Journal of Universal Computer Science, vol. 3, no. 4 (1997), 247-278
submitted: 20/12/96, accepted: 21/4/97, appeared: 28/4/97 © Springer Pub. Co.

The Linear Time Hierarchy Theorems for Abstract State Machines and RAMs

Andreas Blass[1]

Yuri Gurevich[2]

Abstract: We prove the Linear Time Hierarchy Theorems for random access machines and Gurevich abstract state machines. One long-term goal of this line or research is to prove lower bounds for natural linear time problems.

1 Introduction

In [Jones 93], Neil Jones exhibited a computation model where the linear speed-up theorem fails and, instead, a linear time hierarchy theorem holds: There exists a positive constant c such that, for every positive d, $\mathrm{Time}(c{\cdot}d{\cdot}n) - \mathrm{Time}(d{\cdot}n) \neq \emptyset$. In fact, Jones exhibited several computation models of that kind. One long-term goal of this exciting line or research is to prove lower bounds for linear time problems. The models described in detail in Jones's paper were particularly designed for the linear time hierarchy theorem, but he mentioned that the linear time hierarchy theorem also holds for Schönhage's storage modification machines [Schoenhage 80]. In an appropriate sense, storage modification machines are equivalent to random access machine whose only arithmetical operation is successor (Successor RAMs) [Schoenhage 80].

In general, even the Successor RAM model is too restrictive for many applications, e.g., computational geometry. One result of this paper is the Linear Time Hierarchy Theorem for random access machines with the usual arithmetical operations; see Section 9.

Contrary to polynomial time, linear time is not a robust notion. Successor RAMs, RAMs with addition, RAMs with addition and multiplication, etc. all give different versions of linear time. The most versatile machine model we know is the abstract state machine (ASM) model [Gurevich 91, Gurevich 95]. These machines are also known as Gurevich (abstract state) machines or evolving algebras. Unary ASMs (using no basic functions of arity ≥ 2) with limited interaction with the outside world are equivalent to the storage modification machines of Schönhage [Dexter, Doyle and Gurevich 97]. One can easily define a kind of ASMs equivalent to Addition RAMs, Addition and Multiplication RAMs, etc. According to the ASM thesis, any algorithm can be closely simulated, on its natural abstraction level, by an appropriate ASM [Gurevich 91]. The thesis has been confirmed in numerous applications [Boerger 95, dell Castillo 97, Huggins 97]. The ASM model offers an abstract parametrized version of linear time. Tell us

[1] Partially supported by NSF grant DMS–9505118. Mathematics Department, University of Michigan, Ann Arbor, MI 48109-1109, ablass@umich.edu

[2] Partially supported by NSF and ONR. EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, gurevich@umich.edu

what operations you suppose to be performed in constant time and we will tailor an appropriate ASM model.

In this paper, we prove the linear time hierarchy theorem for sequential ASMs with arbitrarily (but finitely) many internal dynamic functions and no restriction on the arities. We presume, however, that there are no external functions and thus the ASMs do not interact with the environment. Elsewhere, we intend to generalize the linear time hierachy theorem to reactive ASMs [Blass and Gurevich 97]. We presume also that initial states do not have any predefined static non-logic universes and functions. But incorporating static universes and functions is relatively easy; we explain the necessary changes.

The paper is organized as follows.

Section 2: Preliminaries. A particular notion of lock-step simulation is introduced. Roughly speaking, a simulation of one algorithm by another is *lock-step*[3] if the simulating algorithm (the "predator") makes only a bounded number of steps to simulate one step of the simulated algorithm (the "prey") [Gurevich 91]. The term "lock-step" replaces the term "real time" used by Michael Rabin and others in 1960s; the reason is that the term "real time" has a different meaning now. The notions of equivalence mentioned above utilize lock-step simulations.

Section 3: Abstract State Machines. We recall the notion of abstract state machine appropriate for this paper.

Section 4: More on Abstract State Machines. We make particular input/output conventions and define (skipping some details) the honest time for ASMs.

Section 5: Diagonalization Theorem. Given an arbitrary vocabulary Υ, we describe a diagonalizing Υ-ASM (the predator) that lock-step simulates any Υ-ASM (the prey) and then changes the output (if any); the actual predator program is found in Section 6. The simulation is with preprocessing, and it is not required that the predator halts if the prey's computation hangs. It is possible to achieve a simulation without preprocessing and with hang-state recognition but the proof is much harder and is not needed for proving the ASM linear time hierarchy theorem. We intend to give the proof in [Blass and Gurevich 97] where these improvements are needed.

Section 6: Diagonalization Program. We interleave the rules of the diagonalization program with explanations.

Section 7: ASM Linear Time Hierarchy Theorem. We prove the hierarchy theorem and give some comments. The theorem has this form: For every vocabulary Υ with enough nullary and unary function names, there exists a constant c such that, for every positive integer d, $\mathrm{Time}(c \cdot d \cdot n) - \mathrm{Time}(d \cdot n) \neq \emptyset$.

Section 8: Comments. We comment on the proof of the ASM linear time hierarchy theorem and, in particular, explain how to generalize it to the case where initial states contain some predefined static universes and functions, e.g. a copy of integers (or reals) with standard order, addition and multiplication.

Section 9: Random Access Machines. We prove the Diagonaliziation Theorem for RAMs and then derive the RAM Linear Time Hierarchy Theorem. In the case of RAMs, the diagonalizing predator lock-step simulates a given prey without preprocessing. A recognition of hang states can be easily achieved as well.

The results of this paper have been announced in [Blass and Gurevich 94].

2 Preliminaries

2.1 Provisos

In this paper, we consider a restricted class of algorithms satisfying the following conditions. An algorithm $\mathcal{A}$ has a program Program($\mathcal{A}$) and a well defined collection States($\mathcal{A}$) of states. The collection States($\mathcal{A}$) has distinguished subcollections of initial and final states. Each initial state *is* an input of $\mathcal{A}$ and may be called an *input state*. Each final state yields a Boolean output.

The states of $\mathcal{A}$ split into *enabling* states where $\mathcal{A}$ is *enabled* and *disabling* states where $\mathcal{A}$ is *disabled*. Any final state is disabling. Non-final disabling states are *hang states*.

In one step, $\mathcal{A}$ transforms a given enabling state A into a state A'; symbolically $A \Rightarrow_{\mathcal{A}} A'$. The state A' is the *successor* of A with respect to $\mathcal{A}$. Further, let $\Rightarrow^k_{\mathcal{A}}$ be the k-fold iteration of the successor relation $\Rightarrow_{\mathcal{A}}$, let $\Rightarrow^+_{\mathcal{A}}$ be the transitive closure of $\Rightarrow_{\mathcal{A}}$, and let $\Rightarrow^*_{\mathcal{A}}$ be the reflexive transitive closure of $\Rightarrow_{\mathcal{A}}$. A state A of $\mathcal{A}$ is *reachable* if $I \Rightarrow^*_{\mathcal{A}} A$ for some initial state I.

In this paper, the letters $\mathcal{A}$ and $\mathcal{B}$ are reserved to denote algorithms.

2.2 Runs

Recall that the first infinite ordinal ω is the collection of all natural (that is non-negative integer) numbers. For every ordinal α, $\alpha < \omega$ if and only if $\alpha \in \omega$. Let λ range over non-empty ordinals $\leq \omega$.

Definition 1 Let ρ be a sequence $\langle A_i : i < \lambda \rangle$ of states of $\mathcal{A}$.

- ρ is a *partial run* of $\mathcal{A}$ if $A_i \Rightarrow_{\mathcal{A}} A_{i+1}$ for all i with $i+1 < \lambda$. If λ is finite, then the number $\lambda - 1$ is the *length* of ρ. (Notice that the length is the number of steps rather than the number of states.)
- A partial run ρ is a *run* of $\mathcal{A}$ if (i) A_0 is an initial state, and (ii) if λ is finite, then $A_{\lambda-1}$ is disabling. (Notice that there is no interaction between an algorithm and the environment. An initial state uniquely defines the run.)
- Suppose that ρ is a run. If λ is finite and the state $A_{\lambda-1}$ is final, then $\mathcal{A}$ *converges* on A_0; otherwise it *diverges* on A_0. If $\mathcal{A}$ converges on A_0, then the output yielded by $A_{\lambda-1}$ is the *output* of $\mathcal{A}$ on A_0. If ρ is finite but $\mathcal{A}$ diverges on A_0, then $\mathcal{A}$ *hangs* on A_0.

□

$\mathcal{A}$ *accepts* (respectively *rejects*) an input state I if it converges on I and outputs *true* (respectively *false*).

2.3 Simulations

To distinguish more vividly between the simulated and simulating algorithms, we call the first a *prey* and the second a *predator*. We are primarily interested in the case where initial states of the prey are also initial states of the predator.

Definition 2 [Preliminary Simulation Definition] Let ψ be a mapping from States($\mathcal{A}$) to States($\mathcal{B}$). States in the range of ψ are *pose states* of $\mathcal{B}$. $\mathcal{B}$ *simulates* $\mathcal{A}$ *with respect to* ψ if, for every run $\langle A_i : i < \lambda \rangle$ of $\mathcal{A}$, the following condition is satisfied.

S0 The pose states of $\mathcal{B}$ in the run of $\mathcal{B}$ on A_0 form the sequence $\langle \psi A_i : i < \lambda \rangle$ (for the same λ). Furthermore, if λ is finite and $A_{\lambda-1}$ is final, then $\psi A_{\lambda-1}$ is final and yields the same output.

