P.M.B. VITÁNYI

AN OPTIMAL SIMULATION OF COUNTER MACHINES

Preprint

An optimal simulation of counter machines[*)]

by

Paul M.B. Vitányi

ABSTRACT

Each multicounter machine can be real-time simulated by an oblivious one-head tape unit in logarithmic space. The solution involves a new positional representation for (multituples of) integers.

KEY WORDS & PHRASES: *multicounter machines, real-time simulation by one-tape Turing machines, number representation, counting, coding*

---

1. INTRODUCTION

The idea of counting, that is, adding or subtracting an unit from any given number, to obtain another one, is the substrate of arithmetic if not of all of mathematics. Thus, it is frequently necessary in computing to maintain many counts simultaneously, while the only information we want to extract at any time is the set of currently zero counts. The process of storing several integer counts, each count independently being incremented or decremented by an unit in each step, governed by the current input and the set of zero counts, is abstracted and formalized in the notion of a multicounter machine. Such machines have numerous connections with both theoretical issues and more of less practical applications. It is of considerable interest, for many questions, to implement multicounter machines as efficiently as possible. We shall show that counting is basically simple, in the computational complexity sense of the word, by demonstrating that each multicounter machine can be real-time simulated by an oblivious one-head tape unit using minimal storage space. Since the presented implementation is optimal in all commonly considered resources at once, the two decade old quest for better simulations of multicounter machines by Turing machines is finalized in one stroke.

Doing arithmetic presupposes number representations. Different representations are better suited to different arithmetical operations. All of arithmetic can be performed by multicounter machines. Because we shall simulate a multicounter machine by a one-head tape unit, we need to straightforwardly represent a multituple of integers as a linear string. No known representation for single integers allows the counter steps to be performed by an oblivious one-head tape unit without unbounded time loss in between

simulated steps. Neither does any known representation, for pairs of integers, allow the counter steps to be performed by a one-head tape unit, oblivious or not, without unbounded time loss in between simulated steps. To achieve our objective, we in effect have to develop a new representation for multituples of integers with the required properties.

Multicounter machines and Turing machines. For the present purpose, machines are viewed as *transducers,* that is, as abstract storage devices connected to input- and output terminals. Thus we consider a machine as hidden in a black box with input- and output terminals. Consequently, the presented simulation results concern the input-output behavior of black boxes and are independent of input-output conventions, or whether we want to recognize or to compute. The abstract storage structure embodied by a *k-counter machine* (k-CM) consists of a finite control connected to an input- and an output terminal, and k *counters* each capable of containing any integer. The states of the finite control are partitioned into *polling* and *autonomous* states. (Here we can assume without loss of generality that all states are polling states.) At the start of a computation the finite control of the k-CM is in a designated *initial state* and all counters are set to zero. A *step* in a k-CM computation is uniquely determined by the state of the finite control, by the symbol scanned at the input terminal if the state is a polling state and the set of counters which contain zero. The action at that step consists of independently altering the contents of each counter by $-1, 0$, or $+1$, changing the state of the finite control and producing a, possibly empty, output string. Thus the machine effects a transduction from input strings to output strings. If you will, the input and output may be thought of as written on input- and output tapes, on which the resident access pointers

(heads) are steered by the finite control. The steering commands issued can be viewed as part of the output. Above we closely followed the formulation in [2] where also a more precise definition can be found. For the more standard concept of *multitape Turing machines* consult [2,6]. Note that, for us, a one-tape Turing machine consists of a finite control connected to an input- and output terminal, and a single head storage tape. A one-head tape unit is a one-tape Turing machine.

Simulation. A machine A *simulates* a machine B in time $T(n)$ if, for all $n > 0$, the input/output behavior of B during the first n steps, the atomic inputs and outputs ordered according to their occurrences in time, is exactly mimicked by A within the first $T(n)$ steps. That is, if for every input sequence $i_1, i_2, \ldots$ read from the input terminal (i) the output sequences written to the output terminals by A and B are the same, and (ii) if $t_1 \leq t_2 \leq \ldots \leq t_k \leq t_{k+1} \ldots$ are the steps at which B reads or writes a symbol, then there are corresponding steps $t'_1 \leq t'_2 \leq \ldots \leq t'_k \leq t'_{k+1} \ldots$ at which A reads or writes the same symbols, and $t'_i \leq T(t_i)$ for all $i \geq 1$. For a *linear time* simulation it is required that $T(n) \in O(n)$; for a simulation with *constant delay* that $T(n+1) - T(n) \leq c$ for some fixed constant c and all n; for a *real-time* simulation that $T(n) = n$. It is well known, that a constant delay simulation can always be sped up to a real-time one, but not a linear time simulation in general. We use *simulation* in the above strong sense of *on-line simulation* [6] throughout.

Obliviousness. A Turing machine is *oblivious* if the movements of the storage tape heads are fixed functions of time independent of the particular inputs to the machine. Many problems seem inherently oblivious: the usual algorithms for computing the four main arithmetic operations, a table look-

up by sequential search, can easily be programmed obliviously without sacrificing worst case time efficiency. Other tasks like binary search or sorting are, it appears, nonoblivious in nature. For many purposes, there are excellent reasons why to restrict attention to oblivious computations [6,7]. Here we show yet another, more heuristic, motive for doing so. Viz., restriction of the considered model of computation to its oblivious version may shift the emphasis in the problem to be solved, from one difficulty to a completely different one, thus directing us to a solution. Whereas the difficulty in real-time simulating k-counter machines by k'-tape Turing machines, $k' < k$, stems from the fact that $k' < k$, the same problem with the simulating machine restricted to its oblivious version knows as difficulty but the obliviousness of the simulating device alone.

For suppose we can simulate some abstract storage device S in time $T(n)$ by an oblivious Turing machine M. Then we can also simulate a collection of k copies of S, say $S_1, S_2, \ldots, S_k$, interacting through a common finite control, by dividing all storage tapes of M into k tracks, each of which is a duplicate of the corresponding former tape, and by an appropriate modification of M's finite control. The same head movements of the resulting machine M' can now do the same job, on each of the k collections of tracks, as they formerly did on the collection of tapes of the original machine M. So the resources used by M' are, apart from sizes of finite control and alphabets, the same as those used by M. In particular this holds for time- and storage complexity. Therefore the following two statements are equivalent:

(i) We can simulate an abstract storage device S by an oblivious Turing machine M in time $T(n)$ and storage $S(n)$.

(ii) For each $k > 0$ we can simulate a collection of k copies of S, interacting through a common finite control, by an oblivious Turing machine

M' in time T(n) and storage S(n), where M' has the same tape/head constellation as M.

We are in particular interested in the following specialization of the above maxim.

Define the *quintessential counter* S as a 1-CM with input commands "add δ", $\delta \in \{-1,0,1\}$. At each step S reads an input command from the input terminal, modifies the stored count in the obvious way, and outputs either "count equal zero" or "count unequal zero" in concordance with the current state of affairs.

PROPOSITION 1. *If we can real-time simulate the quintessential counter S by an oblivious one-head tape unit then we can real-time simulate each multi-counter machine by an oblivious one-head tape unit (which for each multi-counter machine makes the same head movements as a function of time alone).*

Background. Counter machines are relatively old devices in computer science. Unrestricted 2-counter machines were shown to be as powerful as Turing machines in [5]. Subsequently the efficiency of implementations on Turing machines was investigated. On linear arrays, as formalized by Turing machines, the use of a tally representation for each count either requires a separate access pointer (storage tape head) per count or unbounded update time in between simulated steps. Curiously, even with the use of a separate pointer for each count, binary representations also require unbounded update time, although minimal storage space. This sorry state of affairs was improved in [1,2] which both presented linear array simulations using minimal space, while [1] eliminated the unbounded update time at the cost of retaining all access pointers, and [2] eliminated all access pointers but one at the cost of retaining unbounded update time. Thus, [2] exhibited the classic linear time/logarithmic space simulation of multicounter machines by one-tape

Turing machines. Efforts to reduce this simulation to a real-time one using a fixed number of storage heads failed, but did produce some weaker problems. For the Origin Crossing Problem, where the task is to recognize the set of sequences of unit basis vectors in k-space, $k \geq 1$, which leave from and end in the origin, an ingeneous solution by a real-time one-tape Turing machine was constructed in [1]. The result implies that each k-counter machine can be real-time simulated by a k-tape Turing machine in logarithmic space, $k \geq 1$. Next in difficulty comes the Axis Crossing Problem, where the task is to recognize the set of sequences of unit basis vectors in k-space, which leave from the origin and end in one of the (k-1)-dimensional hyperplanes with one zero coordinate, $k > 1$. For no $k > 1$, a real-time solution on but a (k-1)-tape Turing machine was found, for the k-dimensional Axis Crossing Problem, after its proposal in [2]. No headway was made with the roughly stated issue: do less than k accesspointer, in a linear array, suffice to real-time simulate k counters, or k counters handicapped like in the Axis Crossing Problem.

In [8] we made the linear time/logarithmic space one-tape solution of [2] oblivious, retaining the same resource bounds. This is a matter of some significance, since by its nature an oblivious Turing machine is usually far slower than a nonoblivious one. For example, each oblivious multitape Turing machine needs n log n steps to simulate n steps of a single pushdown store, although an oblivious 2-tape Turing machine can achieve this bound [6]. For oblivious one-tape Turing machines the lower bound on this simulation time increases, perhaps, up to n^2. Due to the compact way the counts can be stored, the situation for counter machines was somewhat better. In [7, Corollary 2] it was shown how to simulate each multitape Turing machine, using at most S(n) storage in n steps, by an oblivious 2-tape Turing machine

in n log S(n) steps and S(n) storage. So the previously best simulation of
multicounter machines, by combining [2] and [7], yielded a n log log n time
/ log n storage simulation by an oblivious 2-tape Turing machine. Since the
thrust of [8] was to achieve fast low-cost combinational logic networks im-
plementing multicounter machines, as an expedient intermediate next result
a real-time simulation by, basically, a linear iterative array with a
restricted amount of oblivious local rewriting was proposed. Although not
very elegant, this intermediate model served its purpose in yielding an op-
timal implementation of multicounter machines on combinational logic net-
works and, perhaps more important, the ideas embodied in the method suggest
the approach to the final simulator presented here.

Outline of the paper. The objective is to construct an oblivious one-head
tape unit capable of simulating any multicounter machine in real-time. In
section 2 a stylized version of such a simulator is exhibited and shown to
work. This version is one of many which are possible on the basic underlying
principles, and is choosen because it is at once ammenable to short rigorous
proofs of validity and achieves, it seems, the utmost frugality of machinery.
To a large extend this gain is gotten at the cost of loss of intuition as to
how and why it does what it is supposed to do. To counterbalance this exposi-
tory defect, we insert some informal comments. The reader may also follows
the genesis of the result by consulting [8] and the earlier version in the
STOC Proceedings. In section 3 we enlarge on the optimality of the result,
its connection with number representations, and on additional fruit borne.

2. THE SIMULATION

After some vain attempts to real-time simulate multicounter ma-
chines by Turing machines with a fixed number of tapes, one gets the feeling

that, anyway, a real-time simulation by an *oblivious* one-head tape unit is out of the question. In the event, intuition is wrong; but let us informally consider the matter in some more detail. It quickly becomes apparent that updating a count, in real-time on an oblivious machine, requires a redundancy in notation which seems to make a simultaneous real-time check for zero impossible. To achieve the latter, we allow only encodings of integers such that an integer is zero iff the scanned position of the encoding (the "first" position, so to speak) shows this uniquely. Since the head motion is supposed to be oblivious we must, roughly speaking, update each "initial" $\Omega(\log i)$ length segment (situated around the head) of the encoded integer within each interval of i steps, for all $i \geq 1$. While moving the head to update longer segments of code in an oblivious manner, we may have actually stored small counts which may reach zero during this motion. So the machine has to simultaneously shift and update smaller segments of code, while updating larger segments of code, and so on recursively down to the smallest segments. Such considerations force compact encodings, and, apart from giving us some feel for what behavior is necessarily involved in a simulation as desired, they show that the integer representation used must be positional in nature.

Outline of the simulation. The simulation splits naturally into two parts. First we introduce a redundant binary representation for the integers, and formulate certain minimal requirements for real-time maintaining the representation of the stored integer under the counter operations. These requirements consist in a fixed strategy, of accessing constant length segments of this representation, for all input streams. Second, we construct on oblivious one-head tape unit capable of implementing these requirements in real-time.

The current count of the quintessential counter, as figuring in Proposition 1, is stored on the single tape in a (garbled form of) redundant

binary representation, with marked most signigicant nonzero digit and lead-

ing distinguished blank symbols. As a consequence of preserving some inva-

riants, the stored count equals zero iff the "first" position of the repre-

sentation is a blank. Since this first position shall reside in the finite

control of the simulator, that situation is instantly recognized.

Hence the problem is solved, if we can real-time update the representation

of the current count while preserving the invariants. In the choosen repre-

sentation it suffices to update each segment of the 2i-th through (2i+3)-th

position of the representation at least once within each interval of 3^i con-

secutive steps, for all i ≥ 0, while also processing the current input com-

mands, by an update of the first two positions, in each step. Intuitively

speaking, the timing allows us to propagate carries and borrows (negative

carries) fast anough. Although there is a considerable freedom about how to

implement the required datamovement on an oblivious one-head tape unit, we

choose for frugality in attendant machinery and minimal bit compression

(that is, a small storage tape alphabet). Therefore, we divide the represen-

tation into blocks of two digits each, and store the first three blocks in

the finite control. Each digit of the representation residing on the

tape is tagged with an opening or a closing bracket, viz. the first digit

of a block with an opening bracket and the second one with a closing

bracket. To access each segment of the 2i-th through (2i+3)-th digits of the

representation at least once in every interval of 3^i steps, we develop a

method of recursively transporting the digits of block j, from one side of

the combination of the first i blocks to the other side, back and forth,

for all i,j, 1 ≤ i < j. This transport, which entails moving the total com-

bination of the first i blocks, in its turn supplies the necessary motion

for the combination of the first i + 1 blocks, while it also allows the

single head to access blocks i + 2 and i + 3 within the timing constraints.

The single head, without being able to determine the positional index of the scanned digits (since there will be all in all but four tags, viz. two types of opening brackets and two types of closing brackets), preserves a topology which allows it to single out and update due segments. The net effect will be that, for all i simultaneously, the combination of the first i blocks acts like a very fat head, moving slower the greater i is, but fast anough to do the same job to blocks $i + j$ as the head itself does to blocks j, for all i,j $\geq$ 1.

On notation. To be able to express and prove the subsequent constructions, it is convenient to introduce some notation first. The objects operated upon are *linear arrays* or *strings* of symbols from a finite alphabet. Arrays can be finite or one-way infinite. In a one-way infinite array A[0:∞], A[0] is the first element and A[i] is the (i+1)-th element, i $\geq$ 0. A[i:j] denotes the (j-i+1)-length subarray consisting of the (i+1)-th through (j+1)-th elements, 0 $\leq$ i $\leq$ j. The concatenation A[i:j] A[j+1:k] equals A[i:k], 0 $\leq$ i $\leq$ j $<$ k, and we identify A[i:i] with A[i], i $\geq$ 0. Similar for finite arrays. Arrays are operated upon by functions from arrays to arrays. Since these functions shall be partial we introduce the *undefined* array ϕ. By definition, for any array A, $\phi A = A\phi = \phi$. The undefined array should be distinguished from the *empty* array ε for which by definition, for any array A, $\varepsilon A = A\varepsilon = A$. Mappings from arrays to arrays are defined in terms of length preserving functions from finite arrays to finite arrays. If a function P maps an array S to an array S', with S, S' finite and of equal length, then we write P : S $\mapsto$ S'. By definition P : $\phi \mapsto \phi$ for all functions P. Functions induce relations amongst one-way infinite arrays in essentially two ways. In the first type of relation $\overset{Q}{\Rightarrow}$ the argument of Q determines in-

tegers i,j, i ≤ j, and for all arrays A[0:∞], A'[0:∞] if P : A[i:j] ↦ A'[i:j],

for a function P associated with Q, A'[0:i-1] = A[0:i-1] and A'[j+1:∞] =

A[j+1:∞], then A $\overset{Q}{\mapsto}$ A'. In this case, clearly $\overset{Q}{\mapsto}$ is a *function* from one-way

infinite arrays to one-way infinite arrays. In the second type of relation

$\overset{P}{\mapsto}$, P is a function, and A $\overset{P}{\mapsto}$ A' if A = $S_1 S S_2$, A' = $S_1 S' S_2$ and P : S ↦ S'.

