SIAM J. COMPUT.
Vol. 4, No. 2, June 1975

TRANSLATING PROGRAM SCHEMAS TO WHILE-SCHEMAS*

EDWARD ASHCROFT† AND ZOHAR MANNA‡

Abstract. While-schemas are defined as program schemas without goto statements, in which iteration is achieved using while statements. We present two translations of program schemas into equivalent while-schemas, the first one by adding extra program variables, and the second one by adding extra logical variables. In both cases we aim to preserve as much of the structure of the original program schemas as possible.

We also show that, in general, any translation must add variables.

Key words. program schemas, removing goto statements, while-schemas, flowchart transformations.

Introduction. The program schema approach makes it meaningful to consider the relative "power" of programming language constructs. Most work in this area [13], [4], [5], [14], [15] has considered adding features to program schemas such as recursion and arrays. Here we consider removing, or at least restricting, a feature of program schemas: the goto statement.

There has been much interest lately, following observations by Dijkstra [8], in the possibility and desirability of removing goto statements from programming languages, using instead such statements as the while statement. Programs in such languages should be better structured, easier to understand and, hopefully, easier to prove correct. For example, the elegant formal system of Hoare [10] for proving programs correct requires programs with the sort of "nested" structure that while statements provide. Goto-less programs are clearly an interesting class of programs to study.

We therefore define a class of while-schemas in which iteration is achieved with while statements: **while** ψ **do** S. The tests ψ may be arbitrarily complicated; this feature of our while-schemas is crucial. We show that while-schemas are as powerful as program schemas by giving a translation, Algorithm 1, of program schemas to equivalent while-schemas. This translation is interesting in that it preserves most of the "loop structure" of the program schemas, and gives while-schemas of the same order of efficiency. The translation allows the addition of extra program variables.

Bohm and Jacopini [3] have shown that program schemas can be translated into while-schemas, with the addition of extra *logical variables*. A modification of a technique in Brown et al. [2] would show (by a further translation) that the additional logical variables add no extra power to while-schemas. We present an improvement on Bohm and Jacopini's reduction to while-schemas with logical variables, which we call Algorithm 2. This doesn't give us "pure" while-schemas, since logical variables are used, but the schemas produced are often more "readable"

* Received by the editors November 2, 1973, and in revised form March 1, 1974. This research was supported in part by the Advanced Research Projects Agency of the Office of the U.S. Secretary of Defense under Contract SD-183, and in part by the National Research Council of Canada.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

‡ Department of Applied Mathematics, Weizmann Institute, Rehovot, Israel.

than those produced by Algorithm 1. The method preserves whatever "while-structure" already exists in a program schema, and when applied to a program schema corresponding directly to a while-schema, Algorithm 2 gives us back that while-schema.

Both Algorithm 1 and Algorithm 2 give while-schemas that use more variables in general than the original program schemas. It is natural to ask whether this is a necessary feature of any translation. We show that this is the case by giving a program schema, with one variable, for which there is no equivalent one-variable while-schema. This also means, of course, that program schemas *are* more powerful than while-schemas in the restricted sense that they need fewer variables in general.

The construction and proofs presented here first appeared in abbreviated form in a paper by the authors presented at IFIP Congress 1971 (Ljubliana, Yugoslavia).

1. Program schemas. A *program schema* consists of a finite sequence of *statements*, separated by semicolons. This sequence must start with a *start statement*, e.g., START(x_2, x_4), designating *input variables*, and end with a *halt statement*, e.g., HALT(x_1, x_3), designating *output variables*. The other statements may be of the following types:

(i) *null statements*, i.e., **null**;

(ii) *assignment statements*, i.e.,

$$x_i \leftarrow \tau,$$

where τ is a *term*;

(iii) *conditional statements*, i.e.,

$$\textbf{if } \psi \textbf{ then } S_1 \textbf{ else } S_2,$$

where S_1 and S_2 are statements and ψ is a *formula*;

(iv) *compound statements*, i.e.,

$$[S_1; S_2; \cdots; S_n],$$

where $S_1, S_2, \cdots, S_n$ are statements;

(v) *goto statements*, i.e.,

$$\textbf{goto } L_i,$$

where L_i is a *label*.

Any statement can be labeled by preceding it with a label followed by a colon. A *formula* is any quantifier-free formula of predicate calculus. A *term* is any composition of variables, constants and function symbols.

The statement **if** $\cdots$ **then** $\cdots$ **else null** can be written **if** $\cdots$ **then** $\cdots$ provided no confusion results.

Example. The following is a program schema P_1 with one variable, that will be used often throughout the paper:

```
SCHEMA $P_1$. START($x$);
          $x \leftarrow a(x)$;
          $L$: **if** $p(x)$ **then** [$x \leftarrow e(x)$; **goto** $L$];
          $A$: **if** $q(x)$ **then** $x \leftarrow b(x)$ **else** [$x \leftarrow g(x)$; **goto** $N$];
          $M$: **if** $r(x)$ **then** [$x \leftarrow d(x)$; **goto** $M$];
          $B$: **if** $s(x)$ **then** [$x \leftarrow c(x)$; **goto** $L$] **else** $x \leftarrow f(x)$;
          $N$: **null**;
          HALT($x$).
```

A, B, L, M and N are labels (A and B will be used in later discussions). The symbols a, b, c, d, e, f and g denote functions and the symbols p, q, r and s denote predicates or tests. The expressions $p(x)$, $q(x)$, etc. are (simple) formulas.

2. While-schemas. A *while-schema* is a program schema using only statements of types (i), (ii), (iii) and (iv) and type (vi) below:

(vi) *while statement*, i.e.,

$$\textbf{while } \psi \textbf{ do } S,$$

where S is a statement and ψ is a formula.

Such statements are to be considered as abbreviations for the equivalent statements

$$L: \textbf{if } \psi \textbf{ then } [S; \textbf{goto } L] \textbf{ else null}.$$

Example. The following is a while-schema P_2 with two variables.

```
SCHEMA $P_2$. START($x$);
          $x \leftarrow a(x)$;
          **while** $p(x)$ **do** $x \leftarrow e(x)$;
          $y \leftarrow x$;
          **if** $q(x)$ **then** [$x \leftarrow b(x)$; **while** $r(x)$ **do** $x \leftarrow d(x)$];
          **while** $q(y) \wedge s(x)$ **do**
               [$x \leftarrow c(x)$;
               **while** $p(x)$ **do** $x \leftarrow e(x)$;
               $y \leftarrow x$;
               **if** $q(x)$ **then** [$x \leftarrow b(x)$; **while** $r(x)$ **do** $x \leftarrow d(x)$]];
          **if** $q(y)$ **then** $x \leftarrow f(x)$ **else** $x \leftarrow g(x)$;
          HALT($x$).
```

The schema P_2 uses the same symbols as P_1, denoting functions and predicates, and here we have the (more complicated) formula $q(y) \wedge s(x)$.

3. While-schemas with logical variables. A *while-schema with logical variables* is a program schema using only unlabeled statements of types (i), (ii), (iii), (iv) and (vi), and type (vii) below:

(vii) *logical assignment statements*, i.e.,

$$t_i \leftarrow \textbf{true}$$

or

$$t_i \leftarrow \textbf{false}.$$

The variables appearing in logical assignment statements are called *logical variables*, and they may not appear in ordinary assignments. They may appear, however, in formulas, as if they were *propositions*, i.e., 0-ary predicates.

Example. The following schema P_3 is a while-schema with one logical variable (and one program variable).

```
SCHEMA $P_3$. START($x$);
        $x \leftarrow a(x)$;
        $t \leftarrow$ true;
        while $t$ do
            [while $p(x)$ do $x \leftarrow e(x)$;
             if $q(x)$ then [$x \leftarrow b(x)$;
                             while $r(x)$ do $x \leftarrow d(x)$;
                             if $s(x)$ then $x \leftarrow c(x)$
                                       else [$x \leftarrow f(x)$; $t \leftarrow$ false]]
                       else [$x \leftarrow g(x)$; $t \leftarrow$ false]];
        HALT($x$).
```

P_3 uses the same symbols as P_1 and P_2 denoting functions and predicates, and here t is a logical variable used also as a formula.

