JOURNAL OF COMPUTER AND SYSTEM SCIENCES: 1, 187-212 (1967)

Some Definitional Suggestions for Automata Theory

DANA SCOTT*

Mathematics Department, Stanford University, Stanford, California 94305

There have been many new abstract machines suggested recently, and it seems like a good time to propose adopting a more uniform terminology. In particular, the so-called nondeterministic machines are very popular, and, even though the author helped contribute to this popularity, he now feels that is it simpler to avoid them. It will be adequately explained below how every purpose served by the nondeterministic non-machines can be quite adequately covered by the deterministic philosophy—if one only adopts the appropriate, obviously reasonable definitions. Further, we wish to advocate strongly the separation of the concepts of *program* and *machine.* Not only is this natural, but it helps to unify the presentation. In particular, definitions can be given once for all in sufficient generality that they do not have to be repeated every time an inspiration for a new machine strikes.[1]

The paper has six sections. In Section 1 programs (as linguistic objects) are defined. In Section 2 machines are defined. In Section 3 programs and machines are brought together for the purpose of computation. Section 4 contains examples of some useful machines. Section 5 shows how the various kinds of sets associated with machines

* A less polished version of this paper was presented at the Conference on the Algebraic Theory of Machines, Languages and Semigroups under the title "A Modest Proposal Concerning Nondeterministic Automata" on September 5, 1966. Preparation of the manuscript was supported by NSF Grant GP-3926.

[1] The plans of eliminating nondeterministic machines and giving greater emphasis to programs was formulated by the author in his lectures on automata theory at Stanford during 1964/65. Several other workers have made similar if not identical suggestions. Specific references will be given below. In this connection the referee in his helpful comments on the paper made the following remark which is worth quoting in full: "Although your main object is to unify and simplify the treatment of various kinds of machines, I think it is worth putting in a few more references to related work, since the mathematical theory of programs is in its infancy and it is by no means clear yet what are the best concepts to single out. Here, for example, you might mention that your definition of a program is virtually the same as what Kaluzhnin [*1*] would call an $\mathscr{F}$–$\mathscr{P}$ graph scheme. And that, apart from your (most useful) modification of the treatment of input and output, your machine of Def. 2.1. is what Kaluzhnin calls an interpretation $\{M; \mathscr{F} \to \mathscr{M}(\mathscr{F}); \mathscr{P} \to \mathscr{M}(\mathscr{P})\}$ for $\mathscr{F}$–$\mathscr{P}$ graph schemes. Also you ought to mention Elgot's "Abstract Algorithms and Diagram Closure" (preprint, IBM Laboratory, Vienna, 1966). Elgot's formulation does not accomplish what yours does but it is another very natural way of explicating the notion of program built up from certain basic types of command. He also discusses the notion of a generating algorithm, generable set, computable and semicomputable sets and functions, and various closure properties corresponding to those for the recursive functions."

can best be defined. One of the main points is the emphasis given to *functions* over *sets*. Thus the basic nature of a program is to compute a function according to Section 3. It will be seen in general that the sets fall into place the same way they do in ordinary recursive function theory. Finally, Section 6 collects together some miscellaneous applications.

1. Programs

From an elementary point of view a program is a structured set of instructions which allows a computer (human or mechanical) to successively apply certain basic operations and tests to given initial data until the data have been transformed into some desirable form. One of the simplest ways to indicate the structures of programs is by means of the well-known *flow diagrams*, one of which is illustrated in Fig. 1. We can imagine following the (unique) path of the flow of control in the diagram by

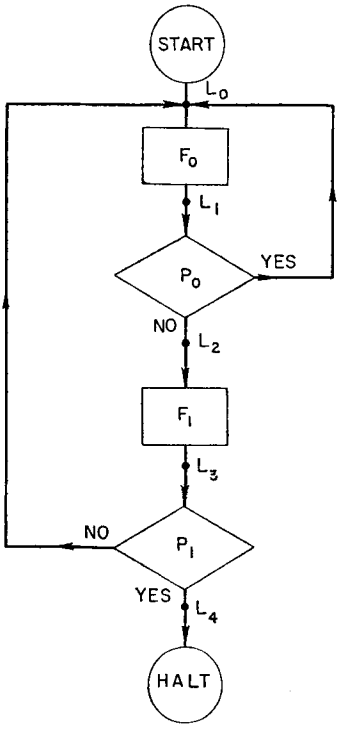

Fig. 1

starting with the initial data at the position marked *start*; applying first the operation given by the function F_0; to that result applying the test given by the predicate P_0; then switching control either back to F_0 or ahead to F_1; and so on until the halt position is reached— if ever. This is all very familiar and is basic to the idea of a program. Of course, people want more elaborate programs making use of subroutines and recursive procedures; but still, even those will be compiled into more direct programs that could be illustrated by such flow diagrams (very big diagrams to be sure!)

If we thus agree that flow diagrams can illustrate a sufficiently rich collection of programs, then we can begin to consider programs in general. Unfortunately, it is somewhat cumbersome to define in a rigorous mathematical way what a flow diagram is in general as a *geometrical* object, but it is easy to extract the essential structure from a diagram by judicious use of labels. In fact, in Fig. 1, some labels have been inserted at nodes on routes leading into each of the boxes. If we simply read the diagram, the following list of instructions is obtained:

start : **go to** L_0

L_0 : **do** F_0 ; **go to** L_1

L_1 : **if** P_0 **then go to** L_0 **else go to** L_2

L_2 : **do** F_1 ; **go to** L_3

L_3 : **if** P_1 **then go to** L_4 **else go to** L_0

L_4 : **halt**

If we then read over this list of instructions and follow the flow they suggest, we arrive back at our original diagram (more or less). Clearly, nothing essential of program structure is lost by considering such collections of instructions.

To be specific, instructions are made up from some basic symbols with the aid of *identifiers* chosen from certain sets $\mathscr{L}$ (the labels), $\mathscr{F}$ (the function or operation symbols), and $\mathscr{P}$ (the predicate or "test" symbols). To be even more specific, the identifiers in these sets could be regarded as strings of symbols from some conventional alphabet; but that is not too important. It is convenient, however, to assume that the three sets are pairwise disjoint and that the set of labels is infinite. We could fix $\mathscr{L}$ once for all, but we might wish to vary $\mathscr{F}$ and $\mathscr{P}$ so that our notation can have mnemonic content, but this is not too important either.

Next we imagine that

start	**if**	:
go to	**then**	;
do	**else**	**halt**

stand for nine remarkable symbols that never occur within any of our identifiers.

(That is a bit strict, but never mind.) Then we can say that an *instruction* is a string of one of the following four forms:

start : go to L

L : **do** F; **go to** L'

L : **if** P **then go to** L' **else go to** L''

L : **halt**

where $L, L', L'' \in \mathscr{L}$ and $F \in \mathscr{F}$ and $P \in \mathscr{P}$. Note that our convention allows us to forget whether we want spaces between the symbols and identifiers. (I know that the symbols :, **do**, **go to**, and; are not really necessary, but style has to count for something.) The first type of instruction is a *start instruction*; while the last type is a *halt instruction*. The other two types can be called *operation* and *test instructions*.

DEFINITION 1.1. A *program* is a finite set Π of instructions containing exactly one start instruction and containing for each label that occurs anywhere in any instruction in Π exactly one instruction that begins with that label.

It might be better to say that Π is merely a program *schema*, because the operation and predicate symbols in Π have not yet been given any meaning. But the shorter term does not seem to be misleading. The problem of giving meaning to the parts and whole of Π is, of course, the task of the next two sections. The reader should not expect any surprises.

2. MACHINES

To be able to follow a program explicitly we must know (i) how to input the data, (ii) how to perform the operations and tests, and (iii) how to output the results when a halt instruction is finally reached. It is just this information that is contained in the specification of a machine. To make the specification precise we will build a mathematical model for a machine by collecting together the relevant pieces of information in certain functions. The reader, of course, understands the abstract concept of a mathematical function, but it is helpful here to adopt some special conventions.

