IA

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 42/75 JULI

R.M. BAER & J. VAN LEEUWEN

THE HALTING PROBLEM FOR LINEAR TURING ASSEMBLERS

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

AMS (MOS) subject classification scheme (1970): 02F15, 68M25

ACM -Computing Reviews- categories: 5.26

Proposed running head:

LINEAR TURING ASSEMBLERS

Send page-proofs to:

Dr. Jan van Leeuwen
Dept. of Computer Science
SUNY at Buffalo
4226 Ridge Lea Road
Amherst, NY 14226

THE HALTING PROBLEM FOR LINEAR TURING ASSEMBLERS

Robert M. Baer[*]

Department of Computer Science, University of California, Berkeley, and Department of Biochemistry & Biophysics, University of California, San Francisco

and

Jan van Leeuwen[**]

Department of Computer Science, University of California, Berkeley, and Department of Mathematics, University of Utrecht, Utrecht, The Netherlands.

[*] The research of this author was supported, in part, by Grant USPHS GM-17539.

[**] The research of this author was supported by the Netherlands Organization for the Advancement of Pure Research (ZWO). A part of this paper was prepared at the Mathematical Centre, Amsterdam 1005, The Netherlands.

Abstract Turing assemblers are Turing machines which operate on n-dimensional tapes under restrictions which characterize a procedure of assembly rather than computation, and which are intended as an abstraction of certain algorithmic processes of molecular biology. It has been previously shown that Turing assemblers with n-dimensional tapes can simulate arbitrary Turing machines for all $n > 1$. Here it is shown that for $n = 1$ even nondeterministic Turing-assemblers have a sharply restricted computational capability, being able to successfully assemble only regular sets. The halting problem for linear Turing-assemblers is therefore algorithmically solvable, and a characterization of the set of achievable final assemblies will be given as a subclass of the context-free languages.

§1. Introduction

The objects of molecular biology are assembled by mechanisms not yet completely understood. Some objects seem to develop through a process of self-assembly, while others seem to be algorithmically assembled by mechanisms such as ribosomes. The latter processes raise the question to what extent notions of assembling are equivalent to the well understood subjects of computability. In the study of cellular automata it appears that the distinction between computing capability and replicating capability can be either emphasized or erased, according to one's taste (i.e., depending on how the notion of replication is formalized). In any case, the relation between a constructing ability and a computing ability is well understood for cellular automata. The same relation appears not to have enjoyed the same depth of study in the case of sequential automata. In Baer [2], a study was initiated with a view toward remedying this state of affairs.

As a model of a mechanism which is to build a structure by assembling together a suitable collection of building-blocks according to some algorithmic prescription, it was proposed in [2] that a restricted version of a Turing machine with n-dimensional work-tape might be suitable. The principal restriction held that the machine could operate only in contact with the structure being erected and that extensions of the structure could be realized only by adjoining building-blocks to the surface of the structure. Such machines are called *Turing assemblers*. The building-blocks are taken to be n-dimensional unit cubes which come in finitely many varieties, and in unlimited quantity for each variety. A collection of blocks is said to constitute an *assembly* if the blocks in the collection form a connected set in n-dimensional space. (Here, "connected" means the transitive closure of "facewise-connected".)

The operation of a Turing assembler is understood as the process which results when the Turing assembler is placed in its initial state, at some canonical position on an initial assembly. If the Turing assembler enters a terminal state, the assembly produced is taken to be the n-dimensional construct which exists upon the Turing assembler's halt. In [2] it was shown that Turing assemblers which operate in n-dimensional space ($n \geqq 2$) can simulate any (ordinary one-head, one-tape) Turing machine, and hence have universal computing capability, even if there is only one variety of building-block available for erecting assemblies. It follows that the halting problem for such Turing assemblers is undecidable.

Turing assemblers which operate in one-dimensional space are called linear assemblers. Such constructors appear to have an algorithmically more interesting behaviour with some useful interpretations in language theory. We shall therefore give a detailed study of linear assemblers, considering both several restricted models of linear assemblers as well as the general non-deterministic linear assembler. We shall relate their domains and ranges to regular and (linear) context-free languages, solve the halting problem for the various models, and also compare the kinds of linear assembler which we distinguish with more traditional types of automata.

Since the halting problem for linear assemblers (in their most general form) is solvable such machines cannot serve as universal computational devices in molecular biology, and are indeed considerably less powerful than the higher-dimensional assemblers.

In §2 we briefly review some of the concepts from automata- and language-theory, and recall some terminology introduced in [2].

In §3 we consider (for any natural number k) the character of assemblies produced by linear assemblers which are constrained to make exactly k (full) passes across their assemblies. We show that a set of assemblies produced by any such a k-pass assembler can also be produced by a 2-pass linear assembler. Assuming that the machine may start on any arbitrary initial assymbly, the range of a k-pass assembler is shown to be a regular set.

In §4 we examine the class of ω-pass linear assemblers which can reverse their motion only at the ends of the assemblies on which they operate but which have no constraint upon the number of passes which they may make across their assemblies. We show that, for each ω-pass assembler T, there is a one-pass assembler $\bar{T}$ which adjoins as a suffix to its initial assembly the sequence of alternating suffixes and prefixes which T adjoins to its initial assembly. A corollary to this result is that the halting problem for ω-pass assemblers is strongly decidable.