4. Equivalence of schemas.

Two schemas having the same input variables and the same output variables are said to be *equivalent* if they compute the same function (from input variable values to output variable values), *no matter what functions or predicates are denoted by the symbols in the schema.* (Of course, the same symbol appearing in the two schemas must denote the same function or predicate.)

More formally, we can first give meaning to the symbols in a schema by using an interpretation. An *interpretation I* consists of a domain D_I from which the variables in the schema may take values and a specification of the functions and predicates over D_I denoted by the function and predicate symbols in the schema. The interpretation also supplies initial values (from D_I) for the input variables. Given an interpretation I, a schema S becomes a *program* (S, I). The program has a finite or infinite computation in the usual way, and if this is finite we let val(S, I) denote the final values of the output variables. If the computation is infinite, val(S, I) is undefined.

Two schemas S_1 and S_2 are then equivalent if, for all interpretations I, val(S_1, I) = val(S_2, I), i.e., both are undefined, or both are defined and have the same values.

In most of the paper we do not need the formal definition of equivalence. In these sections we will use simple equivalence-preserving transformations which are clearly correct. However, we do use the formal definition using interpretations in the last section.

Examples. The schemas P_1, P_2 and P_3 are all equivalent. (In fact, P_2 is the result of applying Algorithm 1 to P_1, and P_3 is the result of applying Algorithm 2 to P_1, as we will see later.

To see informally that P_1 is equivalent to P_2, note that each iteration of the main while statement in P_2 corresponds in P_1 to going from label B back to label B. The variable y in P_2, at the beginning of each iteration, holds the value that x previously held in P_1, the last time computation reached label A.

To see informally that P_1 is equivalent to P_3, note that each iteration of the main while statement in P_3 corresponds in P_1 to going from label L back to label L (the long way, via statement labeled B) or to label N. In the latter case, t is made **false** in P_3, and we subsequently exit from the main while statement.

5. Flowcharts. We will find it useful to consider the flowchart representations of schemas. Program schemas clearly correspond to arbitrary flowcharts, with one start node and one halt node, using assignment and test statements as shown in Fig. 1; ψ is a formula and τ is a term.

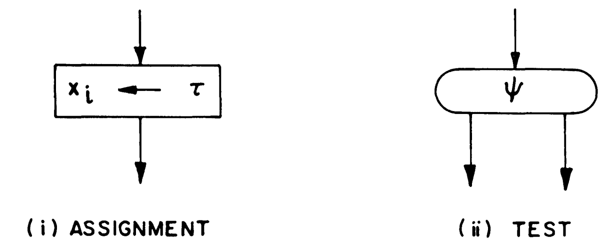

FIG. 1. *Flowchart statements*

We shall be more concerned with *normal forms* for such flowcharts.

5.1. While-chart form. Firstly, it is clear that while-schemas have more restricted "structure" than program schemas, and we define below (inductively) a correspondingly restricted class of flowcharts.

A *while-chart* is a *one-entrance, one-exit* piece of flowchart of one of the five types shown in Fig. 2. The boxes $A_1, A_2, \cdots$ represent while-charts, ψ is a formula and τ is a term. The various cases correspond to the types of statement allowed in while-schemas. Any flowchart of the form of Fig. 3, where A is a while-chart, is said to be in *while-chart form.* For every while-chart form flowchart, there is an equivalent while-schema, and vice versa.

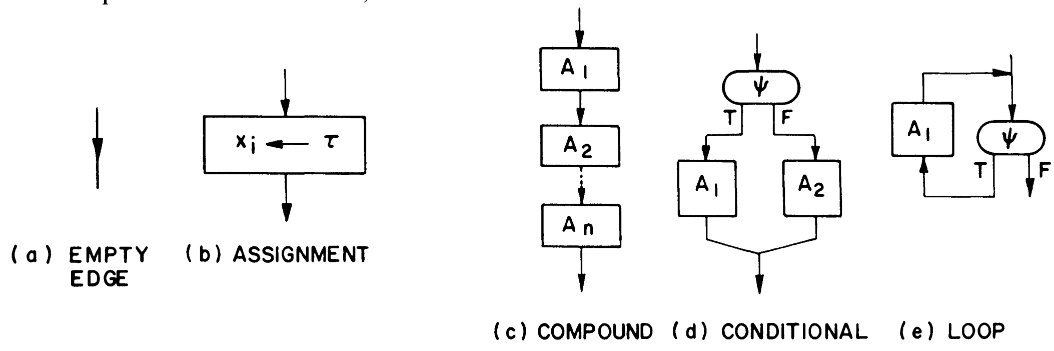

FIG. 2. *While chart constructs*

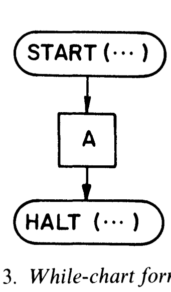

FIG. 3. *While-chart form flowchart*

5.2. Block form. Even general flowcharts can be put into normal forms, by such methods as duplicating nodes, unwinding loops, etc. One such normal form is the block form of Cooper [7] and Engeler [9].

A *block* is a *one-entrance, many-exit* piece of flowchart constructed inductively as follows (we occasionally number the exits from a block, starting at the left):

(i) A *basic block* is a block. A basic block is a one-entrance, many-exit *tree-like* piece of flowchart. An example is shown in Fig. 4.

(ii) The flowcharts in Fig. 5 are blocks, where B_1 and B_2 are blocks.

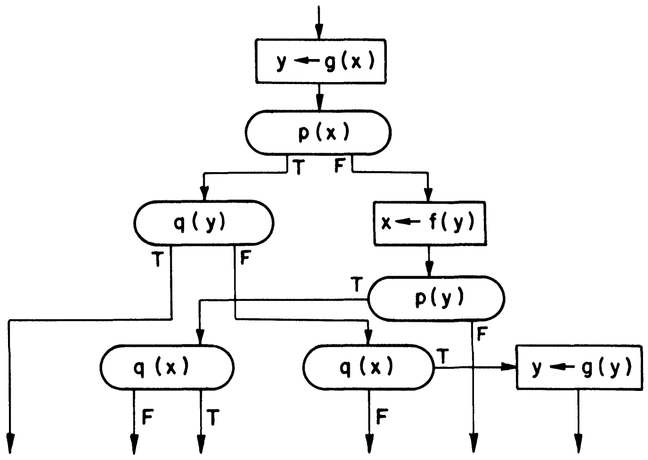

FIG. 4. *Example of a basic block*

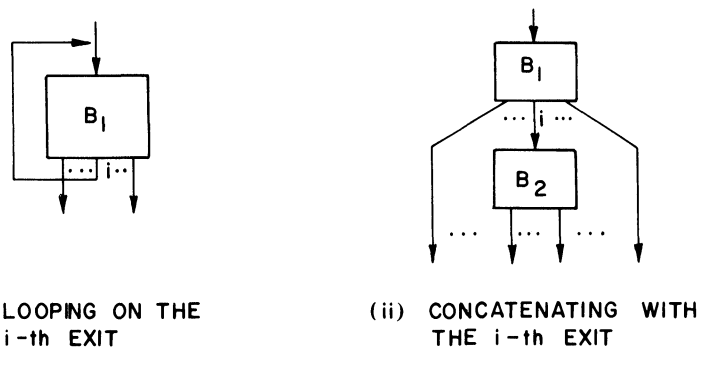
B1
... i ..
B1
... i ...
B2
... ...
(i) LOOPING ON THE i-th EXIT
(ii) CONCATENATING WITH THE i-th EXIT

FIG. 5. *Nonbasic constructs for blocks*

A flowchart is in *block form* if it is of the form shown in Fig. 6, where B is a block.

