G. Urschler

Automatic Structuring of Programs

Abstract: A method is described that allows the translation of a traditionally written (unstructured) program into a set of top-down structured, semantically founded, GOTO-free modules. The method reveals not only the logic of a given program in a most natural way, but it also reduces code duplication to a minimum. It is further shown how the obtained structured program can be mapped back into a GOTO program in such a way that all GOTOS are backwards branches and their number is minimal. The connection between recursively and iteratively structured programs is demonstrated using the WHILE, DO FOREVER, and multilevel EXIT statements. Extensions of the method show the structuring of source programs containing block structures and subroutines.

Introduction

The two major disciplines that can be subsumed under the heading structured programming are the methodology of program development and language design. The methodology proposes to develop reliable and understandable programs in a sequence of steps that show different levels of abstraction, starting with the problem and ending with the final code, and to do this by stepwise refinement. The aim of the language design is to support these ideas by the development of suitable language features. There are two obvious ways to do this. One is to start with specific problem solutions, to find adequate expressional tools for them (which may also provide feedback for better solutions), and then to try to find suitable generalizations. The other is to start with an existing language (which is an expressional tool for an infinite number of problem solutions) and develop an algorithm that automatically transforms an arbitrary program written in the language into a structured form, thereby developing a new programming language as well. We intend this paper to be interpreted primarily in the latter sense, i. e., as a contribution to language design. When developing a "structurizer," we obtain, in addition, a valuable tool for:

- The automatic structuring of existing unstructured programs, which should greatly facilitate the reading of existing programs.
- The education of programmers, a whole generation of whom are said to have been spoiled by FORTRAN.
- The further automatic processing of programs for the purposes of compilation, optimization, parallelization, correctness proving, formatting, and editing.

The structuring method described in this paper was originally developed by the author for the translation of flow diagrams into maximally parallel form. In [1] it was shown that the program structure suggested in this paper is exactly the one required to get dynamic (run-time) parallelism (i.e., to execute, for instance, independent iterations of the same loop asynchronously). To assure the reader that the structuring method presented is not one of the well-known ones in disguise, the following comparisons are made:

1. The oldest structuring method is due to Kleene [2], who showed that finite state machines (which flow diagrams can be considered to be) are equivalent to regular expressions. What makes Kleene's approach different from ours (and all other control structure oriented ones) is the equivalence relation between regular expressions. It is based on the rules of associativity, commutativity, and distributivity of the primitive operations of concatenation and ORing. Most of these rules do not apply to transformations admis-

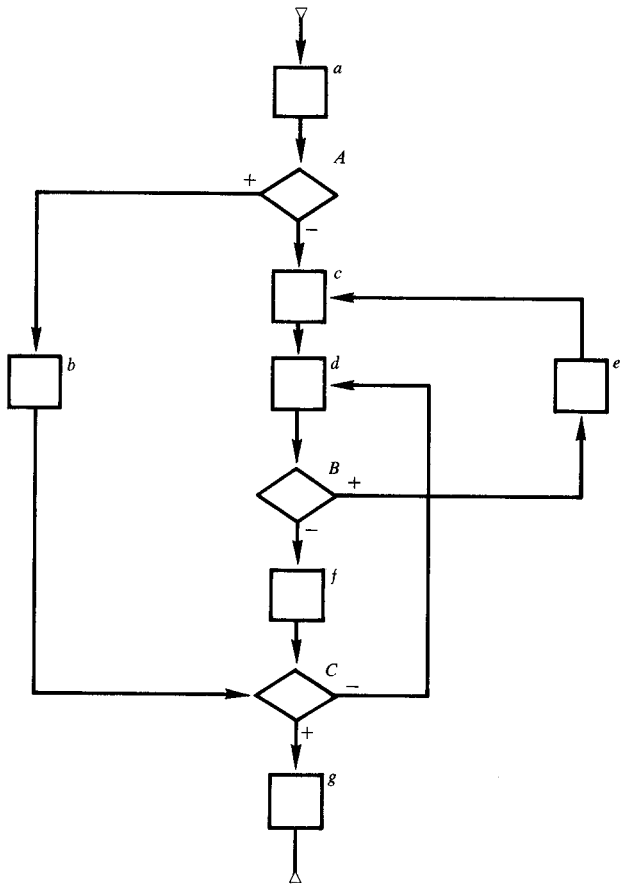

Figure 1 Flow diagram skeleton.

sible in program structuring because they make a program based on a set of computations and deprive it of its original control structure.

2. Böhm and Jacopini [3] modeled the control structure at least partially by means of the data flow (using control switches). In contrast, in this paper no use of the data flow of a program is made.
3. The method of Manna and Ashcroft [4], as an improvement of the Böhm and Jacopini results, treats program transformations as pure control structure transformations and thus preserves the program's "topology." The price paid is a possible exponential increase in program size. With our method, however, we claim to keep the conciseness of the GOTO.
4. Our method may seem to be a contradiction of the more negative implications of the results of Ashcroft and Manna [4], Knuth and Floyd [5], and Bruno and Steiglitz [6], which proved the WHILE statement to be inherently weaker than the GOTO statement. Along these lines Peterson, Kasami, and Tokura [7] finally came to the conclusion that besides the WHILE statement the (more general) DO FOREVER, in connection with the multilevel EXIT statement and node splitting (i.e., the copying of program text), is needed to reach the full capability of the GOTO. In this paper recursive language constructs are used instead of the above mentioned iterative ones. Because recursion is a more powerful language concept than the GOTO, the above results no longer apply. The proposed language feature itself can be regarded as a generalization of Dahl's [8] concept of a recursive loop.
5. The first attempts to translate an iterative program into a recursive form are due to McCarthy [9] and Van Wijngaarden [10]. What makes their approaches different from ours is that they form (recursive) modules on a syntactical rather than a semantical basis. Their modules always have a maximal scope (up to the end of the program), whereas our scopes are minimal and based on the post dominance relation.

One of the many restrictions of the work reported in this paper is that the data flow of programs is disregarded, as are many involved optimization problems. This is not because we think these problems are insignificant, but is rather a consequence of an attempt to clearly isolate the problem.

First we show how simple GOTO programs can be translated into modular, top-down structured programs in such a way that the program logic is exposed. As a next step the obtained program representation is brought into as concise a form as possible. Afterwards the resulting programs are mapped back into GOTO programs in such a way that all GOTOs are backwards branches and their number is minimal. Then the connection between recursively and iteratively structured programs is demonstrated by using the WHILE, DO FOREVER, and multilevel EXIT statements. Finally an extension of the method to a more powerful source language is given, showing that a recently proposed language feature of Zahn [11] fits nicely into the proposed language concept.

