PROGRAM INVERSIONS TO REORDER CODE

by

Raphael A. Finkel

Computer Sciences Technical Report #293

March 1977

ABSTRACT

Different languages provide control structures with different kinds of flexibility. For example, CLU iterators and SIMULA classes allow control to be suspended in the middle, to be resumed later. Other languages, like ALPHARD and PASCAL, do not have such a facility. A technique called inversion is presented in some generality. It brings statements inside loops to positions outside those loops. It is often possible to invert code that employs one set of control structures to create code that does not make use of those features. Although inversion works when the termination test is not the first step of the loop, statements between the loop entry and the termination test cannot be brought outside the loop. Two examples are given to demonstrate inversion: Two programs, in CLU and ALPHARD, that generate all binary trees on n nodes, and two SIMULA programs that generate all partitions of the integer n.

Key Words and Phrases: loops, control structures, program transformation, loop inversion, program inversion, iteration.

CR Categories: 4.12, 4.22, 4.29, 5.24

PROGRAM INVERSIONS TO REORDER CODE

INTRODUCTION

Several new programming languages generalize the FOR loop of ALGOL in such a way that the body of the iteration is executed for all values that a generator might produce. In this way, one can define ways to iterate over objects of user-defined types. In particular, ALPHARD [London 76, Shaw 76a, 76b, Wulf 76] allows a construction called a generator, and CLU [Liskov 77] has a similar construct called an iterator. These two forms share a common goal: to allow separation between the abstraction that supplies values to a loop and the body that uses those values.

These two methods of producing values for the loop variable differ in a fundamental way, as will be shown below. This paper will show that in most cases, CLU iterators can be transformed by a mechanical process into ALPHARD generators. The resulting code looks quite different, even though it represents the same algorithm. Designers of programming languages realize that the constructs they provide influence the quality of the algorithms that are used and the ease both of programming and of understanding the code. The results shown in this paper indicate that in most cases, one can program iteration in either a CLU fashion or an ALPHARD fashion, and then mechanically translate the result into whatever language is actually available.

THE ALPHARD GENERATOR AND CLU ITERATOR

The ALPHARD generator is written as a "form" (the ALPHARD term for module) that extends some type, either programmer-defined or basic. This form has the following two functions: &init, which sets up the first value in the result cell, and returns the boolean value TRUE iff it succeeds, and &next, which uses the current value of the result cell and any other local data to form the next value; it also returns TRUE iff it succeeds. The loops that the user actually writes, however, do not explicitly use these two functions. A loop to sum the numbers from 1 to 10 might look as follows:

```
1  sum := 0;
2  FOR i:upto(1,10) DO
3       sum := sum + i
```

Figure 1

This user-oriented loop is equivalent to the following code that explicitly calls &init and &next:

```
1  (LOCAL i:upto(1,10), ok: boolean;
2  ok := i.&init;
3  WHILE ok DO
4       (sum := sum + i;  ok := i.&next)
5  )
```

Figure 2

The procedures &init and &next can be written quite readily:

```
1  FORM upto (lower, upper:  integer) EXTENDS k: integer =
2       BEGINFORM
3       SPECIFICATIONS ...
4            FUNCTIONS &init(u:upto) RETURNS (b:boolean),
5                      &next(u:upto) RETURNS (b:boolean);
6       IMPLEMENTATION
7            BODY &init = (u.k := u.lower;  b := u.lower ≤ u.upper);
8            BODY &next = (u.k := u.k + 1;  b := u.k ≤ u.upper);
9       ENDFORM
```

Figure 3

The form given above holds only part of the information required of a form; for a complete specification, see [Shaw 76, p. 11].

In CLU, the same loop might be written as follows:

```
1  sum := 0;
2  FOR i:INT IN fromto(1,10) DO
3       sum := sum + 1;
```

Figure 4

Although the iterator has a different name (and in practice has one extra argument, the step size), this code looks very similar to Figure 1 for ALPHARD. The iterator fromto looks like this:

```
1  fromto = ITER (lower, upper: INT) YIELDS (INT);
2       k:INT := lower;
3       WHILE k ≤ upper DO
4            YIELD(k);  k := k + 1
5            END k;
6       END fromto;
```

Figure 5

The iterator signals that it has no more values by falling to the end of its code; when this happens, the loop that depends on it terminates.