Clearly every flowchart in block form is equivalent to some program schema. The result of Engeler and Cooper is that for every program schema we can find an equivalent flowchart in block form (see for example, Manna [14]).

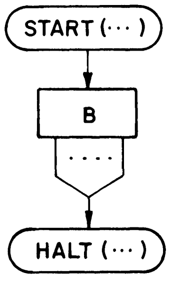

FIG. 6. *Block form flowchart*

Example. Figure 7 shows a flow chart P'_1 in block form which is equivalent to the program schema P_1. The blocks are indicated by broken lines. B_0, B_1 and B_3 are basic blocks. B_2, B_4 and B_6 are constructed by looping, and B_5 and B_7 by concatenation.

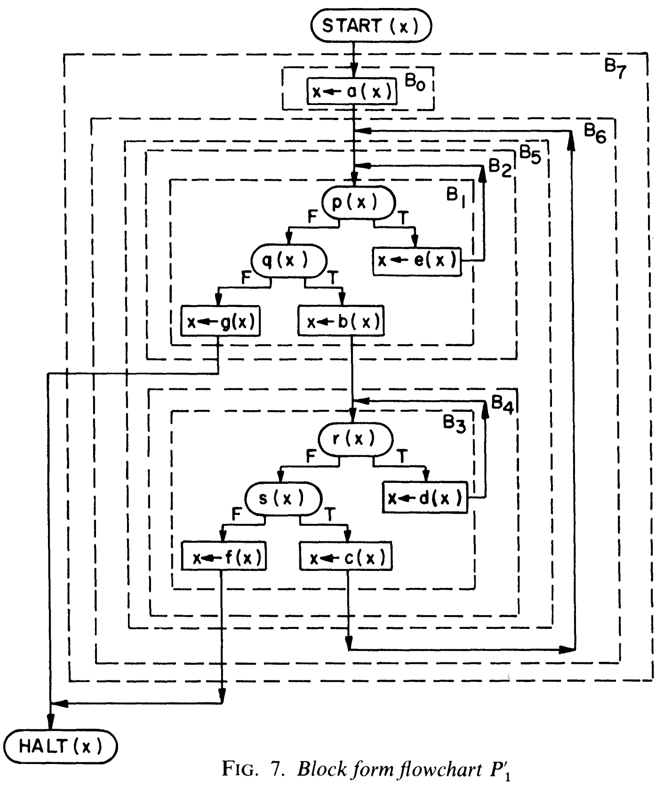

FIG. 7. *Block form flowchart* P'_1

5.3. Properties of basic blocks. Before we consider our next normal form, we observe two useful properties of basic blocks.

Property 1. Given any basic block B (with n exits) and some ith exit of B, there exist a formula i-test(B), a basic block i-pruned(B) (with $n - 1$ exits) and a sequence of assignment statements i-ops(B) such that the flowcharts in Fig. 8 are equivalent.

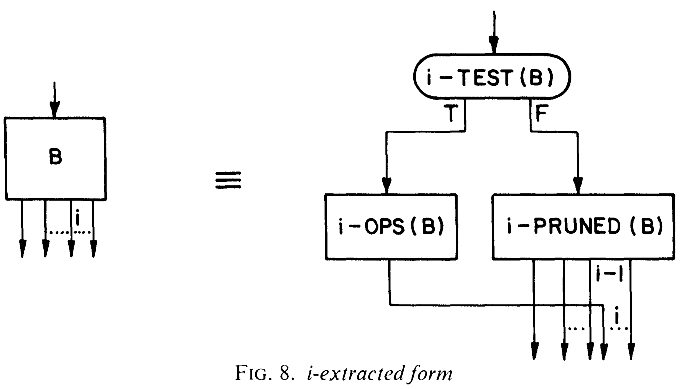

FIG. 8. *i-extracted form*

To see this, note first that the basic block can be put into a form in which the tests on the path on the *i*th exit precede the assignments, by repeated application of the transformation shown in Fig. 9, where ψ' is like ψ but with x_i replaced by τ. It is then a simple matter to find a single test to "extract" the *i*th path by working up the path from the bottom, repeatedly applying the transformations shown in Fig. 10 (or their mirror images). The upper transformation extracts the first path,

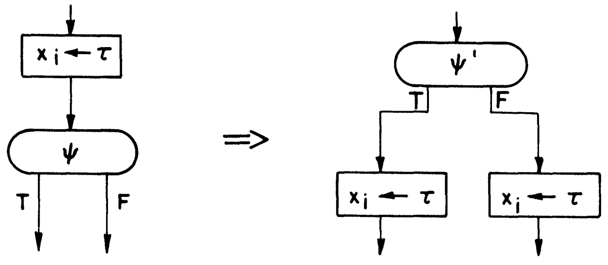

FIG. 9. *Moving tests up*

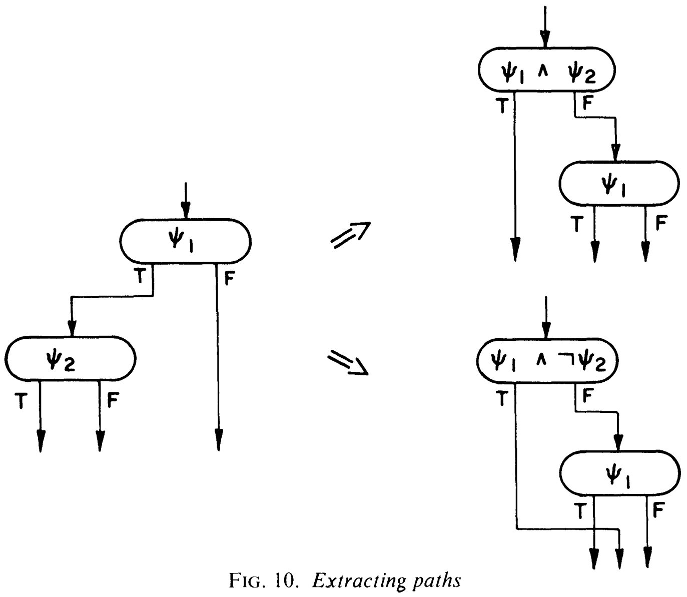

FIG. 10. *Extracting paths*

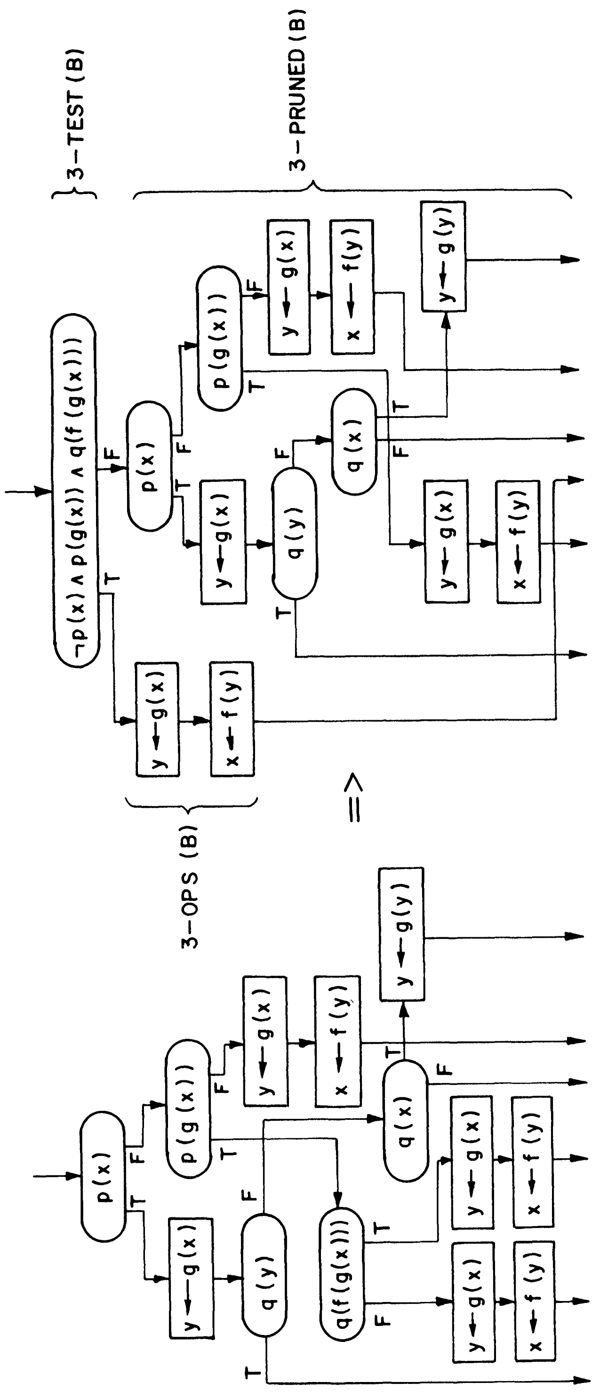

FIG. 11. *The 3-extracted form of the basic block of Fig. 4*

and the lower transformation extracts the second path (the third path is already extracted).

