INFORMATION AND CONTROL 35, 159–171 (1977)

Structured Turing Machines

RONALD E. PRATHER

Department of Mathematics, University of Denver, Denver, Colorado 80208

A structured decomposition theorem for Turing machines is given. The nature of the building blocks and the form of the connections allowed suggest a parallel to the Bohm–Jacopini theorem on structured flowcharts. Thus in a broadest sense, there is obtained an independent machine–theoretic restatement of the fundamental precepts of structured programming. At the same time, the characteristics of the decomposition offer several obvious advantages over known results of this type. In particular, the building blocks (the "simple" machines) are seen to perform total and regular word functions. Furthermore, the connections themselves should prove to be useful as pedagogical tools in the Turing machine theory and as a theoretical framework for top-down machine (or algorithmic) design.

1. INTRODUCTION

In virtually every branch of mathematics or computer science, a "structure theorem" can be found at the heart of the theory. One might think of the fundamental theorem on Abelian groups (Dean, 1966) or the Krohn–Rhodes decomposition theorem (Nelson, 1968) as being typical in the areas of classical group theory and the theory of finite state automata, respectively. In this paper we seek such a result, a structural decomposition theorem for Turing machines.

A parallel goal is that of obtaining a machine–theoretic analogy to the well-known theorem of Bohm and Jacopini (1966) on structured flowcharts. Since the announcement of this result, no single idea in computer science has attracted more attention than that of "structured programming" (Dahl, Dijkstra, and Hoare, 1972). In essence, the Bohm–Jacopini theorem shows that every flowchart admits an equivalent reformulation in terms of the basic structured constructs of Fig. 1 (and also for convenience, if ··· then ···, and do ··· until ···), together with the "simple" assignment statements. This result is now so often cited and reinterpreted that further clarification or elaboration is not necessary here. Besides its theoretical importance, the resulting structured approach to programming has lead to a highly successful algorithmic design procedure, known variously as "top-down" or "step-wise" successive refinement (Wirth, 1973) of the problem. In addition, the structure of the resulting program or flowchart allows one to consider proving the correctness of the algorithm

ISSN 0019–9958

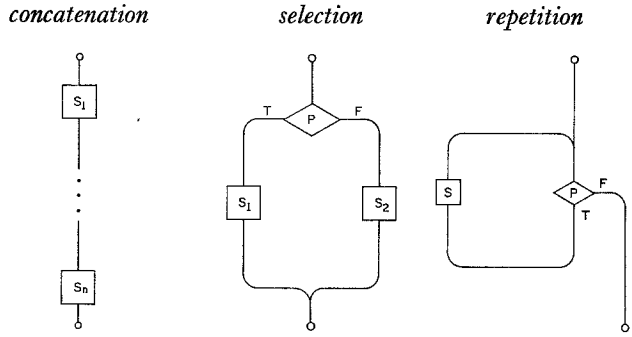


FIGURE 1

(Manna, 1974), a virtual impossibility in the case of an arbitrary unstructured flowchart of moderate size.

Here we present an independent derivation of an entirely analogous system of machine constructs whereby one may realize each Turing-computable function by a structured connection of "simple" machines. It then follows that to a large extent, we are able to extend the attributes of structured programming (including clarity, efficiency, simplicity, etc.) to the realm of structured machine design. In a broader sense, one could view our result as a rephrasing of the fundamental precepts of structured programming, identifying programs, and Turing machines in an appropriate manner.

2. Simple Machines vs Elementary Machines

In the widest interpretation, the idea of combining simpler Turing machines into ever more complicated ones is not entirely new. Hermes (1965) introduces certain connections built up from the *elementary machines* l, γ, r, which halt after moving left one square, printing a symbol γ, or moving right one square, respectively. These are to be compared with the "simple" machines, the building blocks of our structural decomposition theorem.