As long as there are more values to come, the iterator has a control point somewhere in the middle of its code; it is in a simple coroutine relationship to the loop that is employing it. Thus we see the major difference between CLU iterators and ALPHARD generators: the latter explicitly return Boolean values that indicate when the loop ought to terminate, and the &next procedure must begin anew for each value to be returned. We will show that most CLU-type iterators can be transformed to ALPHARD-type generators. However, we will also argue that the ALPHARD-like solutions are often considerably less clear. Thus, in a subjective sense, CLU iterators are more expressive.

AN EXAMPLE: GENERATING BINARY TREES

Consider the following problem: Print all distinct binary trees that have n nodes. This task will require a FOR loop over such trees; we are interested in how to produce those trees in CLU and ALPHARD. One algorithm chooses some subset of the nodes to lie in the left subtree, according to some arrangement in that tree, one node to be the root, and the rest of the nodes to inhabit the right subtree in some arrangement. Such a partitioning produces one tree. To get the next one, find a different arrangement of the nodes in the right subtree, if possible; the result is certain to be different from the first. Continue finding new right subtrees until all combinations have been found. Then, to get the next tree, find the next arrangement on the left, and reset the right to its initial arrangement. When there are

no more combinations on the left, then put another node in the left side, taking it from the right side, and continue. The process stops when there are no more nodes on the right side, and all the arrangements on the left side have been used up.

To program this algorithm in CLU requires an abstraction called "tree" with at least two operations: "create", which returns an empty tree, and "build", which takes two trees and returns a new one that has the two arguments as subtrees. Given these operations, the CLU iterator, which we will call "bingen", looks like this:

```
1  bingen = ITER (n: INT) YIELDS tree;
2      IF n = 0 THEN YIELD tree$create() ELSE
3          FOR root: INT IN fromto(1,n) DO
4              FOR left: tree IN bingen(root - 1) DO
5                  FOR right: tree IN bingen(n - root) DO
6                      YIELD tree$build(left,right)
7                      END right;
8                  END left;
9              END root;
10         END if;
11     END bingen;
```

Figure 6

This recursive iterator is a clear and compact implementation of the algorithm given above; the reader is urged to trace through the execution of bingen(3) to be convinced that it works properly. Note that the YIELD statement falls in the middle of three nested iterations. In order to use this algorithm in ALPHARD, we must rearrange the code so that the &next procedure terminates at the point where the iterator

is ready to yield a value. It is not obvious how to accomplish this rearrangement; we will spend much effort later in formalizing a method. Figure 7 shows the ALPHARD generator that results.

Figure 7

```
1  FORM bingen (n: integer) EXTENDS t: tree =
2       BEGINFORM ...
3       IMPLEMENTATION
4       BODY &init =
5           (LOCAL okroot, okright, okleft: boolean;
6           root := upto(1,n);  okroot := root.&init;
7           left := bingen(root - 1);  okleft := left.&init;
8           right := bingen(n - root);  okright := right.&init;
9           b := TRUE);
10      BODY &next =
11          IF n = 0 THEN b := FALSE ELSE
12          (LOCAL okright:  boolean;
13          okright := right.&next;
14          WHILE NOT okright DO
15              (LOCAL okleft;
16              okleft := left.&next;
17              WHILE NOT okleft DO
18                  (LOCAL okroot;
19                  okroot := root.&next;
20                  WHILE NOT okroot DO
21                      (b := FALSE;  RETURN);
22                  left := bingen(root - 1);
23                  okleft := left.&init);
24              right := bingen(n - root);
25              okright := right.&init);
26          t := treecreate(left,right);
27          b := TRUE)
```

After the reader is convinced that the code in figure 7 works, let us point out a few peculiarities. None of the loops could be made into a standard ALPHARD FOR loop, because the role of &init and &next are reversed, and the condition that is tested is the negation of the usual condition. Furthermore, the loop variable is given a new value in the middle (which is absolutely forbidden in ALPHARD, and is most likely not syntactically correct in any case). For example, the loop at lines 16 to 23 "initializes" left by a call to &next, and gets the "next" value by a call in line 23 to &init. In line 22 a new generator is given to left. The condition in line 17 is the negation of okleft. The ALPHARD code appears to be strangely inverted.

