# On Pointers versus Addresses


Amir M. Ben-Amram and Zvi Galil*

Tel-Aviv University — Tel-Aviv University and Columbia University


LISP programmers know the value of everything, but the cost of nothing.

Alan Perlis


ABSTRACT

What is the cost of random access to memory? We address this fundamental problem by studying the simulation of random addressing by a machine which lacks it, a "pointer machine". We formulate the problem in the context of high level computational models, allowing the use of a data type of our choice. A RAM program of time $t$ and space $s$ can be simulated in $O(t \log s)$ time using a tree. To enable a lower bound proof, we formalize a notion of *incompressibility* for general data types. The main theorem states that for all incompressible data types an $\Omega(t \log s)$ lower bound holds.

Incompressibility trivially holds for strings but is harder to prove for a powerful data type. We prove incompressibility for the real numbers with a set of primitives which includes all functions which are continuous except on a countable closed set. This may be be the richest set of operations considered in a lower bound proof.

We also show that the integers with arithmetic $+, -, \times$ and $\lfloor x/2 \rfloor$, any Boolean operations and left shift are incompressible. The situation is reversed once right shift is allowed.



**Categories and Subject Descriptors:** E.1 [Data Structures]; F.1.1 [Models of Computation]; F.2.0 [Analysis of Algorithms and Problem Complexity]: General

**General Terms:** Algorithms, Theory

**Additional Key Words and Phrases:** Incompressibility, Random Access Memory, Pointer Structures


---


* Research of this author supported in part by NSF grants DCR-85-11713 and CCR-86-05353.

## 0. INTRODUCTION

What should be the capabilities of an abstract "computer" for theoretical study? This question is of fundamental importance in complexity theory. The quest for bounds on the complexity of problems calls for a machine model which is both realistic and theoretically accessible. The random access machine [3] seems to be the machine model in widest use. The general notion is that we want the RAM to model our experience in conventional programming. Thus we let it handle "items" which are of a data type suitable for our needs: integers, real numbers etc. We also let it use *indirect addressing*, which is the feature to which the RAM actually owes its name: the ability to access an arbitrary memory location, selected by an integer *address*. This corresponds to using arrays in a conventional programming language.

The selection of the data type that the machine should use—its data items and the primitive functions that may be applied to them—is an important decision, and since the first use of the RAM for complexity analysis, authors have considered this question, attempting to justify or evaluate possible choices, the data usually being integers [13,17,29], sometimes of limited size [15], or real numbers [36,30,9].

None of these authors considered the exclusion of the basic capability of indirect addressing. However, including it in our instruction set is a very important decision, since the use of *indirect addressing* in problems whose input and output are unrestricted data, which can be used—directly or indirectly—for addressing, makes the RAM very powerful (and hard to analyze). It allows for striking algorithms, such as distribution sort and table lookup by hashing [20].

Only in 1977 did Tarjan suggest the exclusion of this capability from our model in order to obtain non-linear lower bounds for a data structure problem[38]. His machine model was first called "the reference machine" and later got the more intuitive name "pointer machine" [39]. This name suggests its replacement for RAM-type indirect addressing; it accesses memory locations by means of *pointers*. As in high-level languages such as LISP and Pascal, pointers are not numbers and are not subject to the same instructions.

Tarjan did not consider the power of pointer manipulation in general nor the effect of this change on the RAM, but rather used this machine as a context for developing a lower bound for a specific problem, that of maintaining disjoint sets on-line. To this end, he considered mainly the structures held in the machine's memory, while forcing the machine to operate on its data symbolically. However, in realistic algorithms, computation with the input values is frequently of value.

A natural framework for presentation of algorithms is often derived from an actual programming language. We have already noted that the indirect addressing feature appears in most high-level languages, and thus forms a part of the framework usually assumed. We also noted that the *pointer* model is quite common, and in one language, LISP 1.x [37], this is the only method of memory access, which means that users of this language actually view the underlying machine model not as the common RAM but as a different machine which we shall formalize and call the **LISP machine**. Actually many natural, well-known algorithms are designed in the framework of pointer structures.

Our commitment to realistic algorithms motivates us to consider various data types, since the algorithm designer tends to use a data type convenient for his purpose; that is, we may assume the basic data items to be integer, real etc. and the choice of primitive operations may also be varied. In contrast with common practice, we do not define our models with a specific data type; our aim is to cope with this variety. In fact, we wish to formulate concrete definitions of generalized computational models so that the liberty, which one usually takes in describing algorithms, will be reduced to specifying a model in our class. This may also allow such algorithms to compare with compatible lower bounds. The basic model consists of a "CPU"—a device which executes a program and has access to input and output tapes and a finite set of registers, and a "memory unit" which holds information for use by the CPU. The memory model makes the difference between a RAM and a LISP machine, while their CPU uses the same data type which may be varied. Our main results are tight bounds on the degradation in performance suffered by the RAM in its transformation to a LISP machine. The bounds are clearly affected by the choice of the data type.

A *data type* $\mathcal{T}$ is defined by the *domain* of data values and the *primitive function set* which specifies the basic operations that the machine can apply to its data. We distinguish the *primitive set* of a data type from the *instruction set* of a machine; a machine model such as the RAM may be associated with various data types, and a data type may be put to use in various models. Symbolically, $\mathcal{T} = (\mathcal{D}, \mathcal{F}, <)$, where $\mathcal{D}$ is the domain, $\mathcal{F}$ the primitive function set and $<$ an order predicate on $\mathcal{D}$. The machines considered in this paper are of the "high level" breed: they can add, subtract and compare numbers and input or output an arbitrary value. Since we are free to choose the data type, much power can be encompassed in the primitive operations on the data. This is an insurmountable barrier to lower bound proofs. We overcome this problem by defining a class of *incompressible* data types. The *incompressibility* property is inherent in the data type; it is not related to a particular machine model and therefore is a general tool for deriving lower bounds, not only for the LISP machine. By analyzing several data types with respect to incompressibility, we justify the assumption of this property in propositions pertaining to a wide range of models; moreover, incompressibility suggests itself as a criterion of the plausibility of a data type.

The models are defined in §1. In §2 we give a strict definition to the problem of simulating a $\mathcal{T}$-RAM by another $\mathcal{T}$-machine. We consider the general problem of simulating a RAM program (with respect to its input and output). To handle this general problem, we define a specific problem for solution by the simulator. We show that the complexity of this solution determines the most general bounds on complexity of an on-line simulation of the RAM. This specific problem is called RASUS—random access storage unit simulation. It requires the simulator to act as a storage unit accepting LOAD and STORE commands. Its task is to answer a LOAD command by the value last stored in the given address.

On LISP machines, this task is easily accomplished using a binary search tree. The stored data are added to the tree with their address as key, and retrieved in the usual fashion. By using balanced trees we get our upper bound, $O(t \log s)$.

This upper bound seems bound to be optimal. How can we store $n$ items in a linked structure of out-degree 2, without having paths of length at least $\log n$? Indeed, the upper bound is optimal if our "items" have an atomic nature, as do symbols [39]. However, once we are allowed to operate on our data with non-elementary functions, that is, *compute* with them, it is no longer true that each input value must be stored separately. The machine may encode the input in its memory in any way that allows it to decode the information it may need. An example of how a powerful data type enables a faster simulation is given by the data type of integers with *shift* (§6), where we have a simulation in $O(t\alpha(s))$ time where $\alpha$ is a (very slow increasing) functional inverse of Ackermann's function. Indeed, this speed is gained by encoding the data in a compact structure. This seems as an unfair trick. In the area of low-level models, such as Turing machines, lower bounds have been based on the knowledge that a string of length $n$ may require $n$ storage cells for representation [28]. Our new notion of *incompressibility* characterizes high-level data types in this respect. The precise definition, in order to characterize the data type and not a particular machine, is given in terms of decision trees that compute functions on $\mathcal{D}^n$. The definition states that finite decision trees over an *incompressible* data type cannot encode $\mathcal{D}^n$ in $\mathcal{D}^m$ for $m < n$.

Our main theorem states that for such a data type, the machine cannot encode its input in a small memory graph, which entails the desired lower bound. The proof shows that encoding and decoding a memory graph is equivalent to transforming data tuples, as in the definition of incompressibility.

Finite data types provide a simple example of incompressibility, which is (implicitly) the fact used in the theory of Kolmogorov complexity ([22,28]). The power of the new definition is in its applicability to more sophisticated data types, and we proceed to show that useful data types are incompressible. The first data type we deal with is the real numbers. For this data type, we define a class of primitive functions which is larger than any practical set, and obviously stronger than the sets assumed in some older lower bound proofs [6,8,36,30], as well as other attempts to characterize real computable functions [9]. An important representative is the well-known *floor* function.

Denote by $\mathcal{F}$ the set of functions $f : \Re^k \to \Re^m$, for $k, m > 0$, such that for some countable, closed set $C \subset \Re^k$, $f$ is continuous in $\Re^k - C$. We show that the data type of real numbers where all primitive functions are in $\mathcal{F}$ is incompressible. The proof makes use of the topological invariance of dimension, together with Baire's category theorem.

The last result is extended to the field of rationals as well. This relates to previous work ([8,3]) in the same fashion as our result about the reals.

We then proceed to the most important of all, the integer data type. As mentioned above, we show that with a suitable instruction set, the data type is compressible. However, once we exclude *right shifting*, we get an incompressible data type, allowing (besides the standard operations) multiplication, all Boolean (bitwise) operations, unconstrained left shift and integer division by 2 (constrained right shift) or even exponentiation. Actually we add all the operations assumed in the references, as long as the right shift is excluded ([13,17,29]). The proof of incompressibility draws from the intuition that all these operations cannot retrieve information about the lower bits of a number that is "buried" in higher bits.

We conclude with historical notes on previous study of pointer machines.

A preliminary version of some of the results appeared in the first author's M.Sc. thesis [5], and as an abstract in the FOCS symposium [6].

## 1. DEFINITION OF THE MODELS

A *high level model of computation* (or *machine*) consists of a read-only input tape, a write-only output tape, a finite number of registers and (usually) a memory. Each of the input and output tape squares, as well as the memory cells, contains one item of the *ground data type* or *domain* of the machine. A random access machine (RAM) is a high level model where the memory cells are addressed by non-negative integers[1]. It is assumed that the intersection of the domain and $\mathbb{N}$ is a prefix of the latter, possibly infinite. We shall denote the number of registers by $N_{reg}$ and the registers themselves by $R_1$, $R_2$,...,$R_{N_{reg}}$. A well known fact about the RAM is that the actual number of registers is not important for our analysis, since memory cells can be used instead of missing registers, provided that we have a certain (fixed) minimum number of registers needed to execute the necessary instructions. This replacement will cost us only a constant factor in running time. We make it a general requirement of high level models that this property hold. Thus in describing any single program for a high level machine we shall impose no limit on the number of registers used, but only demand that their number be bounded (which is always the case for a fixed program).

The program for the machine is not stored in memory. Thus we are assuming the program does not modify itself. It consists of a (finite) sequence of (optionally) labeled instructions. The set of instructions consists of

| | | |
|---|---|---|
| 1. *I/O instructions*: | $READ$ | $R_i$ |
| | $WRITE$ | $R_i$ |
| 2. *Standard register operations*: | $SET$ | $R_i, constant$ |
| | $SET$ | $R_i, R_j$ |
| 3. *Flow control instructions*: | $JUMP$ | $label$ |

[1] $\mathbb{N}$ denotes the set of non-negative integers.

$$\begin{array}{ll} JEQL & R_i, R_j, label \\ HALT & \end{array}$$

4. *Memory access instructions.*

For the RAM, the last group includes the familiar

$$\begin{array}{ll} LOAD & R_i, R_j \\ STORE & R_i, R_j \end{array}$$

which access the memory cell whose address is in $R_j$. This is *indirect addressing.* To access a cell of constant address, we can SET a register to the address. Thus we may dispense with direct load and store. The address must belong to $\mathbb{N}$; otherwise the program is invalid and will not be considered. We use the notation $(a)$ for the contents of the cell addressed by $a$.