We recall (Prather, 1975) that a *Turing machine* $Z = (S, \Sigma, M)$ consists of

(i) a finite set S of "states" (with $0, 1 \in S$)

(ii) an alphabet Σ (augmented as $\Sigma_\square = \Sigma \cup \{\square\}$)

(iii) a function M (the table of "moves")

$$M:(S - \{1\}) \times \Sigma_\square \to S \times \Sigma_\square \times \{L, N, R\}.$$

As used here, a Turing machine Z ordinarily begins its computation in the *initial state* 0 at the left-hand end of a tape on which the words $x_i \in \Sigma^*$ (the

free monoid on the alphabet Σ, identity $= \epsilon$) are printed with the blank ($\square$) as a word separator. Then Z follows the moves dictated by M unless or until the *final state* 1 is reached. With L, N, R interpreted as left motion, no motion, or right motion of the read–write head on an expandable tape and with a standard convention for adjoining blanks in the case of a "run-off," the table M induces a relation ($\rightarrow$) on the set $\Sigma_\square{}^* \times S \times \Sigma_\square{}^*$ whose elements are the *instantaneous descriptions* (id's) of Z. As indicated above

$$0_x = (\epsilon, 0, x) \qquad \text{with} \quad x = x_1 \square x_2 \square \cdots \square x_n$$

is an *n-initial* i.d. whereas any i.d.

$$1_y = (\square^r, 1, y\square^s) \qquad \text{with} \qquad y = y_1 \square y_2 \square \cdots \square y_m$$

is said to be an *m-final* i.d. More generally, any i.d. with state 1 is called a *terminal* i.d., one for which the machine halts.

Realizing that

$$M(q, \sigma) = (q', \sigma', \mu)$$

means that Z in the state q scanning the symbol σ overprints the symbol σ' and moves according to μ while making a transition to state q', one can easily anticipate the conclusion of Hermes' theory: Every Turing machine can be realized by an appropriate "combination" of elementary machines. One immediately visualizes that these combinations or connections involve state transitions of the sort found in the state diagram representation of finite-state automata. But this accounts for one of the main disadvantages of the Hermes scheme for machine combinations: As in the case of general flowcharts, the connections are wildly unstructured!

A second drawback concerns the nature of the building blocks themselves, the so-called "elementary machines." No attempt is made to sensure that they compute functions f: $(\Sigma^*)^n \rightarrow (\Sigma^*)^m$. In fact, for the choice l, γ, r this is not the case, at least according to any of the established conventions in this regard. But suppose we use the terminology of Davis (1958) in saying that the Turing machine Z is (n, m)-*regular* (we write $Z = Z^{n,m}$) if each n-initial i.d. encounters a terminal i.d. only if it is m-final. This then allows us to adopt the convention that $Z = Z^{n,m}$ *computes the* (partial) *function*

$$\Psi_Z : (\Sigma^*)^n \rightarrow (\Sigma^*)^m$$

as given by

$$\Psi_Z(x_1, \ldots, x_n) = (y_1, \ldots, y_m) \qquad \text{iff} \qquad 0_x \rightarrow \cdots \rightarrow 1_y,$$

with $x = x_1 \square x_2 \square \cdots \square x_n$ and $y = y_1 \square y_2 \square \cdots \square y_m$ as before. Then

it is seen, by way of comparison with the elementary machines of Hermes, that the simple machines of Section 3 are regular and *total* (in that the functions they compute are total functions).

3. Structured Machines

The equivalence between the Turing computable and the partial recursive functions $f: (\Sigma^*)^n \to (\Sigma^*)^m$ is well known. Most recently (Prather, 1975; Eilenberg and Elgot, 1970; Brainerd and Landweber, 1974) there has been an increasing trend toward choosing that particular "cryptomorphic" version of recursive function theory which best suits the discussion at hand. Certainly that best describes the approach taken here. We define the class of *partial recursive functions* (over the alphabet Σ) to be the smallest collection $\mathscr{P} = \mathscr{P}_\Sigma$ of partial functions $f: (\Sigma^*)^n \to (\Sigma^*)^m$ (with $n, m \geqslant 1$) containing the

(i) *binary projection* $b: (\Sigma^*)^2 \to \Sigma^*$

$$b(x, y) = y;$$

(ii) *counting* (or "successor") *function* $c: \Sigma^* \to \Sigma^*$

$$\begin{aligned} c(y) &= \sigma_1 && \text{if } y = \epsilon \\ &= x\sigma_{i+1} && \text{if } y = x\sigma_i \quad (0 < i < n) \\ &= c(x)\,\sigma_1 && \text{if } y = x\sigma_n\,; \end{aligned}$$