In §5 the general class of non-deterministic linear assemblers is considered. After a detailed argument involving the use of crossing-sequences it is shown that the set of all initial assemblies on which such an assembler may halt is a regular set which can be effectively determined from the program of the assembler.

In §6 the solvability of the halting problem for the general linear assembler is shown to follow from the result in §5, and some applications to language theory are presented. In particular it is shown that the range of a linear assembler is a linear context-free language (and each linear context-free language can be so obtained).

§2. Preliminaries

Our notations and terminology will mainly follow standard references like Hopcroft & Ullman [5] or Salomaa [8]. Finite, non-empty sets of symbols are called alphabets. The number of elements in an alphabet Σ is denoted by $\# \Sigma$. Finite sequences or strings of symbols from Σ are called words, and the set of all words over Σ (including the empty word λ) is denoted by Σ^*. For $x, y \in \Sigma^*$ we denote their concatenation by xy, and with this operation Σ^* is recognized as the free monoid generated by Σ. The length of a word x is denoted by $|x|$, $|\lambda| = 0$.

Subsets of Σ^* are called languages (over Σ). The product of languages L_1 and L_2 is $L_1L_2 = \{xy\colon x \in L_1\ y \in L_2\}$, the star of a language L is $L^* = \cup\{L^n\colon n \geq 0\}$ where by definition $L^0 = \{\lambda\}$ and for all $n \geq 0$ $L^{n+1} = L^nL$. A mapping $h\colon L_1 \to L_2$ is called a homomorphism if $h(xy) =$ $= h(x)h(y)$.

We will assume that the reader is reasonably familiar with some of the machine-models discussed in Hopcroft & Ullman [5] like finite-state automata and stack-automata, although no detailed knowledge of it will be required.

A language is called regular just in case it can be recognized by a finite-state automaton. One can alternatively characterize the regular languages as the smallest family of sets containing the finite languages that is closed under the operations of (set-theoretic) union, product, and star.

We shall denote Turing-machines as $T = \langle Q,\Sigma,\pi\rangle$, where Q is a finite set of states, $\Sigma = A \cup \{\square\}$ an alphabet (with $\square$ a distinguished symbol called the blank), and π the program of T. The program is a list of quintuples $\langle p,a,b,q,m\rangle$ ($p,q\in Q; a,b\in\Sigma; m\in\{-1,0,1\}$) which are all interpreted as instructions. If T is in state p when scanning symbol a on its tape, then T will replace a by b, switch to state q, and move left ($m=-1$), right ($m=1$) or not at all ($m=0$), in accordance with some instruction $\langle p,a,b,q,m\rangle$ in its program. If the program π contains no pair of distinct quintuples both beginning with $\langle p,a,\ldots\rangle$, then T is said to be deterministic (non-deterministic otherwise). The Turing-machines we consider are single tape, single head devices. Note that a finite state automaton is a Turing-machine whose program contains quintuples of the form $\langle p,a,a,q,1\rangle$ ($a\neq\square$) and $\langle p,\square,\square,p,0\rangle$ only.

Turing machines begin their computation on tapes which contain just one string over A and which are otherwise blank (i.e., the data on the tape as string over Σ contains no embedded blanks). Moreover, all machines which we consider shall maintain this condition of no embedded blanks during the course of computation.

It shall be useful to distinguish those states of a machine $T = \langle Q,A \cup \{\square\},\pi\rangle$ which drive it to the right (left) whenever the machine scans a non-$\square$ symbol. The set of right-states of T is $\overrightarrow{Q} = \{p \in Q: \langle p,a,b,q,m\rangle \in \pi \;\&\; a \neq \square \Rightarrow m = 1\}$, the set of left-states $\overleftarrow{Q}$ is defined similarly.

A Turing machine is called a linear (Turing-) assembler when the following conditions on its program π hold. If $a \neq \square$ and $\langle p,a,b,q,m\rangle \in \pi$

then $b = a$ (i.e., the machine cannot rewrite any non-$\square$ symbol). Also, if $<p,\square,\square,q_1,m_0> \in \pi$ and $p \in \vec{Q}$ and $<q_i,\square,\square,q_{i+1},m_i> \in \pi$ $(i=1,\ldots,)$, then $m_i \neq 1$ $(i=1,\ldots,k)$. (i.e., the machine, having arrived at a blank square as a result of a move to the right, may not thereafter move further to the right without placing a nonblank symbol in this square). The corresponding condition is to hold when the machine arrives at a blank square as a result of a move to the left. These latter restrictions correspond to a view of the assembler as a machine which adjoins building-blocks to the ends of a linear array of such blocks (nonblank symbols), and whose motion is confined to the physical structure being assembled (see Baer [2]).

A configuration of $T = <Q,A \cup \{\square\},\pi>$ is denoted either as a string of the form UqV $(q \in Q; U,V \in A^*)$ or as a triple of the form $<q,j,W>$ $(q \in Q; W \in A^*;$ j any integer). UqV denotes that the current assymbly is UV and that the machine is scanning the first symbol of V (unless V is empty, in which case the machine is scanning the blank at the immediate right of U). To explain the second denotation we consider the tape-squares of T indexed by the set Z of rational integers; the tape itself is then described by a mapping $W: Z \rightarrow A \cup \{\square\}$. Then $<q,j,W>$ denotes the machine in state q, scanning square j of a tape described by W. (We consider only tapes carrying an assembly and usually identify W with this assembly.)