A fifth group of instructions is the *type-specific instructions*, also called *arithmetic* (although they may not be related to conventional arithmetic operations). This is a set of instructions which operate on registers only and implement functions on the machine's *domain.* All these instructions have the form

$$R_i \leftarrow f(R_{i_1}, \ldots, R_{i_k}), \quad 0 < i, i_j \leq N_{reg} \text{ and } f \in \mathcal{F}$$

where $\mathcal{F}$ is the *primitive function set* of the machine. An additional domain-dependent instruction is

$$JGTR \quad R_i, R_j, label$$

which compares data elements and branches to *label* if the value in the register $R_i$ is greater than that in $R_j$. We do not impose any restrictions on the "greater than" relation except that the integers included in the domain are supposed to retain their natural order.

The ordered triple $\mathcal{T} = (\mathcal{D}, \mathcal{F}, <)$ of the machine's domain, primitive function set and order relation is called the *data type* of the machine. A machine of type $\mathcal{T}$ is called a $\mathcal{T}-$machine. A *parametric* family of machines is defined by a family of types $\mathcal{T}_i = (\mathcal{D}_i, \mathcal{F}, <)$, where the set of instructions is fixed though their semantics are necessarily dependent on the domain. This allows us to write one program which will run on the entire family.

The semantics of a program need not be explained in detail. It is assumed that the machine starts up with all registers zeroed and a blank memory (which, for a RAM, is defined to be all zeros). The machine then advances, according to program instructions and input data, between instantaneous states described by the contents of the registers, the program counter and the contents of non-blank memory (which is always finite).

**Memory Structure**

$$\begin{tikzcd}
R_1 \arrow[d] & R_2 \\
ⓐ \arrow[r, Rightarrow] \arrow[d] & ⓑ \arrow[d] \\
ⓒ \arrow[r, Rightarrow] & ⓓ
\end{tikzcd}$$

**Missing Arcs**

CDR $b$
CAR $c$
CAR $d$
CDR $d$
REG(2)

**Figure 1.1.**

We are now ready to define the **LISP machine** (LM). This is a high level machine whose *storage model* is a directed graph. To be specific, its memory is modeled as a finite set $\mathcal{S} \subset \mathcal{D} \cup \mathcal{P}$, $\mathcal{P}$ being the set of *dotted pairs* which is an infinite set satisfying $\mathcal{P} \cap \mathcal{D} = \emptyset$. This set is augmented with two operators $\mathrm{CAR}, \mathrm{CDR}\colon \mathcal{P} \to \mathcal{S}$, associating to each *dotted pair* two "pointers" to other "memory locations". We select a special constant $nil \in \mathcal{D}$ and define these functions as $nil$ for arguments not in $\mathcal{P}$. In accordance with LISP conventions we refer to the elements of $\mathcal{D}$ as *atoms*.

To simplify matters, we regard the registers too as containing *pointers*. Technically, we have a "register contents" function $\mathrm{REG}\colon \{1, 2, \ldots, N_{reg}\} \to \mathcal{S}$ so that $\mathrm{REG}(i)$ is the "contents" of register $i$. The number of registers is still considered to be an unspecified constant. It is not hard to show that, ignoring a constant factor change in running time, any bounded number of registers (above a necessary minimum) suffices for any program. The state of a LM is given, as for the RAM, by the contents of all the registers, the program counter and the non-blank portion of memory. "Non-blank memory" means here the set of cells accessible from the registers via chains of CARs and CDRs of any length. During the execution of the program, this set may grow as new dotted pairs are created, or shrink—when all references to a cell disappear. Since a cell which is "unlinked" from the structure can no longer influence the behavior of the machine, it can be assumed that such dotted pairs are deleted from $\mathcal{S}$.

The *memory image* of a LM (in some state of execution) can thus be said to have a *structure*, which is a directed graph whose nodes are $\{R_1, R_2, \ldots, R_{N_{reg}}\} \cup (\mathcal{S} \cap \mathcal{P})$, and arcs naturally defined by the functions REG, CAR and CDR, restricted to the set of nodes.

This structure is independent of the *data type* of the machine; the data type gets into the picture when we consider the "arcs" leading outside the structure, i.e. the values of the REG, CAR and CDR functions which are in $\mathcal{D}$. Given the structure alone, we can list the "names" of the missing arcs; **Figure 1.1** explains this concept (note: when drawing structures, the CAR of a dotted pair will be drawn by convention as an ordinary arrow, and the CDR—as a double arrow).

This list is fixed by the memory structure and independent of any data values; complementing it with a matching list of data values completes the description of the LM's memory.

In summary, the instantaneous state of a LISP machine is defined by its "program counter" and the current *memory image*, described as the pair (*structure*, *content*) where *structure* is the memory graph and *content* is a list of domain values, the length of which is fixed for any specific *structure*.

We define the *height* of a memory image to be one plus the height of a BFS tree of its *structure* (the addition of one counts for the "missing arc" at the end of the path).

The instruction set of the $\mathcal{T}$-LM is the same as that of the $\mathcal{T}$-RAM, except for the memory access instructions which are replaced by a set of instructions for operating on dotted pairs. A valid program does not apply any of the arithmetic instructions to a dotted pair. We consider only valid programs.

Let us describe now the new instructions of the LISP machine.

$CONS \quad R_i, R_j, R_k$

this instruction adds to $\mathcal{S}$ a *new* dotted pair $P$ whose CAR and CDR are REG$(j)$ and REG$(k)$ respectively and sets REG$(i) = P$.

$LCAR \quad R_i, R_j$

this instruction sets REG$(i)$ to CAR(REG$(j)$).

$LCDR \quad R_i, R_j$

this instruction sets REG$(i)$ to CDR(REG$(j)$).

The instruction JEQL is also extended to test for equality of pointers. Thus it branches if REG$(i) =$ REG$(j)$ regardless of whether these values belong in $\mathcal{D}$ or in $\mathcal{P}$.

Another useful instruction would be a type test instruction, i.e. something like "jump if REG$(i)$ is a dotted pair", allowing a program to avoid applying pointer operations to data values and vice versa. It is easily seen, however, that programs using this instruction can be rewritten without it with at most a constant factor loss in running time; so the inclusion or omission of this instruction is not important.

The next two instructions separate the world of LISP machines in two, following an important distinction in the LISP language. In the world of **pure** LISP, the above instructions are all there is. The following two instructions distinguish **standard (full)** LISP from **pure** LISP, in that they add the capability to *change the contents* of an existing memory location, where in **pure** LISP a memory location does never change its value. The instructions are:

$RPLACA \quad R_i, R_j$

change CAR(REG$(i)$) to REG$(j)$.

$RPLACD \quad R_i, R_j$

change CDR(REG$(i)$) to REG$(j)$.

If applied to a domain value, the effect of these instructions is undefined.

To sum up, we have in fact two distinct LISP machines: the pure LISP machine, PLM, and the full LISP machine, FLM. The distinction between the two highlights a theoretical problem: what are the implications of including or excluding the capability of changing part of a dotted pair (or more generally a record) *in situ*. The outstanding effect of this capability is that cycles can appear in the memory structure; the structure of a PLM's memory is always an acyclic graph, topologically ordered by the time the nodes where created. Thus it would be interesting to find the complexity of simulating the FLM on the PLM. For the RAM simulation problem, however, the machines exhibit similar performance, that is, none of our results separates the two. The common name "LISP machine" will be used in a generic sense, that is, a statement mentioning LM is to be true when either PLM and FLM are consistently substituted for LM.

Each LISP machine starts off with all the registers initialized to *nil*, and then changes its state according to the program instructions and input data.

To define the running time, or *time complexity*, of a RAM or LM program we shall use the *uniform cost model*, that is, we charge one "time unit" for each instruction performed. Uniform cost is also applied to space complexity in the RAM, which is defined by the number of memory locations used.

Programs for high level models will be written in **pidgin algol** with simple variables and control structures and also procedure calls. It is well known how to compile this language into "machine code" and evaluate their running time (see for example [3]). It is helpful to note that stacks, a useful data structure, are easily implemented in pure LISP. Stacks are represented by the standard LISP data structure known as **list**: for completeness, here is its definition.

A **list** is either *nil*,
or it is a dotted pair, whose CDR is a **list**.

The CARs of the dotted pairs comprising the list are called the **list elements**.

## 2. THE SIMULATION PROBLEM

### 2.1. Simulation of the Storage Unit

The only difference between high level models of the same data type is the memory model. Thus when asking to compare the computational power of two machines, it is natural to see the question as that of comparing the power of their "storage units".

The idea of the *storage unit* of a RAM follows naturally from the definition of the model. It is supposed to be some kind of machine, which can be given *commands* of the form *STORE* $d, a$ and *LOAD* $a$ where $d \in \mathcal{D}$ and $a$ (the *address*) is in $\mathcal{D} \cap \mathbb{N}$. The machine should output in response to each LOAD the value last stored into the specified address by a STORE or 0 otherwise.

A *simulation of the RAM storage unit* (RASUS for short) consists of two programs for a high level model. One program is used to carry out a STORE and the other to carry out a LOAD. Initially the machine starts with a fixed memory structure that corresponds to an all-zero memory of the RAM. Each STORE is carried out by placing $a$ and $d$ in designated registers and executing the *store* program. Each LOAD is executed by placing $a$ in its designated register and executing the *load* program, whose task is to deposit in the $d$ register the value last "stored" into address $a$. The memory image is not changed in any way between execution of the programs. Instead of providing the initial memory structure, it may be convenient to have an *initialization program* in addition to the previous two. This is a one-shot program with no input, so its running time is fixed and of no interest.

Let $T_{sim}(u)$ be the total time spent by the machine in performing the simulation for the sequence of commands $u$. Define the RAM time $t(u)$ to be the number of instructions in $u$, i.e. its length; and the RAM space $s(u)$ to be the number of distinct addresses that appear in STORE commands in $u$. Then we measure the *time complexity* of the simulation by the function

$$T(t, s) = \sup_{u: \substack{t(u) \leq t \\ s(u) \leq s}} T_{sim}(u).$$

We can also consider $T_{load}(t, s)$ and $T_{store}(t, s)$, the worst-case time needed by the *load* and *store* programs, respectively, after any preceding command sequence $u$ such that $t(u) \leq t$ and $s(u) \leq s$.

If $T_{load}(t, s)$ and $T_{store}(t, s)$ are bounded by the function $f(t, s)$, we say the simulation has a *delay* of $f(t, s)$. We further attribute it as *uniform delay* if $f(t, s) = O(\frac{T(t,s)}{t})$. A simulation having constant uniform delay is said to be *real time*.

## 2.2. Machine Simulation

A fundamental approach to the comparison of random access and LISP machines is looking for bounds on the gap between the performance of *any program* on the RAM and a program for the same problem on the LM. Excluding the case of a RAM with finite memory, simulating an LM by the RAM is both simple and efficient (real time). Obtaining the complexity of simulating RAM programs by the LM is less trivial and we shall see that we can learn of time bounds for such simulations by studying the complexity of RASUS on the LM. In fact, we show that for any high level model, the complexity of RASUS determines the time bounds on simulating the RAM in general.

We define one program as simulating another one if their external behavior matches. By external behavior we mean the sequence of input instructions performed, output produced (for a given input) and halt (if done) throughout the execution of the program. Note that this is necessarily an *on-line* simulation (the alternation of input and output operations is preserved). Let $t_0 < t_1 < \cdots < t_i < \cdots$ be the time steps at which the simulated program makes an external operation. Let $t'_0 < t'_1 < \cdots < t'_i < \cdots$ be the corresponding time steps of the simulation. Let $s_i$ be the space used by the RAM in the time interval $[0, t_i]$. We say that the simulation time is governed by a function $T(t, s)$ if for all input sequences, for all $i$ such that $a < t_i \le t$ and $s_i \le s$, we have

$$t'_i \le T(t, s).$$

The simulation is said to be *real time* when there exists a constant $c > 0$ such that for all input sequences we have

$$t'_i - t'_{i-1} \le c(t_i - t_{i-1}) \qquad \forall i \ge 0.$$

In general we say that the simulation has a *delay* governed by $f(t, s)$ if for all input sequences,

$$t'_i - t'_{i-1} \le f(t_i, s_{t_i}) \cdot (t_i - t_{i-1}) \qquad \forall i \ge 0.$$

The simulation *achieves time* $T(t, s)$ *with uniform delay* if its running time is governed by $T(t, s)$ and its delay by $O(\frac{T(t,s)}{t})$.