(iii) *diagonal function* $d: \Sigma^* \to (\Sigma^*)^2$

$$d(x) = (x, x);$$

(iv) *exchange function* $e: (\Sigma^*)^2 \to (\Sigma^*)^2$

$$e(x, y) = (y, x);$$

and closed under the operations of

(i′) *composition*, $(f \circ g)(x) = g(f(x))$,

(ii′) *cylindrification*, $(i \times f)(y, x) = (y, f(x))$,

(iii′) *exponentiation*, $f^*(y, x) = \overbrace{f \circ f \circ \cdots \circ f}^{y \text{ times}}(x)$,

(iv′) *minimalization*, $(\mu f)(x) = \inf\{y: f(x, y) = \epsilon\}$,

for $x \in (\Sigma^*)^n$, $y \in \Sigma^*$. When we say "y times" in (iii′), we refer to the natural ordering of the words of Σ^* as established by the counting function of (ii). Thus we identify ϵ with 0, σ_1 with 1, etc. Similarly in (iv′), the infimum refers to this same ordering of Σ^*.

B

	τ	$\square$
0	0 $\square$ R	1 $\square$ R

C

	$\sigma_i (0 < i < n)$	σ_n	$\square$
0	0 σ_i R	0 σ_n R	2 $\square$ L
2	3 σ_{i+1} L	2 σ_1 L	1 σ_1 N
3	3 σ_i L	3 σ_n L	1 $\square$ R

D

	τ	$\square$
0	q_τ $\square$ R 2	$\square$ L
2	2 τ L 1	$\square$ R
q_σ	q_σ τ R q_σ'	$\square$ R
q_σ'	q_σ' τ R p_σ	σ L
p_σ	p_σ τ L p_σ'	$\square$ L
p_σ'	p_σ' τ L 0	σ R

E

	τ	$\square$
0	q_τ' $\square$ R	$q_\square'$ $\square$ R
s	s τ L	s' $\square$ L
s'	s' τ L	1 $\square$ R
q_σ	q_τ σ R	$q_\square'$ σ R
q_γ'	q_τ' γ R	$q_\square''$ γ R
q_σ''	q_τ'' σ R	p_σ $\square$ L
$q_\square''$	q_τ'' $\square$ R	s $\square$ L
p_σ	p_σ τ L	p_σ' $\square$ L
p_σ'	p_σ' τ L	p_σ'' $\square$ L
p_σ''	p_σ'' τ L	r_σ $\square$ R
r_σ	q_τ σ R	$q_\square'$ σ R

FIGURE 2

It is easily shown (Prather, 1975; Brainerd and Landweber, 1974) that these axioms are equivalent to the more usual ones in which primitive recursion plays a prominent role. In particular

$$d \circ b = i\colon \Sigma^* \to \Sigma^*$$

computes the *identity function* $i(x) = x$ and with cylindrification, identity functions of n arguments can be performed. We can perform arbitrary permutations of n arguments through composition of exchanges, and arbitrary projections from n arguments are also easy to realize. Recursion itself is simulated by an appropriate use of exponentiation.

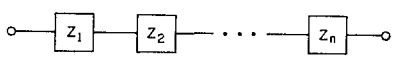
We take as our catalog of *simple machines* the group of tables shown in Fig. 2, noting that by design

$$\Psi_B = b \qquad \Psi_C = c \qquad \Psi_D = d \qquad \Psi_E = e,$$

as is easily checked. For reasons that become apparent only later, we include in our catalog a "right inverse" for C, i.e., a Turing machine $\hat{C}$ which counts down, so that

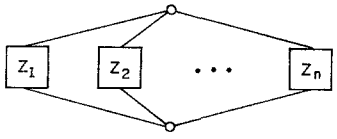
$$\Psi_{\hat{C}} = \hat{c} \qquad \text{with} \qquad c \circ \hat{c} = i.$$

(1) *serial connection* ($Z_i = Z_i^{r_i,s_i}$ with $s_i = r_{i+1}$ for $1 \leqslant i < n$)



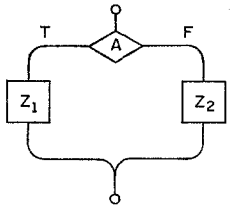
$Z =$ **begin** Z_1 ; Z_2 ;...; Z_n **end**
$= Z_1 \circ Z_2 \circ \cdots \circ Z_n$