The computations of a linear assembler T may now be formally defined as (finite) sequences of successive configurations of the assembler. We denote the set of final assemblies which can be produced by T from the initial configuration q_1U by T(U). When T is non-deterministic it may very well happen that T(U) has many elements. We let domain (T) = $\{U \in A^*: T(U) \neq \emptyset\}$ and range (T) = $\cup \{T(U): U \in$ domain (T)$\}$.

It will sometimes be convenient to identify the set of states Q of a Turing assembler with $\{0,1,\ldots,\#Q - 1\}$. In so doing, we shall always identify 0 as the halting state, and 1 as the initial state (i.e., the state in which the machine, scanning the leftmost symbol of its initial assembly, starts). Whether starting in state 1 or not, if the machine moves from one end of its current assembly to the other with no reversals of motion, we say that the machine has made a *pass* across the assembly.

Still thinking of the tape-squares of an assembler T as being indexed by Z, we define $q_U(i) = q_{j_1} q_{j_2} q_{j_3} \ldots$ to be the sequence of states in which T has crossed the boundary between squares i and i+1 in a computation on U. The sequences $q_U(i)$ are called the crossing-sequences of the computation, a concept due to Hennie [3] and Trakhtenbrot [10] that we shall use in §4.

§3. Assemblies generated by k-pass assemblers

Let k be a natural number. A k-pass assembler is a deterministic linear assembler $T = \langle Q, A \cup \{\square\}, \pi \rangle$ which operates in the following way.

The initial configuration is of the form q_1W, where q_1 is the initial state of T and $W \in A^*$. If k=0 then the machine simply adjoins a prefix to the initial assembly without ever making a left-to-right pass across the assembly. If $k \neq 0$ then T moves steadily to the right of W and upon reaching its end adjoins a bounded suffix to W. If k = 1, the machine then terminates; if k > 1, the machine next makes a right-to-left pass along the current assembly, adjoins a prefix upon reaching the left end of the current assembly, and continues to make passes and adjoin fixes until precisely k passes and fixes have been made, at which point *T* terminates.

Note that k-pass assemblers are somewhat related to a transducer-model described by Schützenberger [9], and can also be viewed as a special type of bounded crossing transducer (Rajlich [7]).

Theorem 3.1. If T is a k-pass assembler, there is a 2-pass assembler T which is equivalent to T.

Theorem 3.2. The range of a k-pass assembler is a regular set.

Recall that if the (say right hand edges of the) squares of a Turing-machine tape are thought of as indexed by Z in the natural way, then one particular gage on the computation of the machine may by defined in the following way. Let $\phi(i,W)$ be the number of times the read/write head passed edge i during the computation which begins with configuration q_1W, and let $\Phi(W) = \max_i \phi(i,W)$. The function Φ is called the *alternation gage* of the machine. By a theorem of Trakhtenbrot [10], if a Turing machine (with a one-sided infinite tape) transforms the set A^* in a way such that there is a constant c for which $\Phi(W) < c$ for all $W \in A^*$, then there is an equivalent Turing machine (again with one-sided infinite tape) whose alternation gage Φ' satisfies $\Phi'(W) = 1$ for all $W \in A^*$; i.e., the equivalent machine produces the same result as the original machine but needs only one pass (i.e., needs no reversal of motion) to produce this result. Any k-pass assembler satisfies the condition of Trakhtenbrot's theorem except for a technicality that stems from Trakhtenbrot's use of one-sided (infinite) tapes. In the next section we show that there is version of Trakhtenbrot's theorem that applies to linear assemblers more general than k-pass assemblers (in that the number of passes need not be bounded).

The proof of 3.1 is trivial for the cases $0 \leq k \leq 2$ and is proved for larger values of k by induction, by way of construction of a suitable equivalent (one-less-pass) machine. We shall need a more elaborate version of this same scheme for the construction in the next section, and save the argument for there.

The proof of 3.2 is almost immediate from the statement of 3.1.

We note that 3.2 is related to a result of Schutzenberger [9], which states that the transform of a regular set by a two-pass transducer is again a regular set. However, transducers rewrite strings whereas assemblers merely extend them.

§4. The halting problem for ω-pass assemblers

By an *ω-pass assembler* we mean a deterministic linear assembler which is required to move uniformly across its current assembly at each pass, without being constrained to make a fixed number of passes (like the k-pass assembler). Thus, if an ω-pass assembler is in a right-state and scanning a nonblank symbol, then the assembler must move right and go into a right-state. There is a corresponding condition for the action due to left-states. An ω-pass assembler may cycle in a given position only if it is scanning a blank square. An ω-assembler assembler halts just in case it enters a halting-state, and without loss of generality it may be assumed that this happens (if at all) only upon completion of a suffix (or prefix).

The ω-pass assemblers are in their operation somewhat related to Hibbard's scan-limited automata ([4]).

An ω-pass assembler may have two different types of divergent behaviour: extending the assembly indefinitely or finally cycling back and forth across the assembly without extending if further.

We shall show that the halting problem for ω-pass assemblers is solvable by reducing it to the halting problem of one-way assemblers. A *one-way assembler* is an assembler whose program contains no quintuple $<\ldots,-1>$; such machines can make no left-moves. A one-way assembler may adjoin many suffices to an assembly, in this way contrasting 1-pass assemblers which always halt after adjoining the first suffix.