### 2.3. Equivalence of the Problems

This section is devoted to proving that the general problem of simulating the RAM is equivalent to the specific problem of random access storage unit simulation (RASUS). In the proof we will need the following technical lemma.

**Lemma 1.** *If RASUS can be solved by a high level machine in time $T_1(t,s)$, then for any $k > 0$ there is a RASUS solution for the same machine whose running time $T_2$ satisfies*

$$T_2(kt,s) \le c_k T_1(t,s),$$

*where $c_k$ is a constant.*

**Proof.** Let $\mathrm{RASUS}_1$ be the RASUS solution which runs in time is $T_1$. We obtain the desired solution $\mathrm{RASUS}_2$ by duplicating the register set used by $\mathrm{RASUS}_1$ $2k^2$ times, as well as the code for the procedures, so that we actually have $2k^2$ $\mathrm{RASUS}_1$ clones operating concurrently, to which we refer as $D(i,j)$ (for *data*) and $T(i,j)$ (for *time stamp* where $0 \le i,j < k$. In addition we count the STORE commands in $N_{store}$. Here is the code for $\mathrm{RASUS}_2$, under the simplifying assumption that all the duplicates share the registers $a$ and $d$ which are not modified except for depositing the result of a successful LOAD. Therefore, these parameters need not be passed explicitly. The *load* procedure uses two auxiliary registers, $r$ and $t$.

```
init_2:
    N_store, N_load ← 0
    for 0 ≤ i, j < k
        init_1(D(i, j))
        init_1(T(i, j))
    end for
store_2:
    N_store ← N_store + 1
    i ← N_store mod k
    for j = 0, ..., k − 1
        store_1(D(i, j))
        d ← N_store
        store_1(T(i, j))
    end for
load_2:
    N_load ← N_load + 1
    j ← N_load mod k
    r ← 0
    t ← 0
    for i = N_store + 1, ..., N_store + k   (mod k)
        load_1(T(i, j))
        if d > t then
            t ← d
            load_1(D(i, j))
            r ← d
        end if
    d ← r
    end for
```

The correctness of the solutions stems from the fact that every *store* effects each column of the $k \times k$ matrix, *load* scans a whole column, and the time stamps ensure that the last STORE into the sought address determines the output.

Thus every *store* and *load* calls at most $2k$ times the corresponding procedure of $\mathrm{RASUS}_1$, while each of the $\mathrm{RASUS}_1$ duplicates only receives up to $1/k$ of the commands. This gives immediately the desired time bound. The constant $c_k$ seems to be $O(k)$, but we have multiplied the number of registers by $k^2$; so we may want to take into account the cost of absorbing this growth in number of registers. On the LISP machines, this problem may be solved with a bit of programming by linking the memory images of the $k^2$ RASUS clones in a suitable list structure. Then the code need not be duplicated, and $c_k$ is a genuine $O(k)$. □

**Lemma 2.** *The general simulation of the RAM and the RASUS problem are real-time equivalent on any high level machine. That is, if the machine can solve RASUS with time complexity $T(t, s)$, then it can simulate every RAM program in time governed by $3T(t, s)$. Vice versa, if there exists a function $T(t, s)$ such that every RAM program can be simulated by our model in time governed by $T(t, s)$, then there it can solve RASUS in time complexity $T_{RASUS}(t, s) \leq cT(t, s)$ for some constant c. This relationship holds also for the delay of uniform delay solutions.*

**Proof.** ($\Rightarrow$) Assume that there is a solution to the RASUS problem which runs in time $T(t, s)$. Let an arbitrary RAM program be given. All the instructions appearing in the program (except for the memory access instructions) can be directly executed by the simulator (they appear in its instruction set). Thus only the STORE and LOAD have to be replaced to form the simulation. These two will be replaced by invocations of the *store* and *load* program of the RASUS; to avoid any conflicts we may equip them with a separate set of registers. Thus the execution of a memory access instruction will be replaced with moving the operands to or from the dedicated registers (at the cost of two instructions) and executing the RASUS procedure.

Assume that for some input, the RAM program uses space $s$ while performing the first $t$ instructions. Let $t_1$ be the number of STORE and LOAD instructions performed. Our simulation will run in time bounded by $t - t_1 + (2t_1 + T(t_1, s)) = t + t_1 + T(t_1, s)$. Since necessarily $T(t, s) \geq t$, and $T(t, s)$ is monotone increasing in $t$, we have

$$t + t_1 + T(t_1, s) \leq 3T(t, s).$$

Using the definition of the simulation time complexity in §2.2, we obtain that our simulation has time complexity bounded by $3T(t, s)$.

This argument can be readily extended to the delay of uniform delay solutions, which completes the proof.

($\Leftarrow$) Assume that the $\mathcal{T}$-machine can simulate the RAM in time $T(t, s)$. We shall build a solution to the RASUS problem running in similar time. Such a solution must be a pair of programs as specified in the RASUS definition (§2.1). That subsection also defines the time complexity of RASUS, a function of the <u>RAM time</u> and <u>RAM space</u> of the input sequence. However, the bound that we are given is on the time that simulation of a RAM *program* may require on the $\mathcal{T}$-machine; and we must convert it to a bound on the RASUS time, dependent only on the aforementioned parameters. The conversion is achieved by choosing a RAM program that is an "interpreter": it accepts the command sequence as input and executes it. Its time and space complexity will be seen to be proportional to the <u>RAM time</u> and <u>RAM space</u> of the input sequence, allowing us to transform a simulation of this program to a RASUS solution of similar complexity.

The RAM program that we use is shown in Figure 2.2.

```
repeat forever
    read command
    if command = STORE then
        read d, a
        (a) ← d
    else   /* command = LOAD */
        read a
        write (a)
    end if
```

**Figure 2.2.**

This program executes an infinite loop. In each iteration of the loop, it reads a command code (one of two predefined constants) and performs the operation requested, reading the operands for this command and for a LOAD, writing out the result. Denote by $\tilde{t}(u)$ and $\tilde{s}(u)$ the time and space, respectively, required by this program for handling a command sequence $u$. It is easy to see that

$$\begin{aligned} &\tilde{t}(u) \leq c_1 t(u) \qquad \text{for some integer } c_1, \text{ and} \\ &\tilde{s}(u) = s(u) \end{aligned} \tag{1}$$

where $t(u)$ and $s(u)$ are defined as in §2.1. Let *prog* be a program for the given machine which simulates the above program in time $T(\tilde{t}(u), \tilde{s}(u))$. From *prog* we form a RASUS solution, namely the three procedures *init*, *store* and *load*. The difficulty in obtaining these procedures lies in the fact that our program may be more complicated than a "read and execute" loop (which would readily yield a RASUS). There may be several READ commands interspersed throughout the program, which alternatingly assume the role of reading the command code or operands. Moreover, this program runs forever. Our procedures are one-shot programs (although the memory image is preserved between the calls) and therefore, to simulate the actions of this program, its "program counter" has to be restored as well. Since the program is fixed, we can label the points where we may stop with a finite set of labels and code a multi-way branch which, according to a preserved index, continues execution at the desired point.

Each of the procedures will follow *prog* up to a certain point: the initialization procedure consists of the (unique) sequence of instructions executed by *prog* up to the first READ instruction. The *store* procedure replaces reading the command code by setting it to STORE and reading the two operands by getting them from the designated registers. At at the next READ, it stops. The *load* procedure is similar. We are now ready for coding the solution, where we assume the arguments to the procedures to be passed in registers $R_a$ and $R_d$ and make use of three additional registers, $R_{pc}$ (for saving the program counter), $R_{inp}$ (for simulating input operations) and $R_{inps}$ (for counting input operations).

The RASUS procedures start with the following lines:

*init*:
$R_{inps} \leftarrow 0$ ;no operands to "read"
JUMP *prog*
*store*:
$R_{inp} \leftarrow STORE$
$R_{inps} \leftarrow 2$ ;"read" both $d$ and $a$
JUMP $L_{R_{pc}}$ ;a multi-way branch
*load*:
$R_{inp} \leftarrow LOAD$
$R_{inps} \leftarrow 1$ ;"read" $a$
JUMP $L_{R_{pc}}$ ;a multi-way branch

In addition, we substitute the following code segment for each occurrence of "READ $R_i$" in *prog*. Here $k$ is a different integer for every READ instruction replaced, yielding a unique label $L_k$.

```
      if R_inps = 0 then
            R_pc ← k
            halt
      else
            if R_inps = 2 then
                  R_inp ← R_d
            else
                  R_inp ← R_a
            end if
            R_inps ← R_inps − 1
      end if
L_k:  R_i ← R_inp
```

Each "WRITE $R_i$" is replaced by

$$R_d \leftarrow R_i.$$

Clearly, the running time of the RASUS obtained is linear in the running time of *prog*. That is, there exists a integer $c_2$ such that for every sequence $u$ of STORE and LOAD commands we have

$$T_{RASUS}(u) \leq c_2 T(\tilde{t}(u), \tilde{s}(u)) \leq c_2 T(c_1 t(u), s(u))$$

(without loss of generality, we may assume $T$ to be increasing). We thus obtained a RASUS solution in time $T_{RASUS}(t, s) \leq c_2 T(c_1 t, s)$. The proof is completed by an application of Lemma 1.

This argument can be readily extended to show that if *prog* simulates the RAM with uniform delay, the RASUS procedures have a proportional uniform delay. □

## 3. TIME BOUNDS ON THE SIMULATION

Now that the tools are prepared we set out to discover the complexity of simulating random access memory (or machines) by a LM.

### 3.1. An Upper Bound

**Algorithm 1:** *A search tree solution to RASUS.*

This algorithm will not be specified in detail, since any of the well-known balanced tree schemes will serve (see, for example, [3]). These tree structures are used to store and retrieve data values by their *keys*, which are in our application the RAM addresses. The *store* program implements the *insert* operation on the tree, and the *load* program—the *find*. The tree structure can be very easily constructed of LISP cells, and the programs for *insert* and find can be readily written in the FLM language.

On the PLM, implementation of the tree algorithms may have to be changed, as they usually use the operation of changing the value at a node, or the identity of its child. However, coercing them to the pure LISP model is not difficult. In all these schemes, the *insert* and *find* programs have the general form of a descent down a particular path from the root to a leaf, then an ascent back, during which modifications are made to the nodes on this path. The locality and order of changes allows for an easy implementation on the PLM, in which pointers to the nodes *neighboring* the path are saved in a stack while descending, and on the ascent, the path is rebuilt anew (with the modifications required) and the stack is used to connect it to the neighborhood. This is actually the "path copying" technique mentioned in [31], where its feature of not modifying existing nodes is used to allow "past generations" of the tree to be retrieved.

The running time of this solution is clearly $O(t \log s)$, as $T_{load}(s) = T_{store}(s) = O(\log s)$ (by known properties of balanced trees). Thus we get our upper bound:

**Theorem 1.** *The LISP machine can simulate the RAM in $O(t \log s)$ time, with uniform delay.*

### 3.2. A Simple Lower Bound

The balanced tree schemes have the reputation of optimality and it is natural to conjecture that the FLM cannot do better than that in the RASUS problem. A "proof" of an $\Omega(t \log s)$ lower bound is easily obtained.

Consider the state of the FLM just before initiating the *load* program, sometime during a simulation. In the first instruction executed by the program, the only *atoms* it can access are those currently linked to registers, i.e. $N_{reg}$ atoms at most. By performing another instruction, no new atoms in memory can be retrieved via a register pointing to an atom, while a pointer to a dotted pair can be used to load any one of its CAR and CDR; thus in two instructions, up to $2N_{reg}$ previously stored atoms can be accessed. Extending this argument by induction, we get that in $k$ instructions starting at any given state, up to $2^{k-1}N_{reg}$ different atoms previously stored in memory can be retrieved.