It will be shown that all such relations of the second type we consider

are also *functions* when restricted to a set of *well formed* arrays. In

both cases, if for some $\overset{Q}{\mapsto}$ and some array A, there is no A' ≠ φ such that

A $\overset{Q}{\mapsto}$ A', then A $\overset{Q}{\mapsto}$ φ. Considering the relation $\overset{Q}{\mapsto}$ amongst arrays as *rewriting*,

the rewriting shall thus be proved to be always *monogenic*, that is, if

A $\overset{Q}{\mapsto}$ A' and A $\overset{Q}{\mapsto}$ A" then A" = A'. We *compose* functions $P_1, P_2, \ldots, P_n$ to a

function P, or *decompose* or *expand* a function P into a sequence of consti-

tuent functions $P_1, P_2, \ldots, P_n$ as follows. If for some $P, P_1, \ldots, P_n$ and all

arrays A there exist arrays $A_1, A_2, \ldots, A_n$ such that A $\overset{P}{\mapsto}$ A_n and

A $\overset{P_1}{\mapsto}$ A_1 $\overset{P_2}{\mapsto}$ A_2 ... $\overset{P_n}{\mapsto}$ A_n then P = $P_1; P_2; \ldots; P_n$. The function composition opera-

tor ";" denotes sequential rewriting from left to right. Whenever necessary,

we denote the value of an array A at time t, t ≥ 0, by A^t and A^0 is the

initial array. We dispense with the superscript if t is understood or when

we view A as a variable.

Main objective. We concentrate on real-time simulating the quintessential

counter of Proposition 1 by an oblivious one-head tape unit.

2.1. An integer representation

Consider a positional base 2 notation for representing the integers,

which may be called *redundant symmetric binary*, using the digits -2,-1,0,1,2.

So the integer c represented by $c_0 c_1 c_2 \ldots c_m$, $c_i \in \{-2,-1,0,1,2\}$, equals

$\sum_{i=0}^{m} c_i 2^i$. Such a representation is *binary* because of the weight of digits

in distinct positions, *symmetric* because of the used digits, and *redundant* since each integer has infinitely many representations, even without leading nonsignificant zero's. To represent the stored integer count on a, potentially infinite, linear tape we essentially use a restricted version of this representation, with a marked most significant nonzero digit and distinguished leading nonsignificant zero's. Let $\Delta = \{-2,-1,0,1,2\}$ and $\bar{\Delta} = \{\bar{-2},\bar{-1},\bar{0},\bar{1},\bar{2}\}$. The barred digits have the same value as their nonbarred counterparts, $\bar{\Delta} - \{\bar{0}\}$ is reserved for the most significant nonzero digit, and "$\bar{0}$", called *blank*, is reserved for the nonsignificant zero's. Let $\Sigma = \Delta \cup \bar{\Delta}$ and let *code*: $\mathbb{Z} \to 2^{\Sigma^{\infty}}$ be a function of the integers into the power-set of Σ^{∞}, where Σ^{∞} is the set of one-way infinite strings over Σ. The function *code* satisfies restrictions (A) - (D) below, for all $c_0 c_1 \dots c_i \dots$
$\dots \in code(c)$, $c \in \mathbb{Z}$.

Separation of a finite significant initial segment and nonsignificant zero's:

(A) $\qquad \exists i \geq 0 \ [c_i = \bar{0}] \ \& \ \forall i > 0 \ [(c_i = \bar{0} \Rightarrow (c_{i-1} \in \bar{\Delta} \ \& \ c_{i+1} = \bar{0})) \ \&$
$$\& \ (c_i \in \Sigma - \{\bar{0}\} \Rightarrow c_{i-1} \in \Delta)].$$

Correct representation:

(B) $\qquad \displaystyle\sum_{i=0}^{\infty} c_i 2^i = c_0$

To identify representations of 0 by just a small initial segment:

(C) $\qquad \forall i \geq 0 \ [(c_{i+1} > 0 \Rightarrow c_i \geq 0) \ \& \ (c_{i+1} < 0 \Rightarrow c_i \leq 0)].$

Under (A) - (C), $(-2)^i 0 \ \bar{1} \ \bar{0}^{\infty}$ represents the integer 2 for all $i \geq 0$. To prevent racing of the most significant nonzero digit to the first position, in just a few steps of the desired single head real-time simulator:

(D) $\qquad \forall i \geq 0 \ [i \text{ is odd} \Rightarrow |c_i| < 2].$

Now if $c_0 c_1 \dots c_{m-1} c_m \dots \in code (c)$, with $c_i \in \Delta$ for $0 \leq i \leq m-1$, and $c_i \in \bar{\Delta}$

for $m \leq i$, then for $m = 0$ we have $0 \leq |c| \leq 2$, for $m = 1$ we have $2 \leq |c| \leq 4$ and in general for $m \geq 2$:

$$2^m - r \leq |c| \leq 2^{m+1} + r'$$

with

$$r = 2 \cdot \sum_{\substack{i=0, \, i \text{ even}}}^{m-2} 2^i + \sum_{\substack{i=1, \, i \text{ odd}}}^{m-2} 2^i$$

$$r' = 2 \cdot \sum_{\substack{i=0, \, i \text{ even}}}^{m-1} 2^i + \sum_{\substack{i=1, \, i \text{ odd}}}^{m-1} 2^i$$

which yields

$$m-3 < \log_2 |c| < m+2.$$

Thus the length of the initial significant segment of the representation of $c \in \mathbb{Z}$ follows by and large the length of the usual binary representation of $|c|$. We are particularly interested in representations for zero. Note that the following proposition holds for code functions satisfying only (A) - (C).

PROPOSITION 2. *Let* $c_0 c_1 \ldots c_m c_{m+1} \ldots \in code(c)$ *with* $c \in \mathbb{Z}$. *Then* $c = 0$ *iff* $c_i = \bar{0}$ *for all* $i \geq 0$ *iff* $c_0 = \bar{0}$.

PROOF. By (A) $c_0 = \bar{0}$ iff $c_i = \bar{0}$ for all $i \geq 0$. So we only have to prove $c = 0$ iff $c_0 = \bar{0}$. Assume $c_i = \bar{0}$ for all $i \geq 0$. By (B) $c = 0$. Assume $c_i \neq \bar{0}$ for some $i \geq 0$. Then by (A) there exists a least $m \geq i$ such that $c_j = \bar{0}$ for all $j > m$, and $|c_m| \neq 0$. For $m = 0$, $|c| \geq 1$ and for $m = 1$ we have $|c| \geq 2$. For $m \geq 2$:

$$|c| = \left| \sum_{i=0}^{m} c_i 2^i \right| \qquad \text{(by (B))}$$

$$\geq \left| 2^m - \left| \sum_{i=0}^{m-1} c_i 2^i \right| \right| \qquad \text{(triangular inequality)}$$

$$\geq 2^m - 2 \sum_{i=0}^{m-2} 2^i \qquad \text{(by (C))}$$

$$= 2. \quad \square$$

2.2. Maintenance of the count.

Let S be the simulated quintessential counter and let $\delta_1, \delta_2, \ldots, \delta_t, \ldots$, $\delta_i \in \{-1,0,1\}$, be any fixed sequence of unit additions/substractions. So at time $t \geq 0$, S contains the integer $\sum_{i=1}^{t} \delta_i$. We maintain the count in an array $C[0:\infty]$ such that the value of the array at time $t \geq 0$ is $C^t[0:\infty] \in$ code$(\sum_{i=1}^{t} \delta_i)$, for any such input stream. The initial array $C^0[0:\infty]$ at time $t = 0$ is defined by $C^0[i] = \bar{0}$ for all $i \geq 0$, and therefore $C^0[0:\infty] \in$ code(0). In the t-th simulated step, $t \geq 1$, the current value C^{t-1} of the array is mapped to the next value C^t by a function COUNT (t, δ_t). The mapping COUNT is defined in terms of a composition of mappings, with the aid of an auxiliary function $I : \mathbb{N} \to 2^{\mathbb{N}}$, called the *parameter selection* function, which has as values sets of bounded cardinality (cardinality four suffices).

DEFINITION. For $t \geq 1$, let $I(t) = \{i_\ell, i_{\ell-1}, \ldots, i_1\}$ with $i_\ell > i_{\ell-1} \cdots > i_1$, and let $\delta \in \{-1,0,1\}$. COUNT (t, δ) is defined as a composition of mappings:

$$\text{COUNT}(t,\delta) \overset{\text{def}}{=} \text{UPDATE}(i_\ell); \ \text{UPDATE}(i_{\ell-1}); \ldots; \ \text{UPDATE}(i_1); \ \text{INPUT}(\delta).$$

Hence

$$C \xrightarrow{\text{COUNT}(t,\delta)} C', \text{ with } C' \neq \phi, \text{ if there exist } C_\ell, C_{\ell-1}, \ldots, C_1 \neq \phi$$

such that

$$C \xrightarrow{\text{UPDATE}(i_\ell)} C_\ell \xrightarrow{\text{UPDATE}(i_{\ell-1})} C_{\ell-1} \cdots \xrightarrow{\text{UPDATE}(i_1)} C_1 \xrightarrow{\text{INPUT}(\delta)} C'.$$

In all other cases $C \xrightarrow{\text{COUNT}(t,\delta)} \phi$.

DEFINITION. Let $i \in \mathbb{N}$ and let $C[0:\infty]$ be a one-way infinite array, $C \neq \phi$.

$$C[0:\infty] \xmapsto{\quad UPDATE(i) \quad} C'[0:\infty],$$

$C'[0:\infty] \neq \phi$, if UPDATE: $C[2i:2i+3] \mapsto C'[2i:2i+3] \neq \phi$, $C'[0:2i-1] = C[0:2i-1]$

and $C'[2i+4:\infty] = C[2i+4:\infty]$, with the function UPDATE: $\Sigma^4 \to \Sigma^4$ defined below.

For convenience we first define UPDATE: $\Delta^4 \to \Delta^4$ and then extend the mapping

to Σ^4.

UPDATE:			
2 0 x y	↦	0 1 x y	for xy $\in$ {00,01,0-1,10,11,20,21}
2 0 x y	↦	0-1 x+1 y	for xy $\in$ {-10,-20,-1-1,-2-1}
2 1 x y	↦	0 0 x+1 y	for xy $\in$ {00,01,10,11}
2 1 0-1	↦	0 0 -10	
2 1 2 y	↦	ϕ	for y $\in$ {0,1}
-2 0 x y	↦	0-1 x y	for xy $\in$ {00,0-1,01,-10,-1-1,-20,-2-1}
-2 0 x y	↦	0 1 x-1 y	for xy $\in$ {10,20,11,21}
-2-1 x y	↦	0 0 x-1 y	for xy $\in$ {00,0-1,-10,-1-1}
-2-1 0 1	↦	0 0 1 0	
-2-1-2 y	↦	ϕ	for y $\in$ {0,-1}
v w x y	↦	v w x y	for v $\notin$ {-2,2}
v w x y	↦	ϕ	for vwxy not in the above list.

Extension of UPDATE to mappings from Σ^4 into Σ^4 : if vwxy $\notin \Delta^4$ then

$$\text{UPDATE: vwxy} \mapsto v'w'x'y'$$

for all vwxy, v'w'x'y' $\in \Delta^*(\bar{\Delta} - \{\bar{0}\})\{\bar{0}\}^* \cup \{\overline{0000}\}$ such that the unbarred

version of the mapping is in the previous list, and UPDATE: vwxy $\mapsto \phi$ in

all other cases. (Recall that if V is a finite alphabet, then V^* is the set

of all finite strings over V including the empty string ε.)

<u>DEFINITION.</u> Let $\delta \in \{-1,0,1\}$ and let $C[0:\infty]$ be a one-way infinite array, $C \neq \phi$.

$$C[0:\infty] \overset{\text{INPUT}(\delta)}{\Longmapsto} C'[0:\infty],$$

$C'[0:\infty] \neq \phi$, if $\text{INPUT}_\delta : C[0:1] \mapsto C'[0:1] \neq \phi$ and $C'[2:\infty] = C[2:\infty]$ with $\text{INPUT}_\delta : \Sigma^2 \to \Sigma^2$ defined below. For convenience we first define $\text{INPUT}_\delta : \Delta^2 \to \Delta^2$ and then extend the mapping to Σ^2.

INPUT_{-1} : $\quad$ x y $\mapsto$ x-1̄ y $\quad$ for $\quad$ xy $\in \{00, 0\text{-}1, -10, -1\text{-}1, 10, 11\}$

$\qquad\qquad\qquad$ 0 1 $\mapsto$ 1 0

$\qquad\qquad\qquad$ x y $\mapsto$ ϕ $\quad$ for $\quad$ xy $\in \{-20, -2\text{-}1, 20, 21\}$

INPUT_0 : $\quad$ x y $\mapsto$ x y $\quad$ for $\quad$ xy $\in \{00, 0\text{-}1, -10, -1\text{-}1, 10, 01, 11\}$

$\qquad\qquad\qquad$ x y $\mapsto$ ϕ $\quad$ for $\quad$ xy $\in \{-20, -2\text{-}1, 20, 21\}$

INPUT_1 : $\quad$ x y $\mapsto$ x+1̄ y $\quad$ for $\quad$ xy $\in \{00, 01, 10, 11, -10, -1\text{-}1\}$

$\qquad\qquad\qquad$ 0-1 $\mapsto$ -10

$\qquad\qquad\qquad$ x y $\mapsto$ ϕ $\quad$ for $\quad$ xy $\in \{-20, -2\text{-}1, 20, 21\}$.

Extension of INPUT_δ to mappings from Σ^2 into Σ^2: if xy $\notin \Delta^2$ then

$$\text{INPUT}_\delta : xy \mapsto x'y'$$

for all xy,x'y' $\in \Delta^*(\bar{\Delta} - \{\bar{0}\})\{\bar{0}\}^* \cup \{\overline{00}\}$ such that the unbarred version of the mapping is in the previous list and INPUT_δ: xy $\mapsto \phi$ in all other cases.

If for some array $C[0:\infty]$ and P = UPDATE(i), i $\geq$ 0, we have

UPDATE: $C[2i:2i+3] \mapsto \phi$ then $C \overset{P}{\mapsto} \phi$, by definition. If for some array $C[0:\infty]$

and $P = \text{INPUT}(\delta)$, $\delta \in \{-1,0,1\}$, we have $\text{INPUT}_\delta: C[0:1] \mapsto \phi$ then $C \overset{P}{\mapsto} \phi$, by definition. For all $P \in \{\text{INPUT}(\delta), \text{UPDATE}(i) \mid \delta \in \{-1,0,1\}, i \geq 0\}$ we have by definition $\phi \overset{P}{\mapsto} \phi$, and thus $\overset{P}{\mapsto}$ is a *mapping* from $\Sigma^\infty \cup \{\phi\}$ into $\Sigma^\infty \cup \{\phi\}$ and not just a relation. Basically, $\text{INPUT}(\delta)$ adds the current input to the currently represented integer and $\text{UPDATE}(i)$ propagates carries and borrows in a segment of the representation, both preserving representations from the code function.

For each input sequence $D = \delta_1, \delta_2, \ldots, \delta_t, \ldots$, with $\delta_t \in \{-1,0,1\}$ for $t \geq 1$, the sequence of mappings

$$\text{COUNT}(I,D) \overset{\text{def}}{=} \text{COUNT}(1,\delta_1); \text{COUNT}(2,\delta_2); \ldots; \text{COUNT}(t,\delta_t); \ldots$$

defines a sequence of (a priori possibly undefined) arrays $C^0, C^1, \ldots, C^t, \ldots$ such that C^0 is the all-blank initial array $C^0[0:\infty] = \bar{0}^\infty$, and for all $t \geq 1$:

$$C^{t-1} \xRightarrow{\text{COUNT}(t,\delta_t)} C^t.$$

Decomposing $\text{COUNT}(t,\delta_t)$ into its constituent functions for all $t \geq 1$, with $I(t) = \{i_{t,\ell(t)}, i_{t,\ell(t)-1}, \ldots, i_{t,1}\}$ and $i_{t,\ell(t)} > i_{t,\ell(t)-1} > \cdots > i_{t,1}$, we obtain for each inputsequence $D = \delta_1, \delta_2, \ldots, \delta_t, \ldots$ the sequence of basic mappings

$$\text{COUNT}(I,D) = \text{UPDATE}(i_{1,\ell(1)}); \text{UPDATE}(i_{1,\ell(1)-1}); \ldots$$

$$\ldots; \text{UPDATE}(i_{1,1}); \text{INPUT}(\delta_1); \text{UPDATE}(i_{2,\ell(2)}); \ldots ,$$

In this sequence, the subsequence of mappings

$$\text{COUNT}(t,\delta_t) = \text{UPDATE}(i_{t,\ell(t)}); \text{UPDATE}(i_{t,\ell(t)-1}); \ldots$$

$$\ldots; \text{UPDATE}(i_{t,1}); \text{INPUT}(\delta_t)$$

is said to constitute the t-th *step* of the maintenance of array C. Starting

from C^{t-1} the sequence of intermediate arrays defined by the t-th step is

$$C^{t-1} (= C_{t-1,0}), \; C_{t,\ell(t)}, \; C_{t,\ell(t)-1}, \; \ldots, \; C_{t,1}, \; C_{t,0} \; (= C^t)$$

defined by

$$C_{t-1,0} \xmapsto{\text{UPDATE}(i_{t,\ell(t)})} C_{t,\ell(t)} \; \ldots \; \xmapsto{\text{INPUT}(\delta_t)} C_{t,0} = C^t.$$

Note that in the decomposition of COUNT(I,D) in the basic mappings UPDATE(.)

and INPUT(.) the parameter t does not occur explicitly; the sequence of ba-

sic mappings is defined totally by the sequence of successive values of I

and the sequence of inputs. This is important in the next sections. In this

section we show in Lemma 1 that, for any inputsequence $D = \delta_1, \delta_2, \ldots$,

$$C^0, C_{1,\ell(1)}, \ldots, C_{1,1} \in \text{code}(0) \cup \{\phi\}$$

and for all $t \geq 1$

$$C_{t-1,0} (= C^{t-1}), \; C_{t,\ell(t)}, \; \ldots, \; C_{t,1} \in \text{code} \left(\Sigma_{i=1}^{t-1} \delta_i \right) \cup \{\phi\}.$$

In Proposition 3 it is demonstrated that for certain choices of the parame-

ter selection function I we have that $C_{t,j} \neq \phi$ for all $t \geq 1$ and all j,

$\ell(t) \geq j \geq 0$, whence $C^t \in \text{code}(\Sigma_{i=1}^t \delta_i)$ for all $t \geq 0$.