□

The intuitive meaning of ψ is encoding. A state A of the prey is encoded as the state ψA of the predator. It is not required in general that ψA_0 is an initial state of the predator; this allow some preprocessing. Thus the predator starts with some preprocessing and eventually arrives at the pose state ψA_0. Then it simulates the first step of $\mathcal{A}$ until it arrives at the pose state ψA_1. And so on.

One natural strengthening of Definition 2 is to forbid preprocessing: $\mathcal{B}$ simulates $\mathcal{A}$ *without preprocessing* with respect to the given encoding ψ if, in addition to S0, the following condition is satisfied.

S1 $\psi A_0 = A_0$.

Another natural strenghtening is that the predator recognizes not only final but all disabling states: $\mathcal{B}$ simulates $\mathcal{A}$ *with disabling state recognition* with respect to the given encoding ψ if, in addition to S0, the following condition is satisfied.

S2 If λ is finite and $A_{\lambda-1}$ is a disabling state, then $\psi A_{\lambda-1}$ is a final state.

On the other hand, Definition 2 is too restrictive. It requires a deterministic encoding of prey states. It makes sense to replace ψ with a mapping Φ from some collection POSE of predator states to the collection of prey states. Intuitively, $A = \Phi B$ means that B is one of the encodings of A.

Definition 3 Let POSE be a collection of states of $\mathcal{B}$, and let Φ be a mapping from POSE to States($\mathcal{A}$). $\mathcal{B}$ *simulates* $\mathcal{A}$ with respect to POSE and Φ if, for every run $\langle A_i : i < \lambda \rangle$ of $\mathcal{A}$, the following condition is satisfied.

S3 The pose states of $\mathcal{B}$ in the run of $\mathcal{B}$ on A_0 form a sequence $\langle B_i : i < \lambda \rangle$ (for the same λ), with $\Phi B_i = A_i$. Furthermore, if λ is finite and $A_{\lambda-1}$ is final, then $B_{\lambda-1}$ is final and yields the same output.

□

We say that $\mathcal{B}$ simulates $\mathcal{A}$ without preprocessing (respectively *with disabling state recognition*) if, in addition to S3, condition S1 (respectively S2) is satisfied.

If $\mathcal{A}'$ is obtained from $\mathcal{A}$ by restricting the collection of initial states to a subset $\mathcal{I}$ and if $\mathcal{B}$ simulates $\mathcal{A}'$, then we say that $\mathcal{B}$ *simulates* $\mathcal{A}$ *on* $\mathcal{I}$. This terminology may be applied to any simulation mode: with or without preprocessing, with or without disabling state recognition.

2.4 Honest Time and Lock-Step Simulations

It is customary to define the computation time of an algorithm $\mathcal{A}$ as the number of steps. But the program of $\mathcal{A}$ may be written on a higher abstraction level which hides details, so that one step of $\mathcal{A}$ involves a number of microsteps. In this connection, we assume that each algorithm $\mathcal{A}$ comes with a function $\mathrm{HT}_{\mathcal{A}}$ whose domain consists of the enabling states of $\mathcal{A}$. Think about $\mathrm{HT}_{\mathcal{A}}(A)$ as the number of microsteps, the *honest time*, necessary to transform A to its successor.

Definition 4 If $A \Rightarrow^*_{\mathcal{A}} B$ and $A_0, \ldots, A_k$ is the shortest partial run with $A_0 = A$ and $A_k = B$, then

$$\mathrm{HT}_{\mathcal{A}}(A, B) = \sum_{i=0}^{k-1} \mathrm{HT}_{\mathcal{A}}(A_i).$$

□

Notice that, since $\mathcal{A}$ is deterministic, any partial run of $\mathcal{A}$ from A to B begins with the shortest partial run from A to B.

Definition 5 Suppose that $\mathcal{B}$ simulates $\mathcal{A}$ (with or without preprocessing, with or without disabling state recognition) with respect to a collection POSE of the predator's states and a mapping Φ from POSE to prey states. The simulation is *lock-step* if there exists a positive integer c such that, for every run $\rho = \langle A_i : i < \lambda \rangle$ of $\mathcal{A}$, the following condition is satisfied. Recall that the run ρ of the prey gives rise to a sequence $\langle B_i : i < \lambda \rangle$ of predator pose states.

S4 $\mathrm{HT}_{\mathcal{B}}(B_i, B_{i+1}) \leq c \cdot \mathrm{HT}_{\mathcal{A}}(A_i, A_{i+1})$ for all i with $i + 1 < \lambda - 1$.

The smallest appropriate c is the *lag factor* of the simulation.
□

3 Abstract State Machines

For the reader's convenience, we recall here Gurevich abstract state machines (ASMs) also known as evolving algebras. More details can be found in [Gurevich 95]. ASM experts can skip most of this section; read only about the normal form at the end of the section.

3.1 States and Updates

Vocabularies and Terms

As usual in logic, a *vocabulary* is a finite collection of relation and function names of fixed arities. However relation names are treated as special function names. Every name in a vocabulary is a function name, but some function names are tagged to be relation names. Similarly, some function names may be tagged to be *static*. It is assumed that every vocabulary contains the following *logic function names*: nullary names *true*, *false* and *undef*, a unary relation name Bool, the names of the standard Boolean operations and the equality sign. All logic names are static.

Terms are defined as in first-order logic. *Boolean-valued terms* are Boolean combinations of terms $P(\bar{x})$ where P is a relation name. A *ground term* is a term with no free variables.

States

A *state* A of vocabulary Υ consists of (i) a set X, called the *superuniverse* of A, and (ii) the interpretations of the function names in Υ on X, called the *basic functions* of A. An r-ary function name is interpreted as a function from X^r to X. Nullary function names *true*, *false*, *undef* are interpreted by different elements. We do not distinguish between these names and their interpretations.

The boolean values *true* and *false* are the only possible values of any relation name. In particular, the equality sign is interpeted as the identity relation. An equality $t_1 = t_2$ can be viewed as a Boolean-valued term or a statement that the terms t_1, t_2 are equal. Usually, the context resolves the ambiguity. A relation F is identified with the set $\{\bar{x} : R(\bar{x}) = \mathit{true}\}$. In particular Bool $= \{\mathit{true}, \mathit{false}\}$.

The nullary function *undef* allows us to deal with partial functions. We say that a basic function f is undefined at a tuple $\bar{a}$ if $f(\bar{a}) = \mathit{undef}$. The Boolean connectives are undefined if at least one of the arguments is not Boolean; over Bool they are interpreted as expected.

Finally, the superuniverse includes an infinite set called the *reserve* of A which is intuitively a "naked" outside set. The reserve can be defined as the collection of elements of A satisfying the following conditions:

- No function produces a reserve value.
- Every basic relation, with the exception of equality, evaluates to *false* if at least one of the arguments is in the reserve.
- Every other function evaluates to *undef* if at least one of the arguments is in the reserve.

(This simpler semantical definition of the reserve differs inessentially from the more syntactical definition in [Gurevich 95].) In this paper, non-reserve elements different from *true*, *false*, *undef* will be called *regular*.

It is often convenient to think about a state A as a map from locations to contents. Locations have the form $(f, \bar{a})$ where f is a function name of some arity r and $\bar{a}$ is an r-tuple of elements of the superuniverse. The content of location $(f, \bar{a})$ is the element $f(\bar{a})$.

Variable Assignments

The only variables we will have are individual variables ranging over the reserve Y of a given state A. Let V be a collection of variable names. A *variable assignment* with domain V over the state A is an injective map $\xi : V \to Y$. The pair (A, ξ) may be called an *expanded state*.

In a sense, there is only one variable assignment with the given domain over the given state: for every variable assignments ξ_1, ξ_2 with the same domain V, there is an automorphism of A that takes every $\xi_1(v)$ to $\xi_2(v)$.

Updates

An *update* of a state A is a pair $\alpha = (\ell, a)$ where ℓ is a location over A and a an element of A. To *fire* (ℓ, a), put a into ℓ (that is, set the context of ℓ to a) and leave the other locations intact.

A set U of updates of a state A is *contradictory* if it contains a pair of updates (ℓ, a) and (ℓ, b) where $a \neq b$; otherwise U is *consistent*. To fire a contradictory update set, do nothing; the new state is identical to the old one. To execute a consistent update set $(\ell_1, a_1), \ldots, (\ell_k, a_k)$, put elements $a_1, \ldots, a_k$ into locations $\ell_1, \ldots, \ell_k$ respectively and leave the other locations intact.

3.2 Rules

Syntactically, update sets are specified by means of *(transition) rules* constructed inductively from atomic rules by means of several rule constructors. In order to apply a rule R to an expanded state (A, ξ), we require that the vocabulary of A should contain every function name that occurs in R and the domain of ξ should contain every variable that is free in R. [Free and bound variables will be defined after the definition of import rules — the only rules that bind variables.] In the rest of this section, we suppose that the vocabulary of a given expanded state is sufficiently rich in this sense.

The evaluation of a rule R at a state A under a variable assignment ξ produces an update set $U = \mathrm{US}(R, A, \xi)$. To fire R, fire U; the result is the *successor* of A with respect to R. R is *consistent* (respectively *contradictory*) at (A, ξ) if U is so.