For any $s > 0$, let $u$ be a sequence of $s$ STORE instructions with addresses $0, 1, \ldots, s-1$ and different data. After executing $u$, the simulating machine must be ready to handle correctly any command of the form $LOAD\ a$ where a is one of the addresses in $u$. According to the previous argument, at least half of the atoms previously stored in memory require $\lfloor \log s - \log N_{reg} \rfloor$ steps to be retrieved. Actually, they are that far from the registers in the memory graph. Thus a load command can now be issued which will force the machine to "travel" a distance of $\Omega(\log s)$ in order to find the requested atom. This argument entails a lower bound on the time for retrieval which holds even with unbounded preprocessing time. Therefore, it remains valid at any later time, and we can create an unbounded sequence of worst-case LOAD commands, obtaining a lower bound of $\Omega(t \log s)$ on simulation time.

The weakness of this argument lies in the assumption that when processing a store, the machine actually stores the given datum in memory, and has to fetch it from this location for a load operation. In fact, the LISP machine might encode the simulated RAM's memory contents in a very clever way, giving rise to a better simulation. This way the *data type* of the machines comes into the picture; only by relying on properties of the data type we may avoid such pitfalls. For a start, the above proof is valid if our machines may handle *symbols*.

The term *symbol* is defined in [38,39] referring to an item on which no operations are permitted except testing for equality. This definition coincides well with the usage of LISP, in which *atoms* may well be symbolic, in addition to numeric.

**Theorem 2.** *Let $\mathcal{T}$ be a data type containing an infinite set of symbols. Then A $\mathcal{T}$-FLM needs $\Omega(t \log s)$ time to simulate a $\mathcal{T}$-RAM.*

**Proof.** None of the instructions in the FLM instruction set can "generate" a symbol; the only instructions whose execution may result in a register pointing to a new symbol are SET, LCAR and LCDR. Therefore the above argument applies. □

### 3.3. Other Data Types

The proof of the lower bound in the previous sub-section relied heavily on the presence of "symbols" in the *domain* of the machines considered. Though it was stated that this is a natural way of thinking in LISP, it is not at all natural to the RAM. So our "goal" now is to show that the $\Omega(t \log s)$ bound carries over to RAM models with more useful data types.

In the following sections three main models of the RAM, i.e. three *data types*, will be analyzed in this respect. The three models are:

(i) The **real number** RAM, whose domain consists of the real numbers.

(ii) The **integer** RAM, whose domain is the integers.

(iii) The **finite word** RAM, whose domain is finite and will be considered as consisting of $k$-bit words, where $k$ is a parameter, making it a model family (§1).

The main tool for the analysis will be a notion of *incompressibility*. This notion is the actual feature of the *symbol* data type that is required for the proof. In §4 this notion is defined and a general proof of the lower bound using this notion is given. This feature is proved for case (iii) in §4 and for (i) in §5. Section 6 is devoted to the integer case; it will be seen that the set of primitive functions should be chosen with care if this data type is to be incompressible.

The wide success of the lower bound proof can be interpreted as an assurance of the general validity of accessibility considerations. The fact that it fails for some data types may be interpreted as indicating that incorporating them in our model is far fetched.

## 4. INCOMPRESSIBILITY AND LOWER BOUNDS

### 4.1. Definitions

The notion of an incompressible data type is that of a data type $\mathcal{T}$ such that a $\mathcal{T}$-machine is not able to "encode" a number of data atoms in a smaller number and so economize on memory accesses. For a precise definition we need a simple model of computation which allows us to concentrate on properties of the data type.

Let $\mathcal{T} = (\mathcal{D}, \mathcal{F}, <)$. Each function in $\mathcal{F}$ has as domain the set $\mathcal{D}^n$ for some $n > 0$; for simplicity of exhibition, we include in the sequel also functions on $\mathcal{D}^0$, which represent constants. We denote by $\mathcal{F}^*$ the *closure* of $\mathcal{F}$ under the following operations. *Aggregation* combines some functions $f_1, \ldots, f_n : \mathcal{D}^m \to \mathcal{D}$ to one vector valued function $f = (f_1, \ldots, f_n) : \mathcal{D}^m \to \mathcal{D}^n$. *Composition* is used to apply a function to the result of another: composing $g : \mathcal{D}^n \to \mathcal{D}$ with $f : \mathcal{D}^k \to \mathcal{D}^n$ yields $g \circ f : \mathcal{D}^k \to \mathcal{D}$.

The model of computation we shall use is the $\mathcal{F}$ *decision tree*. A decision tree of $n$ inputs is an ordered finite binary tree each of whose internal nodes represents a "decision". Actually we associate with each internal node $\nu$ a function $f_\nu \in \mathcal{F}^* : \mathcal{D}^n \to \mathcal{D}$. The computation begins in the root; in every internal node $\nu$ reached, the function $f_\nu$ is applied to the input values, and the computation now proceeds according to its value, "control" moving to the right child if it is greater than zero and otherwise to the left. The expression "greater than" relates to the order relation $<$ of the data type $\mathcal{T}$. Upon reaching a leaf, the computation ends with its result given by the leaf's function, applied to the inputs.

We remark that this is not a comparison tree; since we compare the outcome of functions, whose exact nature is not specified, any test that we wish to allow our models to perform may be included in the tree.

The terminal (leaf) functions have the range $\mathcal{D}^m$ (for some fixed $m$). We say the tree computes a function from $\mathcal{D}^n$ to $\mathcal{D}^m$.

It is quite easy to see that each function computable by a $\mathcal{T}$-PLM or a $\mathcal{T}$-FLM (as well as a $\mathcal{T}$-RAM) program can be computed by an $\mathcal{F}$ decision tree ($\mathcal{F}$ relating to $\mathcal{T}$ as above), provided the program has a finite bound on its running time (we just "unroll" the program and make explicit all calculations to avoid the use of memory). If we are to compute a function of $\mathcal{D}^n \to \mathcal{D}^m$, the running time of our program may depend on $n$ and $m$ and we impose no restrictions on its rate of growth; the finiteness requirement disallows programs whose running time grows indefinitely for fixed $n$ and $m$, depending on the input values (programs of *nonuniform* complexity). Such a program may "compress" any effectively countable domain, e.g. by the conventional "pairing functions". Therefore we shall use the term "finitely computable" to describe a function which can be computed with uniform complexity, where $\mathcal{T}$ can be understood from context.

**Definition.** *A data type $\mathcal{T} = (\mathcal{D}, \mathcal{F}, <)$ is called incompressible if for all $n > 0$, there is no pair $f_n, g_n$ of functions, computable by $\mathcal{F}$ decision trees, such that $f_n$ maps $\mathcal{D}^n$ into $\mathcal{D}^{n-1}$ and $g_n$ is its inverse.*

$\mathcal{T}$ is incompressible if and only if for all $m < n$ there is no pair $f, g$ of finitely computable functions such that $f$ maps $\mathcal{D}^n$ into $\mathcal{D}^m$ and $g$ is its inverse.

**Remark.** The incompressibility of a data type of *finite domain* is implicit in the work of Kolmogorov [22]. The value of this feature to proofs in computational complexity was advocated by Paul et al. [27,28] and other work. Our notion of incompressibility is more general in nature, but is used towards the same end.

### 4.2. The Lower Bound

We now proceed to give the lower bound proof for incompressible data types. To this end we have to consider LISP machines and their memory images instead of the general decision trees. So instead of programs computing functions of their inputs, we consider programs that *transform memory images.* We say a program transforms the memory image $M_1$ to the memory image $M_2$ whenever having set the LM's memory to $M_1$ and started the program, it halts after changing the memory image to $M_2$. For *sets* of memory images $\mathcal{M}_1$ and $\mathcal{M}_2$, we say that $\mathcal{M}_1$ *can be encoded* in $\mathcal{M}_2$ if there is a program that transforms each element of $\mathcal{M}_1$ to an element of $\mathcal{M}_2$, and another program that "decodes" them correctly, i.e. implements the inverse transformation. Both of these programs are required to have their running time bounded by a constant (generally dependent on the sets).

Say that a memory image is *rooted in* $R_1$ if all the other registers are *nil* for that image.

Consider memory images which are **list**s (cf. §1), rooted in $R_1$. It is easy to see that the incompressibility of $\mathcal{T}$, defined generally in the previous subsection, can be formulated for LMs as:

*$\mathcal{T}$ is incompressible if the set of* **list***s of $n$ atoms cannot be encoded in the set of $m$-atom* **list***s, for any $m < n$.*

The lower bound proof will show that if an FLM can simulate the RAM "too fast", that is, it has RASUS programs of low running time, then we can obtain from these programs an encoding of $n$-element lists in $m$-element lists, for $m < n$. Actually the fast RASUS solution amounts to an encoding of $n$ elements in a memory image which is not necessarily a **list** and we must show that we can further encode these images in the set of short **list**s.

**Lemma 3.** *The set of memory images which are rooted in $R_1$ and have height at most $h$ can be encoded in the set of* **list**s *of length $\lfloor \frac{3}{4} 2^h \rfloor$, provided that $\{0, \ldots, 2^h - 1\} \subseteq \mathcal{D}$.*

**Proof.** If $h = 1$, the lemma is trivial since the image is already a **list** of one atom. We continue with $h > 1$.

Let $M$ be a memory image, rooted in $R_1$ and having height at most $h$.

Recall the division of a memory image into *structure* and *contents* (§1). The contents of the memory image are actually *leaves* in a graph, whereas the structure is the subgraph excluding the leaves. Each leaf appears at the end of a directed path, and since $M$'s height is at most $h$, there are at most $2^{h-1}$ different paths to leaves. Hence the number of leaves is at most $2^{h-1}$.

The largest number of leaves occurs when $M$ is a complete binary tree. Then it has $2^{h-2}$ nodes at level $h-1$, each one a parent of two leaves. If we had to encode such an image as a **list**, we would just have to list the values of the leaves; thus **list**s of length $2^{h-1}$ encode these images.

Certainly $M$ is not necessarily a tree. To deal with general structures, we add to each atom in the list a *mark bit* which is either 0 or 1. We encode the marks of pairs of consecutive elements in the list in the bits of one two-bit number (note that $\{0, \ldots, 3\} \subseteq \{0, \ldots, 2^h - 1\} \subseteq \mathcal{D}$). This defines the final form of our **list**: $2^{h-1}$ data atoms accompanied by $2^{h-2}$ mark atoms, encoding one mark bit for every atom. The length of the **list** remains fixed, so we may use array notation and write $A[0], \ldots, A[2^{h-1} - 1]$ for the data atoms and $M[i]$ for the mark bit of $A[i]$ (actually these mark bits are paired as above).

To encode the memory image $M$ in these two arrays, we use a standard correspondence between nodes in the BFS tree of the structure and binary numbers. This BFS tree is an ordered binary tree, with CAR preceding CDR for each dotted pair. We refer to the single dotted pair connected to $R_1$ as the root of the tree (and of the image), and neglect the arc leading into it (we deal only with arcs emanating from dotted pairs). Then by the definition of height (§1), this tree's height is $h - 2$.

The correspondence is obtained by assigning the value 0 to CAR and the value 1 to CDR. We denote by $T$ the unique BFS tree that results from an ordered BFS search in which CAR is handled before CDR. Each node in the structure is the end of a unique path from the root of $T$, whose length is at most $h - 2$. We label it by a number whose binary representation is $b_1 b_2 \ldots b_{h-1}$, where $b_i$ is the value of the $i$th arc along the path, if such an arc exists, and otherwise zero ($b_{h-1}$ is necessarily zero, but it is convenient to keep it). Hence the dotted pair linked directly to $R_1$ is labeled by zero, and the same for its CAR;

its CDR is labeled $2^{h-2}$. The range of possible labels is 0 to $2^{h-1}-1$, corresponding to indices in $A$ and $M$. Obviously, each number may be labelling more than one node. The following facts are easily established.

(i) The set of nodes labeled by $\nu$ has the following form: $u, \mathrm{CAR}\, u, \mathrm{CAR}^2\, u, \ldots, \mathrm{CAR}^\ell\, u$ for some $\ell \geq 0$. Of all these nodes, only the last one is a leaf in $T$ (recall that a leaf of $T$ is not necessarily a leaf of the structure).

