Journal of Universal Computer Science, vol. 3, no. 4 (1997), 279-303
submitted: 20/12/96, accepted: 21/4/97, appeared: 28/4/97 © Springer Pub. Co.

Gurevich Abstract State Machines and Schönhage Storage Modification Machines

Scott Dexter
(University of Michigan, USA
sdexter@eecs.umich.edu)

Patrick Doyle
(Stanford University, USA
pdoyle@cs.stanford.edu)

Yuri Gurevich
(University of Michigan, USA
gurevich@eecs.umich.edu)

Abstract: We demonstrate that Schönhage storage modification machines are equivalent, in a strong sense, to unary abstract state machines. We also show that if one extends the Schönhage model with a pairing function and removes the unary restriction, then equivalence between the two machine models survives.

1 Introduction

Schönhage introduced storage modification machines (Schönhage machines) in [Schönhage 70] (and expanded them in [Schönhage 80]) as a general model of computation. Although developed independently, Schönhage's model generalizes an earlier model presented by Kolmogorov in [Kolmogorov 53] and explained in [Kolmogorov and Uspenskii 63]. In both cases, the goal was to provide a machine model flexible enough to simulate the operation of arbitrary sequential algorithms "in real time." The notion of real-time simulation is defined in [Schönhage 80].

In this paper we confirm the thesis in [Blass and Gurevich 94] that Schönhage's storage modification machines are *lock-step equivalent* (defined below) to unary (i.e. containing only nullary and unary functions) sequential Gurevich abstract state machines (ASMs) without external functions. We then extend this result to show that when we extend the Schönhage machine model with an additional *pairing function* we may remove the unary restriction on the abstract state machine model without violating equivalence.

The notion of "real time" computing has changed since the time of Schönhage's work. [Gurevich 91] defines the notion of "lock-step" as an alternative to Schönhage's notion of real time. For the purpose of this paper, the rather limited definition of lock-step simulation we present below suffices.

The kind of computing devices (algorithms, machines, etc.) we consider here are deterministic devices which interact with the environment in the following way: the input is given ahead of time, output may be emitted (to the environment) at any step, and there is no other interaction. We presume that for each machine there is a well-defined notion of states, initial states, final states, and sequence of states (such that state A_{i+1} succeeds A_i). Furthermore, each state contains a particular binary string, the *input*, and each state transition may or may not yield a one-bit *output*. A *run* of machine $\mathcal{A}$ is a sequence $\langle A_i : i \in \Lambda \rangle$ where Λ is a nonempty initial sequence of $\mathbb{N}$. A_0 is an initial state; if Λ is finite and $i = \max(\Lambda)$ and A_i is not a final state, then A_i is a *hang* state. No A_j, $j < i$, is final.

Definition. A machine $\mathcal{B}$ *simulates a machine* $\mathcal{A}$ *in lock-step with lag factor* c if there exists a mapping ϕ from the states of $\mathcal{A}$ to the states of $\mathcal{B}$ such that for every run $\langle A_i : i \in \Lambda \rangle$ of $\mathcal{A}$, there is a run $B_0, B_1, \ldots$ of $\mathcal{B}$ and a monotone function $J : \Lambda \to \mathbb{N}$ (from indices to indices) such that

1. $J(0) = 0$. Moreover A_0 and B_0 contain the same input.
2. $B_{J(i)} = \phi(A_i)$ and if x is the input at A_i, then x is exactly the input at $B_{J(i)}$.
3. If an output β is emitted during the transition from A_i to A_{i+1}, then there is a unique $l \in [J(i), J(i+1))$ such that an output is emitted during the transition from B_l to B_{l+1}. Furthermore, this output is β. And if no output is emitted during the transition from A_i to A_{i+1} then no output is emitted during any transition from B_l to B_{l+1}, $l \in [J(i), J(i+1))$.
4. If $0 < i < \max \Lambda$ then $J(i) - J(i-1) \leq c$.
5. If Λ is finite, $i = \max(\Lambda)$, and A_i is final, then $B_{J(i)}$ is final.