This eventually gives us the desired form for the basic block. It will be called the *i-extracted form.*

Example. Figure 11 shows the two stages in obtaining the 3-extracted form of the basic block B of Fig. 4.

Property 2. The second property of basic blocks that we need is that every piece of flowchart of the form shown in Fig. 12(a), where B is a basic block, is a while-chart. This can be seen very easily by induction on the number of statements in B. If there are no statements, we have an empty edge, which is a while-chart. If there are $n > 0$ statements, we have either Fig. 12(b) or Fig. 12(c), where B_1 and B_2 are basic blocks. In both cases we have while-charts, since the lower parts are while-charts by the induction hypothesis.

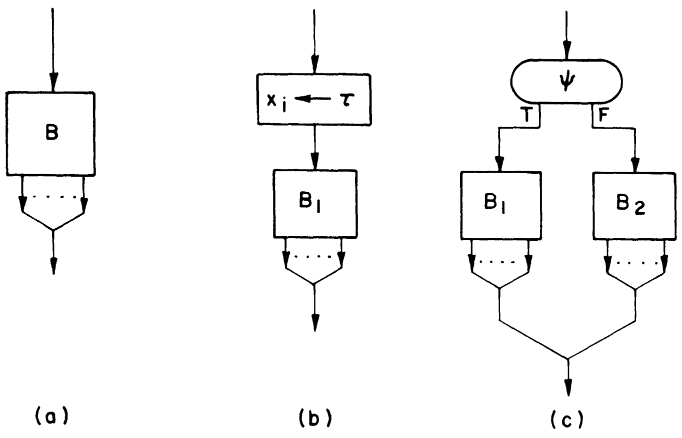

FIG. 12

5.4. Module form. The final normal form for flowcharts which we will consider is module form.

A *module* is either

(i) an assignment statement, or

(ii) a *one-entrance, one-exit* piece of flowchart constructed from modules and tests; these tests are called the tests of the module.

A flowchart is in *module form* if it is of the form shown in Fig. 13, where M is a module.

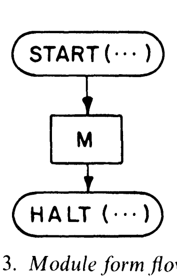

FIG. 13. *Module form flowchart*

This definition may appear surprising since we immediately have that *any* flowchart is in module form, by taking each assignment statement as a module, at one level, and then taking the whole flowchart as a module at the next level. However, we can find an interesting subclass of module form flowcharts:

A *simple module* is either

(i) an assignment statement, or

(ii) a *one-entrance, one-exit* piece of flowchart constructed from modules and *at most one test.*

A flowchart is in *simple module form* if it is in module form and each module is simple.

A simple module either has no tests (and is thus either an empty edge, an assignment statement or the concatenation of modules) or it has one test and can only be of the forms shown in Fig. 14, where A, B, C and D are modules.

The analogy with while-schemas is obvious:

Fig. 14(a) is equivalent to $[C$; **if** ψ **then** A **else** B; $D]$;

Fig. 14(b) is equivalent to $[C$; B; **while** ψ **do** $[A; B]$; $D]$.

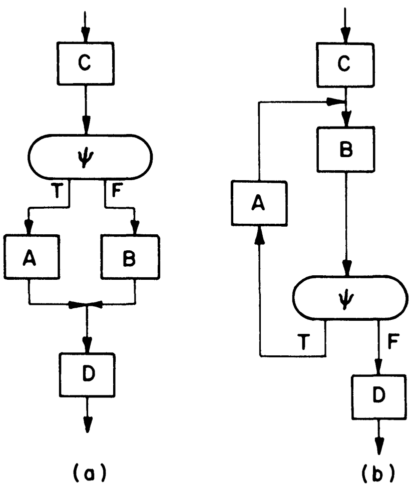

FIG. 14. *Simple modules*

In general, for every flowchart in simple module form there is an equivalent while-schema, and vice versa.

The motivation for module form now becomes clear. At one extreme we can take any flowchart as a module whose submodules are simply assignment statements. If, however, by ingenuity and equivalence-preserving transformations we can get many levels of modules, with fewer tests per module, then we get closer to simple module form and hence closer to while-schemas.

Example. In Fig. 15 we give a module form flowchart P_1'' for the program schema P_1. Modules M_1 and M_2 are simple, but module M_3 is nonsimple since it contains two tests $q(x)$ and $s(x)$.

6. Algorithm 1. To translate program schemas to while-schemas it suffices to consider flowcharts in block form. We show how to transform each block B into an equivalent piece of flowchart consisting of a while-chart W_B followed by a

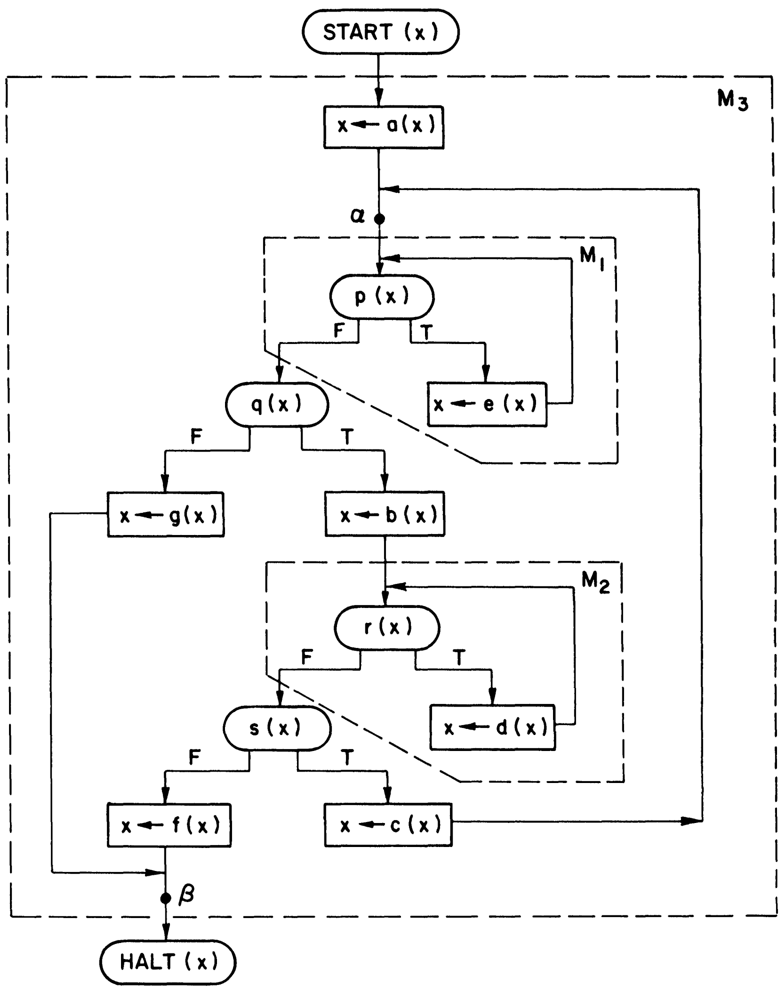

FIG. 15. *Module form flowchart* P_1''

basic block $\bar{B}$. We do this by induction on the block structure as follows:

(i) B is a basic block: we use the transformation of Fig. 16.