A second peculiarity is the use of RETURN in line 21. This manner of leaving a procedure is not actually allowed in ALPHARD, nor is our explicit use of &init and &next, but its function is necessary, since the code at the end of the CLU iterator now appears in the middle of the ALPHARD generator. A third point is that the initialization code is actually a duplicate of some code residing in the &next procedure. If line 21 were changed to this:

```
(b := FALSE; RETURN:
ENTRY &init; root := upto(1,n);  okroot := root.&init),
```

then the rest of the initialization would take place correctly. Fourth, although IF conditionals would serve in place of the WHILE constructs at lines 14, 17, and 20, the WHILE is more general in case &init can fail, although this occurrence is not possible in this particular appli-

cation. These various connections between the code for the two languages leads us to seek a formal way to invert nested loops to arrive at an ALPHARD-like representation given a CLU-like one. These techniques will prove generally useful for coding in languages that do not have the limited coroutine facility of CLU. For the sake of this discussion, we will deal with a simple formalization of the idea of loop, add new constructs to enter and leave loops, and derive the formal mechanism we seek.

LOOP ROTATION

The notation that follows is based on a construct suggested by O. J. Dahl to perform the "n and a half times" loop [Knuth 74]. This construct resembles the ALGOL type of WHILE statement, except that the test may appear at any one place inside the loop. We will write the standard loop in this form:

```
1  loop_j:  LOOP
2  α;
3  ASSURE cond;
4  β;
5  REPEAT loop_j;
```

Figure 8

When the loop is entered, statement α (which may actually be many statements) is executed, and then the Boolean expression cond is evaluated. If its value is TRUE, then control remains inside the loop and continues with statement(s) β . When REPEAT is encountered,

control returns unconditionally to the head of the loop. When the condition is tested and proves to be FALSE, then control exits from the loop to whatever statement follows. A label (here $loop_j$) may appear before the word LOOP, and that same label may be used after the matching REPEAT to make it clear where the matching LOOP is to be found. One standard use of this LOOP construct is to read a stream of input and process it until some sentinel is read. In this case, α in line 2 will read the next data item, the condition in line 3 is that the data are good (anything but the sentinel), and β in line 4 will contain code to process the input.

The first generalization that I would like to introduce is a dual of the loop termination property inherent in ASSURE. This construct, called ENTER, may appear at any place between LOOP and REPEAT; the meaning of the loop is now the same as before, except that the first statement executed will be the one directly following ENTER. For example, if the programmer wishes to write a loop to process input, as mentioned above, and to him the most essential part of the loop is the processing step, then an appropriate way to write the code would be:

```
1  scan: LOOP
2  process input;
3  ENTER;
4  read input;
5  ASSURE not the sentinel;
6  REPEAT scan;
```

Figure 9

Figure 9 presents a rotated form of Figure 8, with α, β and cond made explicit. I do not wish to claim that this program segment is particularly clear; however, we will need the ability to rotate shortly. As an aside, we might note that unbridled use of ENTER and ASSURE can turn a straightforward program into one that looks (speciously) as if it were a loop:

```
1  LOOP
2  ENTER;
3  some code;
4  ASSURE FALSE;
5  REPEAT
```

Figure 10

Or, even worse:

```
1  LOOP
2  ASSURE FALSE;
3  ENTER;
4  some code;
5  REPEAT
```

Figure 11

The reason that these examples are opaque is that a loop construct is being used where it does not belong; the use of ENTER in Figure 11 only exacerbates an already bad situation. However, let us live for the moment with this peculiarity. In one-level loops, then, ENTER allows us to rotate any desired statement to a position directly before the final REPEAT. This facility will eventually allow us to invert CLU iterators by moving the YIELD statement to the end of the outermost loop, where it can function as the termination of the ALPHARD &next procedure.

Consider next a two-level loop, as shown in Figure 12. We wish to move the statement "use(a,b)" to the point directly before the outermost REPEAT.

```
1   a := initial_a;
2       loop1: LOOP
3       ASSURE  a ≤ limit_a;
4       b := initial_b;
5           loop2: LOOP
6           ASSURE b ≤ limit_b;
7           use(a,b);
8           b := b + 1;
9           REPEAT loop2;
10      a := a + 1;
11      REPEAT loop1;
```

Figure 12

No use of ENTER alone can rotate the "use" statement out of the inner loop. This situation is due to the weakness of ENTER. Let us extend ENTER so that it specifies which LOOP is to be entered at that point. (If no loop is specified, it is to be understood that the ENTER refers to the nearest surrounding loop.) The exact dual to this extended ENTER will also be introduced: a labeled LEAVE. The meaning of LEAVE is to unconditionally exit the specified loop (or closest surrounding loop, if none is specified.) Since ASSURE can also break loops, we will insist that each loop have exactly one means of termination. That is, each loop must either contain one ASSURE or one LEAVE that can transfer control to the statement following it.