Lemma 4.1. The halting problem for one-way assemblers is solvable.

Proof. Let $T = <Q,\Sigma,\pi>$ be a one-way assembler and $\#Q = n$. If T starts on an assembly W, then T takes $|W|$ steps to reach the end of W, and then may take at most $n - 1$ steps without halting or repeating some instruction $<q,\square,\ldots>$. $\square$

Lemma 4.2. For each ω-pass assembler $T = <Q,A \cup \{\square\},\pi>$ one can effectively construct a one-way assembler $\bar{T} = <\bar{Q},\bar{A} \cup \{\square\},\bar{\pi}>$ which has the following property: for any string $W \in A^+$, $\bar{T}$ halts on W just in case T halts when applied to W. Further, if T (at the end of successive left-to-right passes) adjoins suffixes $V_1,V_2,\ldots$ to the assembly and (at the end of successive right-to-left passes) adjoins prefixes $U_1,U_2,\ldots$, then $V_1U_1V_2U_2\ldots$ is the suffix which $\bar{T}$ adjoins to W.

Proof. We modify the argument used in Baer [2] to reduce k-pass assemblers (k>2) to 2-pass assemblers. We are here considering only deterministic machines, so let $\pi(q,a)$ denote the unique quintuple in π that begins $<q,a,..>$. Let $(\pi(q,a))_i$ denote the i-th component of the quintuple. Let $Q' = \vec{Q}^n$ (where $n=\#Q$); let $P(Q)$ denote the power set of Q; and let Q" be the collection of n-tuples $<P_1,..,P_n>$ over $\mathcal{P}(\overleftarrow{Q})$ satisfying: $P_i \cap P_j = \emptyset$ if $i \neq j$. We construct $\bar{T}$ as follows. As the set of states of $\bar{\mathcal{T}}$ we take $\bar{Q} = Q \times Q' \times Q''$. The purpose of the second and third components, $T_{\bar{q}}$ and $T'_{\bar{q}}$ of any state $\bar{q} \in \bar{Q}$ is the maintenance of an updated correspondence of the states q' to which states q are driven by left-to-right passes of the current assembly, and right-to-left passes, respectively. If $T = <p_1,..,p_n> \in Q'$ and $s \in \Sigma$, we write Ts in place of the n-tuple $<(\pi(p_1,s))_4,...,(\pi(p_n,s))_4>$. If $T' = <P_1,...,P_n> \in Q''$ and $s \in \Sigma$, we write $s^{-1}T'$ for the n-tuple $<P'_1,...,P'_n>$ in which $P'_i = \{q \in Q: (\pi(q,s))_4 \in P_i\}$ $(i=1,..,n)$. Note that, since π is single-valued, if the components of T' are pairwise disjoint, then the components of $s^{-1}T'$ are also pairwise disjoint.

The program $\bar{\pi}$ of $\bar{\mathcal{T}}$ is a union of sets Π_i of instructions corresponding to different phases of the behavior of $\bar{\mathcal{T}}$.

Corresponding to the initial pass of $\mathcal{T}$ across the starting assembly W, we set

$$\Pi_1 = \{<\bar{p},a,a,\bar{q},1> \mid <p,a,a,p,1> \in \pi \wedge a \in A \wedge \bar{p} = <p,T,T'> \Rightarrow \bar{q} = <pa,Ta,a^{-1}T'>)\}$$

(where it is understood that, in the braces, T and T' range over all permissible values).

Corresponding to $\mathcal{T}$ changing state but motionless on a blank square, we set

$$\Pi_2 = \{<\bar{p},\Box,\Box,\bar{q},0> \mid <p,\Box,\Box,q,0> \in \pi \wedge \bar{p} = <p,T,T'> \Rightarrow \bar{q} = <q,T,T'>\}.$$

Corresponding to T printing a suffix, we set

$$\Pi_3 = \{\langle\bar{p},\Box,a,\bar{q},1\rangle \mid \langle p,\Box,a,q,1\rangle \in \pi \wedge \bar{p} = \langle p,T,T'\rangle \Rightarrow \bar{q} = \langle q,Ta,a^{-1}T'\rangle\}.$$

Corresponding to T printing a prefix, we set

$$\Pi_4 = \{\langle\bar{p},\Box,a,\bar{q},1\rangle \mid \langle p,\Box,a,q,-1\rangle \in \pi \wedge \bar{p} = \langle p,T,T'\rangle \Rightarrow \bar{q} = \langle q_a,\hat{T},\hat{T}'\rangle\},$$

where q_a, $\hat{T}$, and $\hat{T}'$ are defined in the following way: We recall that we identify the states in the set Q with an initial segment of the natural numbers; then $(\hat{T})_n = (T)_n a$ (all $n \in Q$) and

$$(\hat{T}')_n = \cup_k\{(T')_k \colon (T')_k \cap a^{-1}n \neq \emptyset\}$$

and the notation q_a is used to emphasize the fact that when T writes a prefix of length greater than unity, $\bar{T}$ must be programmed to write this same prefix in reverse order. Thus, suppose that the current assembly is W and that T reaches the left end of W and writes the prefix $U = u_1 .. u_k$ and then reverses its direction. The complication which arises when $k \geqq 2$ is dealt with by providing $\bar{T}$ with a set of states which cause $\bar{T}$ to write U from left to right (rather than from right to left as T does). Thus in the definition of the instructions in Π_4 the state q_a is generally distinct from the state q.