When we write

$$f : A \to B$$

we will mean that f is a *partial* function from the set A into the set B. That is, if $f(x)$ is defined, then $x \in A$ and $f(x) \in B$; but we do not assume that $f(x)$ is defined for *all* $x \in A$. We also want knowledge of f to determine the sets A and B uniquely. Very often functions are identified with sets of ordered pairs; thus f would be taken as the set of all pairs $(x, f(x))$ where $f(x)$ is defined. Such a convention would only

allow us to determine a *subset* of A and a *subset* of B from knowledge of f. If we want to be hyperprecise, we can identify the function with the ordered triple consisting of A and B together with the usual set of ordered pairs. Then we would have a mathematical notion of function with the desired features. The reason we want to stress partial functions is twofold: the so-called "don't care" conditions are natural in specifying machines; and programs, as we shall see, only define partial functions anyway.

For notational ease we also invent a symbol Ω to stand for *the undefined*. Thus the equation

$$f(x) = \Omega$$

is short for the statement "$f(x)$ is undefined;" while

$$f(x) \neq \Omega$$

is read "$f(x)$ is defined." We also make the convention that the equation

$$f(\Omega) = \Omega$$

is true. This is necessary for statements involving combinations of functions. Thus if $g(x) = \Omega$, then clearly we want

$$f(g(x)) = \Omega.$$

This result follows from our conventions simply by replacing equals by equals. The same conventions are extended to functions of several variables; hence

$$h(x, \Omega) = h(\Omega, y) = \Omega.$$

(Other conventions might be desirable for special purposes, but in general this seems to be the most straight-forward plan.) Further, we assume that Ω is an object *extraneous* to all the sets we will ever use in building machines or programs; that is,

$$\Omega \notin S$$

is taken as true for all normal sets S.

Partial functions are also very useful in specifying partial predicates and relations. Usually the meaning of a one-place predicate is taken to be simply a subset of a suitable fixed set A. To allow for the "don't care" conditions, we can say that the meaning of a predicate is a partial function

$$p : A \to \{\mathbf{T}, \mathbf{F}\},$$

where **T** stands for the Boolean value **true** and **F** for **false**. Similarly for binary relations we would use an

$$r : A \times A \to \{\mathbf{T}, \mathbf{F}\}.$$

We are now ready for the precise definition of a machine. To help unify the notation

we assume that the symbols **I** and **O** (standing for *input* and *output*) are identifiers *not* belonging to either of the sets $\mathscr{F}$ or $\mathscr{P}$.

DEFINITION 2.1. A *machine* is a function $\mathfrak{M}$ defined on the set $\{\mathbf{I}\} \cup \mathscr{F} \cup \mathscr{P} \cup \{\mathbf{O}\}$ for which there exist sets X (the input set), M (the memory set), and Y (the output set) such that

(i) $\mathfrak{M}_{\mathbf{I}} : X \to M$;

(ii) $\mathfrak{M}_F : M \to M$, for all $F \in \mathscr{F}$;

(iii) $\mathfrak{M}_P : M \to \{\mathbf{T}, \mathbf{F}\}$, for all $P \in \mathscr{P}$; and

(iv) $\mathfrak{M}_{\mathbf{O}} : M \to Y$.

In the above we have used the subscript notation $\mathfrak{M}_{\mathbf{I}}$ as an alternative to the function-value notation $\mathfrak{M}(\mathbf{I})$ so as to avoid the less readable combination $\mathfrak{M}(\mathbf{I})(X)$. Notice, by the way, that the sets X, M, and Y are uniquely determined by the machine $\mathfrak{M}$. Thus our conventions about functions help soften the somewhat clumsy "217-tuple"-style definition of abstract machines that has unfortunately become common.[2]

We shall often want to compare functions of one variable with functions of several variables. A (partial) function of 17 variables on a set A with values in B is indicated by

$$f : A^{17} \to B,$$

where $A^{17} = A \times A \times \cdots \times A$ (17 times). (Conventions should be arranged so taht $A^1 = A$.)

DEFINITION 2.2. A *system of machines* is a sequence $\mathfrak{M}^{(1)}, \mathfrak{M}^{(2)}, \ldots, \mathfrak{M}^{(n)}, \ldots$ of machines for which there exist sets X and Y such that the input set of each $\mathfrak{M}^{(n)}$ is X^n while all the $\mathfrak{M}^{(n)}$ have the same output set Y.

As it stands, this definition is not very informative. For one thing the actions of the various machines have in no way been related. This fault will be eliminated in Section 4, where, after the examination of certain examples, we shall specialize to machines with "standard" input/output for which the comparison of machines in a system will be simple to express.

The next step is to define just what a machine does when confronted with a program, or, to say it the other way around, what is the meaning of a program on a given machine.

3. COMPUTATIONS

Let Π be a fixed program, and let $\mathfrak{M}$ be a fixed machine with input set X, memory set M, and output set Y. We want to define what it means to follow the flow of the program on the machine; the history of this flow is called a computation.

[2] The author is indebted to Michael Harrison for the concept of the 217-tuple.

DEFINITION 3.1. A (*completed*) *computation* by the program Π on the machine $\mathfrak{M}$ is a finite sequence

$$L_0 , m_0 , L_1 , m_1 , \ldots , L_n , m_n$$

of alternating labels of Π and elements of M, where the label L_0 is contained in the start instruction of Π, the label L_n is contained in some halt instruction of Π, and where for $i < n$ we have either an instruction of the form

$$L_i : \textbf{do } F; \textbf{ go to } L'$$

belonging to Π, in which case $L_{i+1} = L'$ and $m_{i+1} = \mathfrak{M}_F(m_i)$, or an instruction of the form

$$L_i : \textbf{if } P \textbf{ then go to } L' \textbf{ else go to } L''$$

belonging to Π, in which case $m_{i+1} = m_i$, and either $\mathfrak{M}_P(m_i) = \mathbf{T}$ and $L_{i+1} = L'$, or $\mathfrak{M}_P(m_i) = \mathbf{F}$ and $L_{i+1} = L''$.

Given a value of m_0 , the start instruction of Π provides the label L_0 , and the rest of the computation sequence is strictly determined As we attempt to follow the program we may find that the developing sequence is *uncompletable* for one of the following reasons:

(1) A label of a halt instruction will never be reached, and the sequence is forced to go on forever;

(2) an instruction

$$L_i : \textbf{ do } F; \textbf{ go to } L'$$

is reached where $\mathfrak{M}_F(m_i) = \Omega$: or

(3) an instruction

$$L_i : \textbf{if } P \textbf{ then go to } L' \textbf{ else go to } L''$$

is reached where $\mathfrak{M}_P(m_i) = \Omega$

We shall reserve the word computation only for the completed sequences of the above definition

DEFINITION 3.2. The (*partial*) *function computed* by a program Π on a machine $\mathfrak{M}$ is that function

$$\mathfrak{M}_\Pi : X \rightarrow Y$$

such that for $x \in X$, $\mathfrak{M}_\Pi(x) \neq \Omega$ if and only if there is a computation

$$L_0 , m_0 , L_1 , m_1 , \ldots , L_n , m_n$$

such that $m_0 = \mathfrak{M}_\mathbf{I}(x)$ and $\mathfrak{M}_\mathbf{O}(m_n) \neq \Omega$, in which case

$$\mathfrak{M}_\Pi(x) = \mathfrak{M}_\mathbf{O}(m_n).$$

Note that $\mathfrak{M}_{\Pi}(x) = \Omega$ might occur simply because $\mathfrak{M}_{\mathbf{O}}(m_n) = \Omega$. In most obvious examples of machines the output function is not so unpleasant and is a totally defined function on M, but there is no reason to assume that it is always total.

The subscript notation used here in $\mathfrak{M}_{\Pi}(x)$ could be criticized because $\mathfrak{M}$ was by Definition 2.1 a *function* with domain $\{\mathbf{I}\} \cup \mathscr{F} \cup \mathscr{P} \cup \{\mathbf{O}\}$, and in that definition $\mathfrak{M}_F$ meant $\mathfrak{M}(F)$. Very true. So let us informally agree to extend the domain $\mathfrak{M}$ to include the set of all programs (assuming that identifiers are not programs!) Then $\mathfrak{M}_{\Pi} = \mathfrak{M}(\Pi)$ and everyone can be happy.