LEMMA 1. *Let array C $\in$ code(c) for some integer c. If, for some i $\geq$ 0,*
$$C \xmapsto{\text{UPDATE}(i)} C', \; C' \neq \phi, \; \text{then } C' \in \text{code}(c). \; \text{If, for some } \delta \in \{-1,0,1\},$$
$C \xmapsto{\text{INPUT}(\delta)} C', \; C' \neq \phi, \; then \; C' \in code(c+\delta).$

PROOF. Let $C \in \text{code}(c)$ for some integer c and $C \xmapsto{\text{UPDATE}(i)} C', \; C' \neq \phi$, for

some $i \geq 0$. If $|C[2i]| \neq 2$ then $C' = C$ and there is nothing to prove. So let

$|C[2i]| = 2$. Then, for $j < 2i$ and $j > 2i+3$, $C'[j] = C[j]$. Since also

$\Sigma_{j=0}^{3} C'[2i+j]2^{2i+j} = \Sigma_{j=0}^{3} C[2i+j]2^{2i+j}$, we have $\Sigma_{i=0}^{\infty} C'[i]2^{i} = \Sigma_{i=0}^{\infty} C[i]2^{i} = c$.

It is easy to check from the definition of UPDATE(i), that if (A), (C) and

(D) hold for C, $C \xmapsto{\text{UPDATE(i)}} C'$ and $C' \neq \phi$, then (A), (C) and (D) also hold

for C'. Hence $C' \in$ code(c). Let $C \in$ code(c) for some integer c and

$C \xmapsto{\text{INPUT}(\delta)} C'$, $C' \neq \phi$, for some $\delta \in \{-1,0,1\}$. Since $C' \neq \phi$ we have

$|C[0]| < 2$. For all $j > 1$, $C'[j] = C[j]$. Because also $C'[0] + 2C'[1] =$

$= C[0] + 2C[1] + \delta$ we have $\Sigma_{i=0}^{\infty} C'[i]2^{i} = c + \delta$. It is easy to check from

the definition of INPUT(δ) that if (A), (C) and (D) hold for C,

$C \xmapsto{\text{INPUT}(\delta)} C'$, and $C' \neq \phi$, then (A), (C) and (D) also hold for C'. Hence

$C' \in$ code(c+δ). $\square$ $\square$

PROPOSITION 3. *Let* T: $\mathbb{N} \to \mathbb{N}$ *be any function such that* $T(i) \leq 3^{i}$ *for all*

$i \geq 0$. *Let the parameter selection function* I: $\mathbb{N} \to 2^{\mathbb{N}}$, *associated with the*

mapping COUNT, *be such that for all indices* $i \geq 0$ *and steps* $t \geq 1$ *there*

exists a t', $t \leq t' < t + T(i)$ *and* $i \in I(t')$. *Then for each input sequence*

$\delta_{1}, \delta_{2}, \ldots, \delta_{t}, \ldots$, $\delta_{t} \in \{-1,0,1\}$, $t \geq 1$, *there exists a sequence of one-way*

infinite arrays $c^{0}, c^{1}, \ldots, c^{t}, \ldots$, *with* c^{0} *the all blank initial array and*

c^{t-1} *is mapped to* c^{t} *by* COUNT(t,δ_{t}) *for all* $t \geq 1$, *such that*

$c^{t} \in$ code$(\Sigma_{i=1}^{t}\delta_{i})$ *for all* $t \geq 0$.

PROOF. Roughly speaking the Proposition states that if, starting from the

all blank initial array c^{0}, UPDATE(i) is executed at least once in every in-

terval of 3^{i} steps, for all $i \geq 0$, and INPUT(δ) is executed each step, with

$\delta \in \{-1,0,1\}$ the currently polled input, then the array at time t represents

the stored integer at time t according to the code function. By Lemma 1 and the

definition of COUNT this is the case if, under the timing assumption on the para-

meter selection function I, each time UPDATE(i) and INPUT(δ) map an array satisfying

(A), (C) and (D), the result is not the undefined array ϕ. The only way

UPDATE(i) can map an array $C[0:\infty]$, satisfying (A), (C) and (D) to ϕ is for

$C[2i:2i+3] \in \{212, 21\bar{2}, -2-1-2, -2-1-\bar{2}\}$. Similarly, the only way INPUT(δ) can

map on array $C[0:\infty]$ satisfying (A), (C) and (D) to ϕ is for

$C[0] \in \{2, \bar{2}, -2, -\bar{2}\}$. Hence we have to prove that, under the assumptions on I,

and starting from the all blank initial array c^0, these undesirable subar-

rays do not occur at the crucial moments. Induction is on the number of steps t.

Base case: the first step. Since c^0 is all-blank, for all $i \geq 0$ we have

$c^0[2i:2i+3] = \bar{0}\,\bar{0}\,\bar{0}\,\bar{0}$. Hence $c^0 \xrightarrow{\text{COUNT}(1,\delta_1)} c^1$ with $c^1[0] = \bar{\delta}_1$ and $c^1[i] = \bar{0}$

for all $i \geq 1$. That is, $c^1 \in \text{code}(\delta_1)$.

Induction: $t \geq 1$. Assume, by way of contradiction, that for the input sequence

$\delta_1, \delta_2, \ldots, \delta_t$ ($\delta_j \in \{-1,0,1\}$, $1 \leq j \leq t$) we have for all j, $1 \leq j \leq t$:

$$c^{j-1} \xrightarrow{\text{COUNT}(j,\delta_j)} c^j, \quad c^j \neq \phi, \qquad \text{(induction assumption)},$$

and

$$c^t \xrightarrow{\text{COUNT}(t+1,\delta)} \phi \qquad \text{(contradictory assumption)},$$

for some $\delta \in \{-1,0,1\}$. For all j, $1 \leq j \leq t$, by Lemma 1, $c^j \in \text{code}(\Sigma_{i=1}^j \delta_i)$.

Let $I(t+1) = \{i_\ell, i_{\ell-1}, \ldots, i_1\}$ and $i_\ell > i_{\ell-1} > \ldots > i_1$. Decomposing

COUNT($t+1,\delta$) into its constituent mappings we have

$$c^t = C_{\ell+1} \xrightarrow{\text{UPDATE}(i_\ell)} C_\ell \xrightarrow{\text{UPDATE}(i_{\ell-1})} C_{\ell-1} \ldots$$

$$\ldots \xrightarrow{\text{UPDATE}(i_1)} C_1 \xrightarrow{\text{INPUT}(\delta)} C_0 = \phi,$$

for some intermediate, possibly undefined, arrays $C_\ell, C_{\ell-1}, \ldots, C_0$. By the

contradictory assumption there must be a first undefined array in this

sequence, say $C_{j-1} = \phi$ and $C_j \neq \phi$ for some j, $0 < j \leq \ell+1$. Note that by

Lemma 1 $C_j \in$ code $(\Sigma_{i=1}^{t} \delta_i)$.

Case 1: $j > 1$. Setting i to i_{j-1}, to avoid subscripts,

$$C_j \xmapsto{\text{UPDATE}(i)} \phi.$$

Since $C_j \in$ code $(\Sigma_{i=1}^{t} \delta_i)$ and therefore satisfies (A), (C) and (D), this can

only happen if $C_j[2i:2i+3] \in \{-2-1-2, -2-1-\bar{2}, 212, 21\bar{2}\}$. Assume $C_j[2i:2i+3] =$

$= 212$, the other cases being symmetrical. Since the initial array C^0 con-

tained only blanks, there must be a t', $0 < t' \leq t$, with $t-t'$ minimal, such

that

$$C^{t'-1} \xmapsto{\text{COUNT}(t', \delta_{t'})} C^{t'},$$

$C^{t'}[2i+2] = 2$ and $C^{t'-1}[2i+2] \neq 2$. (A previous mapping UPDATE(k), with $k > i$,

in the $(t+1)$-th step could not have set $C[2i+2]$ to 2 from another value, so

if $C_j[2i+2] = 2$ then $C_k[2i+2] = 2$ for all k, $\ell+1 \geq k \geq j$. Since $C_{\ell+1} = C^t$

indeed $t' \leq t$.) From the definitions it follows that $C[2i+2]$ can only be set

to 2 from another value by the mapping UPDATE(i). So $i \in I(t')$, and we de-

note by C' the array mapped upon by the occurrence of UPDATE(i) in

COUNT$(t', \delta_{t'})$ = UPDATE(i'_ℓ); UPDATE$(i'_{\ell'-1})$; $\ldots$; UPDATE(i'_1); INPUT$(\delta_{t'})$.

By the definition of UPDATE(i) we must have $C'[2i:2i+2] = 002$. Since during

the mappings, following UPDATE(i) in COUNT$(t', \delta_{t'})$, subarray $C[2i+2:\infty]$ is

not accessed, and we have by Lemma 1 that $C' \in$ code$(\Sigma_{i=1}^{t'-1} \delta_i)$ and

$C^{t'} \in$ code$(\Sigma_{i=1}^{t'} \delta_i)$, it therefore follows that

$$(1) \qquad \sum_{k=0}^{2i+1} C^{t'}[k]2^k = \sum_{k=0}^{2i+1} C'[k]2^k + \delta_{t'} = \sum_{k=0}^{2i-1} C'[k]2^k + \delta_{t'}$$

$$\leq (4^{i+1}-1)/3 \qquad\qquad \text{(by (C) and (D))}.$$

Any first occurrence of an UPDATE(i+1) in a COUNT(t",$\delta_{t''}$), t' < t" < t+1, so in between the mappings by the two occurrences of UPDATE(i) in steps t' and t+1, would have set C[2i+2] to 0, resulting in $|c^{t''}[2i+2]| \leq 1$, contradicting the minimality of t-t'. Therefore, for all t", t' < t" < t+1, i+1 $\notin$ I(t"). By the assumption on I in the Proposition it follows that

$$(2) \qquad t-t' < 3^{i+1}.$$

We are now ready to derive a contradiction. For the only mappings which can alter something in C[2i+2:2i+3] are UPDATE(i) and UPDATE(i+1). However, in between the mappings according to the occurrence of UPDATE(i) in step t' and that of UPDATE(i) in step t+1, no occurrence of UPDATE(i) has changed C[2i+2:2i+3] (since this would contradict the minimality of t-t'), and UPDATE(i+1) has not occurred at all (since $C_j[2i+2] \neq 0$ by assumption, i+1 is not in I(t+1) too). So, by the definitions of COUNT and UPDATE we obtain:

$$(3) \qquad \sum_{k=2i+2}^{\infty} c^{t'}[k]2^k = \sum_{k=2i+2}^{\infty} C_j[k]2^k.$$

Furthermore, by Lemma 1,

$$(4) \qquad \sum_{k=0}^{\infty} c^t[k]2^k = \sum_{k=1}^{t} \delta_k = \sum_{k=0}^{\infty} C_j[k]2^k.$$

Thus:

$$(5) \qquad \sum_{k=0}^{2i+1} C_j[k]2^k = \sum_{k=0}^{\infty} C_j[k]2^k - \sum_{k=2i+2}^{\infty} C_j[k]2^k =$$

$$= \sum_{k=1}^{t} \delta_k - \sum_{k=2i+2}^{\infty} C_j[k]2^k \qquad \text{(by (4))}$$

$$= \sum_{k=1}^{t} \delta_k - \sum_{k=2i+2}^{\infty} c^{t'}[k]2^k \qquad \text{(by (3))}$$

$$= \sum_{k=1}^{t} \delta_k - \sum_{k=1}^{t'} \delta_k + \sum_{k=0}^{2i+1} c^{t'}[k]2^k \qquad \text{(by Lemma 1)}$$

$$\le t - t' + (4^{i+1} - 1)/3 \qquad \text{(by (1))}$$

$$< 3^{i+1} + (4^{i+1} - 1)/3. \qquad \text{(by (2))}.$$

But, by way of contradiction, it was assumed that $C_j[2i:2i+1] = 21$. Therefore,

$$(6) \qquad \sum_{k=0}^{2i+1} C_j[k]2^k = 4^{i+1} + \sum_{k=0}^{2i-1} C_j[k]2^k \ge 4^{i+1} - (4^i - 4)/3 - 4^i/2,$$

for $i \ge 2$ (and ≥ 14 for $i = 1$, ≥ 4 for $i = 0$), by (C) and (D). Since for all

$i \ge 0$ the contradictory assumption leads to the contradictory inequalities

(5) and (6) we conclude that $j = 1$ and case 2 holds.

Case 2. $j = 1$ and

$$C_1 \xrightarrow{\text{INPUT}(\delta)} \phi.$$

However, under the assumptions in the Proposition, $0 \in I(t)$ for all $t \ge 1$, so

COUNT$(t+1,\delta) = \ldots$; UPDATE(0); INPUT(δ). But if $C_1[0:\infty] \ne \phi$ is the value of

UPDATE(0) then $C_1[0] \notin \{-2,-\bar{2},2,\bar{2}\}$. Therefore, the contradictory assumption

also fails in this case and

$$C_1 \xrightarrow{\text{INPUT}(\delta)} C_0 \ne \phi.$$

Since the contradictory assumption has now been proven false, by Lemma 1

$C_{\ell+1}, C_\ell, \ldots, C_1 \in \text{code}(\Sigma_{i=1}^{t} \delta_i)$ and $C_0 \in \text{code}(\Sigma_{i=1}^{t} \delta_i + \delta)$. Setting

$c^{t+1} = C_0$ completes the induction. □

Proposition 3 shows us a way of real-time simulating the quintessential

counter S figuring in Proposition 1. Let c^0 be the all-blank initial array,

and let the parameter selection function I meet the timing conditions in

Proposition 3. If we map in the t-th step, for each $t \geq 1$, the current array
value to the next one by COUNT(t,δ), where "add δ", $\delta \in \{-1,0,1\}$, is the in-
put command polled from the input terminal in the t-th step, then the array
at each time $t \geq 0$ is a representation from code(stored integer at time t).
Since the mapping COUNT(t,δ) = ...; INPUT(δ), and INPUT(δ) maps C[0:1] to a
next value, we can simultaneously output "count equals zero" if the next
value of C[0] = $\bar{0}$, or "count unequal zero" if the next value of C[0] $\neq \bar{0}$,
according to Proposition 2.

Note that the requirement of an initial zero count is not essential. We
can as well prove Proposition 3 starting from C^0 equals a representation of
an arbitrary integer c. For instance, a representation from code(c), con-
taining only equal signed digits of absolute value less than 2, for C^0, lets
Proposition 3 go through as well. Thus, the arrangement can real-time simulate
initially nonzero counters.

2.3. An oblivious one-head tape unit.

Proposition 3 puts a heavy burden on a one-head tape unit: C[0:3] must
always be under scan, C[2:5] within each third step, and in general
C[2i: 2i+3] at least once within *each* interval of 3^i steps, *for
all* $i \geq 0$ *simultaneously*. This requires that, basically, at all times all
C[i] must be on the move, drifting inward or outward from the location oc-
cupied by the single head, so to speak. This data motion must be due to the
swapping of array elements amongst the momentarily simultaneously scanned
tape squares. To be able to scan C[2i: 2i+3] within certain time intervals,
for all $i \geq 0$, it is necessary that at certain times arbitrary many of such
fourtuples are split and the pieces geometrically far apart. The piece

C[2i: 2i+1] must be joined to piece C[2i-2: 2i-1] at certain times and to

C[2i+2: 2i+3] at other times, for all i ≥ 1. Apart from performing the

splitting, moving and glueing, the head must als recognize fourtuples.

C[2i: 2i+3] to perform UPDATE, and also know the relative order amongst pairs

of such foursomes. Hence we need to maintain some order and identification

of the array elements. Yet we cannot identify the individual elements of C

with respect to their position, since such an identification tag for C[i]

needs log i space and log i time to evaluate. All this points in the direc-

tion of a recursive process, but again we cannot maintain depth of recursion

parameters.

The process exhibited below rests on the following intuition. The

goal is roughly to access quadruples of consecutive elements of C, of index

$\theta(i)$, at least once in each interval of $2^{\theta(i)}$ steps, for all

i ≥ 0. We call the individual array elements *cells* and consider them as

packets of information to be swapped amongst simultaneously scanned squares.