Atomic Rules

An *atomic* (or *update*) rule R is an expression $f(t_1, \ldots, t_r) := t_0$ where each t_i is a term and f is a non-static r-ary function name; if f is a relation name then t_0 is a Boolean-valued term. Given an expanded state (A, ξ), evaluate all terms t_i. Let a_i be the resulting value of t_i, $\bar{a} = (a_1, \ldots, a_r)$ and ℓ the location $(f, \bar{a})$. Then $\mathrm{US}(R, A, \xi) = \{(\ell, a_0)\}$.

Block Rules

A *block rule* R has the form

block
    R_1
    $\vdots$
    R_k
endblock

where each R_i is a rule. For readability, the reserved words "block" and "endblock" are often omitted. Given an expanded state (A, ξ), evaluate all rules R_i. $\mathrm{US}(A, \xi) = \bigcup_i \mathrm{US}(R_i, A, \xi)$.

Conditional Rules

A *conditional rule* R has the form

if g_0 then R_0
elseif g_1 then R_1
$\vdots$
elseif g_k then R_k
endif

where $k \geq 0$, each g_i is a Boolean-valued term and each R_i is a rule. A line "(else)if g_i then R_i" is a *clause* with the *guard* g_i and the *body* R_i. If the last guard g_k is the nullary function name *true* then the last clause can be abbreviated to "else R_k".

Given an expanded state (A, ξ), evaluate the guards one after another. If all the guards evaluate to *false*, then $\mathrm{US}(R, A, \xi) = \emptyset$. And if g_i is the first guard that evaluates to *true*, then $\mathrm{US}(R, A, \xi) = \mathrm{US}(R_i, A, \xi)$.

Import Rules

An *import rule* R has the form

import v
  R_0
endimport

where v is a variable and R_0 is a rule, the *body* of R. Given a state A and a variable assignment ξ with domain V, let $V' = V \cup \{v\}$ and ξ' be the variable assignment with domain V' obtained from ξ by picking a fresh (that is outside of the range of ξ) reserve element a and setting or resetting $\xi'(v) = a$. Evaluate R_0 at A, ξ'. $\mathrm{US}(R, A, \xi) = \mathrm{US}(R_0, A, \xi')$.

Free and bound variables and occurrences of variables are defined in the usual way with "import v" binding v. Successive imports can be abbreviated to a rule of the form:

import $v_1, \ldots, v_m$
  R
endimport

Programs

A *program* is a rule without free variables. It can therefore be fired at a state, with the empty assignment, and this firing produces a state. Thus it makes sense to fire a program repeatedly, producing a sequence of states. Usually, we use the word "program" only when such an interation is envisaged.

A *state* of a program Π is any state A of the vocabulary of Π.

3.3 A normal form

Call two programs of the same vocabulary Υ equivalent if they generate the same update set at every Υ-state. Call a program *consistent* if it is consistent at every state. Every program is equivalent to a consistent program. Moreover, all imports can be done at the beginning, so that the program has a prenex form with respect to imports.

Lemma 1 (Normal Form) *Every Υ-program is equivalent to a Υ-program of the form*

```
import v_1,...,v_m
   R
endimport
```

where R is consistent in every Υ-state. Moreover, there is a polynomial time algorithm that transforms any given program to a normal form program.

The proof, found in [Dexter, Doyle and Gurevich 97], consists essentially of three steps. First, rename variables so that no two imports (that is, occurrences of the import command) use the same variable. Second, move all imports to the beginning of the program and all endimports to the end. Finally, add at the beginning of the program a guard that checks the consistency of all the intended updates. Notice, that the normal form of a given program may import extra elements but those elements will be inaccessible and will make no difference.

4 More on Abstract State Machines

For simplicity of exposition, we restrict attention to vocabularies without any non-logic static names. To improve readability, we sometimes write unary functions after their arguments; thus $t.f$ means $f(t)$. In this section, we describe our conventions related to input and output, and then discuss the honest time for abstract state machines.

4.1 The Input/Output Related Conventions

Vocabularies

Every vocabulary contains nullary function names 0, Mode, Output and a unary function name Succ. The name Output is relational, and the other three names are not relational.

Initial States

A state A is *initial* if it isomorphic to a state A_0 satisfying the following conditions.

- The regular elements form a segment $[0, n)$ of natural numbers with $n \geq 2$, the symbol 0 has the obvious interpretation and Succ is the successor function. Furthermore, $\mathrm{Succ}(n-1) = \mathrm{Succ}(\mathit{undef}) = \mathit{undef}$.
- $\mathrm{Mode} = 1$ and $\mathrm{Output} = \mathit{false}$.

A_0 is the *canonic* version of A. The number n is the *size* of A. For clarity, we use "Initial" as an alias for Succ(0), so that $Mode = Initial$ at A.

Final States

A state A is *final* if $\mathrm{Mode} = 0$ at A. The output of a final state A is the value of Output at A. For clarity, we use "Final" as an alias for 0, so that $Mode = Final$ at A.

Remark According to Section 2, final states are disabling. Programs written by us produce no updates at final states. One may want to impose such a requirement on all programs, for example by requiring that every program has the form "if Mode $\neq$ Final then…". We do not impose any syntactical constraint of this sort. Instead we use semantical means, namely the definition of run: the part of a run following the first final state is simply ignored and the final state is considered to be the last state of the run. □

Machines vs. Programs

By default, an abstract state machine $\mathcal{A}$ is a program P (as defined in the previous section) whose states (respectively initial states, final states) are those of the vocabulary of P. However, we reserve the right to restrict the collection of initial states.

A state B of $\mathcal{A}$ is *enabling* if it is non-final and the update set U of P at B (with the empty assignment) is consistent; otherwise B is *disabling*. Thus B is a *hang state* if U is contradictory and B is not final.

4.2 Parse Trees

Trees

Here, a *tree* is a finite structure with one partial unary function Parent satisfying the following condition. There exists a element x (the *root*) such that Parent is undefined at x and every other element y can be transformed to x by repeated applications of Parent. The elements of a tree are called *nodes*.

Oriented Trees

An *oriented tree* is a tree where the children of any node are linearly ordered by means of partial unary functions First Child and Next Sibling. We use abbreviations FC and NS for First Child and Next Sibling respectively.

Labeled Trees

A *labeled tree* is an oriented tree with label function or functions. The domain of a label function L is a set of nodes, so that, in general, L is partial.

A labeled tree T is a *composition* of labeled trees $T_1, \ldots, T_k$ with the same label-function names as T, if the root of T has exactly k childen and the corresponding k labeled subtrees are isomorphic to $T_1, \ldots, T_k$ in this order. Notice that nothing is said about the labels of the root of T.

Abstract Parse Trees

A parse tree is a labeled tree with a total label function Type and a partial label function Nomen.

We define, by induction, abstract parse trees APT(e) for various syntactical entities e.

- If e is a variable, then APT(e) is a one-node tree. The type of (that is the value of Type at) the root is "Var". The nomen of (that is the value of Nomen at) the root is "e" itself.

- If e is a term $f(t_1,\dots,t_r)$, then APT(e) is the composition of parse trees for $t_1,\dots,t_r$. The type of the root is "Fun" and the nomen is "f".
- If e is an atomic rule $f(t_1,\dots,t_r) := t_{r+1}$, then APT($e$) is the composition of the parse trees for $t_1,\dots,t_r,t_{r+1}$. The type of the root is "Update" and the nomen is "f".
- If e is a block rule with constituent rules $R_1,\dots,R_k$, then APT(e) is the composition of the parse trees for $R_1,\dots,R_k$. The type of the root is "Block".
- If e is a clause with guard g and body R, then APT(e) is the composition of the parse trees for g and R. The node type of the root is "Clause".
- If e is a conditional rule with clauses $R_0,\dots,R_k$, then APT(e) is the composition of the parse trees for $R_0,\dots,R_k$. The type of the root is "Cond".
- If e is an import rule with variable v and body R and T is the parse tree of R, then APT(e) is obtained as follows. First, construct the composition of T (all by itself) (that is add a parent to the root of T). Let x be the new root. Second, set the type of x to "Import" and the nomen of x to x itself. Third, for every node y of T whose nomen is v, redefine the nomen of y to x.

Concerning the last item, notice that, for every node y of T with nomen v, the occurrence of v corresponding to y is free in R.

Concrete Parse Trees

Fix a vocabulary Υ and linearly order it arbitrarily. For every function name f in Υ (including the logic names), denote by f-code the ordinal number of f in the linear order. Let P be a program in the vocabulary Υ.

Definition 6 The *concrete parse tree* $\mathrm{CPT}_\Upsilon(P)$ of P is the presentation of the abstract parse tree APT(P) of P obtained as follows.

- Order the nodes of APT(P) in the depth-first manner and then replace every node with its ordinal number (starting with zero).
- Modify the type function, by replacing the labels Block, Clause, Cond, Fun, Import, Update, Var with numbers $0,1,2,3,4,5,6$ respectively.
- Modify the nomen function by replacing every nomen f with its f-code.

□

4.3 Input Structures Encoding Programs

Fix a vocabulary Υ containing (in addition to 0, Succ, Mode and Output) the unary function names Parent, FC, NS, Type and Nomen and let P be a program of vocabulary Υ. Define the *size* of P to be the number of nodes in the concrete parse tree $\mathrm{CPT}_\Upsilon(P)$ of P. Without loss of generality, we restrict attention to programs whose size is at least 6 and at least as large as the total number of function names in Υ (including the logic names).

Definition 7 An initial state A of vocabulary Υ *encodes* an Υ-program P if the canonic version A_0 of A satisfies the following conditions.

- The regular elements with functions Parent, FC, NS, Type, Nomen (restricted to regular elements) form $\mathrm{CPT}_\Upsilon(P)$.