Translation of GOTO programs into recursively structured form

In the following text GOTO programs are assumed to be represented in the form of flow diagrams. Because this paper deals with control structure transformations, only the skeleton of these flow diagrams is shown (sometimes referred to as the outer syntax of the program). Such a skeleton can be regarded as a finite, directed, and labeled graph satisfying the following conditions:

1. It has exactly one beginning (denoted by ∇) and exactly one end (denoted by Δ).

2. Each node in the graph is reachable from ∇, and Δ can be reached from it (i.e., the graph is in reduced form).
3. Edges with the same source have different targets.
4. If two nodes are connected by an edge, at least one of them is either a junction node (i.e., the target of at least two different edges) or a ramification node (i.e., the source of at least two different edges).

In spite of the abstraction made, all of the assertions in this paper are intended primarily to be assertions on programs and not assertions on graphs. Figure 1 shows an example of a flow diagram skeleton.

Ramification nodes are also referred to as decisions (they correspond exactly to the decisions of the underlying flow diagram). They are drawn in the usual way as diamond shaped boxes and are labeled by capital letters (A, B, . . .). Decisions may have more than two outcomes, but identical outcomes are assumed to be combined into a single one (expressed by condition 3 above). In any case the outcomes are assumed to be ordered, as indicated by $+$, $-$ in Fig. 1; if there are more than two outcomes, integers can be used.

Nodes that are the source of a single edge only and are different from ∇ are also referred to as elementary blocks. They correspond to maximal (expressed by condition 4 above) sequences of "basic" statements (i.e., assignment. I/O, and other data oriented statements) in the underlying flow diagram. Elementary blocks are drawn as rectangular boxes and are labeled by lower-case letters ($a, b, \cdots$).

Although the denotations of decisions and elementary blocks are irrelevant for the purpose of this paper, it is sometimes necessary to refer to them. We do this by underlying the node labels. Thus $\underline{A}$, $\underline{B}$, $\cdots$ denote, for instance, not further specified Boolean expressions and $\underline{a}$, $\underline{b}$, $\cdots$ denote not further specified sequences of basic statements.

The example in Fig. 1 was chosen because it is simple enough not to be confusing and complicated enough to show the main problems in the structuring process. (There is a loop in the program, namely $d\ B\ f\ C\ d$, that has two entries and two exits; the program is also irreducible).

The following definitions can be made with respect to flow diagram skeletons:

Definition 1 A node N_2 is a successor of a node N_1 if there is an edge in the graph with source N_1 and target N_2. (In Fig. 1, for instance, A has the two successors b and c, and c has the unique successor d).

Definition 2 A path is a sequence of nodes each of which (except for the first node) is a successor of the node immediately preceding it in the path.

Definition 3 A path is cycle-free if it contains no node twice.

Definition 4 A node N_2 is a post dominator of a node N_1 if each path from N_1 to Δ contains N_2.

Definition 5 A node N_2 is an immediate post dominator of a node N_1 if N_2 is a post dominator of N_1 and if in each path from N_2 to Δ N_2 is the post dominator of N_1 occurring first.

One of the first mentions of dominance relations is in [12]. Since then dominance relations have been widely used in the optimization area, mainly in the form of the predominance relation [13].

Proposition 1 Each node different from Δ has exactly one uniquely determined immediate post dominator.

Proof Each node different from Δ has at least one post dominator, namely Δ. Let us assume there are two cycle-free paths (cycles do not contribute to post dominators) from a node N to Δ containing the two post dominators P_1 and P_2 but in reversed order, i.e., $N \cdots P_1 \cdots P_2 \cdots \Delta$ and N --- P_2 --- P_1 --- Δ. Then $N \cdots P_1$ --- Δ is also a path from N to Δ, and it does not contain P_2, which is a contradiction. Thus in each cycle-free path from node N to Δ, all post dominators of N occur in the same order, and the uniquely determined first of them is the immediate post dominator of N.

The determination of the immediate post dominator of an elementary block is trivial, because in this case immediate post dominator and successor are identical. To determine the immediate post dominator for a decision A, all the cycle-free paths from A to Δ are needed, and the junction point of all of these paths must be found.

In Fig. 1, for instance, the cycle-free paths from A to Δ are $A\ b\ C\ g\ \Delta$ and $A\ c\ d\ B\ f\ C\ g\ \Delta$. The junction point of these paths is C. Thus C is the immediate post dominator of A. From B only one cycle-free path goes to Δ, namely $B\ f\ C\ g\ \Delta$. Thus f is the immediate post dominator of C.

This recipe for finding immediate post dominators is not the best one from an efficiency point of view. For a better one the reader may refer to [14]. Any of these algorithms produces a table of the immediate post dominators of all graph nodes. For our example it is as shown in Table 1.

This table is the key to understanding the entire program logic. By analyzing the given program (∇) we can say it starts with a, does some computation depending on the decision A (the module determined by A), and, independently of what the input data are, each terminating computation comes to module C, executes g afterwards, and finally comes to the end.

Table 1 Immediate post dominators.

Node	∇	a	A	b	c	d	B	e	f	C	g
Immediate post dominator	a	A	C	C	d	B	f	c	C	g	Δ

In a more convenient formal notation we express this main flow of control of the program in the form of the following equation (or production if we use the terminology of the theory of syntax):

$$\nabla = a\,A\,C\,g.$$

Blocks a and g are referred to as terminals in this production because they already have a meaning (the statement sequence denoted by them) and need no further definition. Modules ∇, A, and C are nonterminals, and the definition of ∇ is given by the above equation. The definition of A can be derived in two steps. At first it is shown that A is based upon a decision resulting in one of two possible alternatives. We express this by:

$$A = \{\cdot\cdot|\text{---}\}$$

and read it, for instance, as: IF $\underline{A}$ THEN $\cdots$ ELSE ---.

As a next step the two alternatives are constructed. In the first case (when the condition $\underline{A}$ is true) the statement sequence $\underline{b}$ is executed first. If we trace the main flow of control (i.e., the chain of immediate post dominators) from b to the end we get $b\ C\ g\ \Delta$. However, we need the flow of control only "within the scope" of A, which means that we have to stop the chaining process as soon as the immediate post dominator of the surrounding module is reached. This is because the other part has already been described in another equation.

In our case the surrounding module is A, and its immediate post dominator is C. Thus the alternative $b\ C\ g$ has to be restricted simply to b (the rest belongs to the main module ∇). Analogously we get $c\ d\ B\ f$ as the second alternative of A, and thus we arrive at the equation:

$$A = \{b|c\,d\,B\,f\}.$$

The two missing modules are finally described by

$$B = \{e\ c\ d\,B|\ \varnothing\}$$
$$C = \{\varnothing|d\,B\,f\,C\}$$

where $\varnothing$ denotes the empty alternative.

We refer to ∇ as an unconditional module (no decision is involved), whereas A, B, C are conditional ones. Elementary blocks $(a, b, \cdots)$ can be regarded as unconditional modules too. They define the sequence of basic statements that they represent, and the corresponding equations could be added to the above system.