(ii) B is constructed by looping on the ith exit of B_1: we first transform B_1; and then extract the ith path of $\bar{B}_1$, as shown in Fig. 17.

(iii) B is the concatenation of B_1 and B_2, using the ith exit of B_1: we first transform B_1 and B_2, and extract the ith path of $\bar{B}_1$. It is then possible to move up W_{B_2} past i-pruned($\bar{B}_1$), as shown in Fig. 18.

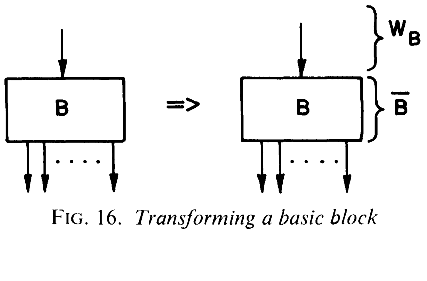

FIG. 16. *Transforming a basic block*

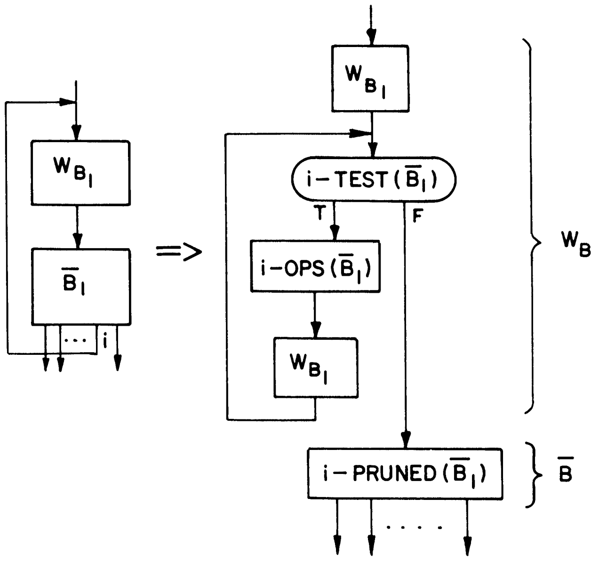

FIG. 17. *Transforming a looping block*

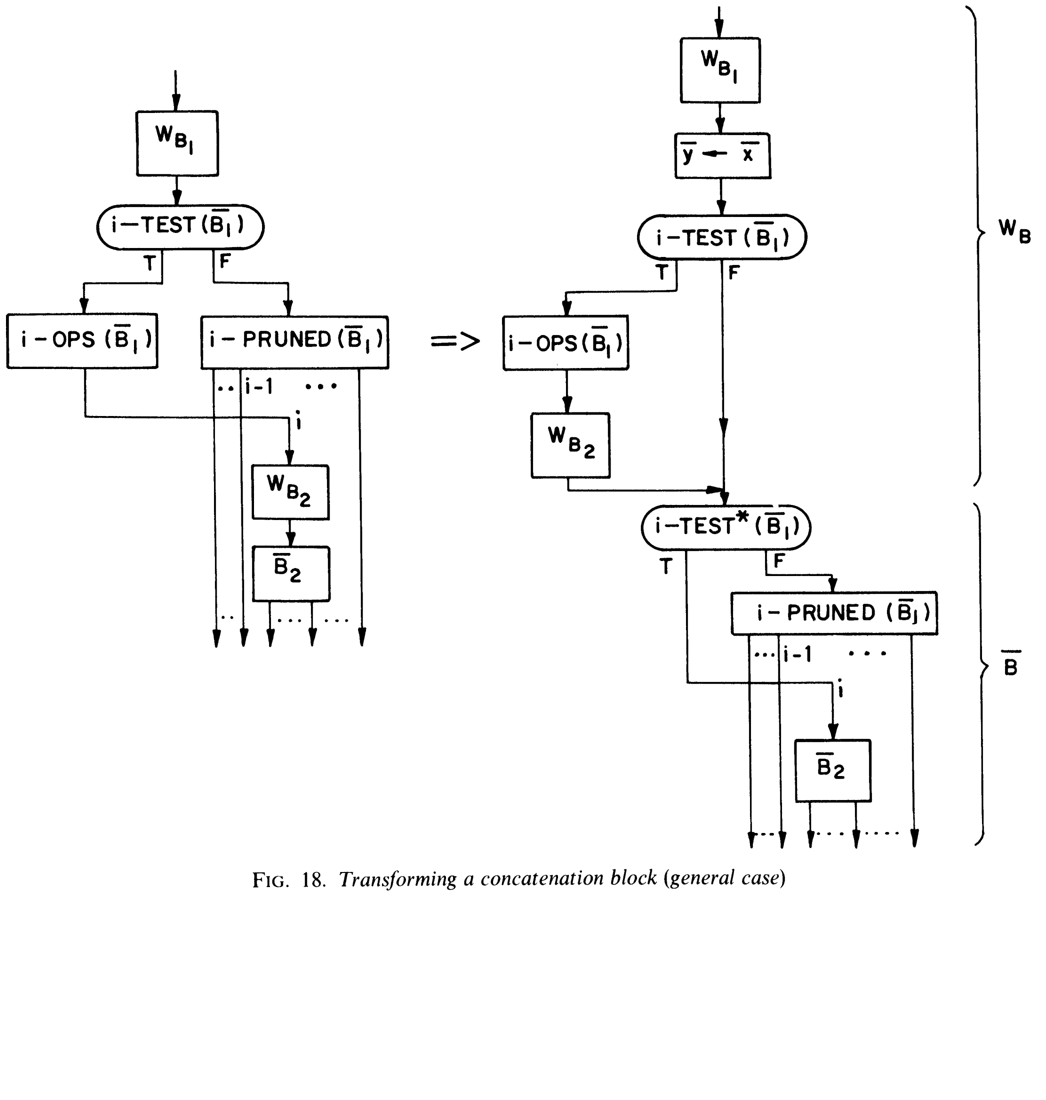

FIG. 18. *Transforming a concatenation block* (*general case*)

In Fig. 18, $\bar{x}$ is a vector of the variables occurring in i-test$(\bar{B}_1)$; $\bar{y}$ is a vector of the same length of *new* variables; and i-test*$(\bar{B}_1)$ is the same as i-test$(\bar{B}_1)$ but with variables $\bar{x}$ replaced by $\bar{y}$, so that any computation must take the same branch out of the two tests.

Note. If B_2 is a basic block, we can simply make the transformation shown in Fig. 19. No new variables are needed in this case.

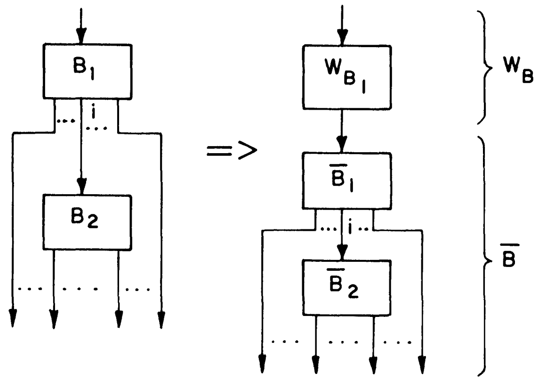

FIG. 19. *Transforming a concatenation block, B_2 basic*

Thus for every block form flowchart, as in Fig. 6, we get a flowchart as in Fig. 20. This flowchart is in while-chart form, by the second property of basic blocks proved earlier, and thus can be simply written as a while-schema.