– Any function name $f \in \Upsilon$ different from 0, Succ, Mode, Output, Parent, FC, NS, Type and Nomen, is interpreted in the default way, that is, every $f(\bar{x}) = \mathit{false}$ is f is relational and every $f(\bar{x}) = \mathit{undef}$ otherwise.

□

Remark Some readers may be disappointed to find out that our inputs are not strings. We could redefine inputs to be strings but do not feel compelled to do that. The popular string representation of inputs is an outcome of a restricted computation model, namely the Turing Machine Model. More flexible machine models do not require their inputs to be strings. □

4.4 Honest Time

How to execute one step of a given program P at a given state A? (Imagine you write an interpreter or compiler for ASM programs.) Here is one way. First you evaluate all the terms (the evaluation phase). Second, you check the updates for consistency (the consistency check phase). Third, you fire the updates if they are consistent (the firing phase).

1. The evaluation phase. Traverse the parse tree depth-first evaluating all the terms but skipping the parts of the parse tree made irrelevant by their guards, namely (i) the bodies of clauses whose guards are false at A and (ii) the clauses having older siblings with guards true at A. During the evaluation traversal, construct a linked list of relevant update nodes.
2. The consistency check phase. Consistency check can be done during one traversal of the update linked list as follows. Let $(\ell_1, a_1), \ldots, (\ell_m, a_m)$ be the updates of the list where each ℓ_i is a location and each a_i is an element. Go through the list doing the following. On stage 1, put a fake version a'_1 of a_1 into ℓ_1. On stage $i+1$, compare the current content a of ℓ_i with a_i. If a is a genuine element, then put a fake version a'_i of a_i to ℓ_i and proceed to the next stage. If a is the fake version of a_i, proceed to the next stage; you may also remove (ℓ_i, a_i) from the linked list. If a is a fake version of some element different from a_i, then halt because you have discovered inconsistency.
3. The firing phase. Traverse the update linked list and fire the updates one after another.

If it is known that the given program P is consistent (for example, if you restrict attention to programs in the normal form described at the end of Section 3) or if you don't care what happens if P is inconsistent, you may want to skip the consistency check. Our diagonalization program will use this simplified execution strategy.

We define inductively the *honest evaluation time* $\mathrm{HET}_A(e)$ for evaluating various syntactical entities e at a state A of sufficiently rich vocabulary.

– If e is a variable then $\mathrm{HET}_A(e) = 1$.
– If e is a term $f(t_1, \ldots, t_r)$, then $\mathrm{HET}_A(e) = 1 + \sum_i \mathrm{HET}_A(t_i)$.
– If e is an atomic instruction $f(t_1, \ldots, t_r) := t_0$, then $\mathrm{HET}_A(e) = 1 + \sum_{i=0}^{r} \mathrm{HET}_A(t_i)$.
– If e is a block rule with constituents $R_1, \ldots, R_k$, then $\mathrm{HET}_A(e) = 1 + \sum_{i=1}^{k} \mathrm{HET}_A(R_i)$.

- Suppose that e is a clause "(else)if g then R". If g holds at A, then $\mathrm{HET}_A(e) = 1 + \mathrm{HET}_A(g) + \mathrm{HET}_A(R)$; otherwise $\mathrm{HET}_A(e) = 1 + \mathrm{HET}_A(g)$.
- Suppose that e is a conditional rule with clauses $C_1, \dots, C_k$. If none of the k guards $g_1, \dots, g_k$ holds in A, then $\mathrm{HET}_A(e) = 1 + \sum_{i=1}^{k} \mathrm{HET}_A(C_i)$. If j is the smallest index such that g_j holds in A then $\mathrm{HET}_A(e) = 1 + \sum_{i=1}^{j} \mathrm{HET}_A(C_i)$.
- If e is an import rule with body R, then $\mathrm{HET}_A(e) = 1 + \mathrm{HET}_A(R)$.

This completes the definition of honest evaluation time.

Lemma 2 *$HET_A(e)$ is the number of nodes in the parse tree of e visited when one traverses the parse tree depth-first skipping (i) the bodies of clauses whose guards are false at A and (ii) the clauses having older siblings with guards true at A.*

Proof An easy induction on e. □

It is convenient to think about $\mathrm{HET}_A(P)$ of a program P as the number of microsteps needed to evaluate P at state A. This includes evaluation of all terms in the program except those whose values are not used because of their guards. We also included in $\mathrm{HET}_A(P)$ the microsteps needed to move around in the parse tree of the program. It is obvious that

H1 $\mathrm{HET}_A(P) \le \mathrm{HT}_A(P)$.

There is also a simple inequality in the opposite direction. In fact, $\mathrm{HT}_A(P) \le 3 \cdot \mathrm{HET}_A(P)$ if the consistency check is done efficiently as explained above. But we will not use this fact in the present paper. Instead, we will use a simpler fact H2 below.

Definition 8 Let K be a collection of states of a program P. The *one-step honest time complexity of P on K* is the maximum of the numbers $\mathrm{HT}_A(P)$ where A ranges over enabling states in K. If K is the collection of all states of P, then the one-step honest time complexity of P on K is the *total one-step honest time complexity of P*. □

Now we are ready for formulate H2.

H2 For every program P, the total one-step honest time complexity of P is finite.

Definition 9 A class K of input structures in the vocabulary Υ is *decidable* within honest time $T(n)$ if there exists an ASM $\mathcal{A}$ of vocabulary Υ such that, for every input structure I of vocabulary Υ in K (respectively outside K), $\mathcal{A}$ accepts (respectively rejects) I within honest time $T(n)$ where n is the size of I. $\mathrm{HT}_\Upsilon(T(n))$ is the collection of input classes K decidable within honest time $T(n)$. □

5 Diagonalization Theorem

Recall that the simulated and simulating algorithms are called the prey and predator respectively. Given a representation R of a prey Turing machine, the classical diagonalizing Turing machine (the predator) starts by copying R (which constitutes the preprocessing stage) and then simulates the steps of the prey using one copy of R as a program and the other as data. (The Turing predator also changes the result of the computation, but we ignore that aspect for the time being). In this section, we generalize this to abstract state machines of a fixed ordered vocabulary Υ. The greater flexibility of the ASM model allows us to achieve a lock-step simulation. More exactly, it is a lock-step simulation with preprocessing and without disabling state recognition, which is sufficient for our purpose (of proving the ASM Linear Time Hierarchy Theorem) in this paper. It is possible to achieve a lock step simulation without preprocessing and with disabling state recognition; this requires a much more subtle construction and will be done elsewhere.

We start with a technicality. Notice that every program P is equivalent to the program

```
block
   P
endblock
```

which will be denoted $P+1$. Even though P and $P+1$ are equivalent and have the same runs, there is a slight difference between the initial states encoding P and $P+1$. We will exploit that difference. Define inductively $P+(i+1)=(P+i)+1$. For every vocabulary Υ and every Υ-program P, let $I(\Upsilon,P)$ be the canonic initial state encoding the Υ-program $P+m$ where m is the number of non-logic nullary symbols in Υ.

Call a vocabulary *diagonalization-solvent* if, in addition to the obligatory function names (the logic function names and the names 0, Succ, Mode and Output; see Section 4), it contains the nullary function names C, LastUpdate and unary function names Parent, FC, NS, Type, Nomen, Val, NextUpdate. Of course, it may contain arbitrarily many additional names.

Theorem 1 (Diagonalization Theorem) *For every diagonalization-solvent vocabulary Υ, there exist an Υ-program Diag, a collection POSE of states of Diag and a mapping Φ from POSE to Υ-states such that Diag lock-step simulates (with preprocessing and without disabling state recognition) any Υ-program P on the initial state $I=I(\Upsilon,P)$, except that Diag changes the output of P (if P converges). Moreover, there exist positive constants c_0,c_1,c_2 (depending on Υ but not on P) such that, if the run ρ of P on I converges, n is the size of I and p is the honest time of ρ, then the honest time of the run of Diag on I is bounded by $c_0+c_1n+c_2p$.*

Here c_0 is the initialization honest time, $c_1 n$ is the preprocessing honest time, and $c_2 p$ is the simulation honest time.

Proof Fix a diagonalization-solvent vocabulary Υ and order it in such a way that

– all non-logic nullary names come first, and
– The ordinal number of 0 is zero, and the ordinal number of Mode is one.

If i is the ordinal number of a name f, then the term $0.\mathrm{Succ}^i$ will be denoted f-ordinal. Our definition of $I(\Upsilon, P)$ was designed to ensure that, for every non-logic nullary name f, the f-ordinal is a block node, that is f-ordinal.Type = Block. We will take advantage of this feature.

The desired Diag is the program exhibited in Section 6 where we also explain how it works provided that the initial state has the form $I(\Upsilon, P)$. To improve readability, we use Final, Initial, Clone, Copy, Pose, Descend, Ascend, Execute as aliases for the terms 0, 0.Succ, $0.\mathrm{Succ}^2$, etc. Since Diag does not alter the values of these terms, Final, Initial, Clone, Copy, Pose, Descend, Ascend, Execute denote numbers $0, 1, 2, 3, 4, 5, 6, 7$.

The desired POSE. A state B of Diag is a pose state if there exists an initial state $I = I(\Upsilon, P)$ such that the following conditions are satisfied. Recall that non-logic non-reserve elements of any state A are called regular elements of A. The interpretation of a basic function name f at a state A will be denoted f_A.