Corresponding to reversing its motion after writing a suffix we set

$$\Pi_5 = \{\langle\bar{p},\Box,\Box,\bar{q},0\rangle \mid \langle p,\Box,\Box,q,-1\rangle \in \pi \wedge \bar{p} = \langle p,T,T'\rangle \wedge$$
$$\bar{q} = \langle q',T,T'\rangle \wedge q' = (\mu i)[q \in (T')_i]\}$$

where we note that (μi) ("the least i such that..") might just as well be written (ιi) ("the unique i such that.."), a point we shall return to, below.

Corresponding to T reversing its motion after writing a prefix we set

$$\Pi_6 = \{\langle\bar{p},\Box,\Box,\bar{q},0\rangle \mid \langle p,\Box,\Box,q,1\rangle \in \pi \wedge \bar{p} = \langle p,T,T'\rangle \wedge$$
$$\bar{q} = \langle q',T,T'\rangle \wedge q' = (T)_q\}.$$

To verify the statement of the Lemma, we need make only a few observations. First, the sets Π_i of instructions, above, form a union which is clearly the program of a one-way assembler. We take as the initial state of $\bar{T}$ the state $\langle q_1,T_1,T_1'\rangle$ where q_1 is the initial state of T, T_1 is a list of the right-states of T, and T_1' is a list of singletons representing the left-states of T. Second, the behavior of $\bar{T}$, when started on the leftmost symbol of an assembly W, acts in the appropriate way. We consider the different phases of this behavior.

As $\bar{T}$ makes its pass across the initial assembly W, it keeps track of the corresponding state of T, were T passing across the same assembly. $\bar{T}$ also, at each step, updates the two lists, T and T', which keep track of the states to which each right-state of T would be driven by W, and a list of sets of left-states corresponding to the left-states to which T would be driven (by right to left passes across the current assembly). Upon reaching the first blank (bordering W on the right), if T hesitates -

changes state without writing and advancing - then, according to Π_2, so does $\bar{T}$. If T writes a suffix, say V, then, according to Π_3, so does $\bar{T}$, and moreover $\bar{T}$ updates the lists T and T to correspond to the current assembly.

When a suffix has been written by $\bar{T}$ (as it would have been written by T), $\bar{T}$ can extract directly from the list T' the state to which T would be then driven by a right-to-left pass across W, and can then accordingly write (in backwards order) the prefix, say U, which T would have written at the end of its right-to-left pass across the assembly. As $\bar{T}$ writes the string U, it updates both T and T'. When U has been written, T extracts directly from the table T the state to which T would be driven by the current assembly from the state in which T would have found itself upon beginning a left-to-right pass across this assembly.

We note that the computation of $\bar{T}$ terminates just in case that of T does, and in the event of termination the final assembly is just as stated in the Lemma. □

From 4.1 and 4.2 we have immediately

Theorem 4.3. Whether any ω-pass Turing assembler halts when started on any assembly W is decidable (by a Turing machine, within $|W| + n \times n^n \times 2^{n^2}$ steps, where n is the number of states of the assembler).

§5. Nondeterministic linear assemblers

In this section we consider the computations of general non-deterministic linear assemblers. (Note that such assemblers are a special, generative counterpart to Hibbard's scan-limited automata [4]).

Any initial assembly on which an assembler may produce a finite, halting computation is called hopeful (for this assembler).

We shall prove that there is an effective characterization of the hopeful assemblies of a general assembler and use it in section 6 to solve the halting problem for the most general case.

We begin with an easily proved, preliminary lemma.

<u>Lemma 5.1</u>. For any linear assembler there is an equivalent linear assembler which extends assemblies by adjoining successive suffixes and prefixes of length at most one.

<u>Theorem 5.2</u>. For any nondeterministic linear assembler, the set of hopeful assemblies is a regular set which can be effectively determined.

Before we give the proof of this theorem we shall discuss the idea behind the tedious construction that we need.

Consider an assembler operating on a particular initial assembly X. We wish to keep track of the position of the leftmost symbol of this initial assembly, so let Ω be the boundary between the square containing this symbol and the (blank) square to its left. Let L denote the part of the tape to the left of Ω, and R denote the part of the tape to the right, i.e., if $X = x_1 \ldots x_k$, then we may represent the situation by the following figure:

$$\underbrace{\cdots\cdots\ \square\ \square\ \square}_{L}\underset{\Omega}{|}\underbrace{x_1x_2\cdots x_k\ \square\ \square\ \square\ \cdots\cdots}_{R}$$

The assembler, starting on x_1, may operate for a while on R and then cross over Ω into L, operate on L for a while, cross back, etc.

The strategy of the following construction is to consider the operation of the assembler on the R and L regions separately. This requires that the assembler, operating on a particular region, should be able to guess the state in which it might return to this region if the region is exited. Then the assembler's behaviors on the left and right regions can be made consistent by requiring that the crossing sequences be consistent. This leads to constructing, from the original assembler, an acceptor of permissible crossing sequences. The acceptor is a modification of the original assembler which uses the region R as a storage tape, and which behaves like a one-way stack-automaton.

We shall prove that the the behavior of the assembler as an acceptor of classes of permissible crossing sequences can be reduced to that of a finite automaton.