If a module is defined in terms of itself (such as B and C), we call it recursive. Thus ∇ and A according to this definition are nonrecursive. A recursive module is the equivalent of a loop in a GOTO program. Module C shows a nesting of loops.

The obtained production system is, in terms of the theory of syntax, a regular one. For the reasons pointed out in the introduction this result is, however, distinctly different from the translation of a GOTO program into a regular expression.

At this point two things are still needed to enable the reader to properly assess the method. The first is to show that the new structured program is still equivalent to the old program. The second is to verify whether the attribute "structured" can really be applied to the obtained program representation. To be able to prove the equivalency we show how the new programs can be executed by a push-down machine with a stack as the main control element. At the beginning of any execution this stack contains only ∇. At each execution step the top element of the stack is processed as follows:

1. If Δ is the top element, the stack is "popped up" and execution ends.
2. If the top element is a terminal ($\neq \Delta$), then the sequence of basic statements denoted by it is executed in exactly the same way that it would be in the corresponding flow diagram. Again the stack is popped up afterwards.
3. If the top element is a nonterminal, then the corresponding decision (if any) is made and the resulting alternative replaces the nonterminal on the stack.

Proposition 2 If the new program is executed according to the above control mechanism, then the same computations are performed that are in the corresponding flow diagram.

Proof (given in more detail in [15]) All that really has to be shown is that the sequence of stack top elements in an execution of the new program forms a path of the original flow diagram. And a sufficient condition for this is that the stack elements always form a chain of immediate post dominators. This is so because for elementary blocks the immediate post dominance relation is identical with the successor relation, i.e., the next top element to be processed is, for these blocks, always their uniquely determined successor. And decision making

does not bring more than one of the possible successors of the decision to the top.

To show that ths stack elements always form a chain of immediate post dominators requires induction as well as going back to the process of construction of alternatives. Each alternative always forms a chain of immediate post dominators (according to construction). After an alternative we always come to the immediate post dominator of the surrounding module (this was the means by which alternatives were restricted to the scope of a decision). Thus when a module is replaced by one of its alternatives, the chain of immediate post dominators remains undisturbed. This was so from the beginning, when the stack consisted only of ∇. This completes the proof.

We now want to analyze the usefulness of the obtained program structure. For one thing it certainly yields to top-down reading. By looking at, for instance, $\nabla = a\ A\ C\ g$, we immediately see that each program always starts with some initialization a, goes through the two main modules A and C, and ends with g. If we already have a rough idea of what modules A and C do, this information may be enough. Only if more detail is needed would we determine the definitions of modules A and C and continue in this way until the desired level of detail is reached. For a practical application a good referencing mechanism might be needed, which again can be generated automatically (this was done, for instance, by the automatic editor of the PL/I definition document [16], in which the Vienna Definition Language (VDL) allowed for modular programs only).

The top-down argument holds up to a point, however, for all kinds of modular programs, which, as the methods described in [9] and [10] show, in no way guarantees readability. What really distinguishes the obtained modules from arbitrary ones is the scope concept. In our case the scopes are always minimal. Whenever a statement occurs in an alternative of a conditional module, then whether or not it will be executed inherently is dependent upon the resolution of the corresponding condition, and this is true for the statements within this scope only.

This minimality of decision scopes gives the derived programs the property that, whenever it is known that an alternative is to be executed, we know that each statement contained in it has to be executed too. This gives a precise look-ahead effect that in [1] led to the dynamic exploitation of maximal parallelism and that should be useful for paging techniques as well.

The specific association of predicates with program blocks shows finally that the obtained modules are of the same category as the WHILE statement and thus justifies the attribute "structured" on a semantical rather than a syntactical basis.

Complete modularization

The modularization method described in the last section ended with a program of the following form:

$\nabla = a\ A\ C\ g$
$A = \{b|cdBf\}$
$B = \{ecdB|\varnothing\}$
$C = \{\varnothing|dBfC\}$

If $a, b, \cdots$ are "hidden" modules (i.e., are not to be expanded at each occurrence but are only referred to and defined somewhere else), a program representation is obtained in which the statements of the original flow diagram occur exactly once. Nevertheless the program contains redundancies in the form of sequences of module references (such as $c\ d\ B$ or $B\ f$), which occur more than once and should (for readability reasons) be combined into new unconditional modules. Because these modules serve the purpose of abbreviation, we refer to them as abbreviation modules. The above program shows that there is no unique way of introducing abbreviation modules. One could combine either $c\ d\ B$ and $d\ B$ or $c\ d$ and $B\ f$ into new modules. The procedure to be described gives one of the possible solutions. It could be merged with the structuring method of the last section into a single step. This is not done for reasons of clarity only.

We start by introducing two new notations. When a module consists of more than one alternative, we refer to these alternatives by properly indexing the module's name. Thus A_1 denotes the first alternative of A, A_2 the second alternative, and so on. Each (hidden or nonhidden) module can also be associated with a list of all those alternatives in which it occurs. With $B/C_2; A_2; B_1$ we denote, for instance, that B is referred to in C_2 as well as in A_2 and in B_1.

From proposition 1 it is known that the set of all program modules is partially ordered by the post dominance relation. Thus a tree graph can be constructed the nodes of which are labeled by module names and which shows exactly this partial ordering. The information contained in this tree together with the knowledge of which modules belong to which alternatives is in essence the information contained in the program representation derived in the last section. If we put this information into graphic form, we obtain what can be called the post dominance tree of the program. For the given example the post dominance tree is shown in Fig. 2 (module ∇ is omitted for convenience).

Each of the alternatives in the graph in Fig. 2 has exactly one begin node and one end node. If alternatives do overlap (i.e., have nodes in common) we put them in the order of their end nodes. One alternative thus comes before another one if its end node is nearer to the bottom of the tree than the end node of the other one

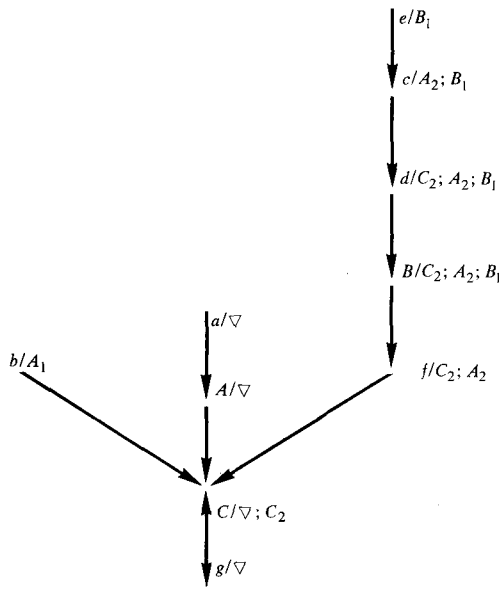

Figure 2 Post dominance tree.