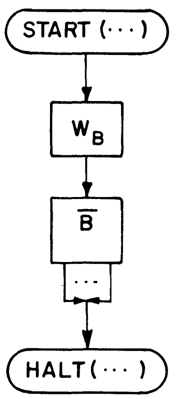

FIG. 20. *Transformed flowchart*

Example. We take flowchart P'_1 of Fig. 7. Blocks B_2 and B_4 are already of the required form; for example, B_2 is shown in Fig. 21. (This is the decomposition that Algorithm 1 yields for looping on the third exit of B_1). The transformed version of B_5 is shown in Fig. 22; note that $q(x)$ comes from 2-test$(\bar{B}_2)$, $x \leftarrow b(x)$ comes from 2-ops $(\bar{B}_2)$ and $x \leftarrow g(x)$ comes from 2-pruned$(\bar{B}_2)$. The transformed version of B_6 is shown in Fig. 23; note that $q(y) \wedge s(x)$ comes from 3-test$(\bar{B}_5)$, $x \leftarrow c(x)$ comes from 3-ops$(\bar{B}_5)$ and $\bar{B}_6$ is simply 3-pruned$(\bar{B}_5)$. The final while-chart form flowchart is shown in Fig. 24, and it corresponds exactly to while-schema P_2.

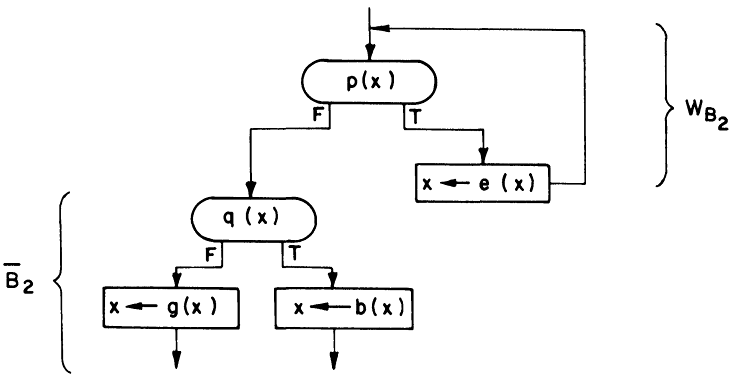

FIG. 21. B_2

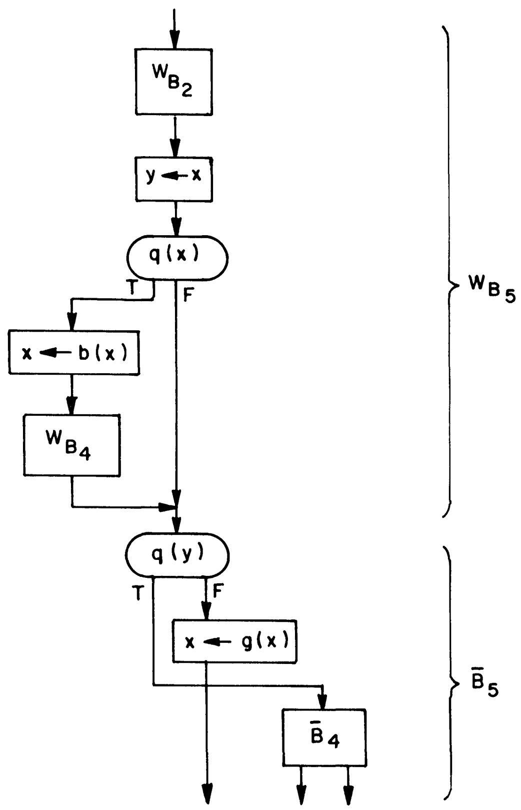

FIG. 22. B_5 *(transformed)*

Comments. (i) Putting a flowchart into block form in general requires some increase in the size of the flowchart. This can be avoided by allowing the exits of any block to be joined together in arbitrary ways and still be a block. In the same spirit we would allow basic blocks that were not tree-like but merely loop-free. Algorithm 1 will work just as well for such block form. The only change needed is in defining the i-extracted form for basic blocks; for example, i-ops(B) becomes a

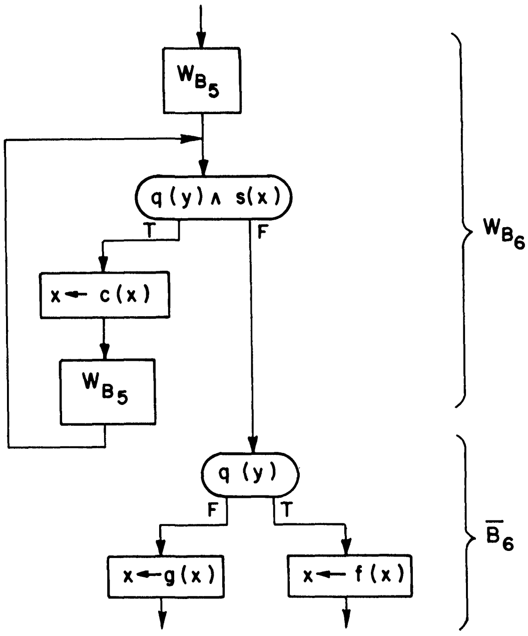

FIG. 23. B_6 (*transformed*)

one-exit basic block rather than a sequence of assignment statements. This modified block form corresponds to interval analysis (see [1]).

(ii) To minimize the number of new variables added by Algorithm 1, we must find block form flowcharts which avoid concatenating blocks except when the second block is basic. Even for while-schemas it is not clear how to do this, and so Algorithm 1 is not an identity mapping on while-schemas. We could avoid this by allowing while-charts to be special cases of blocks. The algorithm is easily modified to deal with this.

(iii) The duplication of W_{B_1} produced by transforming looping blocks (Fig. 17) could be avoided by using a new control construct **repeat** $[S_1$; **exit on** ψ ; $S_2]$ instead of while statements.

7. Algorithm 2. The idea of Bohm and Jacopini's translation of program schemas to while-schemas with logical variables [3] (see also [6]) can be expressed as follows. Suppose the given program schema has n statements including the halt statement, numbered, for our convenience, 1 to n. We construct a while-schema using k additional logical variables, where $2^{k-1} < n \leqq 2^k$. Each statement of the original program schema then corresponds to a particular pattern of values for the k logical variables, e.g., the number of the statement written in binary notation. The while-schema consists of a single while statement, the formula of which will be true provided the "pattern" of logical variable values does *not* correspond to the halt statement. If the formula is true, we enter the body of the while statement, where a series of tests decides to which statement in the program schema the logical variable values correspond. The operation of that statement is then performed, and the values of the logical variables are changed so that their "pattern" corresponds to the next statement to be executed in the program schema. The body of the while statement is repeatedly executed, until we reach the pattern

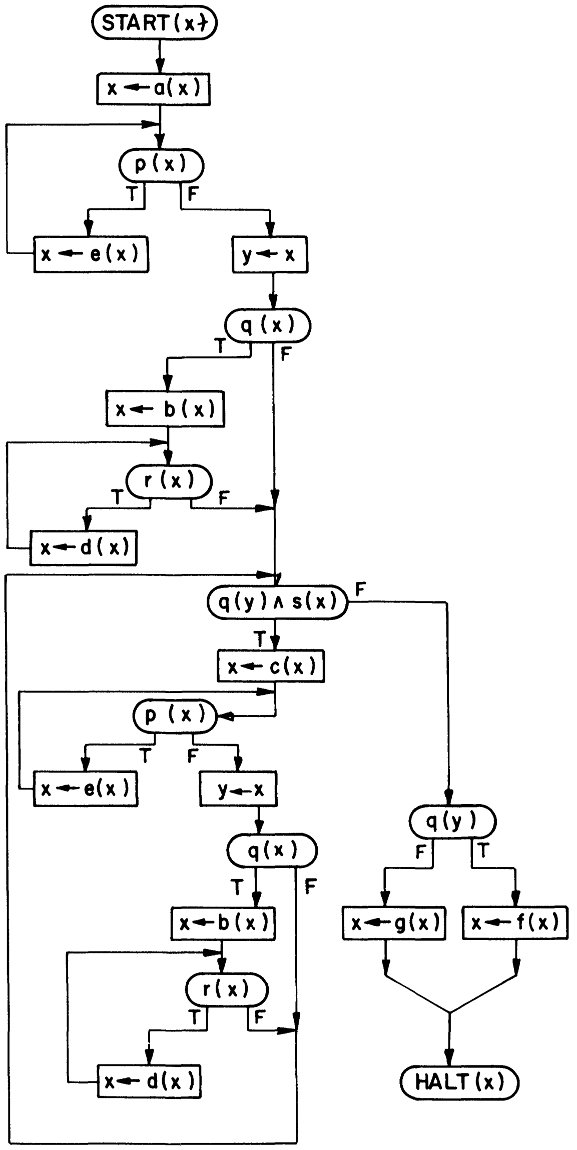

FIG. 24. *Transformed version of* P_1''

for the halt statement in the program schema. When this happens, we exit from the while statement, and reach the halt statement of the while-schema.