(2) *parallel connection* ($Z_i = Z_i^{r_i,s_i}$)



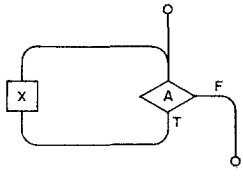
$Z =$ **all** Z_1 ; Z_2 ;...; Z_n **together**
$= Z_1 \times Z_2 \times \cdots \times Z_n$

(3) *alternative connection* ($Z_i = Z_i^{r,s}$ for $i = 1, 2$)

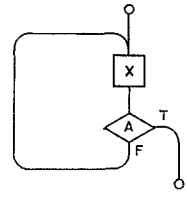


$Z =$ **if** A **then** Z_1 **else** Z_2

(4) *iterative connections* ($X = X^{n,n}$)



$Z =$ **while** A **do** X



$Z =$ **do** X **until** A

FIGURE 3

The reader can easily provide a tabular description of such a machine, allowing that $\hat{e}(\epsilon) = \epsilon$. In this way, all of our simple machines are total and regular.

In order to introduce the concept of *structured machines*, we begin with the postulate that each simple machine is structured. Then we agree that if the Z_i's (respectively, X's) are structured, so are the machines Z formed according to any of the basic connections appearing in Fig. 3. These are intended to behave as follows:

(1) *serial connection*, $Z = Z^{r_1,s_n}$,

$$\Psi_Z(x) = (\Psi_{Z_1} \circ \Psi_{Z_2} \circ \cdots \circ \Psi_{Z_n})(x);$$

(2) *parallel connection*, $Z = Z^{\Sigma r_i, \Sigma s_i}$,

$$\Psi_Z(x^1, x^2, \ldots, x^n) = (\Psi_{Z_1}(x^1), \Psi_{Z_2}(x^2), \ldots, \Psi_{Z_n}(x^n));$$

(3) *alternative connection*, $Z = Z^{r,s}$,

$$\begin{aligned} \Psi_Z(x) &= \Psi_{Z_1}(x) && \text{if } A \text{ halts in state } T \\ &= \Psi_{Z_2}(x) && \text{if } A \text{ halts in state } F; \end{aligned}$$

(4) *iterative connections*, $Z = Z^{n,n}$,

$$\begin{aligned} \Psi_Z(x) &= \Psi_X{}^k(x) && \text{if } A \text{ halts in state } F \text{ after the } k\text{th cycle} \\ &= \text{undefined} && \text{if } A \text{ always halts in state } T, \\ \Psi_Z(x) &= \Psi_X{}^k(x) && \text{if } A \text{ halts in state } T \text{ during the } k\text{th cycle} \\ &= \text{undefined} && \text{if } A \text{ always halts in state } F, \end{aligned}$$

respectively. Note that in (1), (2), the case $n = 2$ would be sufficient; but then our subsequent compositions would be more difficult to describe.

So far, we have only provided diagrammatic and behavioral descriptions of these various connections. Before entering into greater detail, we point out that the automata A are themselves Turing machines, but only elementary recognizers. They do not print, and they always halt at the left in distinguished states T or F, one or the other. For our purposes, they need only have the capability of finite-state automata. In fact, in the sequel, A simply denotes an automaton which recognizes whether or not its tape begins with a nonblank, symbolizing that the first word on the tape is not the null word. The only other recognizer to be encountered is denoted $\neg A$ since it does just the opposite, arriving in state T when its tape does begin with a blank.

The serial–parallel connections have been fully described elsewhere (Prather, 1975). In the alternative case (3), we connect

$$Z_1 = Z_1^{r,s} = (S_1, \Sigma, M_1),$$

$$Z_2 = Z_2^{r,s} = (S_2, \Sigma, M_2),$$

and the recognizer $A = A^{r,r}$ in forming a composite machine

$$\begin{aligned} Z = Z^{r,s} &= (S, \Sigma, M) \\ &= \textbf{if } A \textbf{ then } Z_1 \textbf{ else } Z_2, \end{aligned}$$

as follows. For the set of states, we take a disjoint union

$$S = S_A \vee S_1 \vee S_2 \vee \{1\} \qquad (0_Z = 0_A \text{ and } 1_Z = 1)$$

with the three tables merged into one and extended to include

$$\begin{aligned} M(T, \gamma) &= (0_{Z_1}, \gamma, N), \\ M(F, \gamma) &= (0_{Z_2}, \gamma, N), \\ M(1_{Z_1}, \gamma) &= (1, \gamma, N), \\ M(1_{Z_2}, \gamma) &= (1, \gamma, N). \end{aligned} \qquad (\gamma \in \Sigma_{\square})$$

Then the aformentioned behavior Ψ_Z is easily understood.