0 The superuniverse of I is included in that of B and includes the reserve of B. Logic constants *true*, *false*, *undef* have the same interpretations in the two states. Call elements of I *original elements* of B. Regular original elements will be called *noble*; the other elements of B will be called *plebeian*.
1 First, 0 has the same interpretation in B and I. Second, if f is any of the unary names Succ, Parent, FC, NS, Type, Nomen, then f_I is the restriction of f_B to the elements of I. (Intuitively this means that Diag does not alter 0 and does not alter Succ, Parent, FC, NS, Type, Nomen on original elements.)
2 Every f_B of positive arity takes plebeian values on plebeian arguments.
3 Every nullary f_B is an original element. The range of Val consists of plebeian elements.
4 In B, Mode = Pose.

The desired mapping Φ. Let B be a pose state of Diag. To construct $A = \Phi B$, transform B as follows.

– Remove all nobles.
– Restrict the basic functions of positive arities to plebeians.
– Reset every basic non-logic nullary function f to f-ordinal.Val.

Employing the simplified executing strategy (that is not checking update consistency), Diag simulates any given Υ-program P on $I(\Upsilon, P)$ with respect to POSE and Φ but flips the value of Output when (and if) P arrives to a final state. The simplified strategy is sufficient because we are interested only in converging runs. If $\langle A_i : i \le \lambda \rangle$ is a converging run of P (so that A_λ is final) then P is consistent at every state A_i with $i < \lambda$.

The preprocessing stage splits into two phases. First, Diag clones the regular elements of $I(\Upsilon, P)$. In terminology of the definition of POSE, Diag creates plebeian clones of its noble elements. Second, Diag copies the given parse tree to the clones.

In Section 6, we interleave the rules of Diag with explanations how it runs on input $I(\Upsilon, P)$. Let us explain here why we use the initial state $I(\Upsilon, P)$ rather than the canonic initial state encoding P. The problem is that Diag cannot use a non-logic nullary function f to hold the prey's value of f unless it does not use f for other purposes. Our solution is to record the prey's value of f by means of the term f-ordinal.Val. To this end, we should make sure that these values of Val are not used for other purposes. In fact, the values of Val on the nodes of types Import and Fun, are used for other purposes. If x is an import node then the prey's value of the corresponding variable is represented as x.Val, and if x is the node of some term t then the prey's value of t is represented by x.Val. By augmenting the parse tree of P with initial block nodes, we provided nodes x such that x.Val can be safely used to keep track of the prey's values of nullary functions. Of course, there are numerous other solutions of that problem.

Now let us address the issue of honest time. The desired c_0 is the one-step honest time complexity of Diag on the states with Mode = Initial. The number of preprocessing steps of the predator is $2n$. Let b_1 be the one-step honest time complexity of Diag on states with Mode = Clone or Mode = Copy. Then the preprocessing honest time is bounded by $2b_1n$, so that the desired $c_1 = 2b_1$.

Let b_2 be the one-step honest time complexity of Diag on states where Mode equals Descend, Ascend, Execute or Pose. To simulate one step of the prey, the predator traverses the parse tree of P (except for the parts made irrelevant by guards). It visits every leaf once and every non-leaf node twice. Taking into account Lemma 2 and the fact H1 in Subsection 4.4, it is easy to see that, if a step of the prey takes honest time t, then the predator simulates it in $< 2t$ steps and thus spends honest time $< 2b_2t$. The desired c_2 is $2b_2$. $\square$

Notice that the predator program does not alter 0 and does not alter Succ on the original elements. We formalize this observation for future use.

Corollary 1 *Let B be any state in the run of the program of Section 6 on a canonic initial state of size n. At B, 0 denotes the number zero, and $i.Succ = i+1$ for every $i \le n-1$, and $(n-1).Succ = \text{undef}.Succ = \text{undef}$*

6 Diagonalization Program

We present a program Diag (the predator) and explain how it runs on the initial state $I = I(\Upsilon, P)$ for a given Υ-program P (the prey). Diag has the form

```
block
   Initial Rule
   Preprocess Rule
   Pose Rule
   Evaluation Rule
   Execution Rule
endblock
```

6.1 The Initial Rule

```
if Mode = Initial then
   C := 0, Mode := Clone
endif
```

6.2 The Preprocess Rule

The Preprocess Rule has the form:

```
block
   Clone Rule
   Copy Rule
endblock
```

6.2.1 The Clone Rule

```
if Mode = Clone then
   if C ≠ undef then
      import new   Val(C) := new   endimport
      C := Succ(C)
   else C := 0, Mode := Copy
endif
```

For every regular element x of I, we create a clone x' and assign it to x.Val. We will say that x is a noble element and x' is its plebeian clone. The function Val allows us to access clones at this stage. Later it will be used for other purposes as well.

6.2.2 Copy Rule

The Copy Rule has the form

```
if Mode = Copy then
   if C ≠ undef then
      COPY, C := Succ(C)
   else C := 0, Mode := Pose
   endif endif
```

where COPY is the rule

```
block
   Succ(C.Val) := C.Succ.Val
   Parent(C.Val) := C.Parent.Val
   FC(C.Val) := C.FC.Val
   NS(C.Val) := C.NS.Val
   Type(C.Val) := C.Type.Val
   Nomen(C.Val) := C.Nomen.Val
endblock
```

The successor function and the whole parse tree is copied onto the clones. This takes care of all non-logic positive-arity functions f_I whose range is different from $\{undef\}$. There is no need to copy logic functions or non-logic functions f_I with range $\{undef\}$.

According to the general recipe explained in Section 1, the prey's value of a non-logic nullary function f is represented by f-ordinal.Val. We should ensure that the values f-ordinal.Val are properly initialized during the preprocessing stage. There are only three non-logic nullary functions of I with values different from *undef*, namely 0, Mode and Output. 0-ordinal.Val is automatically initialized to 0.Val because 0-ordinal = 0. Similarly Mode-ordinal.Val is automatically initialized to Initial.Val because Mode-ordinal = 0.Succ = Initial. The initialization of Output is unnecessary because Diag does not play with Output until the very end. It uses Output itself to hold the prey's value of it and thus does not need to use Output-ordinal.Val at all.

Remark. Similarly, Diag does not need to use f-ordinal.Val for any nullary name f that it does not play with. But it is simpler to use the terms f-ordinal.Val for such nullary names f because it makes the program more robust in case we want to augment it with new rules (as we will do in the next section).

6.3 Pose Rule

```
if Mode = Pose then
   if Mode-ordinal.Val = Final then
      Output := not(Output), Mode := Final
   else
      LastUpdate := 0, Mode := Descend
   endif
endif
```

When Mode = Pose, Diag examines the prey's value of Mode. In case the prey halts, Diag changes the output of the prey and halts. Otherwise it initializes LastUpdate and moves to the evaluation phase.

6.4 The Evaluation Rule

The Evaluation Rule has the form

```
if Mode = Descend then
   Descend Rule
   if Type(C) = Import then Import Rule endif
   if Type(C) = Update then Update Rule endif
endif
if Mode = Ascend then
   if Parent(C) ≠ undef then Ascend Rule endif
   if Type(C) = Var then Var Evaluation Rule endif
   if Type(C) = Fun then Fun Evaluation Rule endif
endif
```

Intuitively C is the current position in the prey parse tree. C traverses the prey parse tree depth-first. It moves down when Mode = Descend and it moves right or up when Mode = Ascend. Imports are done while Mode = Descend. Also, the linked list of Updates is set up while Mode = Descend. The evaluation of terms is done while Mode = Ascend.

6.4.1 The Descend Rule

```
if FC(C) = undef
   then Mode := Ascend
   else C:=FC(C)
endif
```

6.4.2 The Import Rule

```
import new
   Val(C) = new
endimport
```

The imported value of a variable is represented as x.Val where x is the corresponding import node of the noble parse tree.

6.4.3 The Update Rule

```
NextUpdate(LastUpdate) := C, LastUpdate := C
```

The nullary function LastUpdate and the unary function NextUpdate are used to create a linked list of the update nodes visited during the evaluation stage. The linked list includes 0 as the initial point even though 0 is not an update node. Notice that, if x is the last node in the linked list, then x.NextUpdate is undefined.

6.4.4 The Ascend Rule

In general, C will move to the right after visiting a node that has a next sibling. There are, however, two exceptions.

- C visits a false guard of some clause. Since the guard is false, there is no need to evaluate the body of the clause.
- C visits a clause with a true guard. There may be more clauses in the same conditional rule but there is no need to evaluate them.

Accordingly the ASCEND Rule is the rule

```
if (C visits a false guard or a true-guard clause) then
   C := Parent(C)
elseif NS(C) ≠ undef then
   C := NS(C), Mode := Descend
elseif C ≠ 0 then
   C := Parent(C)
else
   Mode := Execute
endif
```

where (C visits a false guard or a true-guard clause) is an appropriate abbreviation.

6.4.5 The Var Evaluation Rule

```
Val(C) := C.Nomen.Val
```

C.Nomen gives the appropriate Import node x such that x.Val holds the prey's value of the variable.