Assume we are able to move a *block* of cells, called A_1, by, according to

some regime, moving the head, centered on the cells constituting A_1, from

the left end of A_1, where it scans some squares left adjacent to A_1, to the

right end of A_1, where it scans some squares right adjacent to A_1, and

back again to the left end of A_1. Let A_1 be contained in a block of cells

called A_2. Then A_1 moves by transporting cells of $A_2 - A_1$ through A_1 to the

other side of A_1, while simultaneously shifting the cells of A_1. Thus, we

will shift the total block A_1 from the left end of A_2 to the right end, and

back again to the left end. During such a full sweep of A_1 over A_2, we will

shift block A_2 within a larger block A_3 by a single square. So the relation

between A_2 and A_3 is analogous to that between A_1 and A_2. See Figure 1.
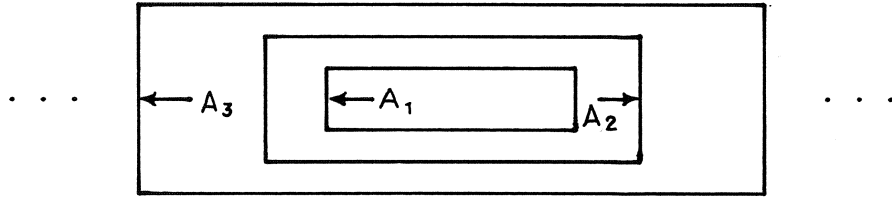
Figure 1. The blocks are individually "moving" in the indicated directions.

In general, we envision an infinite series of nested blocks, $A_1, A_2, \ldots, A_i, A_{i+1}, \ldots$; with A_i properly embedded in A_{i+1}, $i \geq 1$, such that a full sweep of block A_i over block A_{i+1} shifts block A_{i+1} one square in the currently desired direction. In the above arrangement, the head is *always* centered on block A_1, and therefore, since it is allowed to scan but a fixed number of squares, when it is centered at the end of block A_1 it scans but a fixed amount of squares outside. Since the ends of the individual blocks govern the action the single head ought to take, and also cells of $A_{i+1} - A_i$ have to be transported through A_i for arbitrary i, we cannot have the physically present ends of all of blocks $A_2, A_3, \ldots, A_i$ in between the head centered on A_1 and a cell, to be transported, in $A_{i+1} - A_i$.

So we want the blocks to move, in a sense, completely out of each other. That is, an arrangement as below in Figure 2, where we denote the cells in $A_{i+1} - A_i$ as B_{i+1}, for all $i \geq 1$, and A_1 by B_1. ($x \overset{*}{\mapsto} y$ denotes that y occurs after x).
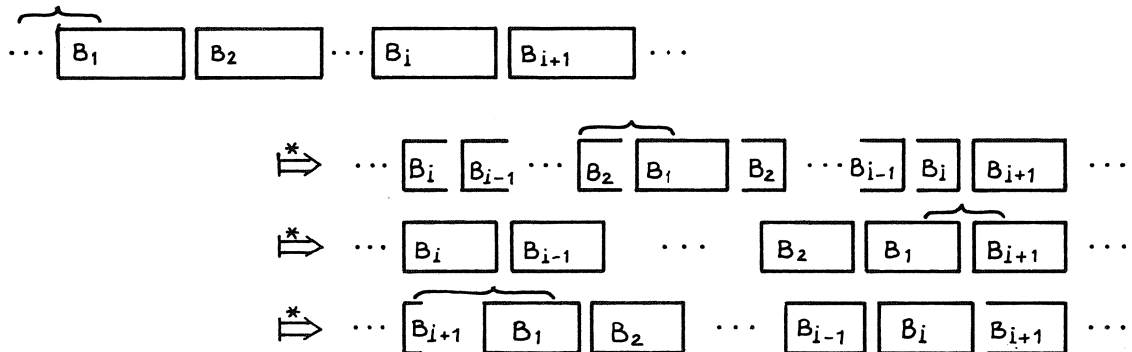
Figure 2.

In this manner we telescope the blocks, as it were, inwards and then outwards in the other direction, subsequently reversing the process. To achieve this behavior, we transport, for all $i \geq 1$, elements of block B_{i+1} through $A_i = \bigcup_{j=1}^{i} B_j$ while simultaneously shifting the cells of A_i to accomodate the transport. The motion of the head through A_i is governed by recursively moving B_{j+1} through A_j, for all j, $j \leq i$. Schematically, level i of the process is depicted in Figure 3.
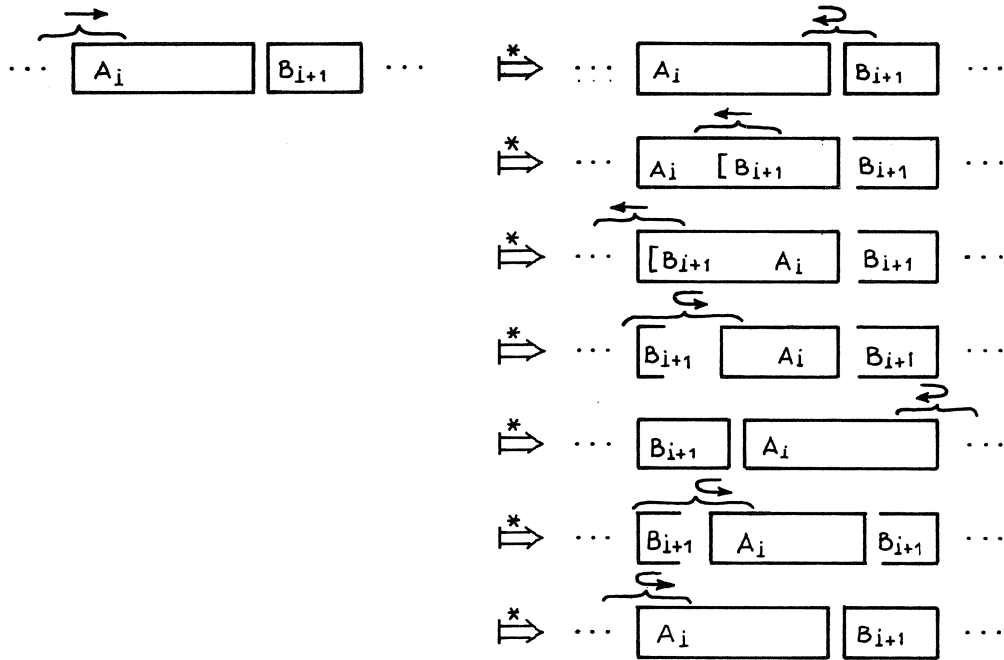
Figure 3. The action of block $A_{i+1} = A_i \cup B_{i+1}$ with respect to blocks B_j, $j > i+1$, is not depicted.

When the head was at the ends of block $A_{i+1} = A_i \cup B_{i+1}$, it now could have picked up or deposited a cell outside, that is, of a block B_j, $j > i+1$. Assume that all blocks B_i, $i \geq 1$, have the same number of cells, say x. By a *full sweep* of the head over block A_i we shall mean the action the head has to perform, starting from one end of A_i, to pick up a cell of B_j, $j > i$, deposit it on the other side of A_i, and finish at the same end of A_i it started from. So basically a full sweep of the head is a traversal of block A_i from one end to the other end and back again. Let a full sweep of the head over A_i take at most S(i) steps, $i \geq 1$. Then to transport all of B_{i+1} from one end of A_i to the other end, and back again, takes at most cxS(i) steps for some constant c. Since this constitutes a full sweep of the head over block A_{i+1}, we have S(i+1) $\leq$ cxS(i), for all $i \geq 1$, and obviously S(1) $\leq$ cx. So S(i) $\in 2^{O(i)}$ for all $i \geq 1$.

In the formal construction below we set the block size to 2, and represent the loosely described block B_i by "$[_i]_i$", in the understanding that the two cells concerned are tagged with "[" and "]". The subscripts on the tags are just there to aid the reader, but do not occur in the actual simulation. An element of block B_j in transport through block A_i, $j \geq i$, is identified by a curly bracket of the appropriate type. Thus each individual cell has permanently assigned to it a tag, consisting of either an opening or closing bracket, which may at different times be square or curly. Figure 4 sketches a descriptive situation:

$$\cdots \quad \boxed{A_i \quad \{_j \qquad }]_j \quad \cdots$$
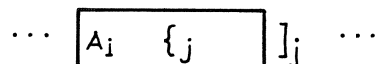
Figure 4.

After these preliminaries we formally define a one-head tape unit M. It

is convenient to view the *instanteneous descriptions* (i.d.'s) (momentary snap-shots of M's tape contents and the head position) as one-way infinite linear arrays T, with "<" or ">" denoting the centre of the head position. We tag the cells, containing elements of the array C of the previous section, with "[", "]", "{" or "}". Below we display only these tags, since for the moment we are not interested in the cell contents. The identity of the underlying squares is not important, but the identity of the tagged cells is fixed, wherever they end up. For convenience of the reader we index the tagged cells (or rather the tags). The eventually defined machine, however, has no indexes associated with the cells, only one out of the four mentioned tags. The *initial i.d.* is, now focussing on the tags only,

$$T^0 = > [_1]_1[_2]_2 \cdots [_i]_i[_{i+1}]_{i+1} \cdots$$

We describe transformations of the array T in the form of six parametrized recursive functions, and four nonparametrized functions, each of two types. Each such function X will, for a unique subarray of T, rewrite this subarray by re-ordering its elements, specified by X(.): $\alpha \diamond \beta \to \alpha' \diamond' \beta'$ with $\alpha,\alpha',\beta,\beta'$ being strings of (for clarity indexed) tags and $\diamond,\diamond' \in \{>,<\}$. A definite requirement for the process is that, at some time, it has to scan "$[_{i+2}]_{i+2}[_{i+3}]_{i+3}$" for the first time. So we define, for all $i \geq 0$:

$$A(>,i): \; >[_1]_1[_2]_2 \cdots [_i]_i[_{i+1}]_{i+1} \; \mapsto \; [_{i+1}[_i]_i[_{i-1}]_{i-1} \cdots [_1]_1 >]_{i+1}.$$

For symmetry we also define:

$$A(<,i): \; [_{i+1}]_{i+1}[_i]_i[_{i-1}]_{i-1} \cdots [_1]_1 < \; \mapsto \; [_{i+1} < [_1]_1[_2]_2 \cdots [_i]_i]_{i+1}.$$

To abbreviate notation we shall henceforth denote shortly, for all $i \geq 0$,

$$\diamond [_I]_I \overset{def}{=} \diamond [_1]_1[_2]_2 \ldots [_i]_i$$

$$[_I]_I \diamond \overset{def}{=} [_i]_i[_{i-1}]_{i-1} \ldots [_1]_1 \diamond ,$$

$$\diamond \in \{<,>\}, \text{ and } [_I]_I = \varepsilon \text{ for } i = 0.$$

So for all $i \geq 0$:

$$A(>,i): \quad >[_I]_I[_{i+1}]_{i+1} \mapsto [_{i+1}[_I]_I>]_{i+1}$$

$$A(<,i): \quad [_{i+1}]_{i+1}[_I]_I< \mapsto [_{i+1}<[_I]_I]_{i+1}.$$

By execution of $A(>,i)$ on the appropriate unique substring of T^0 we have therefore, using the same rewriting denotation as in the previous section:

$$T^0 \xrightarrow{A(>,i)} T^{t_i}$$

with

$$T^0 = >[_1]_1[_2]_2 \ldots [_i]_i[_{i+1}]_{i+1}[_{i+2}]_{i+2}[_{i+3}]_{i+3} \ldots$$

and

$$T^{t_i} = [_{i+1}[_i]_i[_{i-1}]_{i-1} \ldots [_1]_1>]_{i+1}[_{i+2}]_{i+2}[_{i+3}]_{i+3} \ldots$$

where t_i is the number of steps it takes to execute the mapping $A(>,i)$, to be specified later, $i \geq 0$.

With the head scanning at least five squares right of the center position, indicated by " $>$ ", the subarray "$[_{i+2}]_{i+2}[_{i+3}]_{i+3}$" is scanned at time t_i, for all $i \geq 0$, while at time $t = 0$ the subarray "$[_1]_1[_2]_2$" is scanned.

DEFINITION. To achieve the required interchange of tagged cells, define the functions below. Recall that "$\}_j$" denotes the same cell as "$]_j$", only the attached tags have changed. Similarly for "$\{_j$" and "$[_j$". For all $i \geq 0$ and $j > i+1$:

$$A(>,i): \; >[_I]_I[_{i+1}x \;\mapsto\; [_{i+1}[_I]_I>x \qquad\qquad (x \neq [)$$

$$A(<,i): \; x]_{i+1}[_I]_I< \;\mapsto\; x<[_I]_I]_{i+1} \qquad\qquad (x \neq])$$

$$B(>,i): \;]_{i+1}[_I]_I>]_jx \;\mapsto\; \}_j<[_I]_I]_{i+1}x \qquad\qquad (x \neq [)$$

$$B(<,i): \; x[_j<[_I]_I[_{i+1} \;\mapsto\; x[_{i+1}[_I]_I>\{_j \qquad\qquad (x \neq])$$

$$C(>,i): \;]_{i+1}[_I]_I>]_j[_{j+1} \;\mapsto\; \}_j\{_{j+1}<[_I]_I]_{i+1}$$

$$C(<,i): \;]_{j+1}[_j<[_I]_I[_{i+1} \;\mapsto\; [_{i+1}[_I]_I>\}_{j+1}\{_j$$

$$D(>,i): \; >[_I]_Ix \;\mapsto\; [_I]_I>x \qquad\qquad (x \neq [)$$

$$D(<,i): \; x[_I]_I< \;\mapsto\; x<[_I]_I \qquad\qquad (x \neq])$$

$$E(>,i): \; [_{i+1}[_I]_I>]_{i+1}x \;\mapsto\; [_{i+1}]_{i+1}[_I]_I>x \qquad\qquad (x \neq [)$$

$$E(<,i): \; x[_{i+1}<[_I]_I]_{i+1} \;\mapsto\; x<[_I]_I[_{i+1}]_{i+1} \qquad\qquad (x \neq])$$

$$F(>,i): \; [_{i+1}[_I]_I>]_{i+1}[_{i+2}x \;\mapsto\; [_{i+2}[_{i+1}]_{i+1}[_I]_I>x \qquad (x \neq [)$$

$$F(<,i): \; x]_{i+2}[_{i+1}<[_I]_I]_{i+1} \;\mapsto\; x<[_I]_I[_{i+1}]_{i+1}]_{i+2} \qquad (x \neq])$$

$$G(>): \; >\{_j]_i[_{i+1} \;\mapsto\; >]_i[_{i+1}\{_j$$

$$G(<): \;]_{i+1}[_i\}_j< \;\mapsto\; \}_j]_{i+1}[_i<$$

$$H(>): \; >\}_{j+1}\{_j]_i[_{i+1} \;\mapsto\; >]_i[_{i+1}\}_{j+1}\{_j$$

$$H(<): \;]_{i+1}[_i\}_j\{_{j+1}< \;\mapsto\; \}_j\{_{j+1}]_{i+1}[_i<$$

$$J(>): \; >\{_{i+1}]_i]_{i+1} \;\mapsto\; <]_i[_{i+1}]_{i+1}$$

$$J(<): \; [_{i+1}[_i\}_{i+1}< \;\mapsto\; [_{i+1}]_{i+1}[_i>$$

$$K(>): \quad >\}_{i+2}\{_{i+1}]_i]_{i+1} \;\mapsto\; <]_i[_{i+1}]_{i+1}]_{i+2}$$

$$K(<): \quad [_{i+1}[_i\}_{i+1}\{_{i+2}< \;\mapsto\; [_{i+2}[_{i+1}]_{i+1}[_i>$$

The *parametrized* functions A through F set the basic pattern to transport tagged cells from one side of $[_I]_I$ to the other side. (The index j is always greater than i+1). The *nonparametrized* functions G and H serve to move (linked pairs of) curly brackets through "][" interfaces. If curly brackets are adjacent to a (linked) pair like "][" then they have not yet reached their destination. If curly brackets are adjacent to a pair of square brackets of equal type, then they have reached their destination, and are fitted in place and changed back to square brackets, by the functions J and K. In the G and H functions, the index j is again greater that i+1. However, to make the point once more, the indexes are only put there to aid the reader. The intention of the described rewritings is that the arrays concerned consist of nonsubscripted brackets, each bracket viewed as tagging a particular cell, and the rewriting reorders these tagged cells in the array, and possibly changes brackets from square ones to curly ones of the same type, or vice versa, as indicated in the indexed version above. Note that A(>,i): Y ↦ Y' and A(<,i): Z ↦ Z' are related by the fact that Z is the mirror image of Y and Z' is the mirror image of Y'. With *mirror image* we do not mean only the reverse, but the reverse with every constituent symbol changed to its mirror image, so ">" to "<", "[" to "]", "[" to "]", "{" to "}" and "}" to "{". Similarly for the other functions.