Given these tools, it is now possible to arrange the code of Figure 12 to place the "use(a,b)" at the end of the outmost loop:

```
1  a := initial_a;
2      loop2: LOOP
3      b := b + 1;
4          loop1: LOOP
5          ASSURE b > limit_b;
6          a := a + 1;
7          IF a > limit_a THEN LEAVE loop2;
8          ENTER loop2;
9          b := initial_b;
10         REPEAT loop1;
11     use(a,b);
12     REPEAT loop2;
```

Figure 13

It may take some effort to realize that Figure 13 is, in fact, correct. The condition on line 5 is certainly counterintuitive; it will usually cause $loop_1$ to be exited immediately. The body of $loop_1$ is designed to treat the unusual case that b is at the end of its range. Furthermore, b becomes initialized in line 9, which is executed both at the start of $loop_2$ and whenever a is incremented, due to the ENTER at line 8. Also, notice that the code between lines 4 and 10 must be in a loop, not a conditional like "IF b > limit_b THEN ...", since resetting a might change the value of limit_b, if the latter is a procedure.

One bothersome aspect of Figure 13 is its nonuniform treatment of a and b. The termination tests are opposite, and the initializations

are in different places. It is actually possible to treat both variables in a uniform fashion, yielding Figure 14.

```
1  loop0:  LOOP
2  b := b + 1;
3       loop2:  LOOP
4       ASSURE b > limit_b;
5       a := a + 1;
6            loop1:  LOOP
7            ASSURE  a > limit_a;
8            LEAVE loop0;
9            ENTER loop0;
10           a := initial_a;
11           REPEAT loop1;
12      b := initial_b;
13      REPEAT loop2;
14 use(a,b);
15 REPEAT loop0;
```

Figure 14

Now both tests are opposite to their state in Figure 12, and the deeper loops cover successively less-frequently encountered situations. Initialization is handled in lines 10 and 12. Figure 14 represents an inversion of the original program. It is as if the "use(a,b)" were plucked from its inner loop, causing an inversion of each loop that it passed. The reader can easily verify that a program with even deeper nesting can also be inverted to a form similar to that of Figure 14.

LOOP INVERSION

We wish to find a straightforward way to invert a program such as

Figure 12 and automatically get Figure 14. In order to present the answer, we must note that Figure 12 contains only a subset of all possible loops. In particular, no application of ENTER appears there, and all ASSUREs fall directly at the beginning of the loops. Nested loops of this particular flavor can be readily represented by a flow chart:

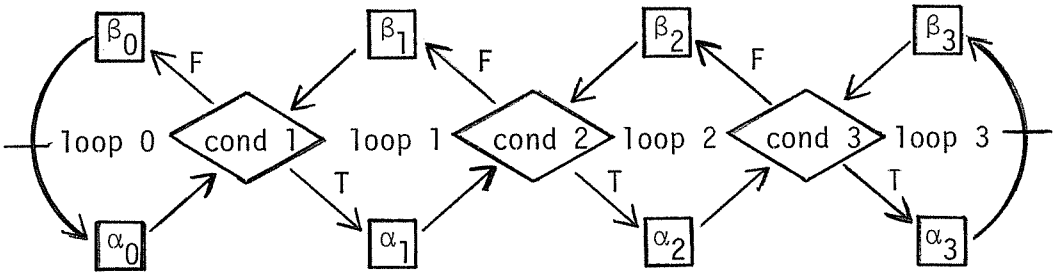

Figure 15

Here, the higher loops are nested within the lower-numbered loops. The program begins with α_0 and ends with β_0; an artificial loop has been introduced at the beginning to make the symmetry of the figure apparent. Figure 16 shows two programs that equally describe the conditions depicted in Figure 15.

```
1   loop0:  LOOP
2   ENTER loop0;
3   α0;
4       loop1:  LOOP
5       ENTER loop1;
6       ASSURE cond1;
7       α1;
8           loop2:  LOOP
9           ENTER cond2;
10          ASSURE cond2;
11          α2;
12              loop3:  LOOP
13              ENTER loop2;
14              ASSURE cond3;
15              α3;
16
17
18              β3;
19              REPEAT loop2;
20          β2;
21          REPEAT loop2;
22      β1;
23      REPEAT loop1;
24  β0;
25  LEAVE loop0;
26  REPEAT loop0;
```

```
1   loop0:  LOOP
2
3   β3;
4       loop3:  LOOP
5       ENTER loop3;
6       ASSURE NOT cond3;
7       β2;
8           loop2:  LOOP
9           ENTER loop2;
10          ASSURE NOT cond2;
11          β1;
12              loop1:  LOOP
13              ENTER loop1;
14              ASSURE NOT cond1;
15              β0;
16              LEAVE loop0;
17              ENTER loop0;
18              α0;
19              REPEAT loop1;
20          α1;
21          REPEAT loop2;
22      α2;
23      REPEAT loop3;
24  α3;
25
26  REPEAT loop0;
```

Figure 16

Figure 16 uses the fullest forms, with all ENTERs explicit. The only difference between the formal structure on the left (the usual way to represent the program) and the right (the inverted format) is the presence of ENTER and LEAVE at lines 2 and 25 on the left and lines 16 and 17 on the right. In both cases, the first executable statement is α_0.

From this example it is possible to derive an algorithm that will transform programs like that on the left to ones like that on the right. To begin, it is necessary to surround the source code by LOOP .. REPEAT with an explicit ENTER and LEAVE. Call this level $loop_0$, the outermost loop. In order to distinguish some statement (here α_3) in the innermost loop, so that it will end up at the very end of the outermost loop, begin to copy the code from the source immediately after the distinguished statement (here, at β_3). The special words LOOP, ASSURE, REPEAT and ENTER get special treatment:

"j: LOOP" becomes "REPEAT j;". Continue with the first statement after "ASSURE $cond_j$".

"ASSURE $cond_j$" becomes "ASSURE NOT $cond_j$;". Continue with the first statement after "REPEAT j".

"REPEAT j" is omitted. Continue with the first statement after "j: LOOP".

"ENTER j: (which is, by default, immediately after "j: LOOP") becomes "j: LOOP; ENTER j" (the "ENTER j" is not, strictly speaking, required). However, the outermost "ENTER" is preserved intact. In any case, continue with the first statement after the "ENTER j". "LEAVE" should only appear in the outermost loop. Leave it intact.

When the distinguished statement (here α_3) has been copied, the result is surrounded by "$loop_0$: LOOP ... REPEAT $loop_0$;"

These rules may seem arbitrary, but reference to Figure 15 shows that they work. Let us now relax the constraint that the test in any loop be at the point of entry. Consider the flowchart of Figure 17.

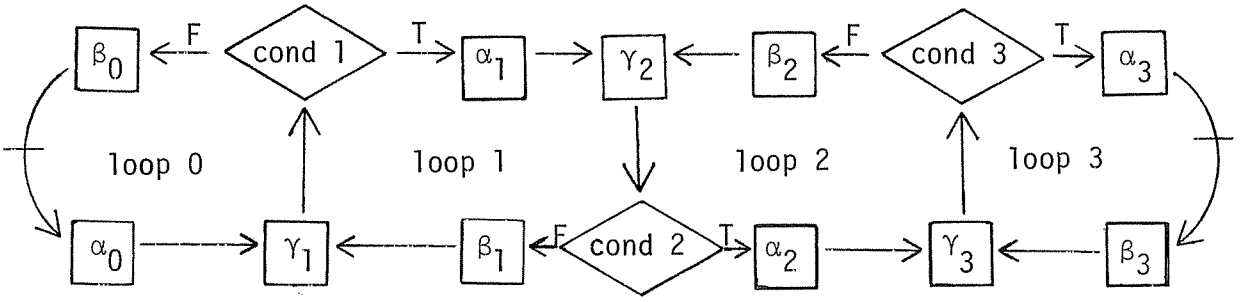

Figure 17

Figure 17 can be best understood as a recursive view of the program schema in Figure 18, as expanded in Figure 19, which also shows the inverted form.

```
1  LOOP
2  ENTER;
3  γ;
4  ASSURE cond;
5  α;
6     recurse
7  β;
8  REPEAT;
```