Remark. One may want instead to require a function ψ from states of $\mathcal{B}$ to states of $\mathcal{A}$, so that ϕ is replaced by a multivalued function ψ^{-1}.

In the case where $c = 1$, we say that $\mathcal{A}$ *strictly* lock-step simulates $\mathcal{B}$.

Two machine models $\mathbb{A}$ and $\mathbb{B}$ are *lock-step equivalent* if (i) for every machine $\mathcal{A}$ of $\mathbb{A}$ there is a machine $\mathcal{B}$ of $\mathbb{B}$ which lock-step simulates $\mathcal{A}$ with finite lag factor, and (ii) for every machine $\mathcal{B}$ of $\mathbb{B}$ there is a machine $\mathcal{A}$ of $\mathbb{A}$ which lock-step simulates $\mathcal{B}$ with finite lag factor. $\square$

The rest of this paper is organized as follows: in Section 2 we review the Schönhage storage modification machine model; in Section 3 we review the abstract state machine model; in Section 4 we prove that every Schönhage machine can be strictly lock-step simulated by an appropriate unary ASM; in Section 5 we prove that every unary ASM can be lock-step simulated by an appropriate Schönhage machine; and in Section 6 we prove that the Schönhage model with pairing and the sequential ASM model with no arity restriction are lock-step equivalent.

We use sans serif text to indicate abstract state machine code; `Courier` indicates Schönhage machine code.

2 Storage Modification Machines

A Schönhage machine (described fully in [Schönhage 80]) consists of a dynamic data structure (called a *Δ-structure*), combined with a finite control *program* that manipulates the structure while reading an input string and writing to an output string. Intuitively, this is a machine that reads from a one-way input tape and uses as storage a dynamic labeled multigraph. Edges in the multigraph are labeled by symbols from an alphabet Δ; elements in the multigraph are named (not necessarily uniquely) by the path to them from a distinguished center node. The machine modifies storage by adding new elements and redirecting edges (so some elements may be rendered unreachable).

Formally, Δ is a finite set (alphabet) of *directions*. The Δ-structure is a triple $S = (X, a, p)$, where X is a finite set of nodes in a graph; $a \in X$ is a distinguished *center* node of the graph; and p is a set (with cardinality $|\Delta|$) of functions from X to X indexed by elements of Δ. Thus each $\delta \in \Delta$ defines a mapping p_δ from X to X; $p_\delta(b)$ is the node found at the end of the edge starting at b labeled by δ. See Figure 1 for an example.

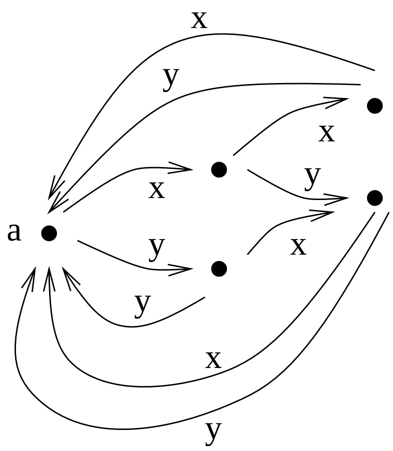

Figure 1: A possible Δ-structure for $\Delta = \{x, y\}$

p can be generalized to the mapping $p^* : \Delta^* \to X$, where we think of each word $W \in \Delta^*$ as defining a path through the structure. So we can define $p^*(\epsilon) = a$, and, recursively, $p^*(W\delta) = p_\delta(p^*(W))$. Thus we can closely associate words in Δ^* and elements of X.

A *state* of a Schönhage machine is given by the remaining input, the accumulated output, a current instruction, and a Δ-structure. In the initial state of a Schönhage machine, the remaining input is the original input string, the accumulated output is empty, the current instruction is the first in the program, and the Δ-structure contains a single node, the distinguished node a with p such that $p_\delta(a) = a$ for all $\delta \in \Delta$. That is, all pointers from the center node point back to the center node.