LEMMA 2. *For all* i > 0, *the functions are related as follows:*

a)   A(>,i) = A(>,i-1); F(>,i-1)

      A(<,i) = A(<,i-1); F(<,i-1)

b) $B(>,i) = B(>,i-1);\ G(<);\ F(<,i-1)$

   $B(<,i) = B(<,i-1);\ G(>);\ F(>,i-1)$

c) $C(>,i) = C(>,i-1);\ H(<);\ F(<,i-1)$

   $C(<,i) = C(<,i-1);\ H(>);\ F(>,i-1)$

d) $D(>,i) = A(>,i-1);\ E(>,i-1)$

   $D(<,i) = A(<,i-1);\ E(<,i-1)$

e) $E(>,i) = B(>,i-1);\ J(<);\ D(>,i-1);\ E(>,i-1)$

   $E(<,i) = B(<,i-1);\ J(>);\ D(<,i-1);\ E(<,i-1)$

f) $F(>,i) = C(>,i-1);\ K(<);\ D(>,i-1);\ E(>,i-1)$

   $F(<,i) = C(<,i-1);\ K(>);\ D(<,i-1);\ E'(<,i-1)$

that is, six parametrized functions recursively calling each other. (Since
$D(>,0)$ and $D(<,0)$ are "no operation"'s which don't change anything we leave
them out, cf. below.)

PROOF. For a) through f) we prove one equality each; the other one is sym-
metric. For all $i > 0$, with $[_{I-1}]_{I-1} = \varepsilon$ for $i = 1$ by definition:

a) For $x \neq [$:

$$^>[_I]_I[_{i+1}x = {}^>[_{I-1}]_{I-1}[_i]_i[_{i+1}x$$

$$\xrightarrow{A(>,i-1)} [_i[_{I-1}]_{I-1}{}^>]_i[_{i+1}x$$

$$\xrightarrow{F(>,i-1)} [_{i+1}[_i]_i[_{I-1}]_{I-1}{}^>x = [_{i+1}[_I]_I{}^>x$$

b) For $x \neq]$:

$$x[_j{}^<[_I]_I[_{i+1} = x[_j{}^<[_{I-1}]_{I-1}[_i]_i[_{i+1}$$

$$\xrightarrow{B(<,i-1)} x[_i[_{I-1}]_{I-1}{}^>\{_j]_i[_{i+1}$$

$$\xrightarrow{G(>)} x[_i[_{I-1}]_{I-1}{}^>]_i[_{i+1}\{_j$$

34

$$\xrightarrow{F(>,i-1)} x[_{i+1}[_i]_i[_{I-1}]_{I-1}>\{_j$$

$$= x[_{i+1}[_I]_I>\{_j$$

c)
$$]_{j+1}[_j<[_I]_I[_{i+1} =]_{j+1}[_j<[_{I-1}]_{I-1}[_i]_i[_{i+1}$$

$$\xrightarrow{C(<,i-1)} [_i[_{I-1}]_{I-1}>\}_{j+1}\{_j]_i[_{i+1}$$

$$\xrightarrow{H(>)} [_i[_{I-1}]_{I-1}>]_i[_{i+1}\}_{j+1}\{_j$$

$$\xrightarrow{F(>,i-1)} [_{i+1}[_i]_i[_{I-1}]_{I-1}>\}_{j+1}\{_j$$

$$= [_{i+1}[_I]_I>\}_{j+1}\{_j$$

d) For x ≠ [:

$$>[_I]_Ix = >[_{I-1}]_{I-1}[_i]_ix$$

$$\xrightarrow{A(>,i-1)} [_i[_{I-1}]_{I-1}>]_ix$$

$$\xrightarrow{E(>,i-1)} [_i]_i[_{I-1}]_{I-1}>x$$

$$= [_I]_I>x$$

e) For x ≠] :

$$x[_{i+1}<[_I]_I]_{i+1} = x[_{i+1}<[_{I-1}]_{I-1}[_i]_i]_{i+1}$$

$$\xrightarrow{B(<,i-1)} x[_i[_{I-1}]_{I-1}>\{_{i+1}]_i]_{i+1}$$

$$\xrightarrow{J(>)} x[_i[_{I-1}]_{I-1}<]_i[_{i+1}]_{i+1}$$

$$\xrightarrow{D(<,i-1)} x[_i<[_{I-1}]_{I-1}]_i[_{i+1}]_{i+1}$$

$$\xrightarrow{E(<,i-1)} x<[_{I-1}]_{I-1}[_i]_i[_{i+1}]_{i+1}$$

$$= x<[_I]_I[_{i+1}]_{i+1}$$

f) For $x \neq \,]$:

$$x]_{i+2}[_{i+1} <[_I]_I]_{i+1} = x]_{i+2}[_{i+1} <[_{I-1}]_{I-1}[_i]_i]_{i+1}$$

$$\xRightarrow{C(<,i-1)} x[_i[_{I-1}]_{I-1}>\}_{i+2}\{_{i+1}]_i]_{i+1}$$

$$\xRightarrow{K(>)} x[_i[_{I-1}]_{I-1}<]_i[_{i+1}]_{i+1}]_{i+2}$$

$$\xRightarrow{D(<,i-1)} x[_i<[_{I-1}]_{I-1}]_i[_{i+1}]_{i+1}]_{i+2}$$

$$\xRightarrow{E(<,i-1)} x<[_{I-1}]_{I-1}[_i]_i[_{i+1}]_{i+1}]_{i+2}$$

$$= x<[_I]_I[_{i+1}]_{i+1}]_{i+2}. \quad \square \; \square$$

The mappings $D(>,0)$: $>x \mapsto >x$ $(x \neq [)$ and $D(<,0)$: $x< \mapsto x<$ $(x \neq])$ are "no operation" or "skip" instructions. Deleting them henceforth in the expansion rules of Lemma 2e) and 2f), for $i = 1$, those become:

ad Lemma 2a) $E(>,1) = B(>,0)$; $J(<)$; $E(>,0)$

$\qquad\qquad\quad E(<,1) = B(<,0)$; $J(>)$; $E(<,0)$

ad Lemma 2f) $F(>,1) = C(>,0)$; $K(<)$; $E(>,0)$

$\qquad\qquad\quad F(<,1) = C(<,0)$; $K(>)$; $E(<,0)$

A *level i expansion* of a function $X(>,j)$ or $X(<,j)$, $j \geq i \geq 0$, consists of expanding that parametrized function with parameter j into a sequence of parameter i functions and nonparametrized functions, according to Lemma 2 (with the "no operation"'s $D(>,0)$ and $D(<,0)$ left out in case $i = 0$). So if $Y_1^{(i)}$; $Y_2^{(i)}$; $\ldots$; $Y_n^{(i)}$ is a level i expansion of $X(.)$ then $X(.) = Y_1^{(i)}$; $Y_2^{(i)}$; $\ldots$ $\ldots$; $Y_n^{(i)}$ with $\qquad Y_\ell^{(i)} \in \{A(>,i),A(<,i),B(>,i),B(<,i),\ldots,F(<,i),G(>),$ $G(<),\ldots,K(<)\} - \{D(>,0),D(<,0)\}$, $1 \leq \ell \leq n$. We extend the concept in the obvious way to sequences of functions $X_1(\diamond_1,j_1)$; $X_2(\diamond_2,j_2)$; $\ldots$; $X_m(\diamond_m,j_m)$, $j_1,j_2,\ldots,j_m \geq i$ and $\diamond_1,\diamond_2,\ldots,\diamond_m \in \{<,>\}$. For example, the level 0 ex-

pansion of A(>,3) is found by way of the level 2 and level 1 expansions:

$$A(>,3) = A(>,2); \; F(>,2)$$

$$= A(>,1); \; F(>,1); \; C(>,1); \; K(<); \; D(>,1); \; E(>,1)$$

$$= A(>,0); \; F(>,0); \; C(>,0); \; K(<); \; E(>,0); \; C(>,0); \; H(<);$$

$$F(<,0); \; K(<); \; A(>,0); \; E(>,0); \; B(>,0); \; J(<); \; E(>,0).$$

The atomic mappings of the level 0 expansions of the parametrized functions are called the *local rewriting rules*, and govern the switching of the tagged cells, in the squares scanned, by the basic steps of the oblivious one-head tape unit M. Note that a fat head covering four squares left and four squares right of the displayed center ">" or "<" suffices to execute these atomic mappings. Below we use superscripts to distinguish the identity of the various tagged cells before and after rewriting.

Local rewriting rules:

$$G(>): \quad >\{^1]^2[^3 \mapsto >]^2[^3\{^1$$

$$G(<): \quad]^1[^2\}^3< \mapsto \}^3]^1[^2<$$

$$H(>): \quad >\}^1\{^2]^3[^4 \mapsto >]^3[^4\}^1\{^2$$

$$H(<): \quad]^1[^2\}^3\{^4< \mapsto \}^3\{^4]^1[^2<$$

$$J(>): \quad >\{^1]^2]^3 \mapsto <]^2[^1]^3$$

$$J(<): \quad [^1[^2\}^3< \mapsto [^1]^3[^2>$$

$$K(>): \quad >\}^1\{^2]^3]^4 \mapsto <]^3[^2]^4]^1$$

$$K(<): \quad [^1[^2\}^3\{^4< \mapsto [^4[^1]^3[^2>$$

$$A(>,0): \quad >[\mapsto [>$$

$$A(<,0): \quad]< \mapsto <]$$

$$B(>,0): \quad]^1 >]^2 x \mapsto \}^2 <]^1 x \qquad (x \neq [)$$

$$B(<,0): \quad x[^1 <[^2 \mapsto x[^2 >\{^1 \qquad (x \neq])$$

$$C(>,0): \quad]^1 >]^2 [^3 \mapsto \}^2 \{^3 <]^1$$

$$C(<,0): \quad]^1 [^2 <[^3 \mapsto [^3 >\}^1 \{^2$$

$$E(>,0): \quad [^1 >]^2 x \mapsto [^1]^2 >x \qquad (x \neq [)$$

$$E(<,0): \quad x[^1 <]^2 \mapsto x<[^1]^2 \qquad (x \neq])$$

$$F(>,0): \quad [^1 >]^2 [^3 \mapsto [^3 [^1]^2 >$$

$$F(<,0): \quad]^1 [^2 <]^3 \mapsto <[^2]^3]^1$$

The only use of the context symbols x in the definitions of $A(>,i)$, $A(<,i)$, $F(>,i)$ and $F(<,i)$ was to force a unique expansion into functions with parameter j, $j < i$, according to Lemma 2. Since $A(<,0)$, $A(>,0)$, $F(<,0)$ and $F(>,0)$ are atomic indivisible actions, because the local rewriting rules shall not be decomposed any further, we do not need these context symbols at the lowest level.

In the sequel it is useful to talk about *well formed* arrays, that is, the set of arrays from which the consecutive i.d.'s of M are taken.

(i) T^0 is a well formed array.

(ii)   If T is a well formed array and X(.) is any local rewriting rule, with the dot standing for any appropriate argument, such that $T \xRightarrow{X(.)} T'$, $T' \neq \phi$, then T' is a well formed array.

(iii) No array is well formed except by (i) and (ii).

Since no mapping either deletes or multiplies a headmarker, i.e., "<" or ">", all well formed arrays contain a single headmarker. By the mutual exclusion of the subarrays they rewrite, if a well formed array T is rewrit-

ten to $T' \neq \phi$ by a local rewriting rule, then T is rewritten to ϕ by all other local rewriting rules. We now show that a well formed array T is always rewritten by some local rewriting rule to a another well formed array, which rewriting rule and array are therefore unique.

Earlier, we observed that, for all $i \geq 0$,

$$T^0 \xRightarrow{A(>,i)} T^{t_i}.$$

If $Y_1^{(0)}$; $Y_2^{(0)}$; $Y_3^{(0)}$; $\ldots$; $Y_n^{(0)} = A(>,i)$ is the level 0 expansion of $A(>,i)$ then, by Lemma 2, there exist well formed arrays $T_0^{(0)}, T_1^{(0)}, T_2^{(0)}, \ldots, T_n^{(0)}$, $T_0^{(0)} = T^0$ and $T_n^{(0)} = T^{t_i}$, such that

$$T_{j-1}^{(0)} \xRightarrow{Y_j^{(0)}} T_j^{(0)}$$

for all j, $1 \leq j \leq n$. By the uniquences of application of local rewriting rules it follows that

$$T_{j-1}^{(0)} \xmapsto{X} \phi$$

for all j, $1 \leq j \leq n$, $X \neq Y_j^{(0)}$ and X is a local rewriting rule. Hence each well formed array in the sequence $T_0^{(0)}, T_1^{(0)}, \ldots, T_n^{(0)}$ has exactly one local rewriting rule which is applicable to it, and the application of this local rewriting rule yields exactly one next well formed array.

By Lemma 2a we have $A(>,i) = A(>,i-1)$; $F(>,i-1)$ for all $i > 0$, which leads to

$$A(>,i) = A(>,0);\ F(>,0);\ F(>,1);\ \ldots;\ F(>,i-1)$$

with

$$T^0 \xRightarrow{A(>,0)} T^{t_0}$$

and

$$T^{t_j} \xrightarrow{F(>,j)} T^{t_{j+1}}$$

for all j, $0 \leq j < i$. Define $A(>,\infty)$ by

$$A(>,\infty) = \lim_{i \to \infty} A(i)$$

$$= A(>,0); \ F(>,0); \ F(>,1); \ \ldots; \ F(>,i); \ \ldots$$

and the level 0 expansion of $A(>,\infty)$ as the infinite, or unbounded, sequence of local rewriting rules resulting from the level 0 expansions of the constituent functions $F(>,i)$, $i > 0$, above. So

$$A(>,\infty) = Y_1^{(0)}; \ Y_2^{(0)}; \ \ldots; \ Y_i^{(0)}; \ \ldots$$

$$= A(>,0); \ F(>,0); \ C(>,0); \ K(<); \ E(>,0); \ldots$$

and there exists an infinite sequence of well formed arrays $T_0^{(0)}, T_1^{(0)}, \ldots$ $\ldots, T_i^{(0)}, \ldots, T_0^{(0)} = T^0$, such that for all $j \geq 1$

$$T_{j-1}^{(0)} \xrightarrow{Y_j^{(0)}} T_j^{(0)}$$

and for no local rewriting rule $X \neq Y_j^{(0)}$ and $T \neq \phi$

$$T_{j-1}^{(0)} \xrightarrow{X} T,$$

i.e., $Y_j^{(0)}$ is the only local rewriting rule applicable to $T_{j-1}^{(0)}$. Consequently, a machine which wants to execute the sequence of local rewritings of the level 0 expansion of $A(>,\infty)$, starting with i.d. T^0, needs only to select the single local rewriting rule $Y_j^{(0)}$, applicable to the current $T_{j-1}^{(0)}$, by considering the length 9 subarray of $T_{j-1}^{(0)}$ with the current headmarker in the center, to obtain the next $T_j^{(0)}$, $j \geq 1$. From the expansions in Lemma 2

we see that a nonparametrized function of G,H,J,K is always followed by a parametrized function from A,B,C,E,F in the level 0 expansion of $A(>,\infty)$. In a single step of M we shall first execute a local rewriting according to G,H,J, or K, if possible, and then execute a local rewriting according to A,B,C,E or F, which by the above is always possible, starting with initial i.d. T^0. So the oblivious one-head tape unit M at each step shall examine the squares around the headmarker, and switches tagged cells and head position amongst the scanned squares according to the only local rewriting rules applicable. Figure 5 shows an initial segment of the sequence of well formed arrays $T_0^{(0)}, T_1^{(0)}, \ldots, T_i^{(0)}, \ldots$ produced by the successive execution of the local rewriting rules in the level 0 expansion of $A(>,\infty)$ using the simple procedure SWITCH below.

Procedure SWITCH:

Step 1. Examine the length 9 subarray, centered on the headmarker, of the current i.d. and switch tagged cells and headmarker position according to the single, if any, local rewriting rule from the G,H,J,K, rules which is applicable. The result is a well formed array T.

Step 2. Examine the length 9 subarray, centered on the headmarker, of array T from step 1, and switch tagged cells and headmarker position according to the single local rewriting rule from the A,B,C,E,F rules which is applicable. The resultant well formed array is the next i.d..