Even without looking at some specific examples of machines and programs, we can already make some interesting definitions and distinctions.

DEFINITION 3.3. Two programs Π and Π' are *equivalent* if and only if for all machines $\mathfrak{M}$

$$\mathfrak{M}_{\Pi} = \mathfrak{M}_{\Pi'}.$$

For example, every program is equivalent to one with a unique halt instruction; because if there is none, we can add one; while if there are several, they can be condensed to one by identifying labels. That is very trivial, and here is another such remark: every program is equivalent to one where no test instruction leads directly to a test instruction involving the *same* predicate symbol.

This notion of equivalence of programs is very strong. If we assume that the sets of identifiers are effectively given as, say, recursive sets of strings over a finite alphabet, then there is an effective decision method for equivalence of programs. This result can be improved to a decision method for equivalence of programs relative to some restricted class of machines.[3] It might be useful to see what is the most general result one can get along this line.

DEFINITION 3.4. A machine $\mathfrak{M}$ is [*effectively*] *reducible* to a machine $\mathfrak{M}'$ if and only if corresponding to each program Π one can [effectively] find a program Π' such that

$$\mathfrak{M}_{\Pi} = \mathfrak{M}'_{\Pi'}.$$

If in addition $\mathfrak{M}'$ is [effectively] reducible to $\mathfrak{M}$, then we say that the two machines are [*effectively*] *equivalent.*

Of the two notions suggested in Definition 3.4, undoubtedly the effective one is the more interesting. Many results about different formulations of the definitions of recursive functions actually establish effective equivalence of machines; we will give some examples in the next section. Note that the definition requires that the input

[3] If the author understands the results correctly, the decidability follows from the work of Yanov (cf. Rutledge [2]).

and output sets of the two machines be respectively equal. For a broader definition, assuming that X and Y are the input and output sets for $\mathfrak{M}$, and X', Y' are the sets for $\mathfrak{M}'$, we might allow encoding and decoding functions

$$e : X \to X' \qquad d : Y' \to Y.$$

We could let $d\mathfrak{M}'e$ be the machine that results from $\mathfrak{M}'$ by replacing its input/output functions by $\mathfrak{M}_I'e$ and $d\mathfrak{M}_O'$. Then if e and d exist so that $\mathfrak{M}$ is reducible to $d\mathfrak{M}'e$, we could say that $\mathfrak{M}$ can be *simulated* on $\mathfrak{M}'$. Whether such a broader notion is actually interesting remains to be seen.[4] These definitions can all be extended in the obvious way to families of machines.

4. Examples

The Turing Machine. The memory set of the Turing machine consists intuitively of two-way infinite tapes almost all squares of which are blank and which have one square under scan by the reading head. It will be enough to have just two states for a square of the tape: *blank* and *marked*. Mathematically we can represent *blank* by 0 and *marked* by 1 and a tape by a two-way infinite sequence $t = (..., t_{-2}, t_{-1}, t_0, t_1, t_2, ...)$, where each $t_i \in \{0, 1\}$. To indicate a tape with, say, the kth square under scan, we need only form the ordered pair (t, k). Thus we take M to be the set of all such pairs (t, k) where the sum $\sum_{i=-\infty}^{+\infty} t_i$ is finite, meaning that almost all squares of the tape are blank.

If we wish the Turing machine $\mathfrak{M}$ to compute number-theoretic functions, we take $X = Y = N = \{0, 1, 2, ...\}$ and define the input/output as follows:

$$\mathfrak{M}_I(m) = (t^{(m)}, 0),$$

where

$$t_i^{(m)} = \begin{cases} 1 & \text{if } i = m, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$\mathfrak{M}_O((t, k)) = \sum_{i=-\infty}^{+\infty} t_i .$$

In other words, we start out with the tape marked just once m spaces directly to the right of the reading head. When we halt, the output is the number of marked squares on the tape. Other conventions are possible, of course.

[4] This concept is related to what Evey ([3], Def. 5.2, p. 2-39) calls *weak equivalence*, except Evey requires that the functions e and d should be recursive.

Next we assume that the following identifiers belong to $\mathscr{F}$: MOVERIGHT, PRINT, MOVELEFT, ERASE, and that this identifier belongs to $\mathscr{P}$: BLANK?. We then define for $F \in \mathscr{F}$

$$\mathfrak{M}_F((t, k)) = \begin{cases} (t, k+1) & \text{if } F \text{ is MOVERIGHT,} \\ (t, k-1) & \text{if } F \text{ is MOVELEFT,} \\ (t(k/1), k) & \text{if } F \text{ is PRINT,} \\ (t(k/0), k) & \text{if } F \text{ is ERASE,} \\ \Omega & \text{otherwise,} \end{cases}$$

where the tape $t(k/i)$ is like t except it has its kth entry replaced by i. For $P \in \mathscr{P}$ we define

$$\mathfrak{M}_P((t, k)) = \begin{cases} \mathbf{T} & \text{if } P \text{ is BLANK? and } t_k = 0, \\ \mathbf{F} & \text{if } P \text{ is BLANK? and } t_k = 1, \\ \Omega & \text{otherwise.} \end{cases}$$

The Register Machine. This machine is, in the author's opinion, far superior to the Turing machine because it is so much more elementary.[5] The machines are effectively equivalent, however, in the precise sense of Definition 3.4. Intuitively, this machine has only two index registers which can be incremented, decremented, and tested for being zero. The contents of the registers can be represented simply by pairs of nonnegative integers (m_0, m_1), the totality of which form the memory set M for this Machine $\mathfrak{M}$. (Since we shall not study any one machine in detail in this paper, we need not introduce special symbols for the different machines.) The input/output are defined by

$$\mathfrak{M}_{\mathrm{I}}(m) = (2^m, 0)$$

and

$$\mathfrak{M}_{\mathrm{O}}((m_0, m_1)) = \begin{cases} k & \text{if } m_0 = 2^k \text{ and } m_1 = 0, \\ \Omega & \text{otherwise.} \end{cases}$$

We assume that the following identifiers belong to $\mathscr{F}$:

$$\mathbf{M}_0 \leftarrow \mathbf{M}_0 + 1 \qquad \mathbf{M}_1 \leftarrow \mathbf{M}_1 + 1$$
$$\mathbf{M}_0 \leftarrow \mathbf{M}_0 - 1 \qquad \mathbf{M}_1 \leftarrow \mathbf{M}_1 - 1$$

and that these identifiers belong to $\mathscr{P}$:

$$\mathbf{M}_0 = 0 \qquad \mathbf{M}_1 = 0$$

[5] The idea of this machine seems to be due independently to Minsky [4], Lambek [5] and Shepherdson–Sturgis [6], except that Minsky realized that only two registers are needed. The machine is also discussed in Evey ([3], Machine 13, pp. 2–78) and Fischer ([7], pp. 376–378).

