are defined by a numbering of the nodes of the dependency graph associated with the equations; this numbering is performed using a depth-first order. The elimination process is performed using three transformations similar to those of the Hecht and Ullman algorithm named S_1, S_2 and S_3. The application of these transformations is restricted to nodes that have only one predecessor. S_1 as T_1 consists of loop-breaking (derecursivation in our framework). S_2 consists of a substitution of a node that has several successors. Then k S_2 transformations are necessary to eliminate a node that has k successors.

The substitution is performed the same way as in the Hecht and Ullman algorithm: each term is substituted successively, rather than substituting the entire right hand side of an equation at once, such as in the Allen and Cocke algorithm. Finally, S_3 eliminates a node that has no successors. This algorithm runs in a time $O(n \log n)$.

7.5 Conclusion

All the algorithms we have presented are refinement to a Gaussian elimination-like algorithm. There are two relationships between the work described above and ours. First, each of the above methods uses a Gaussian elimination method that is similar in some respects to ours. Second, each of them solves a dataflow analysis problem, and much of the difficulty of doing so comes from the irregular structure of the underlying control flowgraph. Using a method like ours, this structure can be made more regular, with the result that the analysis algorithm is made simpler; the normalization of a program anticipates much of the computation of dataflow solutions for the program, so solving a dataflow problem can be much more efficient. Further, the normalization process is performed once on the program, but the simplifications affect both forward and backward dataflow problems to be solved for the program. This is because the backward flowgraph of a normalized program is also normalized, whereas a flowgraph that is merely structured, may be unstructured when its edges are reversed. Finally, even though closures may still be required for loops, especially for non-fast problems [19, 43], these will be simpler than for a non-normalized program.

8 Conclusion

In this paper we have presented an algebraic framework for normalization of control-flow. This framework has made it easy for us to prove that our transformations preserve the semantics of programs. It is applicable to a variety of languages and is more powerful than existing methods in several respects.

First, in methods based upon *node-splitting* [12], and *if conversion* [5], some branching instructions remain after transformations, and code replication may occur even when normalizing programs whose flowgraphs are reducible. Our method eliminates all branching instructions and replicates code only when eliminating irreducibility, a rare condition even in unstructured programs.

Second, while previous methods result in programs with reducible flowgraphs, our method yields programs whose control flowgraphs are more highly structured yet; in particular, all control-flow cycles are normalized into single-entry, single-exit `while` loops. Such loops may be further transformed into conventional `do` loops by induction variable recognition [6, 9]. This makes our method particularly helpful in automatic parallelization, where highly regular loop structures are essential [32], [22]. It simplifies both forward and backward dataflow analyses by transforming a program to one with an obvious, trivial internal structure.

Third, our method makes the separate representation of control dependences unnecessary. In a program normalized by our method, control dependences are, in effect, represented directly by the syntax tree since each conditional structure contains all the expressions it controls. Since the analysis of control dependences is central to the work of parallelizing programs [3], this is a significant simplification.

This work has been implemented in PAF [44], a parallelizer of Fortran programs written at the University of Paris 6, and in MIPRAC, a multilingual parallelizer of programs at the University of Illinois. In Miprac, we measured on a handful of examples, that the ratio of the total normalization time to the total compilation time is approximately 0.02. The total compilation time includes parsing, normalization, interprocedural analysis, intraprocedural dataflow analysis and some restructuring. The average compilation time for these examples was 18.5 seconds. We expect for the ratio to decrease when the time measure will take into account additional passes of Miprac that are currently being implemented.

Acknowledgments

I wish to thank Luddy Harrison for his helpful ideas and suggestions to improve this paper. I also thank the anonymous referees for their comments on earlier drafts of this paper.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1976.

[2] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation and Computing, Vol. II: Computing*. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[3] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the ptran analysis system for multiprocessing. Technical Report RC 13115 (#56866), IBM, New-york, 1987.

[4] F.E. Allen and J. Cocke. A program data flow analysis procedure. *CACM*, 19(3), March 1977.

[5] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. *POPL*, January 1983.

[6] Z. Ammarguellat. *Restructuration des Programmes Fortran en Vue de leur Parallelisation*. PhD thesis, Laboratoire MASI, Université de Pierre et Marie-Curie, Paris 6, France, 1988.

[7] Z. Ammarguellat. Normalization of program control flow. Technical Report CSRD 885, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1989.

[8] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1990.

[9] Z. Ammarguellat and W.L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation - corrected version. In *Sigplan'90 Conference on Programming Languages Design and Implementation*. ACM Press, 1990.