LEMMA 3. *Starting from the initial i.d.* T^0, *a one-head tape unit M, executing SWITCH at each single step, executes exactly the local rewriting rule sequence of the level 0 expansion of* $A(>,\infty)$. *For each* t > 0, *in the first* t *steps M executes this sequence up to and including the* t-th *occurrence of a local rewriting rule of the* A,B,C,E *or* F *type.*

```
Start i.d.        >[][][][][][][][][][]...
⊢A(>,0)⇒        [>][][][][][][][][][]...
⊢F(>,0)⇒        [[]>][][][][][][][][]...
⊢C(>,0)⇒        [}{<]][][][][][][][]...
⊢K(<)⟹          [][>][][][][][][][][]...
⊢E(>,0)⇒        [][]>][][][][][][][]...
⊢C(>,0)⇒        [][]}{<]][][][][][][]...
⊢H(<)⟹          [}{][<]][][][][][][]...
⊢F(<,0)⇒        [}{<]][][][][][][][]...
⊢K(<)⟹          [][>[]][][][][][][][]...
⊢A(>,0)⇒        [][[>]][][][][][][][]...
⊢E(>,0)⇒        [][[]>]][][][][][][][]...
⊢B(>,0)⇒        [][[]<]][][][][][][][]...
⊢J(<)⟹          [][][>]][][][][][][][]...
⊢E(>,0)⇒        [][][]>]][][][][][][][]...
⊢C(>,0)⇒        [][][]}{<]][][][][][][]...
⊢H(<)⟹          [][]}{][<]][][][][][][]...
⊢F(<,0)⇒        [][]}{<]][][][][][][]...
⊢H(<)⟹          [}{][<]][][][][][][]...
⊢C(<,0)⇒        [}{[>}{]][][][][][][]...
⊢K(>)⟹          [}{[<][]][][][][][][]...
⊢E(<,0)⇒        [}{<][]][][][][][][]...
⊢K(<)⟹          [][>[]][][][][][][][]...
⊢A(>,0)⇒        [][[>[]][][][][][][][]...
⊢F(>,0)⇒        [][[[]>]][][][][][][][]...
⊢B(>,0)⇒        [][[[]<]][][][][][][][]...
⊢J(<)⟹          [][[][>]][][][][][][][]...
⊢E(>,0)⇒        [][[][]>]][][][][][][][]...
⊢B(>,0)⇒        [][[][]<]][][][][][][][]...
⊢G(<)⟹          [][[]}][<]][][][][][][]...
⊢F(<,0)⇒        [][[]<][]][][][][][][]...
⊢J(<)⟹          [][[][>[]][][][][][][][]...
⊢A(>,0)⇒        [][[][[>]][][][][][][][]...
⊢E(>,0)⇒        [][[][[]>]][][][][][][][]...
⊢B(>,0)⇒        [][[][[]<]][][][][][][][]...
⊢J(<)⟹          [][[][[]>]][][][][][][][]...
⊢E(>,0)⇒        [][[][[][]>]][][][][][][][]...
⊢C(>,0)⇒        [][[][[][]}{<]][][][][][]...
⊢H(<)⟹          [][[][]}{][<]][][][][][]...
⊢F(<,0)⇒        [][[][]}{<]][][][][][]...
⊢H(<)⟹          [][[]}{][<]][][][][][]...
⊢C(<,0)⇒        [][[]}{[>}{]][][][][][]...
⊢K(>)⟹          [][[]}{[<]][][][][][]...
⊢E(<,0)⇒        [][[]}{<]][][][][][]...
⊢H(<)⟹          [[}{][<]][][][][][]...
⊢C(<,0)⇒        [[}{[>}{]][][][][][]...
⊢H(>)⟹          [[}{[>][}{]][][][][][]...
⊢F(>,0)⇒        [[}{[[]>}{]][][][][][]...
⊢K(>)⟹          [[}{[[]<][}{]][][][][][]...
⊢A(<,0)⇒        [[}{[[<]][}{]][][][][][]...
⊢E(<,0)⇒        [[}{[<]][}{]][][][][][]...
⊢B(<,0)⇒        [[}{[>{]][}{]][][][][][]...
⊢J(>)⟹          [[}{[<]][}{]][][][][][]...
⊢E(<,0)⇒        [[}{<]][}{]][][][][][]...
⊢K(<)⟹          [[][>[]][][][][][][]...
⊢A(>,0)⇒        [[][[>]][][][][][][]...
⊢F(>,0)⇒        [[][[[]>][][][][][][]...
⊢C(>,0)⇒        [[][[[}{<]][][][][][][]...
```

```
⊢K(<)⟹          [[][[[][>]]][][][][][]...
⊢E(>,0)⇒        [[][[[][]>]][][][][][]...
⊢B(>,0)⇒        [[][[[][}{<]]][][][][][]...
⊢G(<)⟹          [[][[[]}][<]]][][][][][]...
⊢F(<,0)⇒        [[][[[]}{<]]][][][][][]...
⊢J(<)⟹          [[][[[][>[]]][][][][][]...
⊢A(>,0)⇒        [[][[[][[>]]][][][][][]...
⊢E(>,0)⇒        [[][[[][[]>]][][][][][]...
⊢B(>,0)⇒        [[][[[][[]<]]][][][][][]...
⊢J(<)⟹          [[][[[][[]>]][][][][][]...
⊢E(>,0)⇒        [[][[[][[][]>]]][][][][]...
⊢B(>,0)⇒        [[][[[][[][}{<]]][][][][]...
⊢G(<)⟹          [[][[[}][}{<]]][][][][]...
⊢F(<,0)⇒        [[][[[}{<]]][][][][]...
⊢G(<)⟹          [[][[}][<]][][][][]...
⊢C(<,0)⇒        [[][[}{>}{]][][][][]...
⊢K(>)⟹          [[][[}{<]]][][][][]...
⊢E(<,0)⇒        [[][[}{<]]][][][][]...
⊢J(<)⟹          [[][[][>[]]][][][][]...
⊢A(>,0)⇒        [[][[][[>]]][][][][]...
⊢F(>,0)⇒        [[][[][[]>]]][][][][]...
⊢B(>,0)⇒        [[][[][[]<]]][][][][]...
⊢J(<)⟹          [[][[][[]>]]][][][][]...
⊢E(>,0)⇒        [[][[][[][]>]]][][][][]...
⊢B(>,0)⇒        [[][[][[][}{<]]][][][][]...
⊢G(<)⟹          [[][[][[}][<]]][][][][]...
⊢F(<,0)⇒        [[][[][[}{<]]][][][][]...
⊢J(<)⟹          [[][[][[][>]]][][][][]...
⊢A(>,0)⇒        [[][[][[][>[]]][][][][]...
⊢E(>,0)⇒        [[][[][[][[]>]]][][][][]...
⊢B(>,0)⇒        [[][[][[][[]<]]][][][][]...
⊢J(<)⟹          [[][[][[][[]>]]][][][][]...
⊢E(>,0)⇒        [[][[][[][[][]>]]][][][]...
⊢C(>,0)⇒        [[][[][[][[][]}{<]]][][]...
⊢H(<)⟹          [[][[][[][[]}{][<]]][][]...
⊢F(<,0)⇒        [[][[][[][[]}{<]]][][]...
⊢H(<)⟹          [[][[][[]}{][<]]][][]...
⊢C(<,0)⇒        [[][[][[]}{[>}{]]][][]...
⊢K(>)⟹          [[][[][[]}{[<]]][][]...
⊢E(<,0)⇒        [[][[][[]}{<]]][][]...
⊢H(<)⟹          [[][[}{][<]]][][]...
⊢C(<,0)⇒        [[][[}{[>}{]]][][]...
⊢H(>)⟹          [[][[}{[>][}{]]][][]...
⊢F(>,0)⇒        [[][[}{[[]>}{]]][][]...
⊢K(>)⟹          [[][[}{[[]<]]][][]...
⊢A(<,0)⇒        [[][[}{[[<]]][][]...
⊢E(<,0)⇒        [[][[}{[<]]][][]...
⊢B(<,0)⇒        [[][[}{[>{]]][][]...
⊢J(>)⟹          [[][[}{[<]]][][]...
⊢E(<,0)⇒        [[][[}{<]]][][]...
⊢H(<)⟹          [[}{][<]]][][]...
⊢C(<,0)⇒        [[}{[>}{]]][][]...
⊢H(>)⟹          [[}{[>][}{]]][][]...
⊢F(>,0)⇒        [[}{[[]>}{]]][][]...
⊢H(>)⟹          [[}{[[]>][}{]]][][]...
⊢C(>,0)⇒        [[}{[[}{<]}{]]][][]...
⊢K(<)⟹          [[}{[[][>]}{]]][][]...
⊢E(>,0)⇒        [[}{[[][]>}{]]][][]...
```

Figure 5. The first 116 well formed arrays $T_j^{(0)}$, $0 \le j < 116$.

The goal of introducing the present bracket manipulator was to scan the subarray $"[_i]_i[_{i+1}]_{i+1}"$ at least once in each interval of $2^{\theta(i)}$ steps, $i \geq 0$. We can express precisely what the t-th i.d. T^t is. T^0 is the initial array at time $t = 0$, and T^t results from an execution of SWITCH on T^{t-1}, for all $t > 0$. According to Lemma 3, t equals the number of occurrences of A,B,C,E,F-type local rewriting rules executed. We need to recognize $"[_i]_i[_{i+1}]_{i+1}"$ as being the correct sequence of cells, which, since the cells are tagged with nonindexed brackets in the manipulator proper, can not go by way of identifying the individual cells. For this purpose, the next Lemma establishes a *topology* for the well formed arrays. Before proceeding, we review a few facts about well formed arrays which are pertinent to the proof of that Lemma. By definition, and the discussion preceeding Lemma 3, the set of well formed arrays equals the set $\{T_j^{(0)} | j \geq 0\}$ defined by the level 0 expansion of $A(>,\infty)$.

$$A(>,\infty) = Y_1^{(0)}; \ Y_2^{(0)}; \ \ldots; \ Y_i^{(0)}; \ \ldots$$

and for all $i \geq 1$

$$T_{i-1}^{(0)} \xmapsto{Y_i^{(0)}} T_i^{(0)} \quad \text{with} \quad T_0^{(0)} = T^0.$$

By the definition of the initial array T^0, and those of the various procedures, each well formed array contains exactly one symbol from $\{<,>\}$ and, for each $i \geq 1$, exactly one symbol from $\{[_i, \{_i\}$ and exactly one symbol from $\{]_i, \}_i\}$. Recall that the indices are not really there but serve to identify the individual cells for the reader by distinguishing between the individual attached tags.

LEMMA 4. *Let* T *be a well formed array, and let* $\diamond \in \{<,>\}$ *and* $\alpha,\beta,\gamma \in \{[,],\{,\}\}^*$. *Then:*

(i) $T = \alpha \diamond \beta [_j]_k \gamma \Rightarrow k = j.$

   $T = \alpha [_k]_j \beta \diamond \gamma \Rightarrow k = j.$

(ii) $T = \alpha \diamond \beta]_j[_k \gamma \Rightarrow k = j+1$

   $T = \alpha]_k[_j \beta \diamond \gamma \Rightarrow k = j+1$

PROOF. We basically prove the Lemma by induction on the sequence of well formed arrays $T_j^{(0)}$, as defined by the level 0 expansion of $A(>,\infty)$, $j \geq 0$. To do so, we consider the initial segments $T_j^{(0)}[0: 2(i+1)]$, $j \geq 0$ and $i \geq 0$, in isolation and show by the claim below that they internally satisfy the Lemma. Viz., in executing the level 0 expansion of $A(>,i)$ to obtain T^{t_i} from T^0 the elements of the sequence of subarrays $T_0^{(0)}[0: 2(i+1)]$, $T_1^{(0)}[0: 2(i+1)]$,...

...., $T_{a(i)}^{(0)}[0: 2(i+1)]$, with $T_0^{(0)} = T^0$ and $T_{a(i)}^{(0)} = T^{t_i}$, will be shown to internally satisfy the Lemma. Since during the execution of $A(>,i)$ the final segment $T[2(i+1):\infty]$ is not changed at all, and T^0 satisfies the Lemma, the elements of the sequence of subarrays $T_0^{(0)}[2(i+1):\infty]$, $T_1^{(0)}[2(i+1):\infty]$, ...

..., $T_{a(i)}^{(0)}[2(i+1):\infty]$ do internally satisfy the Lemma. Because we have an overlap of one symbol between $T_j^{(0)}[0: 2(i+1)]$ and $T_j^{(0)}[2(i+1):\infty]$ for all j, $0 \leq j \leq a(i)$ with $T_0^{(0)} = T^0$ and $T_{a(i)}^{(0)} = T^{t_i}$, we can conclude that each well formed array $T_j^{(0)}$, $0 \leq j \leq a(i)$, satisfies the Lemma. Taking the limit for $i \to \infty$, that is, considering $A(>,\infty)$, it follows that the Lemma holds for all well formed arrays.

CLAIM. Let, for all $i \geq 0$, $X \in \{A,B,C,D,E,F\}$, $\diamond \in \{<,>\}$ and T,T' be well formed arrays such that $X(\diamond,i)$: $T[p:q] \mapsto T'[p:q]$, for some $p,q \geq 0$ and $T \xRightarrow{X(\diamond,i)} T'$, and let $Y_{\ell+1}^{(0)}$; $Y_{\ell+2}^{(0)}$; ...; $Y_{\ell+x(i)}^{(0)}$ be the level 0 expansion of $X(\diamond,i)$ with $T_{\ell+j-1}^{(0)} \xmapsto{Y_{\ell+j}^{(0)}} T_{\ell+j}^{(0)}$ for all j, $1 \leq j \leq x(i)$ with $T_\ell^{(0)} = T$ and

$T_{\ell+x(i)}^{(0)} = T'$. Then, for all j, $\ell \le j \le \ell+x(i)$, $T_{\ell+j}^{(0)}[p{:}q]$ internally satisfies the Lemma.

Proof of claim. *Base case* $i = 0$. Since for $i = 0$ the procedures are essentially but the local rewriting rules, we only have to verify that in the definitions of the various functions the subarrays left and right of the arrow internally satisfy the Lemma. Note that $[_I]_I \overset{\text{def}}{=} \epsilon$ for $i = 0$.

Induction. Assume, by way of contradiction, that for some $X(\diamond, i)$, with $X \in \{A,B,C,D,E,F\}$ and $\diamond \in \{<,>\}$ and $i > 0$, the claim does not hold. But in the execution of the level $i-1$ expansions of six of these functions with parameter i in the proof of Lemma 2, the other six cases being symmetrical, the displayed subarrays all satisfy the claim. Hence it must follow that a nondepicted subarray arising in the execution of the level 0 expansion of some $X'(\diamond', i-1)$, $X' \in \{A,B,C,D,E,F\}$ and $\diamond' \in \{<,>\}$, violates the claim. Regressing in this fashion all the way down to $i = 0$, we contradict the established base case, and the claim is proven. □ □ □

By the discussion preceeding the claim we have established the Lemma. □ □

LEMMA 5. Let T *be a well formed array and let* $\diamond \in \{<,>\}$ *and* $\alpha, \beta, \in \{[,],\{,\}\}^*$. *Then*

$$(T = \alpha \diamond [_j]_k \beta \quad or \quad T = \alpha[_k]_j \diamond \beta) \quad \Rightarrow \quad (k = j = 1).$$

PROOF. That $k = j$ follows already from Lemma 4. Considering the level 1 expansion of $A(>,\infty)$

$$A(>,\infty) = Y_1^{(1)}; \ Y_2^{(1)}; \ \dots; \ Y_j^{(1)}; \ \dots$$

and

$$T_{j-1}^{(1)} \overset{Y_j^{(1)}}{\Longrightarrow} T_j^{(1)} \quad \text{with} \quad T_0^{(1)} = T^0,$$

we observe that it follows from the definitions of the various procedures that, for all well formed arrays $T_j^{(1)}$, $j \geq 0$, the Lemma holds. Expanding each A,B,C,D,E,F function with parameter 1 to level 0, and examining the intermediate well formed arrays $T_{j'}^{(0)} \neq T_j^{(1)}$, j, $j' \geq 0$, yields the Lemma.

□ □

The Lemma's show that a certain topological connectedness between the indexed brackets is preserved throughout the array at all times, and that, in particular, in each well formed array $\alpha \diamondsuit \beta \ [_j]_k[_\ell]_m\gamma$ holds $k = j$ and $\ell = m = j+1$. So whenever there occurs a length four subarray "[][]" right of the headmarker the tagged cells concerned are in the correct consecutive left to right order. Without further proof we give a more exhaustive characterization of the topology. Let T be a well formed array. Then, for each $i \geq 1$, T satisfies precisely one of the following forms.

For $i = 1$:

$$\alpha[_1]_1\diamondsuit\beta$$

$$\alpha\diamondsuit[_1]_1\beta$$

$$\alpha[_1\diamondsuit]_1\beta$$

$$\alpha[_1>\{_j]_1[_2]_2\cdots[_{j-1}]_{j-1}]_j\beta \qquad \text{(some } j \geq 2)$$

$$\alpha[_1>\}_{j+1}\{_j]_1[_2]_2\cdots[_{j-1}]_{j-1}]_j\beta \qquad \text{(some } j \geq 2)$$

$$\alpha[_j[_{j-1}]_{j-1}[_{j-2}]_{j-2}\cdots[_2]_2[_1\}_j<]_1\beta \qquad \text{(some } j \geq 2)$$

$$\alpha[_j[_{j-1}]_{j-1}[_{j-2}]_{j-2}\cdots[_2]_2[_1\}_j\{_{j+1}<]_1\beta \qquad \text{(some } j \geq 2),$$

with the obvious modification for $j = 2$.