Similarly for the iterative connections, we join $X = X^{n,n}$ and $A = A^{n,n}$ in forming the composite machine

$$\begin{aligned} Z = Z^{n,n} &= (S, \Sigma, M) \\ &= \textbf{while } A \textbf{ do } X, \end{aligned}$$

for which

$$S = S_A \vee S_X \qquad (0_Z = 0_A \text{ and } 1_Z = F)$$

with the extended entries:

$$\begin{aligned} M(T, \gamma) &= (0_X, \gamma, N) \\ M(1_X, \gamma) &= (0_A, \gamma, N) \end{aligned} \qquad (\gamma \in \Sigma_{\square}).$$

The other iterative connection is defined analogously, and in either case, the given behavior Ψ_Z is once again quite evident.

4. Examples of Structured Machines

The "identity machine" $I = D \circ B$ is structured, as is each of the parallel connections

$$I^n = \overbrace{I \times I \times \cdots \times I}^{n \text{ times}}, \qquad (n \geqslant 1).$$

Whenever there is little possibility for confusion, we also denote these more general identity machines by I. As a simple illustration, suppose we define a sequence of (n, n)-regular (structured) machines $C^{(n)}$ by writing

$$\begin{aligned} C^{(1)} &= C, \\ C^{(k+1)} &= I \times C^{(k)}. \end{aligned}$$

Then it is clear from the context that $I = D \circ B$. On the other hand, if we were then to write $C^{(n)} = I \times C$ for $n \geqslant 1$, we have in mind a generalized identity machine.

We have ample opportunity to examine the use of more elaborate connections in the next section. So it is perhaps advisable that we limit our attention here to just those auxiliary machines as are found to be convenient in our exposition of the structure theorem. Toward this end, we proceed as above in defining

$$\begin{aligned} B^{(1)} &= B, \\ B^{(k+1)} &= (B \times I) \circ B^{(k)}; \\ D^{(1)} &= D, \\ D^{(k+1)} &= (D \times D^{(k)}) \circ (E^{(k+1)} \times I); \\ E^{(1)} &= E, \\ E^{(k+1)} &= (E^{(k)} \times I) \circ (I \times E). \end{aligned}$$

If at the same time, we assert that $B^{(n)}$ is $(n+1, 1)$-regular, $D^{(n)}$ is $(n, 2n)$-regular, and $E^{(n)}$ is $(n+1, n+1)$-regular, then once again the proper interpretation of the identity machines is clear from the context. We may then summarize the main properties of these machines with

LEMMA. *For all $n \geqslant 1$ the machines $B^{(n)}$, $C^{(n)}$, $D^{(n)}$, $E^{(n)}$ are structured. Furthermore,*

$$\begin{aligned} \Psi_{B^{(n)}}(x_1, x_2, \ldots, x_{n+1}) &= x_{n+1}, \\ \Psi_{C^{(n)}}(x_1, \ldots, x_{n-1}, x_n) &= (x_1, \ldots, x_{n-1}, c(x_n)), \\ \Psi_{D^{(n)}}(x_1, x_2, \ldots, x_n) &= (x_1, x_2, \ldots, x_n, x_1, x_2, \ldots, x_n), \\ \Psi_{E^{(n)}}(x_1, x_2, \ldots, x_{n+1}) &= (x_2, \ldots, x_{n+1}, x_1). \end{aligned}$$

Proof. A straightforward verification by induction can be given in each case.

5. A Structured Decomposition Theorem

Given the description of a Turing machine $Z = (S, \Sigma, M)$ as a table of moves, it may be quite difficult to ascertain and then to verify just what it is that the machine does. The situation is quite analogous to the one we face when asked to analyze an arbitrary flowchart. Rather than to rely entirely on the designers (programmers) insight and ingenuity, one's understanding is greatly enhanced if the machine (program) can be thought of as a structured composition of submachines (subroutines) whose behavior is unquestionably clear. Our structure theorem for Turing machines shows that this position has universal applicability, at least in principle.