The essential point is to eliminate the need for keeping track of the entire L-on R-part of the assembly. Instead, we show that it is sufficient to store only the current right-most symbol of the assembly, together with a few tables that completely describe the behavior, as far as the assembler needs it, of the assembler operating on the initial part of L or on R.

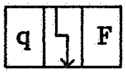
We first describe the tables. (We explain this only for the R-part where the initial assembly occurs. The construction is similar for the L-part).

With each assembly symbol σ (appearing eventually on the R-part), two tables will be associated:

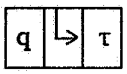
- the <u>top-departure table</u> T_σ describing the behavior with σ currently as a rightmost occurrence when the machine moves stationary or in the inside of R.

- <u>the bottom-departure table</u> B_σ describing the behavior with this instance of σ currently as the rightmost symbol just after the machine returns (in a "guessed" state) across Ω onto R again.

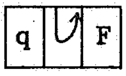
The tables T_σ have entries of the form

(i) 

, with $q \in Q$, $F \subseteq Q$

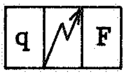
(ii) 

, with $q \in Q$ and τ denoting LOOP or HALT

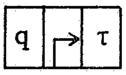
(iii) 

, with $q \in Q$, $F \subseteq Q$

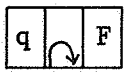
and similarly in the tables B_σ we find entries of the form

(iv) 

, with $q \in Q$, $F \subseteq Q$

(v) 

, with $q \in Q$ and τ denoting LOOP or HALT,

(vi) 

, with $q \in Q$, $F \subseteq Q$.

Entries in T_σ indicate, for any state q, that whenever the machine is currently in state q at the right-most symbol, in an "inside" move it will either cross Ω from R to L in any of the states from F (type (i)), get stuck somewhere on the R-part as indicated by τ (type (ii)), or after circulating in R (but <u>not</u> crossing Ω), return to the top in any of the states of F (type (iii)).

Entries in B_σ similarly indicate the behaviour after the machine crosses Ω from L to R in state q.

When a current $\sigma, T_\sigma, B_\sigma$-combination is known, the tables associated with a symbol σ' that is next to be assembled to the right are easily shown to be effectively computable from σ, T_σ, and B_σ, because these tables (and σ) permit one to completely predict the behavior on the R-part.

Lemma 5.3. For any $\sigma, T_\sigma, B_\sigma$-combination, and next symbol σ', the successor tables $T_{\sigma'}$ and $B_{\sigma'}$ are uniquely and effectively etermined.

Observing that there are only finitely many different tables, it follows from 5.3 that we can compound an extensive list L, showing for each σ, σ' and pair of tables linked to σ what the successor tables for σ' will be.

Lemma 5.4. L is finite and is effectively computable.

Finally, to accomodate an integral consideration of all possible initial assemblies, we will permit (and, in fact, require) assembly symbols to appear between crossing state couples on the input tape, modifying the machine's behavior still one more time in letting it find the "next" assembly symbol (of the initial assembly), at the precise moment that the symbol would occur under the scanner.

We will now describe the finite automaton behavior to which the assembler is reduced.

Input strings are words over $Q \cup \Sigma$.

Let homomorphisms h_Q, h_Σ on $(Q\cup\Sigma)^*$ be determined by

$$h_Q(s) = \begin{cases} s & \text{if } s \in Q \\ \Lambda & \text{if } s \in \Sigma \end{cases}$$

$$h_\Sigma(s) = \begin{cases} s & \text{if } s \in \Sigma \\ \Lambda & \text{if } s \in Q \end{cases}$$

Definition. Let

$\mathcal{B}_1$ = the set of all compatible $w \in (Q \cup \Sigma)^*$ such that $h_Q(w)$ is a crossing sequence which leads to a halt on R when starting on $h_\Sigma(w)$;

$\mathcal{B}_2$ = the set of all compatible $w \in (Q \cup \Sigma)^*$ such that $h_Q(w)$ is a crossing sequence which makes the assembler finally cross Ω to L, when started on $h_\Sigma(w)$.

$\mathcal{B}_3$ = the set of crossing sequences which lead to a halt on L.

$\mathcal{B}_4$ = the set of crossing sequences which make the assembler finally cross Ω from L to R.

Lemma 5.5. The sets $\mathcal{B}_i$ are regular.

Proof. We will show only that $\mathcal{B}_1$ is regular. (The construction for $\mathcal{B}_2$, $\mathcal{B}_3$ and $\mathcal{B}_4$ is similar and left to the reader).

The finite automaton for $\mathcal{B}_1$ will have states with 6 components:

(i) the current state of the assembler (by construction, only when it is at the top or bottom);

(ii) an assembly indicator (0 or 1), showing whether all symbols of the initial assembly already are passed or not;

(iii) a phase indicator (blank, top, cross, or return);

(iv) the top-most assembly symbol; this is, with the assembly indicator at 0, the currently last symbol adjoined;

(v) the corresponding top-departure table;

(vi) the corresponding bottom-departure table.

Thus states are of the form

[<state>, 0 or 1, phase, <symbol>, <table>, <table>].

In addition there are three special states -- r_0, HALT, and LOOP (where r_0 is a start state).

The transitions will correspond to the original assembler's behavior if the appropriate tables are checked when the option is given for making "inside" or "crossing" moves.