We then define for $F \in \mathscr{F}$

$$\mathfrak{M}_F((m_0, m_1)) = \begin{cases} (m_0 + 1, m_1) & \text{if } F \text{ is } \mathbf{M}_0 \leftarrow \mathbf{M}_0 + 1, \\ (m_0 - 1, m_1) & \text{if } F \text{ is } \mathbf{M}_0 \leftarrow \mathbf{M}_1 - 1 \text{ and } m_0 > 0, \\ (m_0, m_1 + 1) & \text{if } F \text{ is } \mathbf{M}_1 \leftarrow \mathbf{M}_1 + 1, \\ (m_0, m_1 - 1) & \text{if } F \text{ is } \mathbf{M}_1 \leftarrow \mathbf{M}_1 - 1, \text{ and } m_1 > 0, \\ \Omega & \text{otherwise.} \end{cases}$$

For $P \in \mathscr{P}$ we define

$$\mathfrak{M}_P((m_0, m_1)) = \begin{cases} \mathbf{T} & \text{if } P \text{ is } \mathbf{M}_0 = 0 \text{ and } m_0 = 0, \\ \mathbf{F} & \text{if } P \text{ is } \mathbf{M}_0 = 0 \text{ and } m_0 \neq 0, \\ \mathbf{T} & \text{if } P \text{ is } \mathbf{M}_1 = 0 \text{ and } m_1 = 0, \\ \mathbf{F} & \text{if } P \text{ is } \mathbf{M}_1 = 0 \text{ and } m_1 \neq 0, \\ \Omega & \text{otherwise.} \end{cases}$$

The Post Machine. This machine is very close to Post's normal systems.[6] The memory set this time consists of strings (finite sequences) of letters from a finite alphabet Σ; that is, $M = \Sigma^*$. To be definite let us take $\Sigma = \{a, b\}$. The input/output sets could be taken simply as Σ^* itself, but for comparison with the previous machines we will again use the integers. We define

$$\mathfrak{M}_I(m) = a^m \; (= aaa \cdots a, \; m\text{-times}),$$

and

$$\mathfrak{M}_O(\sigma) = \begin{cases} m & \text{if } \sigma = a^m, \\ \Omega & \text{otherwise.} \end{cases}$$

The intuitive idea of the action of the machine is that it manipulates a string by reading and erasing symbols on the *left* and writing new symbols on the *right*; thus as the ends of the string undergo modification, the information in the string slowly circulates from right to left. Accordingly, assume that the following identifiers belong to $\mathscr{F}$:

$$\mathbf{M} \leftarrow (\mathbf{M}] \qquad \mathbf{M} \leftarrow \mathbf{M}a \qquad \mathbf{M} \leftarrow \mathbf{M}b$$

and that these identifiers belong to $\mathscr{P}$:

$$\mathbf{M} = a(\mathbf{M}] \qquad \mathbf{M} = b(\mathbf{M}]$$

[6] This machine was not, however, defined by Post. It is defined in Arbib [8] and (independently) in Shepherdson–Sturgis [6].

(If σ is a nonempty string, then $(\sigma]$ is the result of removing the left-most symbol from σ; while $(\Lambda] = \Omega$, where Λ is the empty string.) We then define for $F \in \mathscr{F}$:

$$\mathfrak{M}_F(\sigma) = \begin{cases} (\sigma] & \text{if } F \text{ is } \mathbf{M} \leftarrow (\mathbf{M}], \\ \sigma a & \text{if } F \text{ is } \mathbf{M} \leftarrow \mathbf{M}a, \\ \sigma b & \text{if } F \text{ is } \mathbf{M} \leftarrow \mathbf{M}b, \\ \Omega & \text{otherwise.} \end{cases}$$

For $P \in \mathscr{P}$ we define:

$$\mathfrak{M}_P(\sigma) = \begin{cases} \mathbf{T} & \text{if } P \text{ is } \mathbf{M} = a(\mathbf{M}] \text{ and } \sigma = a(\sigma], \\ \mathbf{F} & \text{if } P \text{ is } \mathbf{M} = a(\mathbf{M}] \text{ and } \sigma \neq a(\sigma], \\ \mathbf{T} & \text{if } P \text{ is } \mathbf{M} = b(\mathbf{M}] \text{ and } \sigma = b(\sigma], \\ \mathbf{F} & \text{if } P \text{ is } \mathbf{M} = b(\mathbf{M}] \text{ and } \sigma \neq b(\sigma], \\ \Omega & \text{otherwise.} \end{cases}$$

Remarkably enough this machine is effectively equivalent to the Turing machine and for some purposes is even better than the Register machine.

The Automation. This machine operates somewhat like the Post machine except that the reading and writing functions are separated and the machine can never reread what it has written. In this case it is more natural to take the input/output sets as both being Σ^*. The memory set is $\Sigma^* \times \Sigma^*$. We define

$$\mathfrak{M}_{\mathbf{I}}(\tau) = (\Lambda, \tau),$$

and

$$\mathfrak{M}_{\mathbf{O}}((\sigma, \tau)) = \sigma.$$

The intuitive idea this time is that the machine is allowed to read the second string symbol by symbol from left to right. Also, it is allowed to write on the first string again from left to right. Specifically we assume that these identifiers belong to $\mathscr{F}$:

$$\mathbf{M}_1 \leftarrow (\mathbf{M}_1] \qquad \mathbf{M}_0 \leftarrow \mathbf{M}_0 a \qquad \mathbf{M}_0 \leftarrow \mathbf{M}_0 b$$

and that these belong to $\mathscr{P}$:

$$\mathbf{M}_1 = a(\mathbf{M}_1] \qquad \mathbf{M}_1 = b(\mathbf{M}_1]$$

We then define for $F \in \mathscr{F}$

$$\mathfrak{M}_F((\sigma, \tau)) = \begin{cases} (\sigma, (\tau]) & \text{if } F \text{ is } \mathbf{M}_1 \leftarrow (\mathbf{M}_1], \\ (\sigma a, \tau) & \text{if } F \text{ is } \mathbf{M}_0 \leftarrow \mathbf{M}_0 a, \\ (\sigma b, \tau) & \text{if } F \text{ is } \mathbf{M}_0 \leftarrow \mathbf{M}_0 b, \\ \Omega & \text{otherwise.} \end{cases}$$

For $P \in \mathscr{P}$ we define

$$\mathfrak{M}_P((\sigma, \tau)) = \begin{cases} \mathbf{T} & \text{if } P \text{ is } \mathbf{M}_1 = a(\mathbf{M}_1] \text{ and } \tau = a(\tau], \\ \mathbf{F} & \text{if } P \text{ is } \mathbf{M}_1 = a(\mathbf{M}_1] \text{ and } \tau \neq a(\tau], \\ \mathbf{T} & \text{if } P \text{ is } \mathbf{M}_1 = b(\mathbf{M}_1] \text{ and } \tau = b(\tau], \\ \mathbf{F} & \text{if } P \text{ is } \mathbf{M}_1 = b(\mathbf{M}_1] \text{ and } \tau \neq b(\tau], \\ \Omega & \text{otherwise.} \end{cases}$$

Note that our conventions here about how the automaton calculates are somewhat different from what is usual in the literature. In the first place, relative to a given program, it is not necessary to have our automaton read the whole input tape before halting; the program can be modified to an equivalent program which acts in this way, if desired. Secondly, it is possible to test the input for being empty since, when $\tau = \Lambda$, the predicates $\mathbf{M}_1 = a(\mathbf{M}_1]$ and $\mathbf{M}_1 = b(\mathbf{M}_1]$ are both false. Thus we need not have any artificial endmarkers on the ends of tapes. In the third place the writing of output is not made to synchronize with the reading of the input; thus the distinctions between Moore, Mealy, and generalized sequential machines do not even suggest themselves, though special programs can be constructed to simulate these machines in an obvious way. Finally, functions computed by the automaton are, in general, partial functions; however, it can be shown that the set on which the function is defined is a regular event, and so the function is the restriction of a function calculated by a generalized sequential machine to this set. Therefore, it seems fair to say that our definition includes the previous ones but is not too general.

If we allowed the automaton to read and write on both strings we would get back to the level of the Turing machine (or the Post machine).[7] Indeed, these operations and predicates would be sufficient:

$$\mathbf{M}_i \leftarrow (\mathbf{M}_i] \qquad \mathbf{M}_1 \leftarrow a\mathbf{M}_i \qquad \mathbf{M}_i \leftarrow b\mathbf{M}_i$$

$$\mathbf{M}_i = a(\mathbf{M}_i] \qquad \mathbf{M}_i = b(\mathbf{M}_i]$$

for $i = 0, 1$. (The reader may easily make this new machine explicit. Notice how the mnemonic character of the identifiers soon allows us to skip some of the tiresome details.) The input/output can be taken to be the same as for the automaton. This machine uses the two strings as *push-down store* memory locations (first-in–last-out); while the Post machine uses its one string as a first-in–first-out memory.