(ii) Let $u$ be a node of label $\nu$, and let $k$ be its depth in the tree. Then $\nu$ is a multiple of $2^{h-1-k}$; all the descendants of $u$ in $T$ have labels of the form $\nu + i$, $0 \leq i < 2^{h-1-k}$; and only descendants of $u$ may bear labels of this form for $i > 0$.

We call a node $v \in M$ a *front node* if it is a dotted pair such that at least one of the arcs $\mathrm{CAR}\, v$ and $\mathrm{CDR}\, v$ does not belong to $T$. It is easy to see that the memory image is reconstructible from the following information: (1) the labels of its front nodes, (2) the depths of these nodes in $T$, and (3) the CAR and CDR of each of these nodes. We show how to encode this information in the arrays $A$ and $M$. Let $v$ be a front node labeled $\nu$, and $k$ its depth in $T$. By Fact (ii), since $k \leq h-2$, all the node labels must be even. If $k = h-2$, then both $\mathrm{CAR}(v)$ and $\mathrm{CDR}(v)$ have to be atomic, and we set $M[\nu] = M[\nu+1] = 0$ and $A[\nu] = \mathrm{CAR}(v)$, $A[\nu+1] = \mathrm{CDR}(v)$. Otherwise, $k < h-2$. Suppose that the CAR arc of $v$ is not in $T$ (CDR is handled similarly). Then by Fact (i) we know that $\nu$ labels no node of depth greater than $k$. Moreover, there may be only one front node with this label and CAR not in $T$. We set $M[\nu] = 1$ and $A[\nu] = k$. We save $\mathrm{CAR}(v)$ in $A[\nu+1]$, which is free since $\nu+1$ might only label descendants of $\mathrm{CAR}(v)$, had the CAR arc of $v$ belonged to $T$ (Fact (ii)). If $\mathrm{CAR}(v)$ is atomic, we set $M[\nu+1]$ to 0 and $A[\nu+1]$ to the value of $\mathrm{CAR}(v)$. Otherwise, we set $M[\nu+1]$ to 1 and $A[\nu+1]$ to the label of $\mathrm{CAR}(v)$. Some of the array elements are not given meaningful values: these are the elements numbered $\nu+2$ to $\nu + 2^{h-2-k} - 1$. However, this will not defy the correct decoding of the information, since by scanning $M$ sequentially we identify these elements when processing $\nu$ and skip them afterwards. All other elements of $M$ and $A$ must have been set by the above procedure (which is an easy exercise to verify).

The above description, while not strictly formal, is not difficult to translate into an LM program, whose running time is bounded by a function of $h$ alone. Likewise it is easy to see that the construction can be reversed, and an inverse transformation can be programmed within a similar time bound. The difficult part of this program is the reconstruction of pointers to other nodes, which have not yet been created. This can be overcome by using two passes: in the first, we represent pointers to nodes by the nodes' number and a special mark. In the second pass, on the FLM we redirect the pointers to their intended target. On the PLM, we use a topological sort to obtain the sequence in which we have to create the nodes. Such a sequence will be obtained since the encoded image has been built by a PLM too. $\square$

**Theorem 3.** *Let $\mathcal{T}$ be an incompressible data type. Then a $\mathcal{T}$-FLM requires $\Omega(t \log s)$ time to simulate a $\mathcal{T}$-RAM.*

**Proof.** Again, we consider the RASUS problem. We will show that for all pairs $s > 0$, $n > 0$, there exists a sequence $u$ of $s$ stores and a sequence $v$ of $n$ loads, such that the simulation of $uv$ must take time $> n\lfloor \log s \rfloor$. Assume to the contrary that there are a RASUS solution $R$ and a pair $s > 0$, $n > 0$, such that for all sequences $u$ of $s$ stores and all sequences $v$ of $n$ loads, the simulation of $uv$ takes at most $n\lfloor \log s \rfloor$ time. Without loss of generality it may be assumed that the memory image left when any of the RASUS procedures terminates is rooted in $R_1$. Fix this solution and the values of $n$ and $s$ for the rest of this proof. We show that it necessarily holds that after simulating $u$ and some part of $v$, $R$ is able to answer every load request using a memory structure whose height is bounded by $\lfloor \log s \rfloor$. We can effectively discover such a structure and obtain an encoding which contradicts incompressibility.

Let a list of $s$ atoms be given. Make up a sequence $u$ by storing these data into addresses $0, \ldots, s-1$. In the rest of this proof we deal with load commands from these addresses only. Call a load operation *costly* if it takes more than $\lfloor \log s \rfloor$ time units, and *cheap* otherwise. A sequence $v'$ of $k \geq 0$ load commands is defined as *costly* if when fed to $R$ following $u$, each of its constituents is a costly load. By our assumption there does not exist a costly sequence of length $n$. Hence, by pursuing any chain of costly loads, we must—after less than $n$ operations—reach a state where every load operation is cheap. Therefore if, in this state, we prune the memory image of $R$ to the height of $\lfloor \log s \rfloor$, that is, change all pointers into farther dotted pairs to *nil*, the flow and outcome of every load operation will not be affected. Therefore, having found such an image, and pruned it, we have a memory image of height at most $\lfloor \log s \rfloor$ which encodes the $s$ input elements.

To complete the proof, here is a description of the encoding and decoding programs. Note that their running time is bounded.

Encoding program:
*Input*: a **list** of $s$ elements, $L$.
*Output*: a memory image of height $\leq \lfloor \log s \rfloor$.
*Method*: A dedicated set of registers is used for simulating $R$, whose procedures, $store_R$ and $load_R$, are only modified by adding an instruction counter $t$ to $load_R$ which allows us to monitor its time. The procedure *CopyImage* copies the memory image rooted in one register to another one (a simple task on an FLM. On a PLM, we use a topological sort to obtain the order in which we will create the nodes so that the pointers can be reconstructed. Such an order always exists, because the copied image was built by a PLM). The rest of the description lies in Figure 4.1, omitting the act of pruning, which is similar to copying the image.

Decoding program:
*Input*: a memory image produced by the encoding program.
*Output*: the reconstructed **list**.
*Method*: for $i = s-1, \ldots, 0$ copy the memory image to $R_1$ and call $\text{load}_R(i)$. Push the retrieved value onto the list.

The conclusion of this construction is that we can encode the **list**s of $s$ atoms in

```
        for i = 0, ..., s − 1
            store_R(pop(L), i)
        end for
    advance:
        CopyImage(R_1, S)
    test:
        for j = 0, ..., s − 1
            CopyImage(S, R_1)
            load_R(j)   /* sets t to the time it spends */
            if t > log s then   /* found a costly load */
                goto advance
            end if
        end for
        /* all the loads were cheap */
        S is the image sought.
```

**Figure 4.1**

memory images at most $\lfloor \log s \rfloor$ high, rooted in $R_1$. By Lemma 3, these images can be further encoded as **list**s of $\left\lfloor \frac{3}{4} 2^{\lfloor \log s \rfloor} \right\rfloor$ atoms, since $\{0, \ldots, 2^{\lfloor \log s \rfloor}\} \in \mathcal{D}$. But this **list** is shorter than $s$, a contradiction to the incompressibility of $\mathcal{T}$.

We have proved that $T(s+n, s) > n \log s$. Since this holds for every $n$ and $s$, we obtain an asymptotic lower bound of $\Omega(t \log s)$. □

Actually, we obtained a lower bound on the time of a *retrieval* operation from a set of $s$ data allowing for unbounded preprocessing time. In general, this is a stronger result than a lower bound on RASUS, which may be more difficult because all instructions must be simulated on-line. Also, it is easy to see that this lower bound holds also in the average where each of the $s$ addresses is equally likely to be queried. Using this observation, we can also deduce that a randomized solution cannot have a better upper bound on expected running time.

### 4.3. An Application: Finite Words

**Theorem 4.** *Let $\mathcal{T}_k$ be the data type of $k$-bit words, with any primitive functions whatsoever. Then $\mathcal{T}_k$ is incompressible.*

**Proof.** Let $\mathcal{D}$ be the set of all $k$-bit words; then $|\mathcal{D}| = 2^k$. Hence $|\mathcal{D}^{n-1}| = 2^{k(n-1)} < 2^{kn} = |\mathcal{D}^{n-1}|$. Therefore, no function from $\mathcal{D}^n$ to $\mathcal{D}^{n-1}$ can be invertible. □

**Corollary 1.** *Let $\mathcal{T}_k$ be the data type of $k$-bit words, with any primitive instructions whatsoever. Then the $\mathcal{T}$-FLM requires $\Omega(t \log s)$ time to simulate the $\mathcal{T}$-RAM.*

In the next two sections incompressibility proofs are given for other, more sophisticated, data types. The proofs apply new techniques to decision trees. The exact method depends on the data type at hand, but in general it can be summarized as follows: discover a property of the primitive functions that prevents them from serving as "encoders" or "decoders". Exploiting the structure of decision trees, show that this property can be extended to a property of the finitely computable functions.

## 5. SIMULATION OF THE REAL RAM

In this section the lower bound of §4 will be shown to apply to the simulation of the **real number** RAM. This model is the common abstraction used in algorithms dealing inherently with real numbers, such as in computational geometry and scientific computation. Shamos defines this model in [36] to have as *domain* the set of real numbers, and as primitive functions the arithmetic operations $(+, -, \times, \div)$ and other common functions (such as exponents and logarithms, trigonometric functions etc), as needed, imposing only the restriction that all functions used be analytic (except for a finite set of points). Mild as it is, this restriction disallows some very useful functions such as the floor function (truncation to an integer). In fact, he shows that one of his lower bounds does not hold with "floor" available. Our data type consists of the real numbers together with a set of primitives from the following set of functions.

**Definition.** *By $\mathcal{F}$ we denote in this section the set of functions $f: \Re^k \to \Re^m$, for any $k, m > 0$, such that for some countable, closed set $C \subset \Re^k$, $f$ is continuous in $\Re^k - C$.*

$\mathcal{F}$ includes all the functions mentioned above, analytic and not. Constants too may be seen as functions in this set, defined on $\Re$, which are just independent of their argument. $\mathcal{F}$ decision trees are similar to *algebraic decision trees* [8], but the latter have a much more restricted primitive set (algebraic functions only).

By selecting $\mathcal{F}$ as our primitive function set, we allow a model which is stronger than previous ones. The restriction on functions in $\mathcal{F}$ is aimed at allowing functions whose continuity is perturbed, but in a limited way so that by composing any number of these functions, the perturbation may not grow too much. (Bijections between $\Re^n$ and $\Re^{n-1}$ will be more "perturbed" than we allow).

Actually we show that under the above definition, the "perturbation" is limited to a special kind of set, a closed boundary set [24]. Recall that a *boundary set* is a subset $A$ of a topological space such that $A^\circ$ (the interior of $A$) is empty (equivalently, the complement of $A$ is a *dense* set). We denote the *complement* of $A$ (with respect to the space under consideration) by $\widetilde{A}$, the *closure* of $A$ by $\overline{A}$ and the *boundary* of $A$ by $\partial A = \overline{A} - A^\circ$. Note that $\partial A$ is always closed, because $\overline{A}$ is closed and $A^\circ$ is open. By a *neighborhood* (of a point) we mean any open set (containing the point). Thus a boundary set is one that contains no neighborhood.

In this paper we refer to $\Re^n$ as a topological space, actually the Euclidean $n$-space. This space is complete [24] which allows us to use the following theorem:

**Theorem 5** [Baire]. *In a complete space, a countable union of closed boundary sets is a boundary set.*

Recall that $\mathcal{F}^*$ denotes the closure of $\mathcal{F}$ under function composition and aggregation.