[10] E. Ashcroft and Z. Manna. The translation of 'goto' programs to 'while' programs. In *Proceedings of IFIP Congress 71*. Amsterdam, 1972.

[11] E. Ashcroft and Z. Manna. Translating program schemas to while-schemas. *SIAM Journal of Computing*, 4(2), 1975.

[12] B. Baker. An algorithm for structuring flowgraphs. *JACM*, 24(1), 1977.

[13] C. Bohm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *CACM*, 9(5), 1966.

[14] R.A. Brooks, R.P. Gabriel, and G.L. Steele Jr. An optimizing compiler for lexically scoped lisp. *ACM SIGPLAN Notices*, 17(6), June 1982.

[15] B.A. Carre. An algebra for network routing problems. *J. Inst. Math. Applic.*, 7, 1971.

[16] J. Chailloux. *Le-Lisp Version 15.* INRIA, Rocquencourt, France.

[17] J. Davies, C. Huson, T. Macke, B. Leasure, and M. Wolfe. The kaps-1: An advanced source-to-source vectorizer for the s-1 mark iia supercomputer. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[18] R. Floyd. Shortest path. *CACM*, 5(6), 1962.

[19] S. Graham and M. Wegman. Fast and usually linear algorithm for global flow analysis. *JACM*, 23(1), January 76.

[20] M.S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, Amsterdam, 1977.

[21] W.L. Harrison III. Compiling lisp for evaluation on a tightly coupled multiprocessor. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1986.

[22] W.L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation: an International Journal*, 2(3), 1989.

[23] W.L. Harrison III. Pointers, procedures and parallelization. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1991.

[24] W.L. Harrison III and Z. Ammarguellat. The design of parallelizers for symbolic and numeric programs. In Takayasu Ito and Robert Halstead, editors, *Proceedings of the 2nd US-Japan Workshop on Parallel Lisp*, June 1989.

[25] W.L. Harrison III and Z. Ammarguellat. Miprac: A compiler for symbolic and numerical programs. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1990.

[26] D.B. Jonhson. Efficient algorithms for shortest paths in sparse networks. *JACM*, 24(1), 1977.

[27] D.E. Knuth. Structured programming with goto statements. *ACM Computing Surveys*, 6(4):261–302, 1974.

[28] D.E. Knuth and R.W. Floyd. Notes on avoiding go to statements. *Information Processing Letters*, 1, 1971.

[29] David J. Kuck, Robert H. Kuhn, Bruce Leasure, and M. J. Wolfe. The structure of an advanced vectorized for pipelined processors. In *Fourth International Computer Software and Applications Conference*, 1980.

[30] A. Mazurkiewicz. Proving algorithms by tail functions. *Information and Control*, 18, 1970.

[31] P.D. Mosses. The mathematical semantics of algol 60. *Technical Monograph*, P.R. G 12, January 1974.

[32] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *CACM*, 29(12), December 1986.

[33] W.W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while , repeat and exit statements. *CACM*, 16(8), 1973.

[34] L. Pointer. Perfect club report. Technical Report 896, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1989.

[35] Proceedings of the IFIP Congress '71. *A Basis for Program Optimization.* IEEE, North- Holland, Amsterdam, 1971.

[36] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms - Theory and Practice.* Prentice-Hall, 1977.

[37] B.G. Ryder. Incremental data flow analysis based on a unified model of elimination algorithms. In *Conference Record of the Ninth Annual ACM Symposium on POPL*, pages 167–176. ACM SIGPLAN, 1982.

[38] B.G. Ryder and M.C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3), September 1986.

[39] R.W. Scheifler. *A denotational Semantics of CLU.* U.S.A, May 1978.

[40] D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, Newton, MA, 1986.

[41] C. Shannon and J.McCarthy. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

[42] J.E. Stoy. *Denotational Semantics: The Schott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[43] R.E. Tarjan. Fast algorithms for solving path problems. *JACM*, 28(3), July 1981.

[44] N. Tawbi, A. Dumay, and P. Feautrier. Paf: un paralleliseur automatique pour fortran. Technical Report 185, Laboratoire MASI , Université de Pierre et Marie-Curie, Paris 6, France, May 1988.

[45] R.D. Tennent. The denotational semantics of programming languages. *CACM*, 19(8), 1976.

[46] J.D. Ullman. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3), 1973.

[47] M.H. Williams and H.L. Ossher. Conversion of unstructured flow diagrams to structured. *The Computer Journal*, 21(2), 1975.

[48] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1980.