The Push-Down Store Machine. Two push-down stores are equal in power to one Turing machine tape. The restriction to *one* PDS is a definite restriction that

[7] This result is due to McCarthy [9]. It is discussed in Evey ([3], Theorem 5.1, p. 2-47) and Fischer ([7], Lemma 2, p. 376).

has gained considerable popularity as a result of the connection of this machine with the context-free languages. The memory set sould be here

$$M = \Sigma^* \times \Sigma^* \times \Sigma^*,$$

where the first string indicates the contents of the PDS memory location; the second, the output; and the third, the input. (Note that in these machines the input/output cannot be conveniently encoded into the working memory as in some of the more powerful machines.) It is therefore clear how to define the input/output functions on the set Σ^*. The operations and predicate identifiers are as follows:

$$\mathbf{M}_0 \leftarrow a\mathbf{M}_0 \qquad \mathbf{M}_0 \leftarrow (\mathbf{M}_0] \qquad \mathbf{M}_1 \leftarrow \mathbf{M}_1 a$$

$$\mathbf{M}_0 \leftarrow b\mathbf{M}_0 \qquad \mathbf{M}_2 \leftarrow (\mathbf{M}_2] \qquad \mathbf{M}_1 \leftarrow \mathbf{M}_1 b$$

$$\mathbf{M}_0 = a(\mathbf{M}_0] \qquad \mathbf{M}_0 = b(\mathbf{M}_0]$$

$$\mathbf{M}_2 = a(\mathbf{M}_2] \qquad \mathbf{M}_2 = b(\mathbf{M}_2]$$

The reader may be trusted with giving the precise definitions of $\mathfrak{M}_F$ and $\mathfrak{M}_P$.

Though for a general theory we probably want to allow for quite diverse input/output sets and for clever encodings and decodings to and from the memory set, it would seem in practice that the most useful set is Σ^*, where Σ is a finite alphabet usually fixed to be $\{a, b\}$. Furthermore, it is rather natural to perform all our encoding and decoding explicitly within the program itself. Thus, we are led to define the concept of a machine with *standard* input/output; this has the advantage of making the *system* corresponding to an initially given machine appear naturally.

DEFINITION 4.1. A machine $\mathfrak{M}$ is a (unary) machine with *standard input/output* if and only if its input/output sets are both Σ^* and its memory set is of the form

$$M = M_0 \times \Sigma^* \times \Sigma^*,$$

where, in addition, for some fixed $m_0 \in M_0$

$$\mathfrak{M}_I(\tau) = (m_0, \Lambda, \tau)$$

and

$$\mathfrak{M}_O((m, \sigma, \tau)) = \sigma$$

for all $\sigma, \tau \in \Sigma^*$ and $m \in M_0$. Further, the operations and tests of $\mathfrak{M}$ are allowed to operate only coordinate-wise on M, and while no restriction is imposed on those for the first coordinate, those allowed for the remaining coordinates are the obvious interpretations of these identifiers:

$$\mathbf{M}_1 \leftarrow \mathbf{M}_1 a \qquad \mathbf{M}_1 \leftarrow \mathbf{M}_1 b$$

$$\mathbf{M}_2 \leftarrow (\mathbf{M}_2] \qquad \mathbf{M}_2 = a(\mathbf{M}_2] \qquad \mathbf{M}_2 = b(\mathbf{M}_2]$$

In other words, we have an unrestricted memory coordinate, but in the output coordinate we can only write, and in the input coordinate we can only read and erase symbols. To generalize now from a given unary machine $\mathfrak{M}$ to its corresponding n-ary machine $\mathfrak{M}^{(n)}$, we have only to replace the memory set by

$$M^{(n)} = M_0 \times \Sigma* \times \Sigma* \times \cdots \times \Sigma* \ (n + 2 \text{ factors})$$

The input/output are given by

$$\mathfrak{M}_{\mathrm{I}}^{(n)}((\tau_0, \ldots, \tau_{n-1})) = (m_0, \Lambda, \tau_0, \ldots, \tau_{n-1})$$

and

$$\mathfrak{M}_{\mathrm{O}}^{(n)}((m, \sigma, \tau_0, \ldots, \tau_{n-1})) = \sigma.$$

All operations and tests on M_0 are retained; while the others are expanded to include the obvious interpretations of

$$\mathbf{M}_i \leftarrow (\mathbf{M}_i] \qquad \mathbf{M}_i = a(\mathbf{M}_i] \qquad \mathbf{M}_i = b(\mathbf{M}_i]$$

for $i = 2, 3, \ldots, n + 1$.

This approach to systems of machines seems better, because we can show (for example) that, if

$$f : \Sigma* \times \Sigma* \to \Sigma*$$

is a function of two arguments computed on $\mathfrak{M}^{(2)}$, then

$$g : \Sigma* \to \Sigma*$$

defined by the equation

$$g(\tau) = f(\tau, \alpha),$$

where $\alpha \in \Sigma*$ is fixed, is computable on $\mathfrak{M}^{(1)}$. This would not be true for the completely general notion of a system in Definition 2.2.

As we gave the examples, the PDS machine has standard input/output. Strictly speaking the automaton does not. The equivalent version of the automaton that fits Definition 4.1 would have $M_0 = \{m_0\}$ with *no* operations or tests on the first coordinate. Actually this restriction is too severe, for it is not only useful but interesting to prove the proposition:

> *Any machine with standard input/output which has a finite M_0 is equivalent to the automaton.*

If the identifiers for the operations and tests of the machine are effectively given along with their meanings (for example, if they are but finite in number), then the equivalence is effective. More generally we can take a given machine and replace

its M_0 by $M_{-1} \times M_0$, where M_{-1} is a new finite set. Any desired operations and tests can be supplied on this new coordinate. No matter what is done, the new machine is equivalent to the given machine.

The Post machine as it stands has integer input/output. To convert it to the desired form with standard input/output, one makes the old memory set into the new M_0 retaining the given operations and tests on this set. The functions computed by the resulting (system) of machines are the partial recursive functions (of any number of arguments) of strings. Unfortunately, the Register machine does not convert so easily into the proper standard form. The reason is that the exponential encoding of the input gave the machine an advantage. When encoding is not allowed, as in the present case, the best way to preserve equivalence with the Post machine is to increase the number of registers from 2 to 3; that is, let

$$M_0 = N \times N \times N,$$

with the obvious operations and tests thereon. For the Turing machine, no change is required: the old memory set becomes the new M_0.

For the remainder of this paper we assume that all unary machines have standard input/output, and that all systems arise from a unary machine in the way just explained. As it stands this convention seems to exclude such machines as the two-way stack automaton,[8] because they treat the input as a weak kind of memory. It will take some further development to see how these new machines will fit into the overall scheme.

5. Sets

Sometimes automata theory is introduced as a study of classification of sets (subsets of the input set which is usually taken to be Σ^*.) The main emphasis is placed on the regular events or the context-free languages, etc. The author (along with many other people) has come recently to the conclusion that the *functions* computed by the various machines are more important—or at least more basic—than the sets accepted by these devices. The sets are still interesting and useful, but the functions are needed to understand the sets. In fact, by putting the functions first, the relationships between the various classes of sets becomes much clearer. This is already done in recursive function theory, and we shall see that the same plan carries over to the general theory.

Let $\mathfrak{M}$ be a given machine with standard input/output and let $\mathfrak{M}^{(1)} = \mathfrak{M}, \mathfrak{M}^{(2)}, \mathfrak{M}^{(3)}, \ldots$ be the corresponding system of machines, as explained in the last section. We have fixed $\Sigma = \{a, b\}$, but, of course, any finite alphabet with at least two letters would be just as good. Using functions computed by the machines of the system we can single out three classes of subsets of Σ^* which seem to be the most important.

[8] See Ginsburg–Greibach–Harrison [*10*] for the definition of these machines.

Definitions similar to the following could be given for arbitrary machines, but then in that generality there do not seem to be any useful theorems forthcoming.