THEOREM. *Every partial recursive function (necessarily realized by some Turing machine) is realized by a structured machine, i.e., a structured connection of simple machines.*

Proof. The functions b, c, d, e are realized by the simple machines B, C, D, E as already remarked. If f, g are realized by structured X, Y, respectively, then it is clear that $Z = X \circ Y$ has the desired behavior for realizing the composition

$$\Psi_Z(x) = \Psi_{X \circ Y}(x) = (\Psi_X \circ \Psi_Y)(x) = (f \circ g)(x).$$

Cylindrification is also easily accomplished, for if $Z = I \times X$ then we have

$$\Psi_Z(y, x) = \Psi_{I \times X}(y, x) = (\Psi_I(y), \Psi_X(x)) = (y, f(x)),$$

as required.

The structured implementation of exponentiation is somewhat more difficult. If $f\colon (\Sigma^*)^n \to (\Sigma^*)$ and $\Psi_X = f$ for the structured machine $X = X^{n,n}$, then we set

$$Y = \hat{C} \times X$$
$$Z = (\textbf{while } A \textbf{ do } Y) \circ (B \times I).$$

It follows that Z is structured and

$$\begin{aligned}\Psi_Z(y, x) &= \Psi_{\textbf{while } A \textbf{ do } Y} \circ \Psi_{B \times I}(y, x) \\ &= \Psi_{B \times I}(\epsilon, \overbrace{\Psi_X \circ \Psi_X \circ \cdots \circ \Psi_X}^{y \text{ times}}(x)) \\ &= \overbrace{\Psi_X \circ \Psi_X \circ \cdots \circ \Psi_X}^{y \text{ times}}(x) \\ &= \overbrace{f \circ f \circ \cdots \circ f}^{y \text{ times}}(x) = f^*(y, x).\end{aligned}$$

Finally, if $f\colon (\Sigma^*)^{n+1} \to \Sigma^*$ and we have a structured machine $X = X^{n+1,1}$ with $\Psi_X = f$ we define the machines

$$T = T^{n,n+2} = (D \times I) \circ ((\textbf{while } A \textbf{ do } \hat{C}) \times I) \circ (D \times I) \circ E^{(n+1)}$$
$$U = U^{n+1,n+2} = D^{(n+1)} \circ (X \times I)$$
$$V = V^{n+2,1} = B^{(n+1)}$$
$$W = W^{n+2,n+1} = C^{(n+2)} \circ (B \times I)$$

together with

$$\begin{array}{l} Z = \textbf{begin } T; \\ \quad \textbf{do begin} \\ \quad\quad \textbf{if } A \textbf{ then } W \textbf{ else } B \times I;\ U \\ \quad\quad \textbf{end} \\ \quad \textbf{until } \neg A; \\ \quad V \\ \textbf{end.} \end{array}$$

According to the lemma T, U, V, W are structured and

$$\begin{aligned}
\Psi_T(x_1, \ldots, x_n) &= (\epsilon, x_1, \ldots, x_n, \epsilon) \\
\Psi_U(x_1, \ldots, x_{n+1}) &= (\Psi_X(x_1, \ldots, x_{n+1}), x_1, \ldots, x_{n+1}) \\
\Psi_V(y, x_1, \ldots, x_n, x_{n+1}) &= x_{n+1} \\
\Psi_W(y, x_1, \ldots, x_n, x_{n+1}) &= (x_1, \ldots, x_n, c(x_{n+1})),
\end{aligned}$$

It follows that Z is structured, and furthermore

$$\begin{aligned}
\Psi_Z(x_1, \ldots, x_n) &= \Psi_T \circ \Psi_{\mathbf{do}\cdots\mathbf{until}\neg A} \circ \Psi_V(x_1, \ldots, x_n) \\
&= \Psi_{\mathbf{do}\cdots\mathbf{until}\neg A} \circ \Psi_V(\epsilon, x_1, \ldots, x_n, \epsilon) \\
&= \Psi_V(\epsilon, x_1, \ldots, x_n, \inf\{y: \Psi_X(x_1, \ldots, x_n, y) = \epsilon\}) \\
&= \inf\{y: f(x_1, \ldots, x_n, y) = \epsilon\} \\
&= (\mu f)(x_1, \ldots, x_n),
\end{aligned}$$

thus implementing minimalization.