(i.e., alternatives with farther reaching ranges come first). In Fig. 2 this ordering has already been anticipated. If overlapping alternatives have the same end node, we say they form a group. Groups in Fig. 2 are separated by a semicolon.

The introduction of an abbreviation module in a tree such as that shown in Fig. 2 is meaningful whenever there is a sequence of at least two nodes contributing to at least two different alternatives. Because we want to keep the number of abbreviation modules minimal, we want to make the sequences to be abbreviated maximal. Thus we process the nodes of the above tree in top-down order (i.e., whenever a node is processed, then all nodes before it have already been processed). Whenever during this processing a node is reached that satisfies the above two conditions (involvement of at least two nodes and at least two alternatives), then an abbreviation module is to be introduced. Two cases can be distinguished:

1. There is more than one alternative with maximal range (i.e., the first group of alternatives attached to the node has a cardinality of at least two). In this case the range of the new module equals this maximal range.
2. There is only one alternative with maximal range. In this case the range of the new module equals the second highest range.

In Fig. 2 for the first time the need for introducing an abbreviation module is recognized when the node c/A_2; B_1 is processed. Because there is only one element in the first group (namely A_2), the range of the new module is the range of B_1 (i.e., nodes c, d, and B have to be combined). As soon as both the begin node and the range of the new module are known, this module is introduced according to the following steps.

1. The begin node of the new module is split in the post dominance tree, and the old node is made a separate entry point of the tree. The new node is labeled by the name of the new module and connected to the successor of the end node of the new module. For the above example this looks as shown in Fig. 3 (the name of the new module being D).
2. Attached to the new node are the group(s) of alternatives in which the new module will occur, and these groups are replaced by the new module within its range. In terms of the above example, this is as shown in Fig. 4.

For the newly introduced node no further processing is needed. In the other involved nodes the number of contributing alternatives has decreased. By iterating the above process we get a terminating procedure that completely modularizes an arbitrarily given program. For the given example the result is shown in Fig. 5. Directly from this diagram the folowing new, completely modularized program can be derived:

$$\nabla = aACg$$
$$A = \{b|Df\}$$
$$B = \{eD|\varnothing\}$$
$$C = \{\varnothing|EfC\}$$
$$D = cE$$
$$E = dB$$

The obtained program form has the property (according to construction) that no sequence of two or more modules occurs more than once. The same module may occur several times, however.

At this point the question naturally arises as to whether this is the ultimate amount of modularization that can be achieved. The answer is no, but we can go farther only if we use data flow information about the program. Into this category fall, for instance, the taking out of statements of a decision scope (if, for instance, the same statement occurs in all alternatives as the first statement and it also has no influence on the involved decision), as well as the recognition of whether or not two different modules are equivalent (and thus could be merged into one).

We regard these transformations (which require program knowledge beyond the flow of control information) as part of the optimization of the program and not as part of the structuring process. Thus they are outside the scope of this paper.

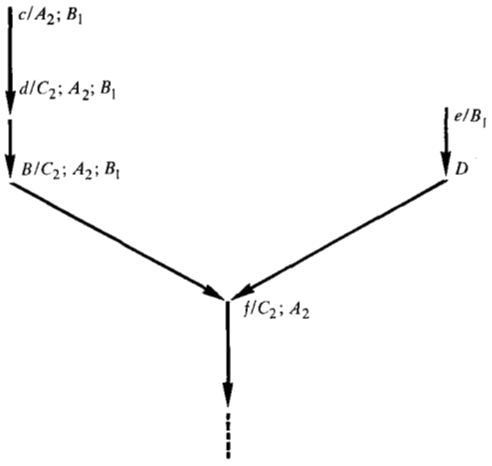

Figure 3 Step 1 for the introduction of an abbreviation module.

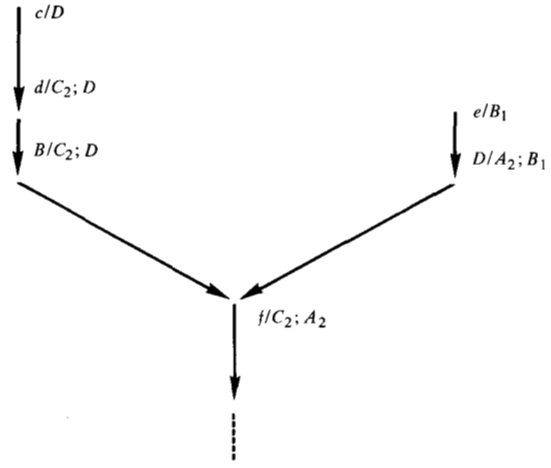

Figure 4 Step 2 for the introduction of an abbreviation module.

Backtranslation of recursively structured programs into GOTO programs

In the second section it was shown how a recursively structured program could be executed by means of a push-down machine. From a practical point of view this does not seem to be the best way to do it, at least not with respect to existing machinery.

A simpler way to execute a module is to make a subroutine call of an assembly language type, i.e., to branch to the module, after storing the return address, and to return to the stored address after execution of the module. Since we started from a pure GOTO program (with no thought of return addresses) even this looks too complicated. In this section we therefore show how the modules can be physically arranged in such a way that the return, and thus the storing of the return address, become superfluous.

Because we are interested in the real code arrangement, we add the definitions for the elementary blocks to the given program. For the chosen example this gives:

$$\nabla = aACg$$
$$A = \{b|Df\}$$
$$B = \{eD|\varnothing\}$$
$$C = \{\varnothing|EfC\}$$
$$D = cE$$
$$E = dB$$
$$a = \underline{a}$$
$$\vdots$$
$$g = \underline{g}$$

As the next step the "spanning tree" of this program is formed (a concept that is modeled after the spanning tree concept of Tarjan [14] but is slightly different).

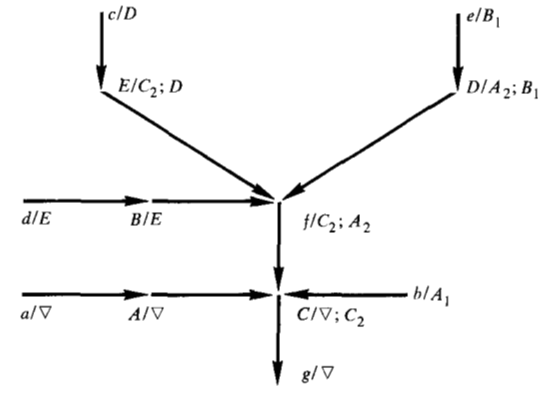

Figure 5 Completely modularized post dominance tree.

This tree is constructed top-down, starting with the main module and expanding each module into its alternatives. This expansion is done, however, only for modules that have not been expanded before. If additionally it is assumed that modules to the left are expanded before modules to the right, we get what has been called a depth-first spanning tree. For the above program it has the form shown in Fig. 6.