For each $i > 1$:

$$\alpha[_i]_i[_{i-1}\beta\diamondsuit\gamma$$

$$\alpha\diamondsuit\beta]_{i-1}[_i]_i\gamma$$

$$\alpha[_i\beta\diamond\gamma\}_i\{_{i-1}]_j[_{j+1}]_{j+1}[_{j+2}]_{j+2}\cdots[_{i-2}]_{i-2}]_{i-1}\delta \qquad \text{(some } j<i-1)$$

$$\alpha\diamond\beta\{_i]_j[_{j+1}]_{j+1}[_{j+2}]_{j+2}\cdots\cdot[_{i-1}]_{i-1}]_i\gamma \qquad \text{(some } j\leq i-1)$$

$$\alpha[_{i-1}[_{i-2}]_{i-2}[_{i-3}]_{i-3}\cdots\cdot[_{j+1}]_{j+1}[_j\}_{i-1}\{_i\beta\diamond]_i\delta \qquad \text{(some } j<i-1)$$

$$\alpha[_i[_{i-1}]_{i-1}[_{i-2}]_{i-2}\cdots[_{j+1}]_{j+1}[_j\}_i\beta\diamond\gamma \qquad \text{(some } j\leq i-1),$$

with the obvious modification for $i-3 \leq j \leq i-1$. Here $\diamond$ can be either "<"
or ">" and $\alpha,\beta,\gamma,\delta \in \{[\ ]\ \{\ \}\}^*$. Considering the fact that

$$T^0 = >[_1]_1[_2]_2\cdots[_i]_i[_{i+1}]_{i+1}\cdots$$

and that, by definition, for $i \geq 0$ and $t \leq t_i$,

$$T^t = \alpha\diamond\beta]_{i+1}[_{i+2}]_{i+2}\cdots[_{i+j}]_{i+j}[_{i+j+1}]_{i+j+1}\cdots,$$

$j \geq 3$, the formats express, but for the choosing of $\diamond$ as "<" or ">", the
format each well formed array T^t can have, by applying in sequence the re-
quirements for $i+1,i,\ldots,1$. According to Lemma 4, whenever we scan a subar-
ray "[][]" right of the headmarker, we know for sure that this is the subar-
ray "$[_i]_i[_{i+1}]_{i+1}$" for some $i \geq 1$. In the next Lemma we give an upper bound
on the number of steps, that is, executions of SWITCH, in between scanning
"$[_i]_i[_{i+1}]_{i+1}$" right of the headmarker, for all $i \geq 1$. To express the timing
we consider expansions of $A(>,\infty)$ of level i, $i \geq 1$:

$$A(>,\infty) = Y_1^{(i)}; \ Y_2^{(i)}; \ \ldots; \ Y_j^{(i)}; \ \ldots$$

and define for all $j \geq 1$

$$T_{j-1}^{(i)} \xmapsto{\ Y_j^{(i)}\ } T_j^{(i)} \quad \text{with} \quad T_0^{(i)} = T^0 .$$

The level 0 expansion of $Y_j^{(i)} = X(\diamond,i)$, with $X \in \{A,B,C,D,E,F\}$ and
$\diamond \in \{<,>\}$, is fixed and, but for the headmarker arguments, is the same

whether $\diamond = <$ or $\diamond = >$. Thus, by Lemma 3, the number of steps of M to execute $X(\diamond,i)$ equals the number of occurrences of A,B,C,E,F local rewriting rules in its level 0 expansion, and does not depend on the orientation of the headmarker $\diamond$, or the position j in the level i expansion of $A(>,\infty)$ where $Y_j^{(i)}$ occurs. We denote the number of steps, used by M, to execute $X(\diamond,i)$, by $T_X(i)$.

LEMMA 6. *There exists a function* S: $\mathbb{N} \to \mathbb{N}$ *such that for each* $t \geq 0$ *there is a* $t' > t$ *such that for some* $\alpha,\alpha',\beta,\beta' \in \{[,],\{,\}\}^*$ *and* $\diamond,\diamond' \in \{<,>\}$

(i) $T^t = \alpha \diamond [_1]_1 \beta \Rightarrow T^{t'} = \alpha' \diamond' [_1]_1 \beta'$ *and* $t'-t \leq S(1)$.

(ii) $T^t = \alpha \diamond [_1]_1 [_2]_2 \beta \Rightarrow T^{t'} = \alpha' \diamond' [_1]_1 [_2]_2 \beta'$ *and* $t'-t \leq S(2)$.

(iii) *For all* $i > 2$ *and* $x \in \{\epsilon,\{,\}\{\ \}$ *there is a* $x' \in \{\epsilon,\{,\}\{\ \}$ *such that:*

$T^t = \alpha \diamond x]_{i-2} [_{i-1}]_{i-1} [_i]_i \beta \Rightarrow T^{t'} = \alpha' \diamond' x']_{i-2} [_{i-1}]_{i-1} [_i]_i \beta'$ *and*

$t'-t \leq S(i)$.

Moreover, $S(i) = 2\,T_A(i) + T_F(i)$, *for all* $i \geq 1$, *is such a function.*

PROOF. Consider the level i expansion of $A(>,\infty)$, $i \geq 1$,

$$A(>,\infty) = Y_1^{(i)};\ Y_2^{(i)};\ \ldots;\ Y_j^{(i)};\ \ldots$$

and

$$T_{j-1}^{(i)} \overset{Y_j^{(i)}}{\longmapsto} T_j^{(i)} \text{ with } T_0^{(i)} = T^0.$$

Then $T_j^{(i)}$ is of the form $\alpha \diamond [_I]_I \beta$ or $\alpha [_I]_I \diamond \beta$, for all $j \geq 0$. All such $T_j^{(i)}$'s, with $Y_j^{(i)}$ not G,H,J or K local rewriting rules, are i.d.'s of M. We restrict attention to the particular subsequence $T_{j_0}^{(i)}, T_{j_1}^{(i)}, \ldots, T_{j_k}^{(i)}, \ldots$ for which $T_{j_k}^{(i)}$ is of the form $\alpha < [_I]_I \beta$ for all $k > 0$ and $T_{j_0}^{(i)} = T^0$. For each $k \geq 0$ there exists a sequence $Y_{j_k+1}^{(i)};\ Y_{j_k+2}^{(i)};\ \ldots;\ Y_{j_{k+1}}^{(i)}$ such that

(1)
$$T^{(i)}_{j_k} \xrightarrow{\quad Y^{(i)}_{j_k+1}; \ Y^{(i)}_{j_k+2}; \ \ldots; \ Y^{(i)}_{j_{k+1}} \quad} T^{(i)}_{j_{k+1}} \ .$$

By the use of the recursive expressions in Lemma 2 we can determine all such

sequences. Subsequently, we have to determine which such sequences take the

most steps to execute. So we first determine $T_X(i)$ for all $X \in \{A,B,C,D,E,F\}$.

It follows from Lemma 3 that $T_X(i)$ equals the number of occurrences of

A,B,C,E,F procedures in its level 0 expansion. We see from Lemma 2 that:

$$T_A(i) = T_A(i-1) + T_F(i-1)$$

$$T_B(i) = T_B(i-1) + T_F(i-1)$$

$$T_C(i) = T_C(i-1) + T_F(i-1)$$

$$T_D(i) = T_A(i-1) + T_E(i-1)$$

$$T_E(i) = T_B(i-1) + T_D(i-1) + T_E(i-1)$$

$$T_F(i) = T_C(i-1) + T_D(i-1) + T_E(i-1)$$

and $T_A(0) = T_B(0) = T_C(0) = T_E(0) = T_F(0) = 1$ and $T_D(0) = 0$. For this system

of recurrence equations with initial values we find $T_A(i) = T_B(i) = T_C(i)$,

for all $i \geq 0$, and consequently $T_E(i) = T_F(i)$, for all $i \geq 0$, which in its

turn yields $T_D(i) = T_A(i)$, for all $i \geq 1$. Hence for all $i \geq 1$:

(2) $$T_E(i) = T_F(i) \geq T_A(i) = T_B(i) = T_C(i) = T_D(i).$$

Now let $Y^{(i)}_{j_k+1}; \ Y^{(i)}_{j_k+2}; \ \ldots; \ Y^{(i)}_{j_{k+1}}$ be a sequence of functions as in (1).

Erasing the G,H,J and K procedures (because they do not contribute to the

number of steps it takes to execute this sequence, by Lemma 3) and replacing

all E's by F's and all B's,C's and D's by A's (because they take the same

number of steps for $i \geq 1$) the resulting sequences are $F(<,i)$, $A(<,i);A(>,i)$

and $A(<,i);F(>,i);A(<,i)$. So for $i = 1,2$, $S(i) = 2 T_A(i) + T_F(i)$ satisfies

the Lemma. For $i > 2$ we note that, for all $k \geq 0$ (with the obvious modifi-

cation for $k = 0$),

$$T_{j_k}^{(i)} = \alpha <[_{I-3}]_{I-3}[_{i-2}]_{i-2}[_{i-1}]_{i-1}[_i]_i \beta$$

$$\overset{Z_i}{\longmapsto} \alpha'[_{i-2}[_{I-3}]_{I-3} >x]_{i-2}[_{i-1}]_{i-1}[_i]_i \beta$$

with Z_i and x one of the following:

$Z_i \in \{J(<);A(>,i-3), K(<);A(>,i-3)\}$ and $x = \varepsilon$;

$Z_i = B(<,i-3)$ and $x = \{$;

$Z_i \in \{C(<,i-3), G(<);C(<,i-3), H(<);C(<,i-3)\}$ and $x = \}\{$.

In all cases, the execution time of Z_i is $T_A(i-3)$, which shows that

$S(i) = 2 T_A(i) + T_F(i)$ satisfies the Lemma for all $i > 2$ too. □ □

COROLLARY. *Let* $S: \mathbb{N} \to \mathbb{N}$ *defined by* $S(i) = 2 T_A(i) + T_F(i)$, *for all* $i \geq 1$.
Then:

(i) *For each* $t \geq 0$ *there exists a* t', $t < t' \leq t + S(1)$, *such that the*
   t'-*th* *i.d. of* M *has the form* $T^{t'} = \alpha <[_1]_1 \beta$ *for some* $\alpha,\beta \in \{ [,],\{,\} \}^*$.

(ii) *For each* $t \geq 0$ *there exists a* t', $t < t' \leq t + S(2)$, *such that the*
   t'-*th* *i.d. of* M *has the form* $T^{t'} = \alpha <[_1]_1[_2]_2 \beta$ *for some*
   $\alpha,\beta \in \{ [,],\{,\} \}^*$.

(iii) *For each* $i > 2$ *and* $t \geq 0$ *there exists a* t', $t < t' \leq t + S(i)$, *such*
   that the t'-*th* *i.d. of* M *has the form* $T^{t'} = \alpha >x]_{i-2}[_{i-1}]_{i-1}[_i]_i \beta$ *for*
   some $\alpha,\beta \in \{ [,] \{,\} \}^*$ *and* $x \in \{ \varepsilon,\{,\}\{ \}$.

It remains to determine S analytically.

LEMMA 7. $S(i) = \dfrac{\sqrt{2}+2}{\sqrt{2}} (1+\sqrt{2})^i + \dfrac{\sqrt{2}-2}{\sqrt{2}} (1-\sqrt{2})^i$, $i \geq 1$.

PROOF. From the system of recurrence equations, and the values for $i = 0$,

in the proof of Lemma 6, follows:

$$T_A(i) = 2\, T_A(i-1) + T_A(i-2), \text{ for } i > 2.$$

The solution for this homogeneous equation is of the form $T_A(i) = ax_1^i + bx_2^i$, where $x_{1,2}$ are the roots of $x^2 - 2x - 1 = 0$ and a and b follow from $T_A(1) = 2$ and $T_A(2) = 4$. So $x_{1,2} = 1 \pm \sqrt{2}$ and

$$a(1+\sqrt{2}) + b(1-\sqrt{2}) = 2$$

$$a(1+\sqrt{2})^2 + b(1-\sqrt{2})^2 = 4$$

yielding $a = 1/\sqrt{2}$ and $b = -1/\sqrt{2}$. Hence

$$T_A(i) = \frac{1}{\sqrt{2}} (1+\sqrt{2})^i - \frac{1}{\sqrt{2}} (1-\sqrt{2})^i, \quad i \geq 1.$$

From the system of recurrence equations, and the identities amongst the functions, it appears that $T_F(i) = T_A(i) + T_A(i-1)$ whence the expression for $S(i)$ follows. □ □

COROLLARY: $S(i) < 3^{i+1}$ *for all* $i \geq 1$. *Viz.*, $S(1) = 6$ *and* $\lim_{i \to \infty} S(i+1)/S(i) = 1 + \sqrt{2}$.

Of course we can obtain that $S(i) < 3^{i+1}$ by a cruder argument. The present analysis, however, is quite straightforward and precise. Running the bracket manipulator on a computer, by way of empirical verification, confirmed the first nine values of S.

2.4. The real-time simulator

Having set the stage in the preceeding sections, we now tie everything together to obtain the desired real-time simulator.

Let M be a one-head tape unit with a one-way inifinite tape divided into two tracks: the *tag track* and the *count track*. The finite control of M

has a special *register* containing the initial segment C[0:5] of the array

C[0:∞] representing the current count as in sections 2.1 -2.2. The single

head of M covers 14 squares and its *position* is the intersquare boundary

in the center. Initially, the head covers the leftmost squares, all squares

on the tape contain special blank symbols and the finite control is in a

distinguished *initial* state, in particular C[0:5] contains $\bar{0}$'s only. Since

M can always initialize previously unscanned squares, still containing

blanks , by keeping a parity bit in the finite control, we assume that the

tape is initially divided in the two tracks as follows. Number the tape

squares from left to right by $-7,-6,\ldots,0,1,2,\ldots$. Square i, i $\geq$ 0, con-

tains initially on the count track $c^0[i+6] = \bar{0}$ and on the tag track a tag

"[", if i is even, and a tag "]", if i is odd. So the initial situation can

be visualized as in Figure 6, with the initial headmarker ">" kept in the
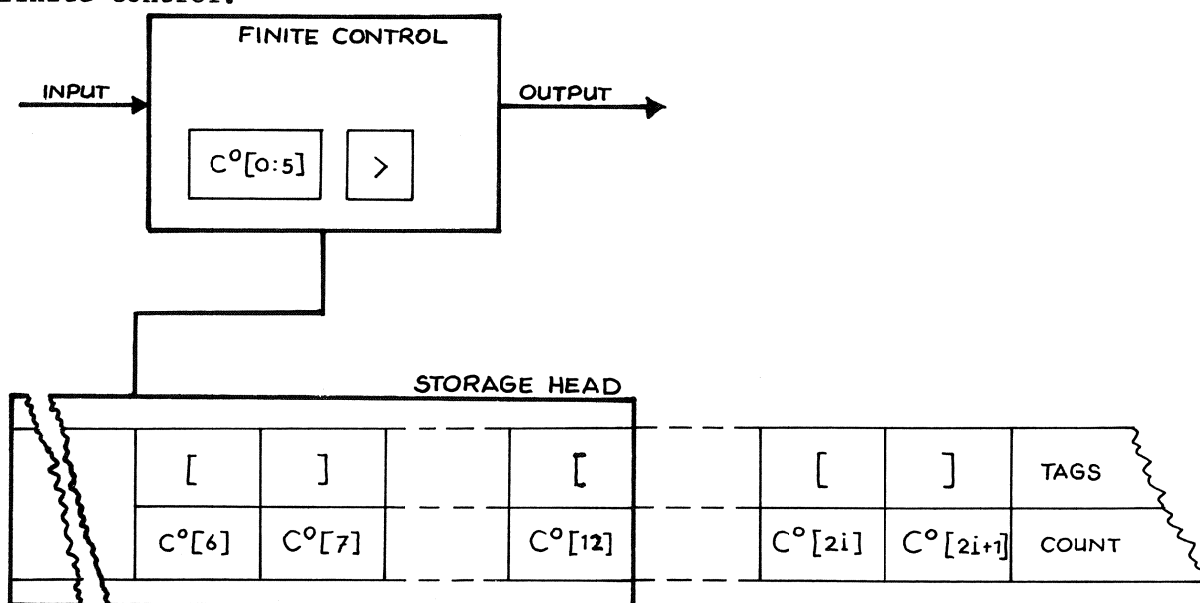
finite control.

Figure 6.

At each step the head rewrites the contents in the squares under scan, and

shifts left, right or not at all. Since the head shifts will be governed by

the local rewriting rules of the last section, the marker ">" or "<", positioned on the center intersquare borders of the scanned squares, can shift at most two squares left or right in a single step. Whether this marker is ">" or "<" can be maintained in the finite control; the initial marker is ">".

Each step of M consists of essentially two parts: first execute COUNT on the representation of the currently stored integer, check whether this integer is zero, and secondly execute SWITCH to switch cells containing digits of the integer representation. The information in the two tracks of a square may be thought of as a cell containing the current digit $C[i]$, which is tagged by the tag on the tag track.

To execute COUNT, M inspects the scanned cells right of the headmarker, so as to determine $I(t)$ in the t-th step, and also identify the squares containing $C[2i]$, $C[2i+1]$, $C[2i+2]$ and $C[2i+3]$ for all $i \in I(t)$. To this purpose first procedure COLLECT is executed. Let P be the current local tape contents, i.e. $P = \dfrac{\tau_1 \tau_2 \tau_3 \tau_4 \tau_5 \tau_6 \tau_7}{\gamma_1 \gamma_2 \gamma_3 \gamma_4 \gamma_5 \gamma_6 \gamma_7}$ is the tape contents on the seven squares right of the headmarker.