The control for a Schönhage machine is provided by a program in a simple programming language. There are two types of instructions in this language: *common* instructions which are the same for all Schönhage machines, and *internal* instructions which depend on Δ. The common instructions are `input`, `output`, `goto`, and `halt`, and the internal instructions are `new`, `set`, and `if`. Each statement may, optionally, have a *label* associated with it. Labels are symbols followed by colons that precede statements. They are used so that other statements in a program may refer to a particular statement. If two statements have the same label, the first one in the program is treated as the only statement with such a label.

Input and output take the form of single binary strings; these strings are manipulated, bit by bit, by the `input` and `output` commands.

The `input` instruction takes the form `input` λ_0, λ_1. A symbol $\beta \in \{0, 1\}$ is read from the input string. If $\beta = 0$, control is transferred to the statement labeled λ_0; if $\beta = 1$, control is transferred to λ_1. If the input string is empty, control is transfered to the next instruction.

The `output` instruction takes the form `output` β. Intuitively, β is emitted to the environment during the execution of this instruction.

The `goto` instruction takes the form `goto` λ, and transfers control to the statement labeled by λ.

The `halt` instruction causes the program to halt. The machine also halts if control passes the end of the program.

The `new` instruction takes the form `new` W. This causes a new node y to be created and added to X; its placement with respect to the other nodes and pointers is determined by W. If W is the empty string (denoted $\Box$), this has the effect of creating a new center node a, with all pointers from the new a pointing to the old a (and no other pointers are changed). If we think of W as having the form $U\delta$, then `new` $U\delta$ causes the δ-pointer from the node indicated by U to be redirected to the new node y, and all pointers from y to point to the original node described by $U\delta$. No other pointers are changed. For example, `new` xy creates a new node that is reached by following the y pointer from the node designated by x. All pointers from this new node point to the node previously designated by xy. See Figure 2.

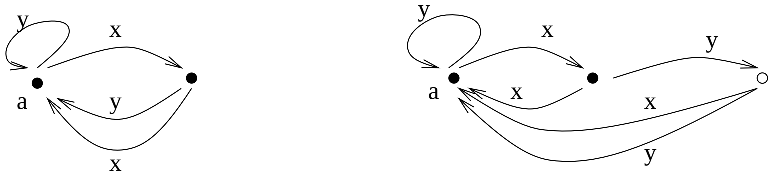

Figure 2: `new` xy

The `set` instruction takes the form `set` W `to` V. This causes a pointer redirection. If W is the empty string, then this has the effect of renaming a to be the node indicated by V. If we think of W as $U\delta$, this causes the δ-pointer from U to be directed to the node indicated by V, and no other pointers are changed. See Figure 3 for an illustration of the effect of this instruction.

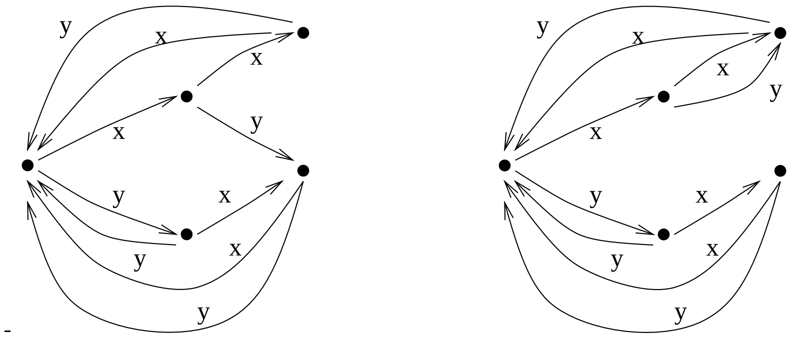

Figure 3: `set` xy `to` xx

Finally, the `if` instruction may take either the form `if` $U = V$ `then` σ or `if` $U \neq V$ `then` σ. σ is an instruction of one of the above types (i.e. not an `if` statement) which is executed iff $p^*(U) = p^*(V)$ or $p^*(U) \neq p^*(V)$, respectively.