The tree defines a sequencing relation between its elements. If we assume Δ to come after ∇, then any element (other than ∇) in the tree is followed by the element immediately to its right. If there is no such element (as is the case at the end of an alternative), then that element is followed by whatever comes after its "father" (i.e., the module from which that element was derived by expansion).

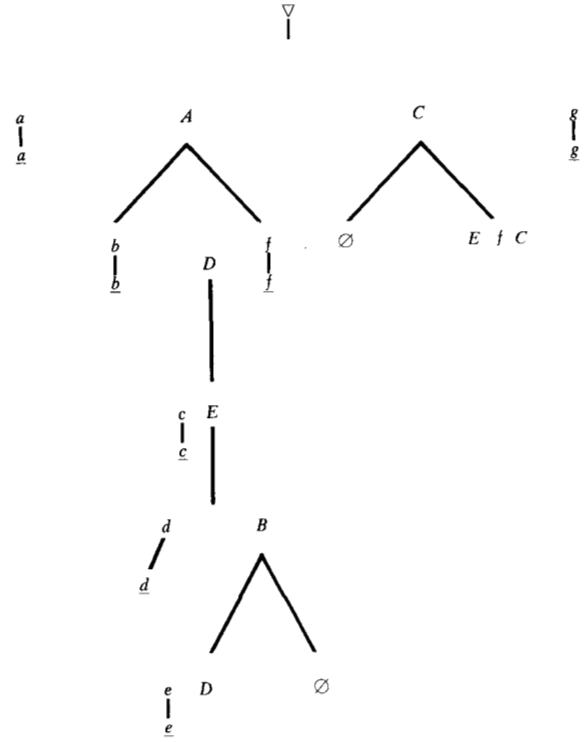

Figure 6 Depth-first spanning tree.

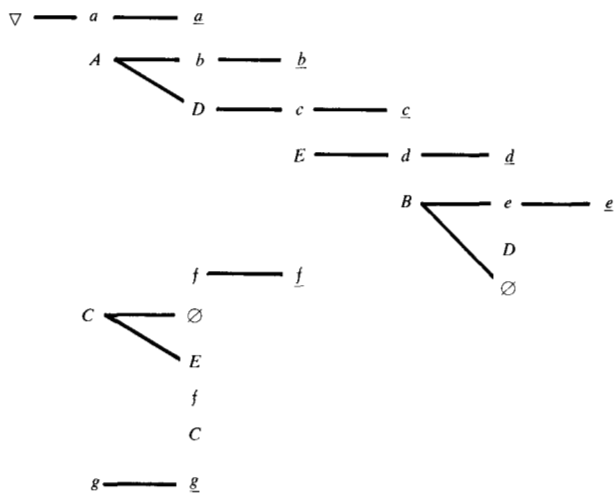

Figure 7 Depth-first spanning tree in left-to-right form.

From the logic of the program construction process it is known that this sequencing relation means a corresponding sequencing in the execution as well. Thus A has to be executed after a, C after A, and so on. To model this execution logic in the physical arrangement of the modules, the above tree is shown once more in Fig. 7, but this time with reversed directions (top-down is translated to left-to-right and vice versa) and using proper identations.

All that has to be done to interpret this tree as a GOTO program is the following:

- Module names at the module definition place (lines to the right) are to be read as labels.
- Module names at a module reference place (no lines to the right) are to be read as GOTOs to the corresponding labels.

In addition the following simplifications can be made:

1. Any GOTOs immediately following other GOTOs can be eliminated, because they never can be executed (in Fig. 7 these are the references to f and C following the reference to E).
2. Labels not being branched to can be eliminated (∇, a, A, $\cdots$ in Fig. 7).

If we apply this simplification to Fig. 7, the following result is obtained (written in a PL/I-like programming style):

```
a,
IF A THEN b; ELSE DO;
   D: c;
   E: d;
   IF B THEN DO;
      e;
      GOTO D;
      END;
   f;
   END;
IF C THEN; ELSE GOTO E;
g;
```

This program (and such programs in general) contains only backwards branches, and there is no duplication of code whatsoever. Because there is no way of avoiding the backwards branches, we can also say that it is a GOTO program with a minimal number of GOTOs and thus the best that can be expected in this respect.

Translation of recursively structured programs into interatively structured ones

The program form obtained in the second and third sections mirrored precisely the logic of the analyzed program. Nevertheless, the specific representation might

still be objectionable. Although syntax does not add meaning by itself, it has a psychological importance that cannot be ignored. To bring, for instance, the program obtained in the second section into a more appealing form, the following steps can be performed.

1. Definitions of nonrecursive modules of the form $M = \{\sigma|\tau\}$ (σ and τ denote arbitrary strings) are written as $M =$ IF $\underline{M}$ THEN σ, ELSE τ.
2. Module definitions of the form $M = \{\sigma M|\varnothing\}$ are written as $M =$ WHILE $\underline{M}$ DO σ and analogously $M = \{\varnothing|\sigma M\}$ is translated into $M =$ WHILE ¬ $\underline{M}$ DO σ. Steps 1 and 2 together give the following result (the semicolon is used now as a statement delimiter, and parantheses are used to group statements together).

$\nabla = a; A; C; g$
$A =$ IF $\underline{A}$ THEN b; ELSE $(c; d; B; f)$
$B =$ WHILE $\underline{B}$ DO $(e; c; d)$
$C =$ WHILE ¬ $\underline{C}$ DO $(d; B; f)$

3. The method described in the third section for complete modularization is applied. (Because alternatives may have shorter ranges now, due to the WHILE construct, different abbreviation modules can be expected.) The result is

$\nabla = a; A; C; g$
$A =$ IF $\underline{A}$ THEN b; ELSE $(D; E)$
$B =$ WHILE $\underline{B}$ DO $(e; D)$
$C =$ WHILE ¬ $\underline{C}$ DO $(d; E)$
$D = c; d$
$E = B; f$

4. In a program representation of this kind many modules are referred to only once, which is very often undesirable. In-line expansion of these modules (which corresponds to an interpretation of the respective module references as macro calls) gives, together with an adding of the hidden modules, the following final result.

$\nabla =$ $\underline{a}$;
IF $\underline{A}$ THEN $\underline{b}$; ELSE $(D; E)$;
WHILE ¬ $\underline{C}$ DO $(d; E)$;
$\underline{g}$
$D =$ $\underline{c}$; d
$E =$ WHILE $\underline{B}$ DO $(\underline{e}; D)$; $\underline{f}$
$d =$ $\underline{d}$

In the rest of this section we concentrate on step 2, i.e., on the question of how recursively structured modules can be expressed in terms of iterative language features, such as the IF, WHILE, DO FOREVER, and multilevel EXIT statements. We restrict ourselves thereby to modules with only two alternatives.