**Theorem 6.** *Let $k, m > 0$. Let $f: \Re^k \to \Re^m$ be computable by an $\mathcal{F}$-decision tree. Then there is an open set $O \subseteq \Re^k$ such that $f$ is continuous in $O$, and $\Re^k - O$ is a boundary set.*

The set $O$ displayed in the proof of this lemma will be called $f$'s *continuity domain.*

**Proof.** We begin by proving the result for functions in $\mathcal{F}^*$, and this by induction on the number of compositions and aggregations used to construct $f$ of functions from $\mathcal{F}$.
*Basis*: $f \in \mathcal{F}$. Then by definition, there is a countable, closed set $C \subset \Re^k$ such that for $O = \Re^k - C$, $f$ is continuous in $O$. $C$ is a boundary set, as every neighborhood in $\Re^k$ contains an uncountable number of points.
*Induction step*: Let us begin with aggregation. Assume $f: \Re^k \to \Re^n = (f_1, \ldots, f_n)$ and assume our statement holds for all $f_i$. Let $O_i$ be the continuity domain of $f_i$. Then $f$ is clearly continuous in $O = \bigcap_{i=1}^n O_i$. Now $\Re^k - O = \widetilde{O} = \bigcup_{i=1}^n \widetilde{O_i}$, a finite union of closed boundary sets. Thus $\Re^k - O$ is a closed boundary set.

Consider now $f = g(h)$ where $g \in \mathcal{F}$ and $h$ has a continuity domain $O \subseteq \Re^k$. Let $g$ be continuous in $\Re^n - C$. Let $\mathbf{x} \in O$; then $h$ is continuous at $\mathbf{x}$. If $h(\mathbf{x}) \notin C$, then $f$ is continuous at $\mathbf{x}$. Also, if $h$ is constant in some neighborhood of $\mathbf{x}$, $f$ is continuous at $\mathbf{x}$. Therefore the subset $X$ of $\Re^k$ where $f$ *is not continuous* satisfies

$$X \subseteq \widetilde{O} \cup \Big( \bigcup_{y \in C} \partial(h^{-1}(\{y\})) \cap O \Big). \qquad (*)$$

Consider the restriction $\hat{h}$ of $h$ to $O$, where it is continuous. Note that $\partial(h^{-1}(\{y\})) \cap O = \partial(\hat{h}^{-1}(\{y\}))$. We show now that $Y = \bigcup_{y \in C} \partial(\hat{h}^{-1}(\{y\}))$ is closed in the subspace $O$. Let $x \in O$ be an accumulation point of $Y$, a limit of a sequence $(x_i)$, where $x_i \in \partial(\hat{h}^{-1}(y_i))$. By continuity of $\hat{h}$ we have $\hat{h}(x) = \lim_{i\to\infty} \hat{h}(x_i)$. For all $i$, $\hat{h}(x_i) \in C$ and since $C$ is closed we have $y = \hat{h}(x) \in C$. Of course $x \in \hat{h}^{-1}(\{y\})$. Suppose it is in its interior. Then there exists some $k$ such that $x_k$ belongs to this interior too (actually infinitely many elements do). $h(x_k) = y$, therefore $x_k$ cannot belong to $\hat{h}^{-1}(y_i)$ for any $y_i \neq y$. Thus by the assumption on $(x_i)$, it must belong to $\partial\hat{h}^{-1}(\{y\})$, a contradiction. Therefore $x \in \partial(\hat{h}^{-1}(\{y\})) \subseteq Y$.

Given that $Y$ is closed in $O$, returning to the space $\Re^k$, we have

$$\overline{Y} \subseteq Y \cup \widetilde{O}.$$

By the induction hypothesis, $\widetilde{O}$ is a boundary set. Assume that $\overline{Y}$ contains a neighborhood $N$. Then $Y$ itself contains $N - \widetilde{O}$ which is an open set as $\widetilde{O}$ is closed, and is not empty as otherwise $N$ would be contained in $\widetilde{O}$. So we have a neighborhood contained in $Y \cap O$. But this contradicts Baire's theorem since $C$ being countable, $Y$ is a countable union of closed boundary sets. Therefore, $\overline{Y}$ is a boundary set. Returning to $X$, by $(*)$ we have

$$X \subseteq \widetilde{O} \cup \overline{Y},$$

which proves that $\overline{X}$ is a boundary set. Moreover, $f$ is continuous in $\Re^k - X \supseteq \Re^k - \overline{X}$, which is open, and hence fulfills our statement.

To extend the result to any function computable by an $\mathcal{F}$-decision tree, note that the following function on $\Re$ belongs to $\mathcal{F}$:

$$t(x) \stackrel{\text{def}}{=} \begin{cases} 1 & x > 0 \\ 0 & x \leq 0. \end{cases}$$

It is easy to see that each part of an $\mathcal{F}$-decision tree consisting of a node $\nu$, parent of two leaves $\lambda_1$ and $\lambda_2$, can be replaced by a single leaf computing

$$t(f_\nu(\mathbf{x}))f_{\lambda_1}(\mathbf{x}) + (1 - t(f_\nu(\mathbf{x}))f_{\lambda_2}(\mathbf{x})).$$

Repeating this process we can contract the whole tree to a single function in $\mathcal{F}^*$. □

We now display the property of continuous mappings which actually prohibits compression. This property is preservation of topological dimension. We begin by reviewing the definition of this concept and some of its properties.

**Definition** [18, III.1]. *The empty set and only the empty set has dimension* $-1$.
*A topological space* $X$ *has dimension* $\leq n$ $(n \geq 0)$ *at a point* $p$ *if* $p$ *has arbitrarily small neighborhoods whose boundaries have dimension* $\leq n-1$.
$X$ *has dimension* $\leq n$, $\dim X \leq n$, *if* $X$ *has dimension* $\leq n$ *at each of its points.*
$X$ *has dimension* $n$ *at a point* $p$ *if it is true that* $X$ *has dimension* $\leq n$ *at* $p$ *and it is false that* $X$ *has dimension* $\leq n-1$ *at* $p$.
$X$ *has dimension* $n$ *if* $\dim X \leq n$ *is true and* $\dim X \leq n-1$ *is false.*

The following two theorems are also from [18].

**Theorem 7.** *A subspace of a space of dimension* $\leq n$ *has dimension* $\leq n$.

**Theorem 8** [Brouwer]. *Euclidean* $n$*-space has dimension* $n$.

It is obvious that the property of having dimension $n$ is topologically invariant. Dimension is, however, not an invariant of continuous transformations: Peano's mapping of an interval on the whole of a square is an illustration of a continuous transformation which raises dimension, and projection of a plane into a line is an illustration of a transformation which lowers dimension. However, this transformation is not invertible, and thus does not interfere with incompressibility. Once the functions are required to be invertible, we obtain the property we need.

**Theorem 9.** *Let $f : X \to Y$ be continuous and invertible, where $X$ and $Y$ are spaces such that* $\dim Y = n$. *Then* $\dim f^{-1}(Y) \leq n$.

**Proof.** Use induction on $n$.
*Basis*: $n = -1$.

$$\dim Y = -1 \implies Y = \emptyset \implies f^{-1}(Y) = \emptyset \implies \dim f^{-1}(Y) = -1.$$

*Induction step*: assume the theorem holds for $n-1$. Let $p$ be an arbitrary point of $f^{-1}(Y)$ and $U$ an arbitrary neighborhood of $p$ therein. $f(U)$ is a subset of $Y$, hence by Theorem 7, $\dim f(U) \leq n$, whence there exists, in the subspace $f(U)$, a neighborhood $O$ of $f(p)$ such that $\dim \partial O \leq n-1$. By $f$ being continuous and invertible it follows that $f^{-1}(O)$ is a neighborhood of $p$, contained in $U$, and also that

$$\begin{aligned} \partial f^{-1}(O) \subseteq f^{-1}(\partial O) &\implies \text{(by the induction hypothesis and Theorem 7)} \\ \dim \partial f^{-1}(O) \leq n-1. & \end{aligned}$$

Thus $f^{-1}(O)$ is a neighborhood of $p$, arbitrarily small, and having boundary of dimension $\leq n-1$; since $p$ is an arbitrary point of $f^{-1}(Y)$, $\dim f^{-1}(Y) \leq n$ and the proof is complete. □

**Theorem 10.** *Let $\mathcal{T} = (\Re, \mathcal{F}, <)$. Then $\mathcal{T}$ is incompressible.*

**Proof.** Assume by contradiction that there is a function $f\colon \Re^n \to \Re^m$ ($m < n$), finitely computable in $\mathcal{T}$ which is invertible. By Theorem 6 there exists a non-empty, open set $O$ where $f$ is continuous. $f(O) \subseteq \Re^m$, hence by Theorem 7 and Brouwer's Theorem we have $\dim f(O) \leq m$. By Lemma 9, also $\dim O \leq m$. But $O$ contains a ball in $\Re^n$, which clearly has dimension $n$, hence by Theorem 7 we have $\dim O \geq n$, a contradiction. □

**Corollary 2.** *The real number FLM requires $\Omega(t \log s)$ time to simulate the real number RAM.*

These results carry over to the field of rational numbers. Here we will allow as primitives functions from $\mathcal{F}_{\mathbb{Q}}$ — the restriction of $\mathcal{F}$ to functions that when applied to rational arguments produce a rational result. We need a stronger version of Theorem 6, obtained by essentially the same proof. For the rest of this section, $\mathcal{X}$ denotes a complete, dense in itself [24] subspace of $\subseteq \Re^k$.

**Theorem 6'.** *Let $f\colon \mathcal{X} \to \Re^m$ be computable by an $\mathcal{F}$ decision tree. Then there is a set $O$, open in $\mathcal{X}$, such that f is continuous in $O$, and $\mathcal{X} - O$ is a boundary set in $\mathcal{X}$.*

**Theorem 11.** *Let $\mathcal{T} = (\mathbb{Q}, \mathcal{F}_{\mathbb{Q}}, <)$. Then $\mathcal{T}$ is incompressible.*

**Proof.** Assume in contradiction that there is an $\mathcal{F}_{\mathbb{Q}}$ decision tree computing a function $f$ that maps $\mathbb{Q}^n$ injectively in $\mathbb{Q}^m$, where $m < n$. Assume further that $g = f^{-1}$ is also finitely computable. Consider these functions as functions over the reals. As shown in Theorem 6, there exists a non-empty open set $O \subseteq \Re^n$ in which $f$ is continuous. $\overline{f(O)}$, as a subspace of $\Re^m$, is complete and dense in itself. By Theorem 6' we have a non-empty set $N' \subseteq \overline{f(O)}$, open therein, in which $g$ is continuous. The set $N = N' \cap f(O)$ cannot be empty and, relative to $f(O)$, is open. Thus $f^{-1}(N)$ is open in $O$, hence in $\Re^n$, and non-empty. We next show that $f$ maps $f^{-1}(N)$ injectively onto $N$. This will complete the proof because, as in the proof of Theorem 10, this cannot be true.

Let $\mathbf{x}^1, \mathbf{x}^2 \in f^{-1}(N)$ such that $f(\mathbf{x}^1) = f(\mathbf{x}^2) = \mathbf{y}$. $\mathbf{x}^1$ and $\mathbf{x}^2$ may not both be rational; but anyway there are rational *sequences* $\mathbf{x}_i^1 \to \mathbf{x}^1$, $\mathbf{x}_i^2 \to \mathbf{x}^2$ in $f^{-1}(N)$. Let $\mathbf{y}_i^j = f(\mathbf{x}_i^j) \in N$. By continuity of $f$ we have

$$\mathbf{y} = f(\mathbf{x}^1) = \lim_{i\to\infty} f(\mathbf{x}_i^1) = \lim_{i\to\infty} \mathbf{y}_i^1$$

entailing (by continuity of $g$)

$$g(\mathbf{y}) = \lim_{i\to\infty} g(\mathbf{y}_i^1) = \lim_{i\to\infty} \mathbf{x}_i^1 = \mathbf{x}^1.$$

Similarly we get $g(\mathbf{y}) = \mathbf{x}^2$. Hence $\mathbf{x}^1 = \mathbf{x}^2$. □

**Corollary 3.** *The rational FLM requires $\Omega(t \log s)$ time to simulate the rational RAM.*

## 6. SIMULATION OF THE INTEGER RAM

The most common data type for the RAM is undoubtedly the integers. The domain of our data type is $\mathbb{N}$ (the non-negative integers). We define the function of *subtraction* on $\mathbb{N}$ as producing 0 if the minuend is smaller than the subtrahend ("positive difference"). The choice of the *instruction set* for this data type is more subtle than for the previous two, since using a rather natural instruction set, integers may be compressed. In fact, there exists a solution to RASUS in the domain of integers which runs in $O(t\alpha(s))$ time, where the function $\alpha(s)$ is a (very slow growing) inverse of Ackermann's function [7]. In this simulation only the operations *addition, subtraction and shift* are used. The power of this algorithm stems apparently from the use of **shift**.