COROLLARY. *Corresponding to each Turing machine Z is a structured connection of simple machines which computes the same (partial) function. The correspondence $Z \to Z^s$ is effective.*

Proof. The main assertion follows from the theorem and the classical equivalence between Turing machines and partial recursive functions. The effectiveness of the correspondence $f \to Z^s$ is already illustrated in the proof of the theorem. But we note that certain well-known constructions (Davis, 1958; Yasuhara, 1971) show that the association $Z \to f$ of partial recursive functions with arbitrary Turing machines is also effective.

6. CONCLUDING REMARKS

It can be argued a full-fledged canonical decomposition theorem would require that the "simple" machines exhibit some characteristic property not shared by all Turing machines. Thus in the fundamental theorem on Abelian groups, the building blocks are cyclic groups, and in the Krohn–Rhodes decomposition theorem, the "simple" machines are either two-state machines or machines whose semigroup is a simple group. So is this respect, our theorem is somewhat less than completely satisfactory. Nonetheless, we feel that the constructs involved and the connections themselves should prove to be useful as pedagogical tools in the Turing machine theory and as a framework for orderly algorithmic design.

In a broadest sense, our decomposition theorem might be viewed as a restatement of the fundamental ideas of structured programming. But surely this

theorem is not unique of its type. Other choices of "simple" machines are quite possible and other connection constructs could also be conceived. Quite obviously, we were looking for those connections which would most closely parallel the familiar structured programming constructs. As for the resulting class of simple machines chosen, we can at least argue that for the most part, they too are parallel to the use of the simple assignement statements of the structured programming theory. Thus C bears a strong resemblance to the assignment statement $y \leftarrow y + 1$ whereas E is aking to the familiar three-assignment exchange: temp $\leftarrow x$, $x \leftarrow y$, $y \leftarrow$ temp. An interesting question, however, in the context of the given set of connections and machines B, C, D, E is whether the machine $\hat{C}$ is actually necessary or whether it is a structured connection of B, C, D, E?

Returning once more to the analogy with structured programming and particularly the Bohm–Jacopini result, we note that a detailed proof of our corollary would point the way toward an effective procedure for transforming any given Turing machine into an equivalent structured machine. But surely this technique would be no more useful than is the Bohm–Jacopini construction for transforming an aribtrary flowchart into a structured flowchart, and in fact, such an approach would miss the point completely. For it seems that the value of either system of structured constructs lies in their applicability toward a "top-down" algorithmic design methodology. In the cas of Turing machines, it is not important that this approach be carried all the way down to the level of the "simple" machines, however these are defined. It is the design method itself which offers the advantages of clarity, simplicity, and in the right hands, even elegance and an assurance that the machine will indeed perform as intended.

RECEIVED: June 11, 1976; REVISED: November 23, 1976

REFERENCES

BOHM, C., AND JACOPINI, G. (1966), Flow diagrams, turing machines and languages with only two formation rules, *Comm. ACM* **9**, 366.

BRAINERD, W. S., AND LANDWEBER, L. H. (1974), "Theory of Computation," Wiley, New York.

DAHL, O. J., DIJKSTRA, E. W., AND HOARE, C. A. R. (1972), "Structured Programming," Academic Press, New York.

DAVIS, M. (1958), "Computability and Unsolvability," McGraw–Hill, New York.

DEAN, R. A. (1966), "Elements of Abstract Algebra," Wiley, New York.

EILENBERG, S., AND ELGOT, C. C. (1970), "Recursiveness," Academic Press, New York.

HERMES, H. (1965), "Enumerability, Decidability, Computability," Academic Press, New York.

MANNA, Z. (1974), "Mathematical Theory of Computation," McGraw–Hill, New York.

NELSON, R. J. (1968), "Introduction to Automata," Wiley, New York.

PRATHER, R. E. (1975), A convenient cryptomorphic version of recursive function theory, *Inform. Contr.* **27**, 178.

WIRTH, N. (1973), "Systematic Programming: An Introduction," Prentice–Hall, Englewood Cliffs, N. J.

YASUHARA, A. (1971), "Recursive Function Theory and Logic," Academic Press, New York.

Printed in Belgium