Procedure COLLECT (P):

Let the seven squares right of the headmarker contain the string $\tau_1 \tau_2 \ldots \tau_7$ on the tag track and the string $\gamma_1 \gamma_2 \ldots \gamma_7$ on the count track. Then we distinguish essentially four cases:

(a) $\tau_1 \tau_2 = [] \ \& \ \tau_3 \tau_4 \neq [] \ \Rightarrow \ I(t) = \{0,1,2\} \ \& \ C[6:7] = \alpha_1 \alpha_2$

(b) $\tau_1 \tau_2 \tau_3 \tau_4 = [][] \ \Rightarrow \ I(t) = \{0,1,2,3\} \ \& \ C[6:9] = \alpha_1 \alpha_2 \alpha_3 \alpha_4$

(c) $\tau_1 \tau_2 \tau_3 \tau_4 \tau_5 =][][] \ \Rightarrow \ I(t) = \{0,1,i\}, \ i > 3, \ \& \ C[2i:2i+3] = \alpha_2 \alpha_3 \alpha_4 \alpha_5$

$\tau_1 \tau_2 \tau_3 \tau_4 \tau_5 \tau_6 = \{][][] \ \Rightarrow \ I(t) = \{0,1,i\}, \ i > 3, \ \& \ C[2i:2i+3] = \alpha_3 \alpha_4 \alpha_5 \alpha_6$

$\tau_1 \tau_2 \tau_3 \tau_4 \tau_5 \tau_6 \tau_7 = \}\{][][] \ \Rightarrow \ I(t) = \{0,1,i\}, \ i > 3, \ \& \ C[2i:2i+3] = \alpha_4 \alpha_5 \alpha_6 \alpha_7$

(d) None of (a)-(c) ⇒ $I(t) = \{0,1\}$.

Modulo the correctness of the implications in the definition of COLLECT, which remain to be proven, the execution of COLLECT(P) in the t-th step of M both determines $I(t) = \{i_\ell, i_{\ell-1}, \ldots, i_1\}$, $i_\ell > i_{\ell-1} > \ldots > i_1$, and identifies the locations where $C[2i_j]$, $C[2i_j+1]$, $C[2i_j+2]$ and $C[2i_j+3]$ currently reside, $1 \leq j \leq \ell$. Since these locations are either under scan on the tape, or in the finite control, viz. $C[0:5]$, the machine can in the t-th step execute COUNT(t,δ) = UPDATE(i_ℓ); UPDATE$(i_{\ell-1})$; $\ldots$; UPDATE(i_1); INPUT(δ) by executing the consecutive mappings in the decomposition on the relevant subarrays of $C[0:\infty]$, without explicitly knowing the value of t. Thus, in each single step, starting from the all-blank tape with the initial headmarker ">" positioned at the left end, the one-head tape unit M will do all of the following.

Procedure STEP:

Step 1. Initialize both tracks of right adjacent previously unscanned squares, still containing primeaval blanks, by writing the correct square bracket on the tag track (check and update parity count in the finite control) and a blank "$\bar{0}$" in the count track of such a square.

Step 2. Execute COLLECT(P).

Step 3. Let the current value of I determined by step 2 be $\{i_\ell, i_{\ell-1}, \ldots, i_1\}$ with $i_\ell > i_{\ell-1} > \ldots > i_1$. READ the current value of δ from the input terminal and execute COUNT (current step, δ), that is,

UPADTE(i_ℓ); UPDATE$(i_{\ell-1})$; $\ldots$; UPDATE(i_1); INPUT(δ).

Step 4. WRITE "count equal zero" or "count unequal zero" to the output ter-

       minal, depending on whether or not $C[0] = \bar{0}$, for the $C[0]$ resulting

       from step 3.

Step 5. Execute SWITCH. That is, switch the contents of the scanned squares,

       considering the combined contents of the tag track and the count track

       on a square as a single package. Interchange these packages amongst

       squares, shift the head position and change the brackets and head-

       marker, governed by the current headmarker, head position on the scan-

       ned squares and scanned brackets alone.

PROPOSITION 4. *The constructed one-head tape unit* M *is oblivious and real-time simulates the quintessential counter.*

PROOF. The one-head tape unit M is oblivious since the headmovement is go-
verned by the tag track, and the headmarker, independent of the input. At-
taching imaginary indexes $i = 3,4,\ldots$ to the initial tag track contents, a shift
of 2 from the ones in the initial i.d. in the previous section, the executions of
SWITCH preserve that pairing of $C[2i]$ with opening bracket indexed i and
of $C[2i+1]$ with closing bracket indexed i, $i \geq 3$. Since $C[0:5]$ resides im-
mobile in the finite control, Lemma's 2-5 ensure that the identification of
array elements by COLLECT(P) in each step remains correct under the inter-
change of the mobile array elements of C on the count track by SWITCH. In
the t-th step, for all $t \geq 1$, COLLECT(P) determines the value $I(t)$ of the
parameter selection function I, as well as the high to low order of elements
in $I(t)$. By Lemma 6 and Corollary, for each $t \geq 0$ and each index $i \geq 2$,
there exists a step t', $t < t' \leq t + S(i-1)$, such that $i \in I(t')$. By the de-
finition of COLLECT(P), $\{0,1\} \subseteq I(t)$ for all $t \geq 1$. Since it follows from
Lemma 7 that $S(i-1) < 3^i$ for all $i \geq 2$, the oblivious one-head tape unit M

real-time simulates the quintessential counter by Propositions 2 and 3. ☐

Let C be any k-counter machine, k ≥ 1. Clearly, C can be thought of as a finite control connected with k quintessential counters $S_1, S_2, \ldots, S_k$. At each step the finite control of C reads an inputcommand from the input terminal if it is in a polling state, checks each S_i, 1 ≤ i ≤ k, for zero contents, and governed by this information issues input commands "add δ_i", $\delta_i \in \{-1,0,1\}$, to each S_i, 1 ≤ i ≤ k, and writes an output string to the output terminal. In the spirit of Proposition 1, we can real-time simulate C by an oblivious one-head tape unit M_C, which is just like M, but with k count tracks (one for each quintessential counter) and one tag track. Storing the first six digits of the representation of each count in the finite control, which is connected to the input- and output terminals through C's original finite control, we finally obtain:

THEOREM. *Each multicounter machine can be real-time simulated by an oblivious one-head tape unit using logarithmic space.*

PROOF. By Propositions 1 and 4. That the space used is logarithmic in the simulated number of steps follows since the head is centered immediately left of the square containing tag "$]_{i+1}$" for the first time after executing A(>,i), which takes $T_A(i)$ steps. To clean up some final details: we can get rid of the fat head, covering 14 squares and sometimes shifting its center two squares in a single step, by cutting out a piece of tape of 14 squares and buffering it in the finite control. The remains of the tape are glued together and the contents of the buffered piece are swapped from the buffer to the scanned tape square and vice versa, according to the desired head motion, cf. the speed up technique in [3]. ☐

On the required bits. Although the preceeding simulation and its proof may not seem easy, the algorithm which does the work is pretty simple. As it happens, we are also frugal in the number of bits. On information theoretical grounds we require about $k \log_2 2n$ bits to represent any k-tuple of integers of absolute values up to n. In the exhibited simulation, we can use four bits for each digit of a count, need not more than $\log_2 n$ digits for each count, and since there are but four tags, each tag can be encoded in two bits. Therefore, we use at most about $(4k+2) \log_2 n$ bits to represent k counts of absolute values at most n. By restricting the most significant non-zero digit to absolute value 1, and appropriately modifying the mappings UPDATE and INPUT, everything goes through as before but code(c) $\subseteq \{-2,-1,0,1,2,\overline{1},\overline{0},\overline{1}\}^\infty$, c $\in \mathbb{Z}$. Thus we only have to use $(3k+2) \log_2 n$ bits to represent k counts of absolute values at most n. Using only digits from $\{-2,-1,0,1,2,\overline{0}\}$ also suffices, but complicates the proof. How good a real-time algorithm is can be measured in the size of the storage alphabet used. Realizing that actual machines use a constant size storage alphabet, we observe that a large, although finite, storage alphabet in an algorithm implies a greater constant delay. That is, the reverse of a speed-up by decreasing the alphabet size. At the cost of a detoriation of the constant delay, implicit in the real-time solution presented, we can do better than using $(3k+2) \log_2 n$ bits. Using in sections 2.1 and 2.2 an analogous redundant symmetric r-ary representation, based on the digits $-r, -r+1, \ldots, -1, 0, 1, 2, \ldots, r-1, r$, we can get the bit count down to about $(1 + 4 / \log_2 r) k \log_2 n$ bits for maintaining k counts of absolute value at most n. The implicit constant delay, however, rises proportional to $\log r$. In the limit, for $r \to \infty$, we achieve about the information theorical minimum in bits, but the constant delay goes to infinity, that is, it takes in-

finite time to execute a single step.

Note, however, that for no fixed finite storage alphabet a real-time simulation of but a single counter on an oblivious multitape Turing machine can reach the information theoretical bit minimum. Such a simulation must use $\Omega(\log n)$ size representations for counts of size n, and we can argue that for each n there must be at least log n representations. Hence we use at least $\log_2 2n + \log_2 \log n$ bits per count.

On the size of the fat head. In the simulation a head covering 10 squares suffices, which can be shown by a slight complication of the proof. Also, the head shift in a single step of M need not exceed one square.

On the initially zero counts. As argued subsequent to the proof of Proposition 3, the assumption of initially zero counts is not essential. The Theorem holds also for multicounter machines with each count initialized to an arbitrary integer.

3. CONCLUSION

For various theoretical and practical reasons, multitape Turing machines, restricted in one or more resources, serve as a standard against which to calibrate the power of other devices, or to compare the power among themselves under different resource restrictions. The commonly considered resources are time, space, number of tapes/storage heads and oblivious versus nonoblivious. The present simulation is, perhaps, the first one which is optimal in all of these resources at once: the use of no resource can be improved by relaxing on the other resource restrictions. Apart from the fact that the simulating device is real-time, oblivious and uses but a single storage head, it is worthwhile to recall that there *do not exist*

on-line Turing machines using $S(n) \in o(\log n)$ space, $S(n)$ unbounded [4]. Thus, the simulation is performed by the simplest (with respect to the considered resources) Turing machine which is not an outright finite automaton. Another resource, which is sometimes considered, is the number of head reversals. Again, it is easy to see that each multitape Turing machine needs, in the worst case, a linear number of head reversals to on-line simulate a counter machine, as does the presented simulation. (Although a multihead Turing machine can simulate a multicounter machine without head reversals [8], the simulation of such a device by a multitape Turing machine needs a linear number of head reversals.)

Some immediate applications. In a computation using k stacks we may want to keep track of which pairs out of the k stacks are of equal height at any time. Without slowing down the computation, we formerly needed k-1 stacks for doing so. Using the present method we need but one extra oblivious one-head tape unit, or two extra oblivious pushdown stores. A single pushdown store does not suffice. Similarly, we can keep track of the headpositions in multihead Turing machine computations.

Number representations. The reader may appreciate the following comment of *John Locke* on the intimate relation between counting and number representation. "For he that will count 20, or have any *idea* of that number, must know that 19 went before, with a distinct name or sign for every one of them, as they stand marked in their order; for whenever this fails a gap is made, the chain breaks and the progress in numbering can go no further. So that *to reckon right, it is required:* (1) that the mind distinguish carefully between two *ideas,* which are different one from another by the addi-

tion or subtraction of one unit; (2) that it retain in memory [a systematic method for deriving] the names or marks of the several combinations, from an unit to that number, and that not confusedly or at random, but in the exact order that the numbers follow one another; in either of which, if it trips, the whole business of numbering will be disturbed, and there will remain only the confused *idea* of a multitude, but the *ideas* necessary to distinct enumeration will not be attained to. "

The one and only basic reason to denote numbers at all is for the purpose of comparing then, of whether the one is greater than the other, for without this capability no arithmetic is possible and with it all arithmetic is possible. Thus we must be able to distinctly represent all numbers, and if we have representations for all numbers up to a given one, then we must be able to derive the next one, or previous one, from the given one, while having a designated point of reference or benchmark number. This is the task expressed in the notion of a counter machine, and multicounter machines enable us to do arithmetic. The exhibited optimal implementation embodies a new representation for multituples of integers suitable for exercising that basic activity using minimal resources. Thus, for each $n = (n_1, n_2, \ldots, n_k) \in \mathbb{Z}^k$, $k \geq 1$, each such representation for n consists of a linear string of symbols, and is about as compact as possible. Such a representation has a distinguished *access position* p, and by considering only the, say, three symbols centered on the access position we can

(i)   add any vector $\delta = (\delta_1, \delta_2, \ldots, \delta_k) \in \{-1, 0, 1\}^k$ to n to obtain such a representation for $n + \delta$;

(ii) for all i, $1 \leq i \leq k$, determine whether $n_i + \delta_i = 0$;

(iii) determine the new access position $p' \in \{p-1, p, p+1\}$, which is also independent of n and δ. In   m   successive additions the distance be-

tween the leftmost and rightmost intermediate access pointer position

is $O(\log m)$, for all $m > 1$.

Note that Gray codes, as representations of integers, have vaguely similar

properties for the case $k = 1$. There, the representation of $n \pm 1$, $n \in \mathbb{Z}$,

can be obtained from the representation of n by changing a single symbol.

However, the symbol in the representation which must be changed to obtain

$n + 1$ from n can lie arbitrary far from the symbol which must be changed to

obtain $n - 1$ from n. Moreover, these positions depend on n and whether we

add or subtract, and do not allow us to test n for 0. The representation de-

rivable from the simulation in [1] is closer to the one above, for the case

$k = 1$, but the new access position p' in (iii) depends on n and δ. None of

these representations have any of the properties (i)-(iii) in case $k > 1$.

Further research may go in the direction of:

Augmented counter machines. Apart from the basic one-step multicounter com-

mands, several other one-step commands can be synthesized using conceiled

auxiliary counters, such as tests for equality amongst counters (by main-

taining all differences on auxiliary counters). It is known [2] that the com-

mands "set counter i to zero" or "set counter i to the value of counter j"

($i \neq j$) can not be synthesized as one-step instructions on a multicounter ma-

chine. At the end of section 2.2 we noted that the requirement of initially

zero counters was not essential for the present simulation. It can be proved

(elsewhere) that with a suitable embellishment the present simulation can al-

so support the one-step instruction "set counter i to the value of counter

j" ($i \neq j$). Define an *augmented counter machine* (ACM) just like a multicounter

machine but with the one-step input command "set counter i to the value of

counter j" (for any pair of counters i,j) added and any initial counter con-

tents in $\mathbb{Z}$ allowed. Such a machine can execute quite powerful instructions in one step. For example:

$$L: \textit{if} \ (x<y \ \& \ y\geq c) \ \textit{then} \ (x \leftarrow z; \ z \leftarrow d) \ \textit{else} \ (x \leftarrow y; \ \textit{goto} \ L') \ \textit{fi}$$

with x,y,z integer variables, c,d integers and L,L' labels, is a one-step instruction for an ACM.

THEOREM. *Each augmented counter machine can be real-time simulated by an oblivious one-head tape unit in logarithmic space.*

Uniform space complexity. Viewed in space-time, the bracket manipulator head describes an interesting curve. This is perhaps best expressed by stating that the two dimensional space-time trajectory described by the centre of the greatest tape segment, delimited by brackets with indices j, $j' \leq i$, is the same as that described by the centre of the greatest tape segment, delimited by brackets with indices j, $j' \leq i-1$, $i > 1$, subsequent to multiplying the time scale of the latter by $S(i)/S(i-1)$ and the space scale of the latter by $i/(i-1)$. This shows that the number of distinct squares visited in each time interval of n steps, for all $n \geq 1$, is $\theta(\log n)$. Generalizing this observation, we say that a multitape Turing machine M uses *uniform logarithmic space* if, for any unbounded input sequence, the total number of distinct squares, visited on M's storage tapes, for each interval of n steps, for all $n \geq 1$, is $O(\log n)$. It can be shown (elsewhere) that each multitape Turing machine using uniform logarithmic space can be real-time simulated by an oblivious one-head tape unit using uniform logarithmic space.

Oblivious simulations. It seems to us that also the converse of the maxim leading to Proposition 1 holds generally. Viz., if we can simulate arbitra-

ry many storage devices by a fixed number of, possibly different, devices then we can do so obliviously retaining the same resource bounds. The point here is that if the multitude of head movements of an arbitrary number of heads can be accommodated by the motion of a fixed number of heads, then there is no reason to suppose that any trajectory of the latter can make significant use of particular input streams.

REFERENCES

[1] M.J. FISCHER & A.L. ROSENBERG, *Real-time solutions of the origin-crossing problem*, Math. Systems Theory 2 (1968), 257-264.

[2] P.C. FISCHER, A.R. MEYER & A.L. ROSENBERG, *Counter machines and counter languages*, Math. Systems Theory 2 (1968), 265-283.

[3] J. HARTMANIS & R.E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc. 117 (1965), 285-306.

[4] J.E. HOPCROFT & J.D. ULLMAN, *Some results on tape-bounded Turing machines*, Journal ACM 16 (1967), 168-177.

[5] M. MINSKY, *Recursive unsolvability of Post's problem of tag and other topics in the theory of Turing machines*, Ann. of Math. 74 (1961), 437-455.

[6] N. PIPPENGER & M.J. FISCHER, *Relations among complexity measures*, Journal ACM 26 (1979), 361-384.

[7] C.P. SCHNORR, *The network complexity and Turing machine complexity of finite functions*, Acta Informatica 7 (1976), 95-107.

[8] P.M.B. VITÁNYI, *Efficient simulations of multicounter machines*, Proc. 9-th ICALP, Lecture Notes in Computer Science, Springer Verlag, 1982, 546-560.

69 F 11
69 F 13
69 F 22
69 B 31
69 E 10
69 E 40
94 B 60
60÷00