A *run* of a Schönhage machine is a sequence of states such that each state is computed from the previous state by executing the previous state's current instruction.

3 Abstract State Machines

An abstract state machine $\mathcal{A}$ (described fully in [Gurevich 95]) is given by a *signature*, a *program*, and an initial *state*. For the purposes of this paper, we restrict our attention to sequential abstract state machines without external functions. The *signature* (or *vocabulary*) of $\mathcal{A}$ is a finite collection of function names, each with a fixed arity. Some function names will be regarded as relation names. A *state* of $\mathcal{A}$ is a set, the *superuniverse*, together with interpretations of the function names in the vocabulary. The superuniverse does not change as $\mathcal{A}$ evolves; the interpretations of the functions may. In particular, the interpretations of *dynamic* functions may change; *static* functions maintain a single interpretation during the course of a run.

The superuniverse X contains distinguished elements true, false, and undef which allow us to deal with relations and partial functions, where $f(\bar{a}) = \mathsf{undef}$ intuitively means f is undefined at $\bar{a}$. These three elements are *logical constants*.

An r-ary function name is interpreted as a function from X^r to X; an r-ary relation name is interpreted as a function from X^r to $\{\mathsf{true}, \mathsf{false}\}$. Boolean terms are built by combining terms $f(\bar{t})$, where f is a relation name, using the Boolean operators and, or, and not.

A *universe* U is a unary relation usually identified with the set $\{x\colon U(x)\}$. The universe $\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$ is another logical constant. When we speak about a function f from a universe U to a universe V, we mean formally that f is a unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = \mathsf{undef}$ otherwise. We use intuitive notation like $f\colon U \to V$, $f\colon U_1 \times U_2 \to V$, and $f\colon V$. The last means that the nullary function (or *distinguished element*) f belongs to V.

We assume that every ASM has the universe $\mathsf{Modes} = \{\mathsf{Initial}, \mathsf{Working}, \mathsf{Final}\}$ in its vocabulary; the distinguished element Mode holds the current mode of the program.

The ASM model [Gurevich 95] does not include input/output conventions, allowing users some freedom. Here we adopt the following conventions. Input is a binary sequence. Input is represented by a universe $\mathsf{InputPositions}$ with distinguished elements 0 and Last, and unary functions Succ and Bit. 0, Last, and Succ give an ordering on the elements of the universe; $\mathsf{Succ}(\mathsf{Last}) = \mathsf{undef}$. We may abbreviate $\mathsf{Succ}(0)$ by 1, $\mathsf{Succ}(1)$ by 2, etc. Bit maps $\mathsf{InputPositions}$ to $\{0, 1\}$ (where 0 and 1 are elements of $\mathsf{InputPositions}$; we assume $\mathsf{InputPositions}$ contains at least these two elements). The input string itself is represented by the distinguished element $\mathsf{InputString}$ (this represents the "current position" in the input string; thus $\mathsf{Bit}(\mathsf{InputString})$ represents the current bit in the input string).

We represent output with a nullary function Output. Intuitively, if $\langle A_i : i \in \Lambda\rangle$ is a run, $\mathsf{Output} = \beta \neq \mathsf{undef}$ at A_i, and $i + 1 \in \Lambda$, then β is emitted to the environment during the transition from A_i to A_{i+1}. We assume $\mathsf{Output} = \mathsf{undef}$ at A_0.

A program of $\mathcal{A}$ is a transition rule. The simplest transition rule is an *update*, which has the form

$$f(t_1, \ldots, t_r) := t_0$$

where we may abbreviate $t_1, \ldots, t_r$ as $\bar{t}$. When an update rule is fired, the function f at $\bar{t}$ is changed so that its value in the next state is t_0.