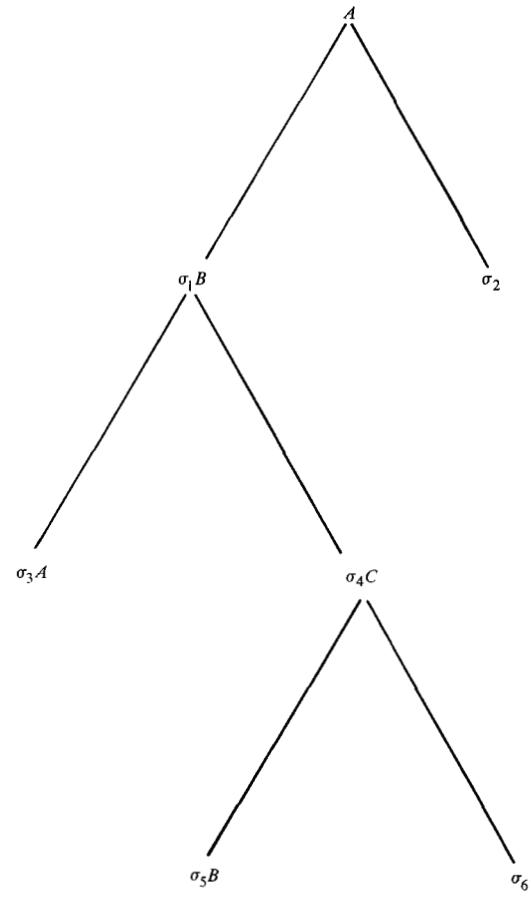

Figure 8 Expansion tree.

In the previous section the concept of a depth-first spanning tree was used. In it module names occurred in three different ways: to denote the definition place of the module (there was a subtree under the module's name), to denote a back reference (the name occurred within the definition tree of the module), or to denote a cross reference (the name occurred outside of the definition tree).

The existence of cross references indicates that the module is used independently in different contexts. Cross references could always be avoided if a copy of the module were made at each cross reference place, such that all references become back references only. In the following text, the concept of an expansion tree is introduced. It is similar to the spanning tree concept but has the following differences:

1. The root module can be an arbitrary module M.
2. All those module references and only those module references are expanded that are not back references and that yield, directly or indirectly, either M or one of that module's ancestors (a module from which it was derived by expansion).

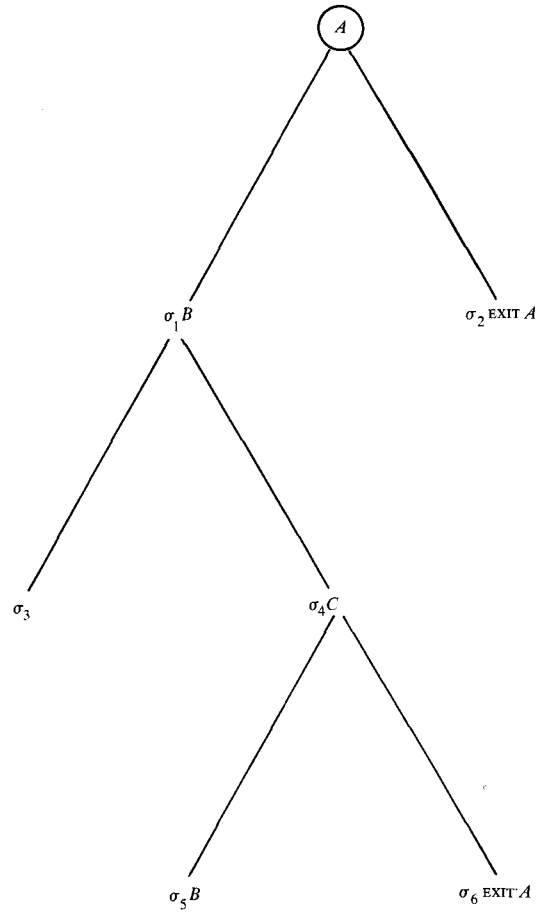

Figure 9 Expansion tree in partially iterative form.

In contrast to the depth-first spanning tree, modules are expanded not only at the leftmost occurrence but at every occurrence that is not a back reference. Thus during the expansion process several copies of the same module may be made. If we distinguish them from each other by unique names (and correspondingly change the involved back references), we get a tree that we call the expansion tree of the module M. An example of an expansion tree with nested recursion is given in Fig. 8.

One of the basic properties of an expansion tree (again in contrast to the depth-first spanning tree) is that only those modules in it are expanded that occur in a module sequence in a rightmost position. This is so because otherwise a module M could be expanded into a sequence such as --- M N ---, where N would be both an immediate post dominator of M and within the scope of M. The leaves in the expansion tree are module sequences in which no further expansions occur. Such sequences that do not end with a back reference are called the exits of the tree.

The basic philosophy of the iterative language feature is to anticipate eventual recursions from the very beginning. If we do this, then this means, in terms of the expansion tree, that all references to the root module become redundant and can be deleted. On the other hand we do have to plug the exits of the tree to avoid the occurrence of an invalid iteration in these cases. This can be done by adding an EXIT M statement to the right of each exit, which forces the control to leave the module at these places.

The expansion tree of Fig. 8 is thus brought into the form shown in Fig. 9 (the circle stands for the anticipated permanent repetition of the module).

Now the other involved recursive modules have to be translated into iterative form by processing them in top-down order in the same way as the root module was processed. Care has to be taken only of what the exits of the new subtrees are. Everything that is already plugged by an EXIT statement can no longer be regarded as an exit (an exit from a module in a higher position implies an exit from the modules in a lower position as well). Exits are now those leaves in the tree where recursive references have been deleted (σ_3 in Fig. 9). The result derived from Fig. 9 after proceeding in this way is shown in Fig. 10.

The resulting tree is always free from any back references (recursions). Of course it may contain references to other "self-contained" recursive modules; these recursions do not show up in the tree, however. Thus it can always be directly coded by using nested IF statements in connection with labeled DO FOREVER statements. From Fig. 10, for instance, the following code can be directly derived.

```
A: DO FOREVER;
   IF A THEN DO;
      σ1;
      B: DO FOREVER;
         IF B THEN (σ3; EXIT B); ELSE (σ4; IF C THEN
            σ5; ELSE (σ6; EXIT A));
         END B;
   ELSE (σ2; EXIT A);
   END A;
```

The DO FOREVER statement, particularly in its nested versions, is not really convincing with respect to its readability. We now give the three special cases in which we can do without it:

1. Module M is defined as $M = \{\sigma M|\varnothing\}$. Instead of writing

```
M: DO FOREVER;
   if M THEN σ; ELSE EXIT M;
   END M;
```

we certainly prefer the shorter WHILE M DO σ.

2. Only the left alternative of M is recursive, and the right alternative is nonempty (this implies that the recursiveness is an indirect one), i.e., M is defined as $M = \{\sigma\ N|\tau\}$, and expansion of N gives M again. Also in this case the module M can be translated similarly to 1), but this time the WHILE (analogously to the IF) has to be combined with an ELSE clause (containing the right alternative of M). The point of the ELSE clause is that is belongs to M too and thus is bypassed by an EXIT M.
3. The module M is recursive in its right alternative only. In this case the Boolean expression belonging to M is negated, the alternatives are swapped, and the module is treated as described under 1) and 2).