6.4.6 The Fun Evaluation Rule

The Fun Evaluation Rule is a block of rules

```
if Nomen(C) = f-ordinal then
   EVALUATE-f
endif
```

where f ranges over Υ. Recall that f-ordinal is the term $0.\mathrm{Succ}^i$ where i is the ordinal number of f in Υ. Let r be the arity of f, Child-1 $=$ FC(C) and Child-$(i+1)$ $=$ NS(Child-i). EVALUATE-F is the obvious rule.

```
Val(C) := f(Child-1.Val,...,Child-r.Val)
```

if $r > 0$ or if f is a nullary logic name or the name Output. Otherwise (if f is a non-logic nullary name different from Output), EVALUATE-f is the rule

```
Val(f) = f-ordinal.Val
```

6.5 Execution Rule

```
if Mode = Execute then
   if NextUpdate(C) ≠ undef then
      C := NextUpdate(C), NextUpdate(C):=undef
   else C := 0, Mode := Pose
   UPDATE
endif
```

Diag traverses the update linked list, executing updates and removing the portions of the liked list which are not needed anymore and thus making room for the next linked list (if any).

UPDATE is the block of rules

```
if Nomen(C) = f-ordinal then
   UPDATE-f
endif
```

where f ranges over non-logic functions in Υ. Let r be the arity of f, Child-$1 =$ FC(C) and Child-$(i+1) =$ NS(Child-i). UPDATE-f is the rule

f(Child-1.Val,...,Child-r.Val) := Child-($r+1$).Val

if $r > 0$ or if f is Output. Otherwise (that is if f is nullary and different from Ouput), UPDATE-f is the rule

Val(f-ordinal) = Child-1.Val

7 ASM Linear Time Hierarchy Theorem

Call a vocabulary Υ *hierarchy-solvent* if, in addition to the obligatory names (the logic names and the names 0, Succ, Mode and Output), it contains at least five non-relational nullary names and at least seven non-relational unary names. Of course, it may contain arbitrarily many additional names. Recall the definition of honest time classes HT($T(n)$) in Section 4.

Theorem 2 (ASM Linear Time Hierarchy Theorem)
For every hierarchy-solvent vocabulary Υ, there exists a constant c such that, for every positive integer d,

$$HT(c \cdot d \cdot n) - HT(d \cdot n) \neq \emptyset.$$

Proof Without loss of generality, we may assume that a given hierarchy-solvent vocabulary Υ is diagonalization solvent (see Section 5) and contains three additional non-relational nullary names Hour, HourLimit, Minute. Order Υ as in Section 5. We augment the program Diag of Section 6 with two additional rules: Clock Initialization Rule and Clock Advance Rule. The latter rule depends on a given d. More exactly, let Timed Diag be the program of the form

```
block
   Clock Initialization Rule
   Clock Advance Rule
   if NOCLASH then Diag endif
endblock
```

where Diag is the program of Section 6. The Clock Initialization Rule is the rule

```
if Mode = Initial then
   if 0.Succ^{d+2} = undef then Mode := Final
   else Hour := 0, Minute := 0, HourLimit := 0.Succ^{d+2}
   endif
endif
```

The Clock Advance Rule is the rule

```
if Mode ≠ Initial and Mode ≠ Final then
   if Hour = HourLimit then Mode := Final
   elseif Succ(Minute) = undef then
      Hour := Succ(Hour), Minute := 0
   else Minute := Succ(Minute)
endif
```

Finally, NOCLASH is the following guard

$\texttt{0.Succ}^{d+2} \neq$ ***undef*** `and (Mode = Initial or Hour` $\neq$ `HourLimit)`

which shuts Diag down when the clock sets Mode to Final; the guard allows us to avoid a possible clash between Diag and the clock (when they try to set Mode to different values) and the ensuing hang state.

The idea behind our clock is quite obvious. Starting from 0, Minute advances every step till it reaches $n-1$. During the next step, Minute is reset to 0, and Hour is advanced. Then again Minute advances until it reaches $n-1$ and then again it is reset to 0 and Hour is advanced. This goes on until the program halts or Hour reaches HourLimit, at which point the clock shuts the program down. Notice that the clock counts neither the initial step, which sets up the clock, nor the final step, which happens *after* Hour reaches HourLimit. Thus the clock shuts the program down after $(d+2)n+2$ steps (unless it does so on the very first step). The guard $0.\mathrm{Succ}^{d+2} \neq$ *undef* and Hour $\neq$ HourLimit disables Diag when the clock shuts the program down, so that Diag does not prevent the clock from setting Mode to Final.

Lemma 3 *Let I be an initial state of size n. The length of any run of Timed Diag on I is $\leq (d+2)n+2$.*

Proof Without loss of generality, I is a canonic initial state. The new rules do not alter 0 or Succ and thus Timed Diag satifies Corollary 1 of the Diagonalization Theorem.

If $n \leq d+2$, then the Clock Initialization Rule stops Timed Diag at once. Assume that $n > d+2$ and let $\langle B_0, B_1, B_2, \ldots\rangle$ be the run of Timed Diag on I, so that $B_0 = I$. If the length of the run (that is the number of steps) is $< (d+2)n+2$, we have finished. We suppose that the length is $\geq (d+2)n+2$ and prove that it equals $(d+2)n+2$.

By the Clock Initialization Rule, Hour = Minute = 0 at B_1. Now we use the Clock Advance Rule. At every B_i with $i \leq n$, Minute $= i-1$, so that Minute.Succ = *undef* at B_n and therefore Hour = 1, Minute = 0 at B_{n+1}. Similarly Hour reaches 2 at B_{2n+1}, reaches 3 at B_{3n+1} and reaches $d+2$ at state $B_{(d+2)n+1}$, so that Hour = HourLimit at $B_{(d+2)n+1}$, Mode = Final at $B_{(d+2)n+2}$ and the run length equals $(d+2)n+2$. $\square$

As in the case of Diag, there are constants c_0, c_1, c_2 satisfying the following condition: if P is an Υ-program, $I = I(\Upsilon, P)$, n is the size of I, the run ρ of P on I converges, and p is the honest time of ρ, then the honest time of the run of Timed Diag on I is bounded by $c_0 + c_1 n + c_2 p$. This time around, the constants are somewhat larger because of the two clock rules.

Let L be the collection of initial Υ-states I accepted by Timed Diag. Since Timed Diag always halts, it decides L. Let b be the one-step honest time complexity of Timed Diag. By Lemma 3, the honest time of any computation of Timed Diag is bounded by $b(dn+2n+2)$. Choose any c such that $cdn \geq b(dn+2n+2)$ for all positive integers d and all n. Then $L \in \mathrm{HT}(cdn)$. (Actually, it suffices to chose c such that $cdn \geq b(dn+2n+2)$ for all d and sufficiently large n.)

It remains to check that $L \notin \mathrm{HT}(dn)$. Toward a contradiction, assume that some Υ-program P decides L within honest time dn. Let $I = I(\Upsilon, P)$ be the initial state coding P and n the size of I. (If desired, P can be enlarged without altering the classes of input structures that it accepts or rejects; thus n can be made sufficiently large.) Since P witnesses $L \in \mathrm{HT}(dn)$, the run ρ of P on I converges. Since the honest time of ρ is $\leq dn$, it involves at most dn microsteps. It follows that the evaluation part of the run R of Timed Diag on I has at most dn steps. Hence the length of R is $\leq 1 + 2n + dn$, so that R halts without being shut down by the clock. But then R produces a different output, which contradicts the assumption that P accepts L. □

8 Comments

Special Form of Timed Diag

Only one variable is used in Timed Diag.

Minimizing the Solvency

We did not attempt to minimize the vocabulary necessary for proving the linear hierarchy theorem. The clarity of exposition was our main concern.

Notice that only nullary and unary non-logic functions are used explicitly in Timed Diag. Of course, if a given vocabulary contains functions of arities ≥ 2, then Timed Diag may have to deal with them (to evaluate them and maybe update), but it does not introduce any non-logic functions of arity ≥ 2.

Preprocessing and Disabling-State Recognition

Timed Diag simulates its prey with preprocessing and without disabling state recognition. That suffices for the Linear Time Hierarchy Theorem. It is possible to get rid of preprocessing and achieve disabling-state recognition, but this is not trivial. We will do that improvement elsewhere [Blass and Gurevich 97] where we will be able to take advantage of these features.

Static Functions

One may want to have initial states with universes and various static functions. For example, one may want that every initial state contains a copy of integers with the standard order, addition and multiplication. Such generalizations are easier than getting rid of preprocessing. In fact, the proof of hierarchy theorem may be simplified if one can take advantage of static functions.

The presence of static function requires some changes in the Diagonalization Theorem, but the only major change is this. After copying a given parse tree, use the copy, not the original, for the bookkeeping purposes. Imagine, for example, that the original parse tree "lives" on the given copy of integers and thus its nodes participate in too many relationships; you cannot copy all the information onto the clone of the parse three. The clone parse tree should be not accessible from outside, so that that the prey cannot modify it. The range of the function Val should be the clone parse tree. The notion of f-ordinal should be redefined so that f-ordinals live on the clone parse tree, and all values f-ordinal.Val should be initialized.

9 Random Access Machines

In this section, we establish the analogs of Theorems 1 and 2 for random access machines (RAMs) in place of abstract state machines. That is, we exhibit a diagonalizing RAM program, which, given (the code of) an arbitrary RAM program P as input, simulates in lock-step the operation of program P on the same input, stops after simulating at most $d|P|$ steps (where d is a specified constant and $|P|$ is the length of P), and, if the simulated computation has produced an output by this time, changes that output. As for abstract state machines, the existence of such a diagonalizing RAM implies a linear time hierarchy theorem.

There are numerous RAM models in the literature. We generally follow [Aho, Hopcroft and Ullman 74], but for simplicity we avoid tapes by using registers not only for computation but also for input and output. Also, we adopt the convention that the content of a register must be a non-negative integer.