DEFINITION 5.1. A subset $S \subseteq \Sigma^*$ is called *decidable* on $\mathfrak{M}$ if and only if there is a program Π such that for all $\tau \in \Sigma^*$:

$$\mathfrak{M}_\Pi(\tau) = \begin{cases} a & \text{if } \tau \in S, \\ b & \text{if } \tau \notin S. \end{cases}$$

DEFINITION 5.2. A subset $S \subseteq \Sigma^*$ is called *acceptable* on $\mathfrak{M}$ if and only if there is a program Π such that for all $\tau \in \Sigma^*$:

$$\mathfrak{M}_\Pi(\tau) = a \quad \text{if and only if} \quad \tau \in S.$$

DEFINITION 5.3. A subset $S \subseteq \Sigma^*$ is called *generable* on $\mathfrak{M}$ if and only if there is a program Π and an integer $n > 0$ such that for all $\tau \in \Sigma^*$:

$$\mathfrak{M}_\Pi^{(n)}(\tau, \tau_1, \ldots, \tau_{n-1}) = a \quad \text{for some} \quad \tau_1, \ldots, \tau_{n-1} \in \Sigma^* \quad \text{if and only if} \quad \tau \in S.$$

Clearly every decidable set is acceptable, and every acceptable set is generable (because $n = 1$ is allowed in Definition 5.3). The word "decidable" was chosen because the program allows the machine to give a definite yes-or-no answer to every question of membership in the set. The word "acceptable" is meant to convey the feeling that the elements actually in the set are accepted, but we don't care what the program tells us to do in the opposite case. The word "generable" indicates that we have to search through all the $(n - 1)$ tuples of the input set to find one to use as auxiliary input to generate a positive answer. Notice in all definitions the programs used all operate in a deterministic fashion. The connection between these concepts and those defined with the aid of "nondeterministic" machines will be discussed in full below. The notions can easily be extended to relations (subsets of Σ^{*m}), and we shall leave the exact formulation to the reader.

The three concepts are not in general equivalent; this follows from known results. First, in the case of the automaton they are all the same; namely, they coincide with the notion of a *regular event*. In the case of the PDS machine, decidable equals acceptable but generable is definitely broader. Finally, in the case of Turing machines, decidable is stricter than acceptable which equals generable (i.e., domains of definition of partial recursive functions are the same as the recursively enumerable sets.) It seems likely that there are other reasonable machines for which all three notions are simultaneously distinct.

In these definitions we have used a to stand for **T (true)** and b for **F) false)**. This was necessary because the functions computed on $\mathfrak{M}$ take values in Σ^* not in $\{\mathbf{T}, \mathbf{F}\}$. The choice of a and b was conventional, however. It can be proved directly, or it follows from the Composition Theorem presented in the next section, that a set S

is decidable on $\mathfrak{M}$ if and only if there is a program Π' such that $\mathfrak{M}_{\Pi}$ is a *total* function and for all $\tau \in \Sigma^*$,

$$\mathfrak{M}_{\Pi'}(\tau) = \Lambda \quad \textit{if and only if} \quad \tau \in S.$$

On the other hand S is acceptable if and only if there is a program Π' such that for all $\tau \in \Sigma*$,

$$\mathfrak{M}_{\Pi'}(\tau) = \Lambda \quad \textit{if and only if} \quad \tau \in S,$$

and, in fact,

$$\mathfrak{M}_{\Pi'}(\tau) \neq \Omega \quad \textit{if and only if} \quad \tau \in S.$$

Thus to accept a string τ we run the program Π' with this input until it halts. In other words, S is the domain of definition of a partial function that gives at most the empty output. A similar result holds for the generable sets. These remarks are familiar from recursive function theory, but we see that they hold for general machines with standard input/output.[9] We did not choose this alternate form for the basic definition, however, because from the stated form of 5.1 it is clear that the complement of a decidable set is decidable. Other closure properties for either the decidable or acceptable sets are not apparent.

Turning our attention now specifically to the generable sets, we can prove first that this class of sets is closed under union. Suppose Π_0 and Π_1 are two programs generating two sets. We can clearly assume that the same machine $\mathfrak{M}^{(n)}$ with $n > 1$ is used in both cases. We shall generate the union of the sets on $\mathfrak{M}^{(n+1)}$. By a suitable choice of notation assume that the labels of Π_0 and Π_1 are different. Then remove the start instructions from each of the programs, take the union, and add the following instructions:

start : **go to** L

L : **if** $\mathbf{M}_{n+1} = a[\mathbf{M}_{n+1}]$ **then go to** L_0 **else go to** L_1

where L is a new label and L_0 and L_1 are the labels of the start instructions of Π_0 and Π_1, respectively. It is obvious that this new program generates the union. This simple observation of how an input location can act as a switch to direct the program into one or the other of two channels beings us directly to the topic of *nondeterministic programs.*

In a deterministic program, each instruction leads to a well-determined next instruction. In a so-called nondeterministic program, one is given a choice of which instruction to execute next. Without going into formal details we can easily imagine a format for such programs. Without loss of generality, we can also imagine that each

[9] Evey [3] notes such facts for the collection of machines he considers which are all variants of multiple PDS machines. The context here is more general, however; and the results are seen to be simple properties of input/output logic without reference to the internal memory.

choice is a *binary* choice. To be a little more definite, let us suppose the program is a routine for a nondeterministic *acceptor*. Thus there is only one input, and the trick is to make the right combination of choices terminating in a halt. Now the technique used in converting this non-program into a real program is based on the maxim that nothing should ever be left to chance. We simply supply an extra input location whose content, a string of a's and b's, tells us the proper sequence of choices to make (a stands for the first choice and b for the second in the binary situation.) It is a straightforward exercise to rewrite the acceptor program as one that generates the same set in the sense of Definition 5.3. Of course, it is really no simpler to run a generator than to run a non-deterministic acceptor, because it is just as hard to find the right combination of inputs as it is to make the right combination of choices. In fact, by using a little "label logic" we can show conversly that every generable set is acceptable by a non-deterministic program. *Thus the two concepts are equivalent.*[10]

The author prefers the notion of the generable set, however, because the deterministic idea of a program is the only natural one, it should be fixed once for all, and then the *use* to which a program is put can be varied at will. It is technically just as easy to use the notion advocated here, and we are saved not only from making unnecessary definitions but also from abusing the generally understood word "program." Therefore, let us leave nondeterminism to the realm of philosophy (or better: probability).

Looking back on what was just suggested, we can isolate a mathematical fact which is independent of the methodological issue.

> *Every set generable on* $\mathfrak{M}^{(n)}$ *with large* n *is, by what we have said, generable on* $\mathfrak{M}^{(2)}$.

We could argue this directly without going through the nondeterministic programs. We can even go a step further. The two input strings can be merged into a single string. Think of it this way: the symbols of the two strings are used up in a definite order by taking a little from one, then a little from the other, and so on. Splice together these little pieces of the two strings and form a new string using a different "color" (or coding) to refer to the two original strings. The generable set we get this way is not all that interesting. But look, as we run this one-string program hopefully to a halt, we can give some *output*. Indeed why not untangle the two strings by decoding the contribution from the original first string. This "improved" program for $\mathfrak{M}^{(1)}$ will define a *function* whose range of values is just the original generable set we started with. We can thus show:

[10] In Rabin–Scott [*11*], the connection is noted for the automaton in one direction (generable implies nondeterministically acceptable). The result is related to Fischer ([7], Theorem 2, pp. 371–372).

A set is generable if and only if it is the range of a computable (partial) function.[11]

That fact is well known in recursive function theory, and it has this much broader range of applicability. For comparison, we might note:

A set is acceptable if and only if it is the range of a computable (partial) function that is a fragment of the identity function.

That is probably of very little use, but it sounds pleasant. What is really nice for comparison is the earlier established fact:

A set is acceptable if and only if it is the domain of a computable (partial) function.

There is probably not too much more to be proved of a general nature about these sets without making more definite assumptions about the operations and tests on M_0. A very simple and general assumption is to suppose that some sequence of operations (and maybe tests also) will *reset* the content of M_0 back to m_0 (the convential initial content) without requiring any input/output. Under this assumption one very easily proves that the generable subsets of Σ^* are closed under formations of products and the $*$.