Figure 18

```
1   loop0: LOOP
2   ENTER loop0;
3   α0;
4       loop1: LOOP
5       ENTER loop1;
6       γ1;
7       ASSURE cond1;
8       α1;
9           loop2: LOOP
10          ENTER loop2;
11          γ2;
12          ASSURE cond2;
13          α2;
14              loop3: LOOP
15              ENTER loop3;
16              γ3;
17              ASSURE cond3;
18              α3;
19
20
21              β3;
22              REPEAT loop3;
23          β2;
24          REPEAT loop2;
25      β1;
26      REPEAT loop1;
27  β0;
28  LEAVE loop0;
29  REPEAT loop0;
```

```
1   loop0:  LOOP
2
3   β3;
4       loop3: LOOP
5       ENTER loop3;
6       γ3;
7       ASSURE NOT cond3;
8       β2;
9           loop2: LOOP
10          ENTER loop2;
11          γ2;
12          ASSURE NOT cond2;
13          β1;
14              loop1: LOOP
15              ENTER loop1;
16              γ1;
17              ASSURE NOT cond1;
18              β0;
19              LEAVE loop0;
20              ENTER loop0;
21              α0;
22              REPEAT loop1;
23          α1;
24          REPEAT loop2;
25      α2;
26      REPEAT loop3;
27  α3;
28
29  REPEAT loop0;
```

Figure 19

It is evident that the method used before holds even in this more complicated case. It should be pointed out that even though Figure 19 poses the source program with every loop having its entry immediately

after its beginning, this situation by no means represents any restriction on the source program; the same target code would have resulted from rotating any or all of the source loops.

Having given some rules for inversion and demonstrated that they work in particular cases, let us investigate why these rules are correct. Both Figures 15 and 17 are symmetric, with control tending to the right as tests succeed. Each loop has an entry point, a conditional test for moving right, and a final statement before encountering the entry point again. Inversion changes the tests so that success moves control to the left. So the statement executed after a successful test will be what used to be the first statement after the loop terminates. Therefore our rule changes assertions to their negations and continues outside the loop. When the excursion to the left has finished, as signaled by encountering the original loop again, it is time to close the current loop, which was opened when we saw its "ENTER". At this point, REPEAT is generated, and the remaining case, failure at the termination test, is finally treated. For this reason, encountering "LOOP" generates "REPEAT" and continues copying after the "ASSURE".

UNINVERTABLE LOOPS

Figures 17 and 19 raise a troubling question: How can γ_3 be brought to the distinguished spot in the outermost loop? If we follow the rules as they stand, the very first statement to be translated is "ASSURE $cond_3$" in line 17, and the code that will be generated will be another ASSURE. However, there will be no current loop open. Even though the eventual number of LOOPs and REPEATs will match, there will

be places in the target code where there have been more of the latter than the former. The (bad) target code is shown in Figure 20.

```
1   loop0:  LOOP
2       ASSURE NOT cond3;
3       β2;
4           loop2:  LOOP
5           ENTER loop2;
6           γ2;
7           ASSURE NOT cond2;
8           β1;
9               loop1:  LOOP
10              ENTER loop1;
11              γ1;
12              ASSURE NOT cond1;
13              β0;
14              LEAVE loop0;
15              ENTER loop0;
16              α0;
17              REPEAT loop1;
18          α1;
19          REPEAT loop2;
20      α2;
21      REPEAT loop3;
22  α3;
23  β3;
24      loop3:  LOOP
25      ENTER loop3;
26      γ3;
27  REPEAT loop0;
```

Figure 20

It is evident that $loop_3$ is illegally rotated with respect to $loop_0$; its LOOP in line 24 falls after its REPEAT in line 21. The difficulty encountered here stems from the fact that the distinguished statement,

γ_3, is positioned between the ENTER and the ASSURE in a loop. Our earlier example was free of this problem since only WHILE-type loops were allowed; in such loops, there can be no statements between the entrance and the first test.

The program of Figure 20 can be understood, however, if the meaning of "REPEAT $loop_j$" is taken to be "GO TO after $loop_j$", the meaning of "$loop_j$: LOOP" is "GO TO after ENTER $loop_j$", and "LEAVE $loop_j$" means "GO TO after REPEAT $loop_j$". These interpretations are consistent with the usual meanings when loops are well-formed, and are therefore an extension for this special case. In order to complete this rewording, each ASSURE must be converted to the equivalent LEAVE statement (with appropriate loop information). The process just described will turn what was a fairly clear program into a web of absolute control transfers. The significant observation is that in the usual case, the control structures that result from inversion are simple loops.