We begin by describing the particular RAM model that we use. Next, we indicate how programs are to be coded in order to serve as input to computations. Then we indicate how the diagonalizing machine is to be constructed. Finally, we explicitly write out this machine's program in a sort of pseudo-code that we hope is readable (for people) yet sufficiently detailed to leave no doubt that the diagonalizing machine performs as claimed.

Our RAMs have a potentially infinite supply of *registers*, indexed by the non-negative integers, each capable of holding (as its *content*) an arbitrary non-negative integer. Register 0 plays a special role and is called the *accumulator*. The input to a RAM is a finite sequence of non-negative integers, initially stored in registers 1 through l for some l; the accumulator and all registers after l initially hold 0. The result of a computation is the content of the accumulator when (and if) a HALT operation is executed. (If a Boolean output is desired, use the parity of the number in the accumulator.)

A RAM *program* is a list of instructions, each having the form "operation operand" where the operand is a non-negative integer and the operation is one of the following, where we have also indicated how to execute each instruction when the operand is a. The phrase "Put t into register i" means to change the content of register i to be t while leaving the contents of all other registers unchanged; $c(i)$ means the content of register i.

1. LOAD= Put a into the accumulator.

2. LOAD Put $c(a)$ into the accumulator.

3. LOAD* Put $c(c(a))$ into the accumulator.

4. STORE Put $c(0)$ into register a.

5. STORE* Put $c(0)$ into register $c(a)$.

6. ADD= Add a to the accumulator,
i.e., put $c(0) + a$ into the accumulator.

7. ADD Add $c(a)$ to the accumulator.

8.	ADD*	Add $c(c(a))$ to the accumulator.
9.	SUB=	Subtract a from the accumulator and if the result is negative then replace it with 0.
10.	SUB	Subtract $c(a)$ from the accumulator and if the result is negative then replace it with 0.
11.	SUB*	Subtract $c(c(a))$ from the accumulator and if the result is negative then replace it with 0.
12.	MULT=	Multiply the accumulator by a.
13.	MULT	Multiply the accumulator by $c(a)$.
14.	MULT*	Multiply the accumulator by $c(c(a))$.
15.	DIV=	Divide the accumulator by a and round down to the next smaller integer.
16.	DIV	Divide the accumulator by $c(a)$ and round down to the next smaller integer.
17.	DIV*	Divide the accumulator by $c(c(a))$ and round down to the next smaller integer.
18.	JUMP	Continue with the instruction in position a in the program.
19.	JGTZ	If $c(0) > 0$ then continue with the instruction in position a; otherwise continue with the next instruction after the current one.
20.	JZERO	If $c(0) = 0$ then continue with the instruction in position a; otherwise continue with the next instruction after the current one.
21.	HALT	Halt.

For simplicity, we assume that the last instruction in a program is always HALT, with an (irrelevant) operand of 0. (Adding this at the end of a program will cause no essential change. At most, a program that previously hung may execute the final HALT instruction.)

A *state* of a RAM consists of the content of its registers, its program, and a program counter pointing to the next instruction in the program to be executed. It is an initial state if register 0 and all but finitely many of the other registers contain 0 and the program counter points to the first instruction in the program. It is a final state if the program counter points to a HALT instruction. It is a hang state if the next instruction would require division by zero or continuing with a non-existent instruction. In all enabling states, after the next instruction is executed, the program counter is moved forward one step in the program except as specified in the instructions JUMP, JGTZ, and JZERO.

For simplicity, we use unit cost measures for RAMs. That is, the length of an input is the number of the last register with non-zero content (or 1 if there is no such register), and the time of a computation is the number of intsruction executions in it. If we used a logarithmic cost measure instead, then our predator would not simulate the prey in lock-step and in fact the cost of the predator's computation would not be bounded by a linear function of the prey's computation cost. The reason is that, in order to copy the prey's program, the predator uses registers with indices comparable to the prey program's length, and the prey's computation may be much shorter than the length of its program. Notice, however, that this difficulty arises only when the prey's computation cost is sublinear and the predator's cost is linear. Thus, the linear time hierarchy theorem remains valid if we use logarithmic cost measures.

In order to use a program as an input to a RAM computation, we must represent it as a finite sequence of non-negative integers. We do this by replacing each operation by a code, namely its number in the list exhibited above. In addition, we put at the beginning of the code the length of the program, i.e., the number of instructions in it. Thus, a program of length l becomes a sequence of $2l + 1$ integers, where term number 1 is l, term number $2i$ is the code of the operation in the ith instruction, and term number $2i + 1$ is the operand of that instruction.

Theorem 3 (Diagonalization Theorem for RAMs) *There is a positive integer c such that for any positive integer d, there is a RAM program Π that operates in linear time and diagonalizes in the following sense. Given, as input, the code of any RAM program P, it simulates the first $d|P|$ steps of the computation of P (or the whole computation if it's shorter than $d|P|$) on that same input and then adds 1 to the output (if any). Furthermore, the simulation is done in lock-step with lag factor c and without preprocessing.*

It turns out that the elimination of preprocessing is considerably easier for RAMs than for ASMs, so we have included "without preprocessing" in this theorem. It would be easy to include "with recognition of disabling states" as well, by modifying the predator to check before each division whether the divisor is 0 and to check before each jump whether the target of the jump is beyond the end of the program. If either of these happens, the predator should simply halt.

Corollary 2 (Linear Time Hierarchy Theorem for RAMs) *There exists c such that for every d*

$$HT(c \cdot d \cdot n) - HT(d \cdot n) \neq \emptyset.$$

Proof of the Corollary With notation as in the preceding theorem, let L be the set of all x such that when Π is run on input x it produces an odd number as output. Thus L is decided in time $c \cdot d \cdot n$ by Π. If P were a RAM program deciding L in time $d \cdot n$, then it would to produce the same (parity of) ouptput as Π when the input is the code x of P itself. But Π and P produce, on input x, ouputs that differ by 1. $\square$

Proof of the Theorem We first describe approximately how the diagonalizing predator RAM is to work. (The description is only approximate because it includes some preprocessing that will be eliminated later and because it glosses over some details that will be filled in later.) Given as input the code of any prey RAM program it makes a second copy of this code and then shuttles back and forth between the two copies, using the first (which it never alters) to read what the prey would do and using the second, plus registers beyond it, to do exactly the same thing (only in different registers), except that if the prey halts then the diagonalizing machine alters the output by adding 1 to it and then halts. To make this approximate description more precise, the following points must be taken into account.

First, we must make sure that the predator operates in linear time on *all* inputs (with a specified proportionality factor in the linear function). More precisely, we have a constant d such that the predator should simulate $d|P|$ steps of P if its input codes a RAM program P. This will require at most $\frac{1}{2}kdn$ steps of the predator, where $n = 2|P|$ is the length of the input and k is the number of predator steps needed to simulate one prey step. This k is a universal constant that could be read off from the detailed description of our predator below, and $\frac{1}{2}k$ is the c of the theorem. Furthermore, if the input to the predator has length n and fails to code a RAM, then the predator should still take at most $\frac{1}{2}kdn$ steps before halting.

In particular, if it is simulating a prey program which, on input equal to its own code, runs for too many steps, then the predator must break off the simulation when time runs out. So we equip it with a clock that counts the steps of the prey as they are being simulated and causes the computation to halt when the time limit is reached.

Determining the appropriate time limit is easy when the input codes a RAM program, for then the number l of instructions in the program is the content of register 1, and the length of the input is $2l$ (because the last operand is zero). For other inputs, doubling the content of register 1 may seriously over- or underestimate the actual length of the input. Underestimates do not present a problem, but overestimates do, for they would result in the predator's clock allowing more time than it should. Fortunately, the problem is easy to circumvent. The predator should read the content l of register 1 and check whether register $2l$ contains 0. If it doesn't, then $2l$ does not overestimate the input length, so it's safe to set the clock accordingly. If register $2l$ does contain 0, then the input is not the code of any program (because in the code of a program register $2l$ would contain the code 21 for a HALT operation) so it is safe for the predator to immediately halt.

Second, we must set aside a small number of registers for the predator to use for "scratch work" like keeping track of which instruction in P it is simulating.

Finally, the trickiest point is to avoid pre-processing. This means that we cannot simply make a second copy of the input before starting the simulation. The copying would prevent the simulation from being in lock-step. In fact, since the computations of some programs P on their own codes are considerably shorter than the programs themselves, the copying would make the simulation of such a program take more time than a constant multiple of the prey's time. Our solution to this problem is that instead of copying the whole input at the beginning, we copy it as needed. This entails a bit more bookkeeping, since we must not confuse the two situations (a) a register contains 0 that has been copied and (b) a register contains 0 because nothing has yet been copied to it. So when we copy we increase all numbers by 1 (so any 0 is necessarily of type (b)), and of course we must compensate for this addition when doing arithmetic. There is nothing particularly difficult about this, but it requires a certain amount of patience. The reader lacking that patience is invited to skip the following, more detailed presentation of the diagonalizing RAM's program.

To describe the diagonalizing machine, we adopt the following conventions. As indicated above, the prey program initially occupies registers 2 through $2l+1$ (though the content of register $2l+1$ is 0) while register 1 contains the number of instructions in the prey program; the content of these registers will not be altered during the computation. The predator will use register $2l+2$ as a program counter, pointing to the register containing the operation or operand needed at the moment. Register $2l+3$ serves as a clock, initialized to $d \cdot l$ and decremented by 1 every time a step of the prey is simulated. A small number of registers, from $2l+4$ to $2l+s-1$, and the accumulator, register 0, are used for the predator's scratch work. The registers from $2l+s$ on are used to simulate the prey's computation, with predator register $2l+s+i$ corresponding to prey register i but with the contents raised by 1 as described above. (When the prey has n in register i, the predator has $n+1$ in register $2l+s+i$, unless the necessary copying has not yet been done, in which case the predator has 0 there.)