Description of transitions:

-- for r_0

In this state we have to bring out the first and maybe only symbol of the initial assembly.

Thus, for any $\sigma \in \Sigma$:

$$\phi(r_0,\sigma) = \{[q_0,0,\underline{top},\sigma,T_1,T_2],\ [q_0,1,\underline{top},\sigma,T_1,T_2]\}$$

where T_1 and T_2 are the top- and bottom-departure tables related to the assembly σ, which are computed directly from the assemblers program a similar fashion as in Lemma 5.3.

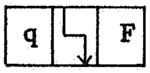
-- for states $[q,0,\underline{top},\sigma,T_1,T_2]$

Here the compounding of the initial assembly is not yet complete, so when simulating a move to the right, we have to read a next Σ-symbol necessarily:

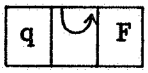
$$\phi([q,0,\underline{top},\sigma,T_1,T_2],\sigma') = \{[q',0,\underline{top},\sigma',T_1',T_2'],\ [q',1,\underline{top},\sigma',T_1',T_2'] \mid (q',1) \in \delta(q,\sigma) \text{ and } \sigma',\ T_1',\ T_2' \text{ is the successor of } \sigma,\ T_1,\ T_2 \text{ according to } L\}.$$

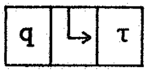
When not moving right, we have to consult the tables for what can happen, thus on λ input

$\phi([q,0,\underline{\text{top}},\sigma,T_1,T_2],\lambda) = \{[r,0,\underline{\text{cross}},\sigma,T_1,T_2] \mid$ all r such

that $\exists_F$  $\in T_1$ and $r \in F\}$

$\cup \{[r,0,\underline{\text{top}},\sigma,T_1,T_2] \mid$ all r such that

$\exists_F$  $\in T_1$ and $r \in F\}$

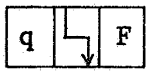
$\cup \{\tau \mid$  $\in T_1\}$

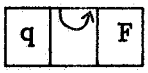
-- <u>for</u> <u>states</u> $[q,1,\underline{\text{top}},\sigma,T_1,T_2]$

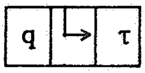
This time we know the initial assembly has been scanned completely, and when we are moving right, we will get to a blank for which we have to consider adding it on to the assembly. Also on λ input, the behavior of the machine when not moving right is recorded.

$\phi([q,1,\underline{\text{top}},\sigma,T_1,T_2],\lambda)$

$= \{[q',1,\underline{\text{blank}},\sigma,T_1,T_2] \mid (q',1) \in \delta(q,\sigma)\}$

$\cup \{[r,1,\underline{\text{cross}},\sigma,T_1,T_2] \mid \exists_F$  $\in T_1$ and $r \in F\}$

$\cup \{[r,1,\underline{\text{top}},\sigma,T_1,T_2] \mid \exists_F$  $\in T_1$ and $r \in F\}$

$\cup \{\tau \mid$  $\in T_1\}$

-- <u>for</u> <u>states</u> $[q,1,\underline{\text{blank}},\sigma,T_1,T_2]$

The instruction to either add the new cube on or not is recorded.

$\phi([q,1,\underline{\text{blank}},\sigma,T_1,T_2],\lambda)$

$= \{[q'1,\underline{\text{top}},\sigma,T_1,T_2] \mid (q',\lambda) \in \delta(q,\square)\}$

$\cup \{[q',1,\underline{\text{top}},\sigma',T_1',T_2'] \mid (q'\sigma') \in \delta(q,\square)$ with σ', T_1', T_2 the successor of σ, T_1, T_2 according to $L\}$

-- <u>for states</u> $[q,i,\underline{\text{cross}},\sigma,T_1,T_2]$

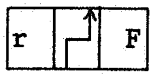
In this case we simulate crossing Ω in state q, which therefore has to be read from tape.

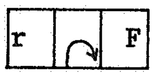
$\phi([q,i,\underline{\text{cross}},\sigma,T_1,T_2],q) = \{[q,i,\underline{\text{return}},\sigma,T_1,T_2]$

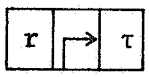
-- <u>for states</u> $[q,i,\underline{\text{return}},\sigma,T_1,T_2]$

Here we simulate returning on the R-part. The return state must be found on the input.

$\phi([q,i,\underline{\text{return}},\sigma,T_1,T_2],r)$

$= \{[q',i,\underline{\text{top}},\sigma,T_1,T_2] \mid$  $\in T_2$ and $q' \in F\}$

$\cup \{[q',i,\underline{\text{cross}},\sigma,T_1,T_2] \mid$  $\in T_2$ and $q' \in F\}$

$\cup \{\tau \mid$  $\in T_2\}$.

This completes the transition behavior (all other combinations are empty).

Clearly, when using HALT as a final state, the machine exactly accepts B_1.

Note that by taking states $[\ldots,1,\underline{\text{cross}},\ldots,\ldots,\ldots]$ as final, the same automaton accepts B_2.