EXTENSIONS

An interesting extension of the initial rules is to cover cases in which the distinguished statement, which from now on we restrict from lying between the ENTER and the ASSURE of a loop, is not in the innermost loop. An example would be to distinguish β_2 in line 23 of Figure 19. All of $loop_3$ is irrelevant to β_2 and will not get inverted. This consideration leads to a simple rule: Loops deeper than the distinguished statement do not follow the special handling reserved for LOOP and the

other constructs; instead, the entire deeper loop is copied intact. An example of the result of such action taken on the source program of Figure 19 can be seen in Figure 21.

```
1   loop0:  LOOP
2       loop2:  LOOP
3       ENTER loop2;
4       γ2;
5       ASSURE NOT cond2;
6       β1;
7           loop1: LOOP
8           ENTER loop1;
9           γ1;
10          ASSURE NOT cond1;
11          β0;
12          LEAVE loop0;
13          ENTER loop0;
14          α0;
15          REPEAT loop1;
16      α1;
17      REPEAT loop2;
18  α2;
19      loop3:  LOOP
20      ENTER loop3;
21      γ3;
22      ASSURE cond3;
23      α3;
24      β3;
25      REPEAT loop3;
26  β2;
27  REPEAT loop0;
```

Figure 21

Not all cases have yet been covered. If one takes a threefold-nested loop, such as that shown in Figure 16 left, and tries to arrive at the right side of Figure 16 by applying three successive inversion steps, then the first will invert lines 11 to 20, the second lines 7 to 22 and the third lines 3 to 24. In each step an artificial loop is introduced about the code to be inverted; Figure 16 shows this artificial loop about lines 3 to 24 on the left side. In performing these inversions, one comes upon situations of deeper loops that have multi-level ENTERs and LEAVEs that cannot be copied verbatim. These occurrences lead to yet another rule for inversion: Deeper loops are copied verbatim, except that occurrences of "LEAVE $loop_j$", where $loop_j$ is not a deeper loop, are omitted; copying continues after "REPEAT $loop_j$". Likewise, when "$loop_j$: LOOP" is encountered, after it is replaced by "REPEAT $loop_j$", if there is an "ENTER $loop_j$" in a deeper loop, control proceeds to after that ENTER. The other case works as before, with control continuing after the matching ASSURE. To demonstrate these new rules, consider the second inversion that will take place during a three-step inversion as mentioned above. This inversion step is demonstrated in Figure 22.

```
1   loop0: LOOP
2   ENTER loop0;
3   α1;
4       loop2: LOOP
5       ENTER loop2;
6       ASSURE cond2;
7           loopt: LOOP
8           β3;
9               loop3: LOOP
10              ENTER loop3;
11              ASSURE NOT cond3;
12              β2;
13              LEAVE loopt;
14              ENTER loopt;
15              α2;
16              REPEAT loop3;
17          α3;
18          REPEAT loopt;
19      REPEAT loop2;
20  β1;
21  LEAVE loop0;
22  REPEAT loop0;
```

```
1   loop0: LOOP
2   β3;
3       loop3: LOOP
4       ENTER loop3;
5       ASSURE NOT cond3;
6       β2;
7           loop2: LOOP
8           ENTER loop2;
9           ASSURE NOT cond2;
10          β1;
11          LEAVE loop0;
12          ENTER loop0;
13          α1;
14          REPEAT loop2;
15      α2;
16      REPEAT loop3;
17  α3;
18  REPEAT loop0;
```

Figure 22

The reader should verify that this transformation follows the rules given, and that by three successive inversions, the program on the left in Figure 16 can be transformed into the one on the right.

AN APPLICATION

Consider Figure 23, which is a SIMULA class that generates all partitions of the integer n, that is sequences $i_1,\ldots,i_k$ which sum to n.

```
CLASS partition (n);  INTEGER n;  COMMENT  n  is to be partitioned;
1   BEGIN
2   INTEGER first;  COMMENT  the first part of the partition;
3   REF(partition) REST;  COMMENT  a list of the rest of the partitions;
4   BOOLEAN more;  COMMENT  when false there are no more partitions;
5   PROCEDURE print;
6       IF rest =/= NONE THEN
7           BEGIN  COMMENT  print this element, recurse for the rest;
8           output(first);
9           rest.print;
10          END
11  more := TRUE;                                          Figure 23
12  IF n=0 THEN detach
13  ELSE BEGIN  COMMENT:  Chop off initial chunk, recurse for rest;
14      FOR first := 1 STEP 1 UNTIL n DO
15          BEGIN
16          rest := NEW partition(n-first);
17          WHILE rest.more DO
18              BEGIN
19              detach;
20              call(rest);
21              END while loop
22          END for loop;
23      END conditional;
24  more := FALSE;
25  END partition;
```