The while statement simply acts as a one-loop interpreter, performing one operation of the original program schema on each iteration. The logical variables simply represent a "program counter".

An improvement upon this method, due to Cooper [private communication], reduces the number of logical variables required. We take the flowchart representation of the program schema and choose a "cut set" of the edges between the assignment and test statements from which it is composed, i.e., we choose at least one edge per loop. We add the edge leading to the halt statement to this set. These edges are then numbered, and coded up as logical variable value patterns as before. The while statement "interpreter", on each iteration, now performs the operations of the original program schema from one cut set edge to the next cut set edge, and updates the logical variables accordingly. This technique is used in Algorithm 2 below.

We consider flowcharts in module form, and, for good results, we try to get as many simple modules as possible. We then translate each module M into a statement of while-schema W_M by induction on the module structure.

(i) If M is an assignment statement, W_M is that assignment statement.

(ii) If M is a simple module, W_M is the corresponding statement of while-schema (see Fig. 14).

(iii) If M is a nonsimple module, then we apply Cooper's version of the Bohm and Jacopini reduction. We choose a cut set of the edges between modules and tests comprising the module M, and add the single exit edge of M. We then take sufficient "new" logical variables to represent these positions in M, and construct a statement of while-schema. This statement will be a compound statement $[S_1; S_2]$. Statement S_1 will perform the operations from the entrance of M up to the first cut set edge, and set the logical variables to correspond to that edge. Statement S_2 is then the while statement which "interprets" the module M. Its formula checks that the current pattern of logical variable values does *not* correspond to the exit edge. The body determines the current cut set edge, performs the operations to the next cut set edge (using the while-schema statements corresponding to the modules of which M is composed) and updates the logical variable values accordingly. This is possible as a statement of while-schema since the use of a *cut set* of edges ensures that there is a bound on the number of tests and modules that can be performed between one cut set edge and the next.

Example. The modules of flow chart P_1'' (Fig. 15) correspond to statements of while-schema as follows: M_1 and M_2 are simple modules and correspond to

$$\textbf{while } p(x) \textbf{ do } x \leftarrow e(x)$$

and

$$\textbf{while } r(x) \textbf{ do } x \leftarrow d(x),$$

respectively. M_3 is nonsimple, so we choose cut set edges, for example α and β in Fig. 15 (there is only one loop in M_3 and we must add the exit edge of M_3). We then need one logical variable t, say, to keep track of the cut set edge—**true** corresponds to α, **false** corresponds to β. W_{M_3} is then the following statement of while-schema

```
[[$x \leftarrow a(x)$; $t \leftarrow$ true];
 while $t$ do [$W_{M_1}$;
               if $q(x)$ then [$x \leftarrow b(x)$;
                              $W_{M_2}$;
                              if $s(x)$ then $x \leftarrow c(x)$
                                         else [$x \leftarrow f(x)$;
                                               $t \leftarrow$ false]]
                         else [$x \leftarrow g(x)$; $t \leftarrow$ false]]].
```

Enclosing W_{M_3} between start and halt statements then gives us the while-schema P_3.

Comment. No reasonable algorithm is known for finding the *optimal* equivalent module form for a program schema, optimal in the sense that Algorithm 2 adds the smallest number of logical variables. However, it is clear that the flow charts of while-schemas (i.e., while-charts) are in simple module form, so that Algorithm 2 is the identity mapping on while-schemas.

8. The necessity of adding variables. We show that any translation from program schemas to while-schemas must in general add variables. We prove that for a particular one-variable program schema there is no equivalent while-schema that also uses only one variable.

Similar results have been demonstrated by several authors: Knuth and Floyd [11], Scott [private communication] and Kosaraju [12], for example. However, these results are weaker than ours, either because the notion of equivalence used is more restrictive than ours, requiring the equivalence of computation sequences (i.e., the sequences of assignments and tests in order of execution) and not just the equivalence of final results, or because complex formulas are not allowed in while statements. For example, the following program schema has no "equivalent" while-schema if we consider execution sequences, or disallow compound tests:

```
START($x$);
$x \leftarrow a(x)$;
$L$: if $p(x)$ then [$x \leftarrow b(x)$; if $q(x)$ then [$x \leftarrow c(x)$; go to $L$]
                                   else $x \leftarrow d(x)$]
          else $x \leftarrow e(x)$;
HALT($x$).
```

However, if we apply Algorithm 1, we get an equivalent while-schema, which happens to use only one variable:

```
START($x$);
$x \leftarrow a(x)$;
while $p(x) \wedge q(b(x))$ do $x \leftarrow c(b(x))$;
if $p(x)$ then $x \leftarrow d(b(x))$ else $x \leftarrow e(x)$;
HALT($x$).
```

Since our result is stronger than the previous results, it needs a more complicated program schema to demonstrate it. The one we use is the following program schema P_5.

```
SCHEMA $P_5$. START($x$);
     $L$: if $p(x)$ then [$x \leftarrow e(x)$; go to $L$];
          if $q(x)$ then $x \leftarrow e(x)$ else [$x \leftarrow e(x)$; go to $N$];
     $M$: if $q(x)$ then [$x \leftarrow d(x)$; go to $M$];
          if $p(x)$ then [$x \leftarrow d(x)$; go to $L$] else $x \leftarrow d(x)$;
     $N$: null;
          HALT($x$).
```

This schema is similar to P_1, but is simpler since it only uses two functions and two predicates. It is especially interesting because for most other simpler versions of P_1 there *are* equivalent one-variable while schemas. For example, the program schema

```
START($x$);
$x \leftarrow a(x)$;
$L$: if $p(x)$ then [$x \leftarrow e(x)$; go to $L$];
    if $q(x)$ then $x \leftarrow b(x)$ else [$x \leftarrow g(x)$; go to $N$];
$M$: if $q(x)$ then [$x \leftarrow b(x)$; go to $M$]; .
    if $s(x)$ then [$x \leftarrow c(x)$; go to $L$] else $x \leftarrow f(x)$;
$N$: null;
HALT($x$)
```

is equivalent to the following one-variable while-schema:

```
START($x$);
$x \leftarrow a(x)$;
while $p(x)$ do $x \leftarrow e(x)$;
while $q(x) \wedge q(b(x))$ do $x \leftarrow b(x)$·
while $q(x) \wedge s(b(x))$ do
        [$x \leftarrow c(b(x))$;
         while $p(x)$ do $x \leftarrow e(x)$;
         while $q(x) \wedge q(b(x))$ do $x \leftarrow b(x)$];
if $q(x)$ then $x \leftarrow f(b(x))$ else $x \leftarrow g(x)$;
HALT($x$).
```

Our proof that there is no one-variable while-schema P'_5 equivalent to P_5 must therefore depend crucially on special features of P_5. The essential property of P_5 is the following:

In any unfinished computation of P_5, if p is true and q is false, then the *next-but-one* function that will be applied is e, whereas if q is true and p is false, then the *next-but-one* function that will be applied is d. If both p and q are false, the computation will terminate after applying one more function.