Using the sets B_i we can now give the

Proof of Theorem 5.1. Observe that the machine may either cross Ω during a computation, or stay on the R-part. Hopeful assemblies in the first case form the set

$$h_\Sigma(h_Q^{-1}(h_Q(\mathcal{B}_1) \cap \mathcal{B}_4) \cap \mathcal{B}_1) \cap h_\Sigma(h_Q^{-1}(h_Q(\mathcal{B}_2) \cap \mathcal{B}_3) \cap \mathcal{B}_1) ,$$

which is regular. Hopeful assemblies in the second ase are simply $\mathcal{B}_1 \cap \Sigma^*$, which is also regular. Hence, the collection of all hopeful assemblies is regular.

§6. The Halting Problem and Some Further Applications

Theorem 5.2 gives a criterion for deciding the halting problem of linear assemblers. We can state a slightly stronger result:

Theorem 6.1. The halting problem for linear assemblers is equivalent to the membership problem for regular sets.

Proof. The reduction of the halting problem follows directly form 5.2.

To show the converse, let $\mathcal{A} = <Q,\Sigma,\delta,q_0,F>$ be a finite automaton accepting a given regular language R.

Define a linear assembler which scans an (input-) assembly as does $\mathcal{A}$, halts when it reaches the first blank on the right in a final state, but keeps moving back and forth on that block and the right-most non-blank if it arrives in a non-final state.

Thus the halting corresponds precisely to acceptance.

Since $\mathcal{A}$ can be chosen to be deterministic, so can the assembler.

The construction in §5 not only shows decidability of hopefulness, but by changing the set of final states one can also show that the collection of assemblies on which the machine might eventually diverge or loop is a regular set. Deleting it from the collection of hopeful assemblies, we get precisely the assemblies on which the machine always halts, whatever non-deterministic choices are made during the computation. Moreover,

Theorem 6.2. The collection of assemblies on which a non-deterministic linear assembler halts is a regular set which is effectively determined.

Let the transform of a set X be the collection of all final assemblies which can be produced by a linear assembler when started on initial assemblies from X.

We can characterize the transform of regular sets of initial assemblies quite precisely.

Theorem 6.3. The transform of a regular set by a nondeterministic linear is a linear context-free language.

Proof. Modifying the construction in §5, it is easy to let the automaton also read the symbols that are assembled onto the initial array (rather than having it done on λ input).

Defining $\mathcal{B}_1'$, $\mathcal{B}_2'$, $\mathcal{B}_3'$, $\mathcal{B}_4'$ much as before, but now inserting $\bar{\Sigma}$ symbols for blocks that are assembled during the (proper) computation, the theorem follows by the argument below.

Let $h_{\bar{\Sigma}}$ be defined by $h_{\bar{\Sigma}}(\alpha) = \begin{cases} \alpha & \text{if } \alpha \in \Sigma \\ \lambda & \text{if } \alpha \end{cases}$.

When the machine does not cross Ω to the L-part during the computation, the collection of transforms we get is $h_{\Sigma}^{-1}(h_{\Sigma}(B_1') \cap R) \cap B_1$ which is regular.

If the machine crosses Ω, the set of hopeful assemblies is regular, and cuts another piece R' from R.

We now have to determine the following sets:

$$C_1 = \{x^R y \mid x \in B_4',\ y \in B_1',\ \text{and } h_Q(x) = h_Q(y),\ h_{\Sigma}(y) \in R'\}$$

$$C_2 = \{x^R y \mid x \in B_3',\ y \in B_2',\ \text{and } h_Q(x) = h_Q(y),\ h_{\Sigma}(y) \in R'\}$$

where x denotes the crossing couples, together with the assembled symbols written left-to-right (since they are actually assembled right-to-left, we have to take reverses) on the L-part.

C_1 and C_2 are both easily seen to be linear context-free languages. Their union, together with the regular set, forms again a linear language.

One can show that on the other hand all linear context-free languages may be obtained as the transform of a regular set. Theorem 6.3 is of interest largely because the motions of a linear assembler may be very irregular, such compared to other proposals of machine-models for linear context-free languages (see e.g. Amar & Putzolu [1]).

Corollary 6.4. Let R be a regular set, x an assembly. It is decidable whether or not x is the transform of an element of R by means of a given non-deterministic linear assembler.

Proof. Reduce it to the membership problem for a linear context-free language.

References

1. V. Amar and G. Putzolu, Generalizations of regular events, in: E.R. Caianello (ed.), *Automata Theory*, Acad. Press, New York (1966) 1-5.
2. R.M. Baer, Computation by assembly, Tech. Rep. 12, Dept. of Computer Sc., Univ. of California, Berkeley (1973) (to appear in J. Comp. Syst. Sci.).
3. F.C. Hennie, Crossing sequences and off-line Turing-machine computations, Proc. 6th Annual Symposium on Switching Theory (1965) 168-172.
4. T.N. Hibbard, A generalization of context-free determinism, Inf. & Control 11 (1967) 196-238.
5. J.E. Hopcroft and J.D. Ullman, *Formal languages and their relation to automata*, Addison-Wesley, Reading, Mass. (1969).
6. M.O. Rabin, Real-time computation, Israel J. Math 1 (1964) 203-211.
7. V. Rajlich, Bounded crossing transducers, Inf. & Control 27 (1975) 329-335.
8. A. Salomaa, *Formal languages*, Acad. Press, New York (1973).
9. M. Schützenberger, A remark on finite transducers, Inf. & Control 4 (1961) 185-196.
10. B.A. Trakhtenbrot, Turing computations with logarithmic delay, Algebra i logica 3 (1964) #4, 33-48 (in Russian).