In the preceding conventions, as well as in some comments below, we tacitly assume that the input to the predator RAM is the code of a prey RAM program. For any other input, it will not matter what the predator does, as long as it halts within time $\frac{1}{2}kdn$, where n is the length of the input.

We shall use the following abbreviations in presenting the program of the diagonalizing RAM. Let $c(i)$ denote the content of register i. Let $l = c(1)$ (the number of instructions in the prey program), $pt = c(2l+2)$ (the pointer to the current operation or operand), and $tm = c(2l+3)$ (the remaining time, i.e., the maximum number of prey steps that can still be simulated). Also, by the *pseudo-content* $pc(i)$, for any register number $i \geq 2l+s$, we mean $c(i)-1$ if $c(i) > 0$ and we mean $c(i-2l-s)$ if $c(i) = 0$. The idea behind pseudo-content is this. A non-zero number n in a register $i \geq 2l+s$ of the predator represents the presence of the number $n-1$ in the corresponding register $i-2l-s$ of the prey at the corresponding stage of the computation. A zero in such a register i means that this part of the input has not yet been duplicated, and has therefore not yet been accessed by the prey (for the predator will duplicate the contents of a register the first time the prey accesses the register). So the content, at that stage, of the corresponding prey register $i-2l-s$ can be ascertained by looking into the predator's register $i-2l-s$, and that is exactly what pseudo-content does. Thus, the content of prey register j at any stage of the computation is the pseudo-content of predator register $2l+s+j$ at the corresponding stage.

With these explanations, we hope that the description below of the diagonalizing RAM will be understandable. That it uses only a constant number of predator steps to simulate one prey step should be clear by inspecting the modules in the definition once one realizes the following two points. First, the availability of truncated subtraction and testing for 0 allows one to test for equality and inequality in a constant number of steps. Second, the initialization module, which may look suspiciously like preprocessing, takes a constant number of steps, independent of the input size, so it can safely be counted as part of the simulation of the first step of the prey's computation. It increases k a little, but does no real harm.

As already indicated, we present the diagonalizing RAM program as a sequence of modules. There is one module for each of the 21 RAM operations, and in addition there are four special modules: Initialize, which initializes pt, tm, and $c(2l+s)$ (the program location, the clock, and the predator's copy of the prey's accumulator); ReadOp, which reads the current operation and passes control to the appropriate module; Step, which advances pt to the next operation (unless this was overriden by a jump instruction); and Cycle, which, after checking that pt has not gone past the end of the program and that time has not run out, decrements the clock and starts the simulation of the prey's next step. Here are the explicit instructions in all the modules. [We use square brackets for comments; these are for the reader's benefit and are not part of the program.]

Initialize

If register $2l$ contains 0 then pass control to module HALT.

Write 2 into register $2l+2$. [This initializes pt.]

Write $d \cdot l$ into register register $2l+3$. [This initializes tm using the specified factor d.]

Write 1 into register $2l+s$. [This initializes the predator's copy of the prey's accumulator. Remember that 0 in a prey register is represented by 1 in the predator's copy.]

Pass control to module ReadOp.

ReadOp

If $c(pt) = 1$ [meaning that the operation to which the program counter points is LOAD=], then increment register $2l+2$ by 1 [so that pt points to the operand for the LOAD= instruction] and pass control to module LOAD=.

Similarly for the other 20 operations.

LOAD=

Add 1 to $c(pt)$ [the operand of the LOAD= instruction] and write the result into register $2l+s$.

Then pass control to module Step.

LOAD

Write $1 + pc(2l + s + c(pt))$ into registers $2l+s$ and $2l+s+c(pt)$. [$c(pt)$ is the operand of LOAD, the address whose content the prey would write into the accumulator. The corresponding address for the predator is $2l+s+c(pt)$, so the predator writes the pseudo-content of this register plus 1 into its copy of the prey's accumulator, namely register $2l+s$. It also writes the same thing into register $2l+s+c(pt)$, which doesn't change the content of this register unless this content is 0. In that case, when the input has not yet been copied into this register, the copying is done as part of this module.]

Then pass control to module Step.

LOAD$*$

Abbreviate $c(pt)$ as z, abbreviate $pc(2l + s + z)$ as y, and abbreviate $pc(2l + s + y)$ as w.

If register $2l + s + z$ contains 0, then write $y + 1$ there.

In any case, write $w + 1$ into registers $2l + s$ and $2l + s + y$.

Pass control to Step.

STORE

Write the content of register $2l + s$ into register $2l + s + c(pt)$. [Recall that register $2l + s$, the predator's copy of the prey's accumulator, was set to 1 by module Initialize. So we needn't use pseudo-contents here.]

Then pass control to module Step.

STORE$*$

Abbreviate $pc(2l + s + c(pt))$ as y.

If register $2l + s + c(pt)$ contains 0 then write $y + 1$ there.

In any case, write the content of register $2l + s$ to register $2l + s + y$.

Pass control to module Step.

ADD=

Add $c(pt)$ to the content of register $2l + s$ and write the result into register $2l + s$.

Pass control to module Step.

ADD

Abbreviate $pc(2l + s + c(pt))$ as y.

If the content of register $2l + s + c(pt)$ is 0 then write $y + 1$ there.

In any case, add y to the content of register $2l + s$ and write the result into register $2l + s$.

Pass control to module Step.

ADD$*$

Abbreviate $pc(2l+s+c(pt))$ as y and abbreviate $pc(2l+s+y)$ as w. If register $2l + s + c(pt)$ contains 0 then write $y + 1$ there.

If register $2l + s + y$ contains 0 then write $w + 1$ there.

In any case, add w to the content of register $2l + s$ and write the result into register $2l + s$.

Pass control to module Step.

SUB=, SUB, ..., DIV*

The remaining arithmetical operations are handled like addition except that every phrase of the form "add t to the content of register $2l+s$" is replaced by "subtract (resp. multiply, divide) t from (resp. by, into) the pseudo-content of register $2l+s$ and add 1 to the result." [Since the content of register $2l+s$ isn't 0, the "pseudo" here just subtracts 1. In the case of ADD, this just cancels the "add 1 to the result," but not in the other three cases.]

JUMP

Write $2c(pt)$ into register $2l+2$. [$c(pt)$ is the operand of the current JUMP instruction. Doubling it gives the location, in the coded program, of the corresponding instruction. This becomes the new value of pt.]

Pass control to module Cycle. [We don't go to module Step, since that updates pt in non-jumping situations.]

JGTZ

If the content of register $2l+s$ is >1 then write $2c(pt)$ into register $2l+2$ and pass control to module Cycle.

Otherwise pass control to module Step.

JZERO

If the content of register $2l+s$ is 1 then write $2c(pt)$ into register $2l+2$ and pass control to module Cycle.

Otherwise pass control to module Step.

HALT

Write $c(2l+s)$ into register 0. Then halt. [Register $2l+s$ is the predator's copy of the prey's accumulator. Its content is one more than the prey's output. So this module ensures that the output of the predator differs (by one) from the output of the prey.]

Step

Increment register $2l+2$ by 1. [The program counter, which pointed to the operand of the instruction just executed, is advanced to point to the operation of the next instruction.]

Pass Control to module Cycle.

Cycle

If $pt > 2l+1$ or if $tm = 0$, then pass control to module HALT.

Otherwise, decrement register $2l+3$ by 1 and pass control to module ReadOp.

$\square$

References

[Aho, Hopcroft and Ullman 74] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974.

[Boerger 95] Egon Börger, "Annotated Bibliography on Evolving Algebras," in "Specification and Validation Methods", ed. E. Börger, Oxford University Press, 1995, 37–51.

[Blass and Gurevich 94] Andreas Blass and Yuri Gurevich, "Evolving Algebras and Linear Time Hierarchy", in "IFIP 1994 World Computer Congress, Volume I: Technology and Foundations", eds. B. Pehrson and I. Simon, North-Holland, Amsterdam, 383–390.

[Blass and Gurevich 97] Andreas Blass and Yuri Gurevich, "The Linear Time Hierarchy Theorem for Reactive Abstract State Machines", in preparation.

[dell Castillo 97] Giuseppe del Castillo, editor, Abstract State Machines, Paderborn Home Page, http://www.uni-paderborn.de/cs/asm.html.

[Dexter, Doyle and Gurevich 97] Scott Dexter, Patrick Doyle and Yuri Gurevich, "Gurevich Abstract State Machines and Schönhage Storage Modification Machines", J. of Universal Computer Science, this issue.

[Gurevich 91] Yuri Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292. Originally published in the Bulletin of European Association for Theoretical Computer Science 43 (1991), 264–284.

[Gurevich 95] Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", in "Specification and Validation Methods", Ed. E. Börger, Oxford University Press, 1995, 9–36.

[Huggins 97] James K. Huggins, editor, Abstract State Machines, Michigan Home Page, http://www.eecs.umich.edu/gasm.

[Jones 93] Neil D. Jones, "Constant Time Factors *Do* Matter", ACM Symp. on Theory of Computing, 1993, 602–611.

[Schoenhage 80] Arnold Schönhage, "Storage modification machines", SIAM J. Computing, 9 (1980), 490-508.