This program makes heavy use of the SIMULA feature by which a class object may temporarily halt in its execution ("detach") or may be continued from such a halt ("call"). It is therefore not possible to

directly represent this program in ALPHARD, although it should be fairly easy in CLU. We would like to create a procedure "next" from Figure 23. We can do this by inverting lines 14 through 22, changing the "call" to an explicit invocation of &next, as shown in Figure 24.

```
1   PROCEDURE next;
2       BEGIN  COMMENT  causes the next partition to exist;
3       rest.next;  COMMENT  perhaps can just change deeper down;
4       WHILE NOT rest.more DO
5           BEGIN  COMMENT  Must make a change at this level instead;
6           first := first +1;
7           WHILE first > n DO
8               BEGIN  COMMENT:  termination condition;
9               more := FALSE;
10              RETURN;
11              init:  first  := 1;  more := TRUE;
12              END  while loop;
13          rest := NEW partition(n-first);
14          END while loop;
15      END next;
```

Figure 24

Figure 24 is by no means clearer than Figure 23; it takes extra effort to be convinced that the initialization works correctly and that the special cases are correctly handled. For example, it is essential that line 7 and 4 both contain WHILE loops, even though at first glance it would appear that IF conditionals would work as well. By now we are used to these peculiarities of inverted programs. It is an easy exercise to translate Figure 24 into the extended ALPHARD of Figure 7.

CONCLUSIONS

Not all programming languages support the same constructs, but the way we develop programs depends on the constructs available to us. A class of recursive generators is readily programmed using CLU iterators or SIMULA classes, but the language we are using may not have these ideas. This paper has shown a way to transform a program written in one manner into a functionally equivalent program that uses less fancy language features. Now we can pretend our language allows CLU-like iterators to some extent, since many programs written with them can be inverted to avoid using internal returns.

It must be emphasized that the CLU approach to iteration is not always most natural. When Figures 6 and 7, the CLU and ALPHARD versions of bingen, are translated into SIMULA, they occupy precisely the same length (as one might expect from the inversion rules). A colleague of mine found the method of Figure 6 natural when he wrote a SIMULA program to implement bingen; I chose the Figure 7 style. Neither of us was at that time familiar with inversions. Our programs were dual to each other; each could be found by inverting the other.

Further work is required to settle the case where the distinguished statement falls between loop entry and the conditioned test that terminates the loop. This situation might be considered freakish, and it may truly be impossible to invert at that place. Another unfinished issue is the presence of conditionals in the code; this paper has completely ignored that case. Perhaps all ASSUREs should be expressed as LEAVEs inside conditionals; this arrangement may well be clearer.

ACKNOWLEDGEMENTS

The author would like to thank David Gries, Larry Landweber, Barbara Liskov, Mary Shaw, and Marvin Solomon for fruitful discussions that led to the ideas presented here.

BIBLIOGRAPHY

[Knuth 74] Knuth, D. E. "Structured programming with go to statements", Computing Surveys 6, 4 (December 1974), 261-301.

[Liskov 77] Liskov, B., Snyder, A., Atkinson, R., Schaffert, C., Abstraction mechanisms in CLU, Massachusetts Institute of Technology Laboratory for Computer Science, Computation Structures Group Memo 144-1 (January 1977).

[London 76] London, R. L., Shaw, M., Wulf, W. A. Abstraction and verification in Alphard: A symbol table example, Carnegie-Mellon University Technical Report (1976).

[Shaw 76a] Shaw, M. "Abstraction and verification in Alphard: Design and verification of a tree handler", Proc. Fifth Texas Conference on Computing Systems (1976, 86-94.

[Shaw 76b] Shaw, M., Wulf, W. A., London, R. L. Abstraction and verification in Alphard: Iteration and generators, Carnegie-Mellon University Technical Report (1976).

[Wulf 76] Wulf, W. A., London, R. L., Shaw, M. Abstraction and verification in Alphard: Introduction to language and methodology, Carnegie-Mellon University Technical Report (1976).