For the **shift** operation, we denote by $\text{shift}(x, y)$ the result of shifting $x$ *left* by $y$ bits, i.e. $x \cdot 2^y$. We denote by $\text{shift}(x, -y)$ the result of shifting $x$ *right* by $y$ bits, i.e. $\lfloor x/2^y \rfloor$.

The next theorem demonstrates that the right shift instruction is the source of the "breakthrough" in the above algorithm, by showing that excluding it from the instruction set leaves an incompressible data type.

## 6.1. An Incompressibility Theorem for Integers

**Theorem 12.** *Let $\mathcal{T} = (\mathbb{N}, \mathcal{F}, <)$, where $\mathcal{F}$ includes addition, subtraction, multiplication, any Boolean (bitwise) operations, integer division by two and left shift. Then $\mathcal{T}$ is incompressible.*

The property shared by all these functions, leading to the incompressibility result, is expressed by the following lemma.

**Lemma 4.** *Let $\mathcal{F}$ be the primitive function set of Theorem 12. Let $f\colon \mathbb{N}^n \to \mathbb{N}^m \in \mathcal{F}^*$, and $\mathbf{x}, \mathbf{y} \in \mathbb{N}^n$. Let $h$ be the number of sub-expressions of the form $\lfloor g/2 \rfloor$ that occur in $f$, and let $k > h$. Assume that for each sub-expression of $f$ of the form $(f_1 - f_2)$, or* $\mathrm{shift}(g_1, g_2)$*, it is guaranteed that*

$$\begin{aligned} f_1(\mathbf{x}) \geq f_2(\mathbf{x}) &\iff f_1(\mathbf{y}) \geq f_2(\mathbf{y}), \\ \text{and} \qquad g_2(\mathbf{x}) < k &\iff g_2(\mathbf{y}) < k. \end{aligned}$$

*It then follows that*

$$\mathbf{x} \equiv \mathbf{y} \pmod{2^k} \implies f(\mathbf{x}) \equiv f(\mathbf{y}) \pmod{2^{k-h}}.$$

**Proof.** We prove that each of the primitives composing $f$, except $\lfloor g/2 \rfloor$, preserves congruence modulo $2^k$. This is well known for addition, subtraction and multiplication. Our use of *positive difference* does no harm because we are guaranteed that in evaluating $f(\mathbf{x})$ and $f(\mathbf{y})$, the operation either coincides with integer subtraction for both or produces zero for both. For the Boolean operations, the right $k$ bits of the result are determined by the right $k$ bits of the operands, and therefore they preserve congruence. The right $k$ bits of $\mathrm{shift}(a, b)$ are determined by the right $k$ bits of $a$ and by the value of $b$. We have two cases: either $b < k$ for both $\mathbf{x}$ and $\mathbf{y}$, which implies $g_2(\mathbf{x}) = g_2(\mathbf{y})$, or $b \geq k$ for both, whence the right $k$ bits of both results are zero. Hence the left shift too preserves congruence $\bmod 2^k$.

The lemma follows by structural induction on $f$.
*Basis*: $f \in \mathcal{F}$. There are two cases. (i) $f$ is not $\lfloor x/2 \rfloor$. Then as shown above, $f$ preserves congruence modulo $2^k = 2^{k-h}$ and the lemma holds. (ii) $f(x) = \lfloor x/2 \rfloor$. Then it is well known that

$$x \equiv y \pmod{2^k} \implies f(x) \equiv f(y) \pmod{2^{k-1}}.$$

Since $h = 1$, the lemma holds.
*Induction step*: a non-primitive $f$ may be constructed from simpler functions by means of *function composition* and *aggregation*. We first note generally that for $0 \leq i \leq j \leq k$, congruence modulo $2^{k-i}$ entails congruence modulo $2^{k-j}$. We begin with aggregation. Let $f = (f_1, \ldots, f_n)$ satisfy the lemma's assumptions. Then they are clearly satisfied by each of the $f_i$. For $i = 1, \ldots, n$ let $h_i$ be the number of sub-expressions of the form $\lfloor g/2 \rfloor$ occurring in $f_i$. Then the number of such occurrences in $f$ is $h = \sum_{i=1}^{n} h_i$. In particular,

$h \geq h_i$ for all $i$. Hence by the induction hypothesis, each $f_i$ preserves congruence modulo $2^{k-h_i}$ and therefore modulo $2^{k-h}$. Put together, $f$ preserves congruence modulo $2^{k-h}$.

Consider now $f = g(\phi)$. Fulfillment of the lemma's assumptions by $f$, $\mathbf{x}$ and $\mathbf{y}$ means that they are satisfied by $\phi$, $\mathbf{x}$ and $\mathbf{y}$ and by $g$, $\phi(\mathbf{x})$ and $\phi(\mathbf{y})$. Let $h_1$ be the number of occurrences of division by two in $\phi$ and $h_2$ the number of such occurrences in $g$. By the induction hypothesis,

$$\begin{aligned}\mathbf{x} \equiv \mathbf{y} \pmod{2^k} &\Rightarrow \phi(\mathbf{x}) \equiv \phi(\mathbf{y}) \pmod{2^{k-h_1}} \\ &\Rightarrow g(\phi(\mathbf{x})) \equiv g(\phi(\mathbf{y})) \pmod{2^{k-h_1-h_2}}.\end{aligned}$$

For $f$ the number of occurrences is $h = h_1 + h_2$ and hence we get $f(\mathbf{x}) \equiv f(\mathbf{y}) \pmod{2^{k-h}}$. □

We are now ready to prove the theorem.

Assume by contradiction that there is a decision tree in our model that computes a surjective function from $\mathbb{N}^m$ to $\mathbb{N}^n$ where $n > m$ (this is required for a "decoding algorithm"). Let $t$ be the number of leaves in this tree. For each leaf $\lambda$, let $R_\lambda$ be the set of $n$-tuples produced at this leaf. Then

$$\bigcup_\lambda R_\lambda = \mathbb{N}^n.$$

Let $h$ be the maximal number of occurrences of division by two in the functions of the leaves. Choose $k > \log t + h$. For each leaf $\lambda$ denote by $X_\lambda$ the set of $m$-tuples for which the tree delivers the result at $\lambda$. Then $R_\lambda = f_\lambda(X_\lambda)$, with $f_\lambda \in \mathcal{F}^*$. We can assume that for all pairs $\mathbf{x}, \mathbf{y} \in X_\lambda$ the assumptions of Lemma 4 hold with respect to $f_\lambda$, since otherwise we can enforce them by replacing $\lambda$ with a finite subtree whose nodes test all the relations mentioned in the lemma such that for each leaf only one outcome of these tests is possible.

Hence it follows from the lemma that

$$|R_\lambda \bmod 2^{k-h}| = |f_\lambda(X_\lambda) \bmod 2^{k-h}| \leq |X_\lambda \bmod 2^k| \leq |\mathbb{N}^m \bmod 2^k| = 2^{km}.$$

Therefore

$$\begin{aligned}|\mathbb{N}^n \bmod 2^k| &= |\bigcup_\lambda R_\lambda \bmod 2^k| \\ &\leq 2^h |\bigcup_\lambda R_\lambda \bmod 2^{k-h}| \\ &\leq 2^h \sum_\lambda |R_\lambda \bmod 2^{k-h}| \\ &\leq 2^h \cdot t \cdot 2^{km} < 2^k \cdot 2^{km} = 2^{k(m+1)} \leq 2^{kn}\end{aligned}$$

while actually, $|\mathbb{N}^n \bmod 2^k| = 2^{kn}$ — a contradiction. □

**Corollary 4.** *The integer FLM (with the primitives of Theorem 12) requires $\Omega(t \log s)$ time to simulate the integer RAM.*

**Remark.** Mathematically, *left shift* is a special case of *exponentiation* $a^b$, which could have been entered into our primitive set as such; and we would still have incompressibility, although the proof gets a little harder. Exponentiation has a more arithmetic flavor than left shift, and correspondingly, we would add *integer division* instead of adding *right shift* in our next theorem. This would allow to implement the right shift as $\lfloor x/2^y \rfloor$.

### 6.2. A Compression Scheme Using Shifts

Let the binary representation of $x$ be $b_n \dots b_1 b_0$. We denote by $\mathrm{Right}_d(x)$ the right $d$ bits of $x$, i.e. the number $b_{d-1} \dots b_1 b_0$. $\mathrm{Right}_d$ is computable by

$$\mathrm{Right}_d(x) = x - \mathrm{shift}(\mathrm{shift}(x, -d), d).$$

By $\mathrm{Mid}_{i,d}(x)$, where $i, d \geq 0$, we denote the number $b_{i+d-1} \dots b_i$, computed by

$$\mathrm{Mid}_{i,d}(x) = \mathrm{Right}_d(\mathrm{shift}(x, -i)).$$

Our compression function encodes $n$ integers, for any fixed $n$, in just a pair of numbers. The compressed structure is called *a packet.*

A packet $P$ represents an $n$-tuple of integers $P[0], \dots, P[n-1]$. We write $P = P^{(n)}$ for emphasis. The actual representation of a packet is as the pair $(a, b)$ where for all $0 \leq i < n$, $P[i] \in \{0, \dots, 2^b - 1\}$ and $a = \sum_{i=0}^{n-1} \mathrm{shift}(P[i], ib)$.

Building a packet, given the $n$ numbers, is straightforward; we are free to choose $b$ at our convenience, for example as the maximum of $P[0], \dots, P[n-1]$. In order to do without multiplication, we can use for the "block size" of the packet not $b$ but $2^b$, exploiting the fact that $i \cdot 2^b = \mathrm{shift}(i, b)$. The packet would then be defined by $a = \sum_{i=0}^{n} \mathrm{shift}(P[i], i \cdot 2^b)$, and multiplication would not be required.

Each of the packet members can be retrieved in constant time, using the following relation (using the form with multiplication):

$$P[i] = \mathrm{Mid}_{ib,b}(a).$$

This allows for a decompression function of $O(n)$ running time. We omit a detailed verification which is straight-forward. It is worth mentioning, however, how this compression scheme can be employed by a fast RASUS. Clearly, by encoding all the used portion of the RAM's memory in a packet, a constant time *load* is easily achieved. However, for each store we may have to recreate the packet because of the new datum being larger than the block size. To obtain a fast *store*, we have to use a more complex structure. A solution for this problem is given in [7], where packet hierarchies are maintained to obtain a RASUS of $O(\alpha(s))$ time per operation, where $\alpha(s)$ is a functional inverse of Ackermann's function. This function grows slower than every inverse of a primitive recursive function. The computational model used in [7] is weaker than the PLM; it has for memory just a finite number of registers.

## 7. HISTORICAL NOTES

In the introduction we described a machine model used by Tarjan [38,39] which resembles the LISP machine. However, its usage in this paper was very specific, not using the power of high-level data types. Thus it was closer, as Tarjan states, to a machine model previously suggested by Knuth [19]. His model, called a *linking automaton*, has the same structure as Tarjan's pointer machine (similar to the FLM), but was constrained to operate on a finite alphabet of symbols.

This machine belongs to the class of *combinatory* or *discrete* machines, such as Turing's machine, which arose from the theory of computability, and are still dominant in the more fundamental areas of complexity theory. These machines operate on a finite alphabet, or even less: a similar model, called a *storage modification machine* (SMM), has been independently proposed by Schönhage [33,34,35]. This machine operates on a memory which does not store any kind of "data"; the information is comprised in the linking pattern of the nodes. The SMM is easily proved to be real-time equivalent to the linking automaton. Schönhage's comprehensive paper [35] defines the model, shows that the SMM can simulate a multidimensional Turing machine in real time, and proves Schnorr's observation that they are real-time equivalent to the so-called successor RAMs [32]. The *successor RAM* (SRAM) is a machine with a memory addressable by $\mathbb{N}$, but an input/output alphabet of $\{0, 1\}$ and an internal data type of integers including only the successor function and comparison for equality. Thus this machine falls short of our "high-level" assumptions (§0), and this explains its weakness: it has such a feeble data type that instead of addressing memory with its "data", you can do as well with pointers.