Let d, e and h be symbols and $D = \{d, e\}^*$. We shall consider the interpretations I_z, where $z \in D \cdot h \cdot D$, defined as follows:

(i) $D_{I_z} = D$

(ii) for $y \in D_{I_z}$, $d(y) = yd$,

$e(y) = ye$,

$p(y) \equiv [|y| < |z| \wedge z(|y| + 1) = e]$[1],

$q(y) \equiv [|y| < |z| \wedge z(|y| + 1) = d]$,

(iii) the initial value of the input variable x is Λ, the empty string.[2]

Note that the predicates p and q are mutually exclusive, and from the essential property of P_5, the computation of (P_5, I_z), where $z = uhv$ ($u, v \in D$), must terminate with $\text{val}(P_5, I_z) = eu$ (the symbol h makes both p and q false and makes the computation halt). Also for any interpretation $I_{w\alpha uhv}$ ($\alpha \in \{d, e\}, w, u, v \in D$), when the value of x becomes ew, the future course of the computation is determined by uhv, since this substring will determine the possible future values of the predicates p and q. This property also holds for any *one-variable* schema P'_5 equivalent to P_5 (it will be called the main property of P'_5), and will be used to show that such a schema cannot exist.

Let us assume therefore that there exists a one-variable while-schema P'_5

[1] Here $|y|$ denotes the length of string y; $z(i)$ denotes the ith symbol in string z.

[2] This is a "Herbrand" or "free" interpretation (see [13]).

equivalent to P_5. Without loss of generality we can assume there is some while statement S in P'_5, say **while** ψ' **do** S_1, which is not contained in or followed by any other while statement, and for which S_1 is executed in the computation for some I_z. We can also assume that there is no bound on the number of iterations of S for computations for such interpretations I_z. (All such bounded while statements could be "unwound" the corresponding number of times, leaving only "unbounded" while statements and while statements never entered for any I_z.)

Let the maximum "depth" of functional composition in any formula in P'_5 be M. Then in computation of (P'_5, I_z), if we evaluate a formula ψ for value w of variable x, then the outcome of ψ is determined by $z(|w| + 1), z(|w| + 2), \cdots, z(|w| + M + 2)$. We define visible$(z, w)$ as this substring of z starting at $z(|w| + 1)$ and ending at $z(|w| + M + 2)$.

LEMMA. *For all $n \geqq 0$ there exist strings $u, w, y \in D$, $|w| = n$, such that for all $v \in D$, the computation of (P'_5, I_{uvwhy}) exits from S with a proper prefix of euv as the value of variable x.*

This technical lemma has the following informal corollary.

COROLLARY. *For every $n \geqq 0$ there exists a computation of P'_5 which exits from S with more than n functions still to be applied.*

This corollary contradicts the fact that S is not followed by or contained in another while-statement; the number of functions that can be applied after exiting from S is bounded. Hence while-schema P'_5 cannot exist.

Proof of lemma. The proof is by induction on n.

Base step ($n = 0$). Since S is unbounded, there exists an interpretation $I_{z'}$ whose computation enters S_1 before the end of the computation is "visible", i.e., more than M function applications from the end. In other words, $z' = u'\alpha yv'hy'$ where $u', y, v', y' \in D$, $\alpha \in \{d, e\}$ and $|y| = M + 1$, and the computation of $(P'_5, I_{z'})$ reaches S with eu' as the value of x. Moreover, since S_1 is entered, the formula ψ' must be true for this value of x. Note that the truth of ψ' is determined by visible$(z', eu') = y$.

Consider now the interpretation $I_z = I_{uvhy}$, where $u = u'\alpha y$ and v is any string from D. The computation must reach S as for $I_{z'}$, i.e., with value eu' for variable x, since the changes in the interpretation are not "visible" by this point. However, when it subsequently exits from S, it can not do so with value euv (the final value) for x, since visible$(z, euv) = y$, and for this value the formula ψ' must be true. Thus it must exit from S with a *proper prefix* of euv as the value of x.

Induction step. Assume we have strings $u, w, y \in D$, with $|w| = n$, such that for all $v \in D$, the computation of (P'_5, I_{uvwhy}) exits from S with a proper prefix of euv as the value of x.

We shall find a string $w' \in D$, $|w'| = n + 1$, such that for all $v' \in D$, the computation of $(P'_5, I_{uv'w'hy})$ exits from S with a proper prefix of euv' as the value of x.

There are three cases to consider (in order):

(i) For all $v = v'e$ (for all $v' \in D$) in the induction hypothesis, the corresponding proper prefix of euv is also a proper prefix of euv'. In this case we take $w' = ew$.

(ii) For all $v = v'd$ (for all $v' \in D$) in the induction hypothesis, the corresponding proper prefix of euv is also a proper prefix of euv'. In this case we take $w' = dw$.

(iii) For some $v = v''e$ in the induction hypothesis, the corresponding proper prefix of euv is euv'', i.e., the computation C of $(P'_5, I_{uv''ewhy})$ exits from S with value euv'' for variable x. Note that the rest of the computation adds ew to the value of x.

Consider now the interpretations $I_{uv'dwhy}$, for all $v' \in D$. By the induction hypothesis, the value of x on exiting from S must in each case be a proper prefix of $euv'd$. But the main property of P'_5 ensures that in no case can this value of x be euv', otherwise the future course of this computation, being determined by why, would be the same as for C, giving x a final value of $euv'ew$ instead of $euv'dw$. Thus with $w' = dw$, the computations of $(P'_5, I_{uv'w'hy})$ (for all $v' \in D$) exit from S with a proper prefix of euv' as the value of x. Q.E.D.

Acknowledgments. We are indebted to David Cooper for stimulating discussions and for his modification of the Bohm and Jacopini's reduction, which we have used in Algorithm 2. We are also grateful to Donald Knuth for his critical reading of an earlier version of this paper, and subsequent helpful suggestions.

REFERENCES

[1] F. E. Allen, *A basis for program optimization*, Proc. IFIP Congress, Ljubliana, Yugoslavia, 1971.

[2] S. Brown, D. Gries and T. Szymanski, *Program schemas with pushdown stores*, this Journal, 1 (1972), pp. 242–268.

[3] C. Bohm and G. Jacopini, *Flow diagrams, Turing machines and languages with only two formation rules*, Comm. ACM, 9 (1966), pp. 366–371.

[4] A. K. Chandra, *On the properties and applications of program schemas*, Ph.D. thesis, Computer Science Dept., Stanford Univ., Stanford, Calif., 1973.

[5] R. L. Constable and D. Gries, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[6] D. C. Cooper, *Bohm and Jacopini's reduction of flowcharts*, letter to the Editor, Comm. ACM, 10 (1967), p. 463, p. 473.

[7] ———, *Programs for mechanical program verification*, Machine Intelligence 6, Edinburgh Univ. Press, 1970.

[8] E. Dijkstra, *Goto statement considered harmful*, Comm. ACM, 11 (1968), pp. 147–148.

[9] E. Engeler, *Structure and Meaning of Elementary Programs*, Symp. on Semantics of Algorithmic Languages, Springer-Verlag, Berlin, 1971.

[10] C. A. R. Hoare, *An axiomatic approach to computer programming*, Comm. ACM, 12 (1969), pp. 576–580, p. 583.

[11] D. E. Knuth and R. W. Floyd, *Notes on avoiding goto statements*, Information Processing Letters, 1 (1971), pp. 23–31.

[12] S. Kosaraju, *Analysis of structured programs*, Proc. of 5th SIGACT Conf., 1973, pp. 240–252.

[13] D. Luckham, D. Park and M. Paterson, *On formalized computer programs*. J. Comput. Systems Sci., 4 (1970), pp. 220–249.

[14] Z. Manna, *Introduction to Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[15] M. Paterson and C. Hewitt, *Comparative schematology*, Conf. Record of Project MAC Conf. on Concurrent Systems and Parallel Computation, ACM, New York, 1970.