Altogether in the translation of a conditional module M into iterative form the following cases can be distinguished:

1. Module M is nonrecursive.
   Iterative counterpart: IF statement.
2. Module M is recursive but in one alternative only.
   Iterative counterpart: WHILE statement in connection with the EXIT statement.
3. Module M is recursive in both alternatives.
   Iterative counterpart: DO FOREVER statement in connection with the EXIT statement.

If a whole program is translated into iterative form (and not just a single module), then the modules of the program should be translated in top-down order, in the sense of the depth-first spanning tree, and only those modules need be translated that have been previously referred to (this saves the redundant translation of those modules that occur at an intermediate position in other expansion trees only).

A great inconvenience in the described translation process is the permissive duplication of code. The duplication is necessary when the same module occurs within different "contexts." By context we mean the list of those ancestor modules that occur in the scope of the module again. They are exactly the modules that cause the introduction of the EXIT statements and thus make the new module context-dependent. Thus we can do without the duplication if the two module occurrences are within the same context. Even then the result is not too impressive. It leads to stand-alone modules containing EXIT statements that are less readable (because of the missing target) than the original GOTO statements were that they replaced.

Altogether a mixture of the iterative and the recursive approach seems to provide the most satisfying solution to the structuring problem: To take the IF and WHILE (and perhaps the UNTIL) statements in the simple cases (where no EXIT statements are involved) and to make use of (recursive) references in all others.

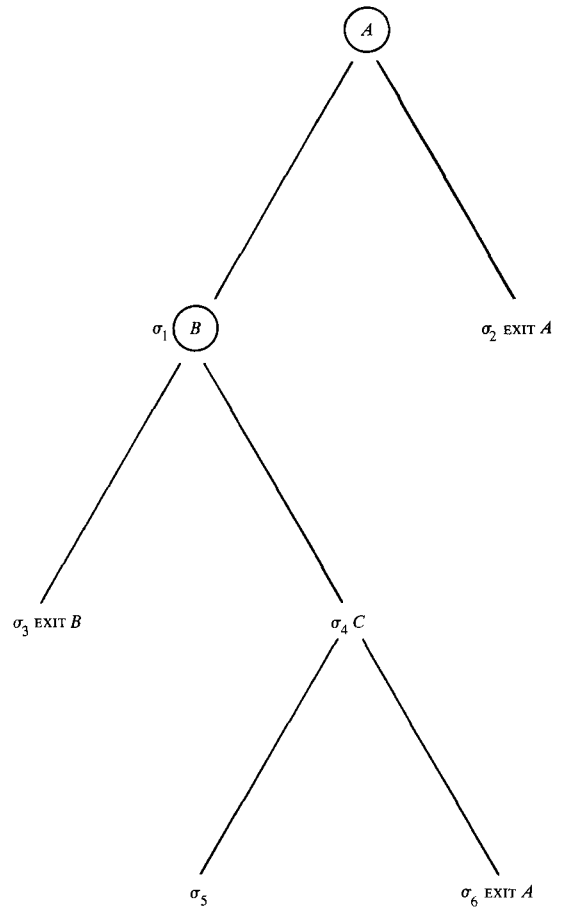

Figure 10 Expansion tree in completely iterative form.

Extensions

• *Multi-exit blocks*

One of the most common features in higher level languages is blocks—either of the simple parentheses-like type (DO; ··· END; or BEGIN; ··· END;) or of the DO loop type (DO $\underline{a}$; ··· END;—where a is the header of the block controlling the interation). When using blocks, the programmer associates meaning with them. Thus this structure should be carried over into the structured program version. As long as blocks have only one exit the shown structuring process can be immediately applied. All that has to be done is to create an unconditional module for the block. In this section we want to solve the problem of structuring a program containing blocks with several exits (i.e., blocks with embedded GOTOs that leave the block abnormally). Consider, for instance, the following very common program (skeleton) fragment (see [17]):

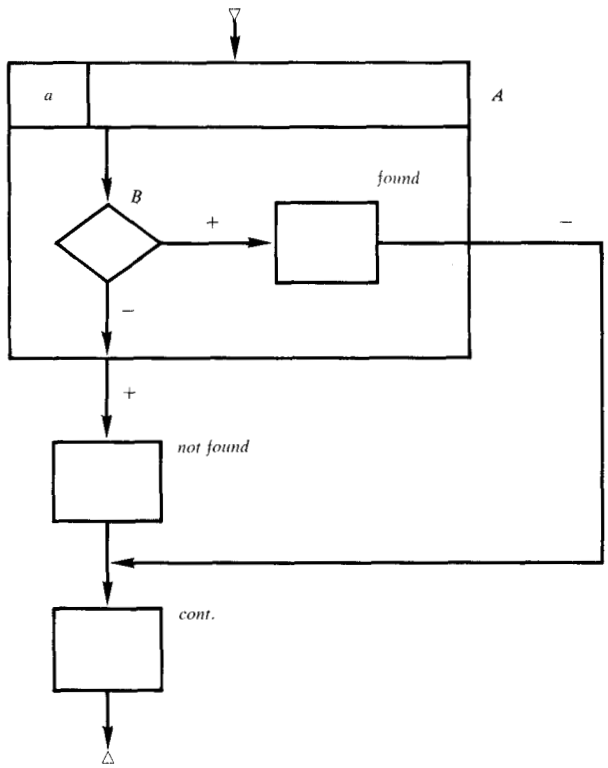

Figure 11 Flow diagram skeleton for multi-exit block.

```
DO a;
   IF B THEN (found; GOTO CONT);
   END;
   not-found;
CONT: cont;
```

A specific interpretation of this skeleton could show, for instance, a search routine through an array where, dependent upon the finding of a certain element, different routines (*found* and *not-found*) are executed. In the two-dimensional flow chart skeleton form the above example could be shown as in Fig. 11. The *A* in Fig. 11 stands for the whole block and *a* for its header only. The single arrow ending at the block contour denotes the conditional exit (depending upon the header); all other exits are unconditional ones. A programmer confronted with the task of structuring the above program will normally end with the following "structured" version of it:

```
DO a WHILE ¬ B;
   END;
IF f(a, B) THEN found; ELSE not-found;
cont;
```

With $f(a, B)$ we thereby denote a test on the outcome of the search. There are two things wrong with the above approach. For one it is an example of tricky (and thus opaque) programming (the DO loop pretends to do nothing but nevertheless passes a result hidden, for instance, in a control variable). For the other the connection between the two program parts is not visible in the flow of control but simulated in the data flow. Especially if the two parts are placed remotely from each other or if $f(a, B)$ is a tricky expression again, the connection may be quite difficult to find.