In the case of relations it is obvious that the projection (existential quantification) of a generable relation is again generable. It can also be proved that a generable set is always the projection of a *decidable* relation instead of just an acceptable relation as in 5.3. (Hint: use one input tape as a "clock" to mark time. If the clock "runs down" before a proper halt is reached, then give some nonempty output. If the halt is reached "in time," then halt.) Also easy to show is the fact that the cartesian product of decidable (resp. acceptable, generable) sets is a decidable (resp. acceptable, generable) relation. Some closure properties of a "mixed" nature are mentioned in the next section.[12]

6. Applications

To have an application, one must have something to apply. What we have to apply is the idea of *the function*. That is to say, once we agree that functions are better than sets (the theme of Section 5), then problems and solutions begin to appear rather naturally. We shall exhibit three examples.

[11] Evey [3] seems to have first established this fact for his special collection of machines. See also Fischer ([7], Theorem 2, pp. 371-372).

[12] Fischer ([7], Theorem 1) gives a result for certain special machines that shows that the generable sets are closed under reversal. The author does not see whether this generalizes to the present context.

After functions have been looked at one at a time as in the previous sections, the next step is to combine them by composition. Thus if $f : \Sigma^* \to \Sigma^*$ and $g : \Sigma^* \to \Sigma^*$ are both computable on $\mathfrak{M}$, the obvious question to ask is whether $gf : \Sigma^* \to \Sigma^*$ is computable, where, of course,

$$gf(\tau) = g(f(\tau))$$

for all $\tau \in \Sigma*$. In general, the answer is, rather surprisingly, "no." It all depends on the kind of machine you use. On a very powerful machine that can store the output of the first function f within its internal memory M_0 and still compute g, the composition is computable. The Turing machine is a good example. The PDS machines are bad examples, however, because it is known that PDS generable sets are (primitive) recursive. But it is also known than an arbitrary recursively enumerable set can be found as the range of the composition of two PDS functions. Therefore, the PDS functions are *not* closed under composition.[13]

We can generalize the composition problem by having f computed on one kind of machine, g on another. Then we ask what kind of machine is needed for gf. There are actually some useful answers to be found. Suppose we consider only machines with standard input/output. Thus the automaton is allowed—it is, so to speak, the weakest kind of machine we are willing to consider. Then, if we assume merely that *one* of f or g can be computed by the *automaton*, then the composition can be computed on the *same* machine as that of the *other* function.

To understand this last assertion, let us assume for sake of argument that f is computed on the automaton. The function g is computed on some fancy machine that we will not even have to specify exactly. Consider that the program for g takes as its input the output from f. But every computation on strings operates by reading or destroying just one symbol at a time. So let us combine the programs for the two functions by setting up an alternate, back-and-forth style of computation. The details of control can be taken care of by the labels. The idea is to let g have the main control. Start the computation off in the usual way until the first call for input is encountered. Then interrupt and let f's program compute away until the first symbol of output is produced. Interrupt f at this point and return to the computation of g. And so on. The number of labels required is finite: the product of the number of labels in the two programs ought to suffice. Notice, however, that the memoryless character of f's program is essential to the argument: the state of the memory used by g must not be disturbed while f is being computed. Notice also, that the roles of f and g can, by the same argument, be interchanged. One important point to watch, however, concerns the halting rule. If g, the "outside" function, finally wants to halt, it must remember to go back to the computation for f and wait until it halts. The reason for this is that $f(\tau) = \Omega$ might be possible, and we want in this case $g(f(\tau)) = \Omega$.

[13] This follows from the method employed in Ginsburg–Hibbard–Ullian ([*12*], Lemma 2.5, p. 326) and in Hartmanis [*13*]. The author also discovered the same proof independently.

This composition result already has a pleasant application:

The image of a set generable on a more powerful machine by a function computable by the automation is generable on the same more powerful machine.

In particular, the atomaton function could be simply the identity function restricted to a regular event. We thus conclude in suitable generality:

The intersection of a generable set with a regular event is generable.

This includes several known results.[14]

An obvious further generalization of the composition problem concerns functions of several variables. First the negative result: even if f_0, f_1, and g are all automaton functions, the composition

$$h(\tau) = g(f_0(\tau), f_1(\tau))$$

as a function of *one* variable need not be computable on the automaton. The reason is that we can choose f_0, f_1, and g so the domain of h is the set

$$\{a^n b^n : n = 0, 1, 2,...\}$$

which is *not* a regular event. In fact, let f_0 and f_1 be defined only on the set

$$\{a^n b^m : n, m = 0, 1, 2,...\}$$

where

$$f_0(a^n b^m) = a^n,$$

and

$$f_1(a^n b^m) = a^m.$$

Then we let g be such that

$$g(a^n, a^m) = \begin{cases} \Lambda & \text{if } n = m, \\ \Omega & \text{otherwise.} \end{cases}$$

The conclusion then follows.

The positive result, for what it is worth, is as follows: a composition

$$h(\tau_0, \tau_1) = g(f_0(\tau_0), f_1(\tau_1)),$$

as a function of *two* variables where at most *one* of f_0, f_1, g requires a more powerful machine than the automaton, can itself be computed on the same more powerful machine. We can call this *composition with disjoint variables*. The trouble with overlapping variables is that the programs for the different functions will consume the inputs at different rates. With disjoint variables this problem is avoided.

[14] See, e.g., Ginsburg ([*14*], Theorem 3.2.1).

Just as we generalized the intersection result for regular events and generable sets, so we can generalize the so-called Substitution Theorem.[15] This result includes the product and $*$-closure theorems and requires that we assume our machines to possess the reset capability. Suppose then that S is generable on one kind of machine and that $f : \Sigma^* \to \Sigma^*$ is computable on another, where at least one of the machines is the automaton. Then on the more powerful machine we can generate the set of strings

$$\{f(\xi_0)f(\xi_1) \cdots f(\xi_{n-1}) : \xi_0\xi_1 \cdots \xi_{n-1} \in S\}.$$

To understand what is intended here the resulting set can be described in words: take any string in S and decompose it into any number of parts, say $\xi_0\xi_1 \cdots \xi_{n-1}$. *If* $f(\xi_i)$ is defined for all $i < n$, *then* put the combination $f(\xi_0)f(\xi_1) \cdots f(\xi_{n-1})$ into the new set. It is the convention allowing partial functions f that makes the result useful.

One of the most interesting facts about the PDS machine is the well-known theorem that the PDS-generable sets are the same as the context-free languages.[16] The proof in one direction is just like our earlier proof relating generable with nondeterministically acceptable sets. Suppose the set S is given by a context-free grammar. Now it is easy to prove that the generation of strings by the grammar can always be arranged so that the rules are applied to the *left-most* nonterminal symbol. (The alphabet Σ will now in general be larger than just $\{a, b\}$.) We can very directly write down a program for a PDS-function whose range is S. An input to the program is regarded as a code word giving us a string of rules from the grammar. The content of the PDS-memory location is the portion of the generated word which still has some nonterminal symbols. The program acts in this fashion : The symbol at the top of the PDS is checked. If it is terminal, it is given as output. If it is nonterminal, the next segment of the input is read. If the input has not presented a rule appropriate for the nonterminal, the program loops (that is, the function is made undefined—note that no memory is required here). If an appropriate rule is presented, then the nonterminal is replaced on the top of the PDS by the indicated string and the program goes through the cycle again. When the PDS is finally emptied, the program halts. In the very first phase of the computation, the single nonterminal that is the "axiom" of the grammar is, of course, entered in the initially empty PDS.

For the proof in the other direction let S be any PDS-generable set where we can write

$$S = \{f(\xi) : \xi \in \Sigma*\}$$

[15] Cf. Ginsburg ([*14*], Theorem 1.7.1).

[16] Cf. Chomsky [*15*] for references to earlier work. Evey [*3*] first established the result in the form stated here. The proof that context-free are PDS-generable is the same as Evey's. The proof for the converse outlined below is due to Mr. William Ogden of Stanford and seems to be simpler than Evey's.