We should also note the first appearance of an abstract computing machine which operates on a graph structure, proposed by Kolmogorov and Uspenskii [21,23]. Their model was intended to give a flexible and realistic definition of an algorithm. It is different from the above by having its memory graph *undirected*, with a bounded node degree. The "programming" of the machine is also different, since it is mathematically oriented and not similar to actual programming as in the other machines. Kolmogorov-Uspenskii machines (KUMs) can be simulated in real time by SMMs, but whether this is possible in the inverse direction is not yet known[1].

Following Tarjan's paper, several authors have used the pointer machine as a model for studying data structure problems. Blum [10] complements Tarjan's paper in studying the disjoint set union problem. Harel and Tarjan [15] demonstrate a lower bound of $\Omega(m \log\log n)$ for performing $m$ *nearest common ancestor* queries on a pointer machine. In contrast, an $O(m)$ algorithm is given for a RAM. They present this problem as a case for comparing the pointer machine with the RAM. However, their result does not shed light on our problem, since their formulation of the problem for the pointer machine lower bound differs from that for the RAM; while the RAM represents tree nodes as integers, the pointer machine is to represent them as unique nodes.

---

[1] This appears as problem 8 of [14].

A similar result appears in [25], where Mehlhorn, Näher and Alt show tight bounds for some variants of the union-split-find problem. One of the variants, which can be solved by a RAM in $O(1)$ steps per operation, requires $O(\log\log n)$ pointer machine steps per operation or $O(\log n)$ steps if set separability is assumed (a further restriction on the representation of sets in the pointer machine).

Chazelle [11] also gives a lower bound for a pointer machine. He addresses *orthogonal range searching* in $d$ dimensions (where all the $s$ answers to the query are to be reported). On the pointer machine, a query time of $O(s + \mathrm{polylog}(n))$ time can only be achieved at the expense of $\Omega(n(\log n/\log\log n)^{d-1})$ space, which is optimal. On the contrary, as he showed in [12], a RAM can process a query in time $O(s + \log n)$ using $O(n \log^{\epsilon} n)$ storage, for any $\epsilon > 0$. Thus in his case, the pointer machine requires asymptotically more storage to achieve the same query time. However, here again the problem is presented to the pointer model in a way which makes it harder.

Paige [26] presents considerations similar to ours with respect to the choice of models and defines a high level RAM as the machine model of an *array processing language* and a pointer machine as modeling a *list-based language*. This resulted in models that are practically equivalent to our $\mathcal{T}$-RAM and $\mathcal{T}$-LM. He even discusses the simulation of a RAM by the list machine, and complements the results of this paper by describing a large class of practical algorithms that can be simulated by a LM without asymptotic overhead.

## 8. CONCLUSION

We have defined the *incompressibility* property for a general data type. Using this property we have proved an $O(t \log s)$ tight bound on the time required by a pointer machine to simulate a RAM. Our models of computation are defined in a general way for a high-level data type, extending the traditional definition of such models. This may be the first time that a complexity lower bound is proved in such a general and powerful model.

Actually, we presented the RAM and LISP machines as representatives of a class of high-level models, the $\mathcal{T}$-machines. These make an attractive setting for the design and analysis of algorithms because in fact, algorithm designers have already assumed such flexibility. Incompressibility proved to be a tool strong enough for proving lower bounds in models of this kind.

We have proved the incompressibility property for the data types of finite words, integer and real numbers with a generous set of primitives. The primitive sets are large enough to be useful in practical, high level algorithms and are perhaps the strongest for which lower bounds have been achieved. For the integers, we exclude the right shift operation, which is shown to induce compressibility.

The proof of the lower bound followed the following scheme: we first proved a lower bound in a model whose power is restricted by assuming the data to be *symbols*. The proof generalized to more complex data types in a natural way by replacing the restriction of symbols by the property of incompressibility. This scheme may carry on to other problems where a lower bound is known assuming symbols are used, or where other similar assumptions have been made, as in Aggarwal and Vitter's lower bounds on I/O complexity [2], who call their restrictive assumption "indivisibility of the records" and pose as an open problem the task of relaxing this assumption.

The family of high-level models includes further variations on the theme of memory access. The *Hierarchical Memory Model* (abbreviated HMM), introduced by Aggarwal et al. [1], operates exactly as the RAM, but memory access instructions are charged according to the address used. While all other instructions cost one, an access to memory location $a$ costs proportionally to $\log a$ (or to another cost function). The virtues of this model are discussed in [1] and we will not repeat them here. A natural extension of our study is comparing this model to the LM just as we did for the RAM. This problem is addressed in [4]. It is shown that the HMM relates to the LM much as the LM relates to the RAM. The proofs of this relationship resemble those of the previous chapters, providing further evidence to the adequacy of incompressibility arguments to this kind of problems.

## NOTE ADDED IN PROOF

It was remarked in Section 7 that some of the published complexity results attributed to "pointer machines" are incomparable with results for the RAM, because the pointer machine was not given the same problem. To make this distinction clear, we suggest to adopt the term *pointer algorithms* for procedures which are given a linked structure and whose task is to modify it in some manner. Such procedures can be used as subroutines in larger programs, but have no independent value. The results in [11,15,25] pertain to pointer algorithms and are not applicable to solutions of the stand-alone problem where the input is given in raw form, say as numbers. It is the latter type of problem which was solved by the RAM algorithms of [12,15,25]. On the other hand, among the papers referenced, stand-alone algorithms for *pointer machines*, allowing for comparison of machine models, only appear in Paige's work [26] and in our papers.

## ACKNOWLEDGMENTS

The authors wish to thank Michael Tarsi for suggesting the use of Brouwer's Theorem in Section 5. Also the comments of an anonymous refree helped much in debugging the paper.

## REFERENCES


[1] A. Aggarwal, B. Alpern, A. K. Chandra and M. Snir, "A model for hierarchical memory," *Proc. 19th Annual ACM Symp. on Theory of Computing* (1987), 305–314.

[2] A. Aggarwal and J. S. Vitter, "The I/O complexity of sorting and related problems," *Proc. 14th ICALP* (1987), 467–478.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass. 1974.

[4] A. M. Ben-Amram, "On adresses versus pointers," manuscript.

[5] A. M. Ben-Amram, *On Pointers versus Addreses*. MSc Thesis, Department of Computer Science, Tel-Aviv University, 1988.

[6] A. M. Ben-Amram and Z. Galil, "On pointers versus addresses," *Proc. Twenty-Ninth Annual IEEE Symp. on Foundations of Computer Science*, White Planes, NY 1988.

[7] A. M. Ben-Amram and Z. Galil, "On the power of shifing integers," manuscript.

[8] M. Ben-Or, "Lower bounds on algebraic computation trees," *Proc. Fifteenth Annual ACM Symp. on Theory of Computing* (1983), 80–86.

[9] L. Blum, M. Shub and S. Smale, "On a theory of computation over the real numbers; NP completeness, recursive functions and universal machines," *Proc. Twenty-Ninth Annual IEEE Symp. on Foundations of Computer Science*, White Planes, NY 1988.

[10] N. Blum, "On the single-operation worst-case time complexity of the disjoint set union problem," *SIAM J. Comput.* 15:4 (1986), 1021–1024.

[11] B. Chazelle, "Lower bounds on the complexity of multidimensional searching," *Proc. Twenty-Seventh Annual IEEE Symp. on Foundations of Computer Science* (1986), 87–96.

[12] B. Chazelle, "A functional approach to data structures and its use in multidimensional searching," Brown Univ. TR CS-85-16 (1985). Preliminary version in *Proc. Twenty-Sixth Annual IEEE Symp. on Foundations of Computer Science* (1985), 165–174.

[13] S. A. Cook and R. A. Reckhow, "Time bounded random access machines," *J. Comput. System Sci.* 7:4 (1973), 354–375.

[14] P. Duris, Z. Galil, W. Paul and R. Reischuk, "Two nonlinear lower bounds for on-line computations," *Information and Control* 60 (1984), 1–11.

[15] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.* 13 (1984), 338–355.

[16] S. Hart and M. Sharir, "Nonlinearity of Davenport-Schinzel sequences and of a generalized path compression scheme," *Proc. Twenty-Fifth Annual IEEE Symp. on Foundations of Computer Science*, Singer Island, Fl., 1984.

[17] J. Hartmanis and J. Simon, "On the power of multiplication in random access machines," *15th Annual Symp. on Switching and Automata Theory* (1974), 13–23.

[18] W. Hurewicz and H. Wallman, *Dimension Theory.* Princeton University Press, Princeton 1948.

[19] D. E. Knuth, *The Art of Computer Programming*, Vol. 1: *Fundamental Algorithms.* Addison-Wesley, Reading, Mass. 1968.

[20] D. E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching.* Addison-Wesley, Reading, Mass. 1975.

[21] A. N. Kolmogorov, "On the notion of algorithm," *Uspehi Mat. Nauk.* 8 (1953), 175–176.

[22] A. N. Kolmogorov, "Three approaches to the definition of the concept of 'amount of information'," *Problemy Peredači Informacii* 1, 1 (1965), 3–11. English translation in *Selected transl. in Math. Statistics and Probability*, Vol. 7, 293–302.

[23] A. N. Kolmogorov and V. A. Uspenskii, "On the definition of an algorithm," *Uspehi Mat. Nauk.* 13 (1958), 3–28. English translation in *AMS transl.* II Vol. 29 (1963), 217–245.

[24] K. Kuratowski, *Introduction to Set Theory and Topology*, L. F. Boron (transl.), Addison-Wesley, Reading, Mass. 1962.

[25] K. Mehlhorn, S. Näher and H. Alt, "A lower bound for the complexity of the union-split-find problem," *Proc. 14th ICALP* (1987), 479–488.

[26] R. Paige, "Real-time simulation of a set machine on a RAM," in *Computing and Information ICCI '89*, Vol. 2, R. Janicki and W. Koczkodaj (eds.), 1989.

[27] W. J. Paul, "Kolmogorov complexity and lower bounds," *Fundamentals of Computation Theory FCT '79*, Lothar Budach (ed.), Akademie-Verlag, Berlin, 1979, 325–334.

[28] W. J. Paul, J. I. Seiferas and J. Simon, "An information-theoretic approach to time bounds for on-line computation," *J. Comput. System Sci.* 23:2 (1981), 108–126.

[29] V. R. Pratt, M. O. Rabin and L. J. Stockmeyer, "A characterization of the power of vector machines," *Proc. 6th annual ACM Symp. on Theory of Computing*, Seattle, Washington, 1974.

[30] F. P. Preparata and M. I. Shamos, *Computational Geometry: an Introduction,* Springer-Verlag, New-York, 1985.

[31] N. Sarnak and R. E. Tarjan, "Planar point location using persistent search trees," *Comm. of the ACM* 29, 7 (July 1986), 669–679.

[32] C. P. Schnorr, *Rekursive Funktionen und ihre Komplexität.* Teubner, Stuttgart, 1974.

[33] A. Schönhage, "Real-time simulation of multi-dimensional Turing Machines by storage modification machines," *Project MAC Technical Memorandum 37*, MIT (1973).

[34] A. Schönhage, "Storage modification machines," *Theoretical Computer Science — 4th GI Conf.*, Aachen 1979.

[35] A. Schönhage, "Storage modification machines," *SIAM J. Comput.* 9:3 (1980), 490–508.

[36] M. I. Shamos, *Computational Geometry.* PhD Thesis, Yale 1978.

[37] L. Siklóssy, *Let's Talk LISP.* Prentice-Hall Inc., Englewood Cliffs, NJ 1976.

[38] R. E. Tarjan, "Reference machines require nonlinear time to maintain disjoint sets," *Proc. Ninth Annual ACM Symp. on Theory of Computing*, Boulder, Colo., 1977.

[39] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. System Sci.* 18 (1979), 110–127.