If we look at Fig. 11 more carefully, we almost immediately see that the whole block *A* is nothing more than a kind of decision that requires, instead of evaluating a simple Boolean expression, execution of a more complex program fragment. Decisions of this kind involve flow of control. Thus we can no longer limit ourselves to abstractions of them. To make this fact explicit we introduce a new module type, called a decision module. It occurs in front of the definition of a conditional module and is put in parentheses (for instance $(\alpha)\ \{\cdots \mid ---\}$). Its meaning is that the selection of alternatives is dependent upon the outcome of this module. For convenience we assume that the first of these alternatives is the one that is executed when the block is left normally.

With these preparations we can specify the structuring process for programs containing multi-exit blocks as follows:

Step 1 Outmost multi-exit blocks are interpreted as (visible) decisions, and besides this the structuring process is performed as it was before for simple flow diagram skeletons. For the above example this gives:

$\nabla = A$, *cont*
$A = (\alpha)\ \{$*not-found*$|\varnothing\}$

Step 2 The definitions for all decision modules encountered in step 1 are derived. Because we want to keep the block structure, this is an almost trivial step. We need, however, a new notation and introduce ⟨block header/block body⟩ as a self-explanatory notation for describing a block.

Step 3 All encountered block bodies are structured (treating nested blocks in the same way as the outermost ones). To be able to use the base algorithm for this purpose we need the one-begin and one-end structure of the respective block bodies. This is achieved by closing all abnormal exits with a CASE*i* statement (where *i* gives the connection to the succeeding alternative) and connecting them afterwards to the main exit. For the above example steps 2 and 3 give:

$\alpha = \langle a/B \rangle$
$B = \{$*found*; CASE 2$|\varnothing\}$

Step 4 As soon as a multi-exit block is regarded as a kind of a decision, it should contain only code serving the purpose of making that decision. Thus whenever a CASE*i* statement is immediately preceded by some code that is "self-contained" (i.e., does not contain other exits out of the same block), this code should be moved out of the block body and placed in front of the alternative indicated by the respective CASE*i* statement. For the above example the resulting structured program text thus is:

$\nabla = A;\ \underline{cont}$
$A = (\alpha)\ \{\underline{not\text{-}found} | \underline{found}\}$
$\alpha = \langle \underline{a} / B \rangle$
$B = \{\text{CASE } 2 | \varnothing\}$

If we now look at the definition of module A, we can see a much more meaningful structure than the one shown before. It shows that a (programmed) decision α has to be performed depending on which one of two possible alternatives has to be executed. This is exactly the structure that would be obtained if we had programmed top-down correctly and used levels of abstraction. It exposes the search routine α (which could also be defined in another way) as well as the resulting actions, and it gives, independently of their specifications, the proper connection.

In PL/I a block with two exits could be coded by means of a truth value function. For more than two exits a new language feature (not yet in PL/I) is needed to obtain a proper structuring tool.

• *Label variables*

In a GOTO program a GOTO with a label variable as target is nothing more than a disguised conditional branch statement (dependent upon the value of the label variable, a branch is made). To be able to define the corresponding conditional module the "range" of the label variable at this point has to be determined. This can be done by either worst case assumptions (all labels occurring in the program) or by making a flow analysis as used in optimization techniques.

• *Subroutines*

We assume subroutines to be left by return statements only. The structuring is then performed quite simply by structuring each subroutine separately (subroutine calls are treated as data transformations, i.e., as basic statements). If a subroutine has several entry points, then the subroutine is structured as often as there are entry points, assuming in each case a different entry point as the program beginning ∇.

As a rather open problem we consider the structuring of programs involving (recursive) subroutines, label variables and their aliases, abnormal exits, external and static attributes in arbitrary combination. Further investigations along these lines seem to be necessary to get a deeper understanding of the nature of structured language concepts.

Summary

It has been shown that GOTO programs can be transformed into recursively structured ones in such a way that both the program logic is exposed and the conciseness of the GOTO is kept.

An extension of the method to GOTO programs containing block structures and subroutines has been given.

Acknowledgments

The author is grateful for the many useful suggestion made by the referees, which had a significant impact on the final version of this paper. He also acknowledges the influence of the enjoyable paper by Knuth [15].

References

1. G. Urschler, "The Transformation of Flow Diagrams into Maximally Parallel Form," *Proc. 1973 Sagamore Comp. Conf. on Parallel Proc.*, Syracuse University.
2. S. C. Kleene, "Representation of Events in Nerve Nets." *Automata Studies*, Princeton University Press, 1973, p. 3.
3. C. Boehm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules," *Comm. ACM* **9,** 366 (1966).
4. E. Ashcroft and Z. Manna, "The Translation of GOTO Programs to WHILE Programs," *Proc. IFIP Congress 1971*, p. 250.
5. D. E. Knuth and R. W. Floyd, "Notes on Avoiding GOTO Statements," *Info. Proc. Letters* **1,** 23 (1971).
6. J. Bruno and K. Steiglitz, "The Expression of Algorithms by Charts," *J. Assoc. Comput. Mach.* **19,** 517 (1972).
7. W. W. Peteron, T. Kasami, and N. Tokura, "On the Capabilities of WHILE, REPEAT, and EXIT Statements," *Comm. ACM* **16,** 503 (1973).
8. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972.
9. J. McCarthy, "Towards a Mathematical Science of Computation," *Proc. IFIP Cong.*, North Holland Publishing Co., Amsterdam, 1962.
10. A. Van Wijngaarden, "Recursive Definition of Syntax and Semantics," *Formal Language Description Language for Computer Programmers*, North Holland Publishing Co., Amsterdam, 1966.
11. C. T. Zahn, "A Control Statement for Natural Top-Down Structured Programming," Symposium on Programming Languages, Paris, 1974.
12. R. T. Prosser, "Application of Boolean Matrices to the Analysis of Flow Diagram," *Proc. Fall Joint Computer Conference 1959*, p. 133.
13. E. S. Lowry and C. W. Medlock, "Object Code Optimization," *Comm. ACM* **12,** 13 (1969).
14. R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Computing* **1,** 146 (1972).
15. G. Urschler, "The Inherent Parallelism of Flow Diagrams," *Technical Report TR25.129*, IBM Laboratory, Vienna, Austria, 1972.

16. K. Walk, K. Alber, M. Fleck, H. Goldmann, P. Lauer, E. Moser, P. Oliva, H. Stigleitner, and G. Zeisel, "Abstract Syntax and Interpretation of PL/I," *Technical Report TR25.098*, IBM Laboratory, Vienna, Austria, 1969.
17. D. E. Knuth, "Structured Programming with GOTO Statements," *Report STAN-CS-74-416*, Stanford University, 1974.

Received April 2, 1974; revised September 20, 1974

The author is located at the IBM System Development Division Laboratory, Vienna, Austria.