(This time we can go back to $\Sigma = \{a, b\}$.) Without loss of generality, we can assume that the program for f never has to test either the input or the PDS for being empty. (Hint: both locations can be assumed to operate in code where a certain unique pattern indicates the end of the string before emptiness is actually encountered.) Further, we can assume that reading instructions are always immediately followed by instructions to destroy the symbol read (and reading the empty string produces the undefined.) In the notation we have been using, $\mathbf{M}_0$ stands for PDS, $\mathbf{M}_1$, the output and $\mathbf{M}_2$ the input. By way of abbreviation let

$$R[\mathbf{M}_i, L', L'']$$

stand for the sequence of instructions to read and destroy the left-most symbol of M_i and transfer to the statement labeled L' or L'' according as the symbol read is a or b. Similarly, let

$$W[\lambda, \mathbf{M}_i, L']$$

be the instruction to write a letter λ on $\mathbf{M}_i$ and transfer to L'. (Remember that the λ's go on the *right* of the output and on the *left* of the PDS.) Let the labels of the program be $L_0, L_1, \ldots L_p$, where L_0 is the label of the start instruction and L_p is the label of the halt instruction. Let $\mathscr{R}$ be the set of all $r < p$ for which L_r is the label of a PDS read statement, i.e., the statement

$$L_r : R[\mathbf{M}_0, L_{\mathbf{a}(r)}, L_{\mathbf{b}(r)}]$$

occurs in the program. Here $\mathbf{a}(r)$ and $\mathbf{b}(r)$ are the indices specifying the transfer to be made after reading a symbol a or b. Our context-free grammar for S is going to be based on terminal symbols a, b, and nonterminals L_i^r where $i \leqslant p$ and $r \in \mathscr{R} \cup \{p\}$. The axiom is L_0^p. The rules of the grammar are read off the instruction in the program as follows:

$$\left.\begin{matrix} L_i^r \to L_j^r \\ L_i^r \to L_k^r \end{matrix}\right\} \quad \text{for } L_i : R[\mathbf{M}_2, L_j, L_k];$$

$$L_i^r \to \lambda L_j^r \quad \text{for } L_i : W[\lambda, \mathbf{M}_1, L_j];$$

$$L_i^i \to \Lambda \quad \text{for } L_i : R[\mathbf{M}_0, L_{\mathbf{a}(i)}, L_{\mathbf{b}(i)}];$$

$$L_i^r \to L_j^s L_{\mathbf{a}(s)}^r \quad \text{for } L_i : W[a, \mathbf{M}_0, L_j], \text{ all } s \in \mathscr{R};$$

$$L_i^r \to L_j^s L_{\mathbf{b}(s)}^r \quad \text{for } L_i : W[b, \mathbf{M}_0, L_i], \text{ all } s \in \mathscr{R};$$

$$L_p^p \to \Lambda \quad \text{for } L_p : \mathbf{halt}.$$

What one must show is that a sequence of replacements of left-most nonterminals which successfully eliminates all nonterminals exactly corresponds to a completed computation by the given program. Note that in the rules corresponding to the write instructions for the PDS, a nonterminal may be replaced by any one of several different strings. The trick is to look ahead and choose the s to be the index of the instruction that will *read* the symbol just written. That shows why the rule corre-

sponding to the read instruction can be so trivial. Be careful, however; if a wrong s is chosen, you will find yourself eventually with a nonterminal L_i^s, $i \neq s$, i the index of a read instruction, and there are *no* rules for eliminating such symbols.

As a last application we shall prove the recursive unsolvability of Post's Correspondence Problem.[17] For this purpose, the Post machine is by all odds the best. One first establishes in the usual way the unsolvability of the Halting Problem for programs on the Post machine. Note that for halting questions the whole difficulty lies in the transformations of the content of the memory—input and output can be forgotten. Further, we can so encode our memory that it is the *emptiness* of the memory that triggers a halt (rather than a loop as in the previous discussion.) Thus consider a program for the Post machine having labels $L_0, \ldots, L_p$, where L_0 is for **start**, L_p is for **halt** and the instructions are of these two types:

$$L_i : R[L_j, L_k, L_p],$$

which is short for reading and destroying the *left-most* symbol and transferring to L_j if a is read, L_k if b is read, and to **halt** if the empty string is read; and

$$L_i : W[\lambda, L_j],$$

which is short for writing λ as the *right-most* symbol and transferring to L_j. Further, we can assume the *start* leads to this conventional instruction:

$$L_0 : W[a, L_1].$$

From this program, we shall now effectively construct a system of corresponding pairs of words from the alphabet $\{a, b, e, L_0, \ldots, L_p\}$ such that this correspondence problem has a solution *if and only if* the program ever halts after being started with the initial state of the memory being empty. Thus it will be shown that the Halting Problem reduces to the Correspondence Problem, and therefore the latter is unsolvable.

We begin our set of corresponding pairs with

$$(ea, ae) \quad \text{and} \quad (eb, be),$$

and then adjoin the following which are read off the program:

$$\begin{array}{rl} (L_0, L_0 eae L_1 e) & \text{for } L_0 : W[a, L_1]; \\ (eL_i, \lambda e L_j e) & \text{for } L_i : W[\lambda, L_j]; \\ \left.\begin{array}{r}(eL_i ea, L_j e) \\ (eL_i eb, L_k e) \\ (eL_i e L_p, L_p)\end{array}\right\} & \text{for } L_i : R[L_j, L_k, L_p]. \end{array}$$

[17] Few people have seemed to read Post [*16*] even though they have applied his result many times. When the author was forced to give the theorem in lectures, he discovered the proof given below. At the time Post wrote [*16*], he apparently did not know that the word problem for arbitrary normal systems could be reduced to that for monogenic (deterministic) systems. This probably explains why his proof in [*16*] was given for the more complicated reduction.

One should now verify for himself that a product of words from the first members of these pairs can equal a corresponding product from the second members in one and only one way. Indeed the solution (if it exists) of this correspondence equation transcribes almost word for word a completed computation. Conversely, the completed computation gives a solution for the equation. The idea of this proof is certainly the same as in the original Post argument, but we have been able to avoid certain tiresome reductions because our programs are *deterministic*. And so again it seems best to forget about nondeterminism.

References

1. A. Kaluzhnin, *Problems of Cybernetics* **2**, 371-391 (1961).
2. J. D. Rutledge. *J. ACM* **11**, 1-9 (1964).
3. R. J. Evey. *The theory and applications of pushdown store machines.* Doctoral dissertation, Harvard University (1963).
4. M. Minsky. *Ann. Math.* **74**, 437-454 (1961).
5. J. Lambek. *Canadian Math. Bull.* **4**, 295-302 (1961).
6. J. C. Shepherdson and R. E. Sturgis. *J. ACM* **10**, 217-255 (1963).
7. P. C. Fischer. *Info. Control* **9**, 364-379 (1966).
8. M. A. Arbib. *J. Australian Math. Soc.* **3**, 301-306 (1963).
9. J. McCarthy. "Recursive Functions of Symbolic Expressions and their Computation by Machine (The LISP Programming System)" [*Quart. Progr. Rept. No. 53*, Research Laboratory of Electronics, M.I.T. Cambridge, Massachusetts (1959)].
10. S. Ginsburg, S. Greiback, and M. Harrison. *J. ACM* **14**, 172-201 (1967).
11. M. Rabin and D. Scott. Finite automata and their decision problems, *IBM J. Res. Develop.* **3**, 114-125 (1959).
12. S. Ginsburg, T. N. Hibbard, and J. S. Ullian. *Illinois J. Math.* **9**, 321-337 (1965).
13. J. Hartmanis. Context-free languages and Turing machine computations (Unpublished).
14. S. Ginsburg. "The Mathematical Theory of Context-Free Languages." McGraw Hill, New York, 1966.
15. N. Chomsky. *Handbook Math. Psyc.* **2**, 323-418 (1963).
16. E. L. Post. *Bull. Am. Math. Soc.* **52**, 264-268 (1946).

Printed in Belgium