The other transition rules are defined inductively. If R_0 through R_k are transition rules, g_0 through g_k are Boolean terms (built using Boolean operators from terms of the form $f(\bar{t})$ where f is a relation name), and k is a natural number, then the following are transition rules:

(i) block $R_0 \ldots R_k$ endblock

(ii) if g_0 then R_0
elseif g_1 then R_1
⋮
elseif g_k then R_k
endif

(iii) import v R_0 endimport

(i) is the *block rule*; a block of transition rules is fired by firing all rules simultaneously (the block/endblock notation is often omitted when the meaning is clear from context). During the firing of these rules, all terms are evaluated and the set of updates to execute is computed. If there is some pair of updates which attempt to set the value of one location to two different values, then we say the update set is *inconsistent*; in this case *no* updates are fired and the state does not change. Otherwise the set of updates is *consistent* and all updates in the set are simultaneously fired.

(ii) is the *conditional rule*; the *guards* $g_0, g_1, g_2, \ldots$ are evaluated sequentially until some g_i evaluates to true, at which point the corresponding R_i is fired. If no g_i evaluates to true then no R_i is fired and the rule does not change the state.

(iii) is the *import rule*; this is used when we need to create a new element (e.g. add a new node to a graph or create a new message in some protocol). v refers to an element which is brought from the special universe Reserve; typically v appears in R.

A *run* of a program is a sequence of states such that each state is computed from the previous state by applying the updates determined by the program.

3.1 Normal Forms

We introduce a variety of normal forms for abstract state programs which will be of use to us in reasoning about properties of general ASM programs.

We may suppose without loss of generality (wlog) that an abstract machine program does not reuse variables; that is, each occurrence of import v has a different variable v. Furthermore, an ASM program may be written without the use of elseif; e.g. we may rewrite

if g_0 then R_0 elseif g_1 then R_1 elseif g_2 then R_2 endif	as	if g_0 then R_0 endif if not g_0 and g_1 then R_1 endif if not g_0 and not g_1 and g_2 then R_2 endif

3.1.1 First Normal Form

An arbitrary abstract state program may be put in NF1

```
import v1,...,vk
  R
endimport
```

by rewriting

if g_0 then
  import v
    R
  endimport
endif
	as	import v
		  if g_0 then
		    R
		  endif
		endimport

and rewriting

import v
  R
endimport
import w
  S
endimport
	as	import v,w
		  R
		  S
		endimport

where v and w are distinct under our assumption that variables are not reused.

3.1.2 Second Normal Form

Programs in Second Normal Form (NF2) have the form

```
import v1,...,vk
    if g1 then R1 endif
    ⋮
    if gn then Rn endif
endimport
```

where $R_1, \ldots, R_n$ are updates.

We may translate programs in NF1 to programs in NF2 by rewriting

```
if g then
    if h then
        R
    endif
endif
```

as

```
if g and h then R endif
```

and rewriting

```
if g then R1,...,Rn endif
```

as

```
if g then R1 endif
if g then R2 endif
⋮
if g then Rn endif
```

(recall that all updates are executed simultaneously). In general, this will produce a program of the form

```
import v1,...,vk
    if g1 then R1 endif
    ⋮
    if gl then Rl endif
    R(l+1)
    ⋮
    R(l+m)
endimport
```

where $R_1, \ldots, R_{l+m}$ are updates. We then rewrite this as

```
import v_1,...,v_k
    if g_1 then R_1 endif
    ⋮
    if g_l then R_l endif
    if true then R_{l+1} endif
    ⋮
    if true then R_{l+m} endif
endimport
```

3.1.3 Third Normal Form

Programs in Third Normal Form (NF3) have the form

if CONS then R endif

where

1. R is a program in NF2.
2. For every state A satisfying CONS, R is consistent (i.e. generates a consistent update set) at A.

Let if g_i then $h_i(\bar{t}_i) := \tau_i$ be a transition rule in R. The desired CONS is the conjunction of the Boolean terms

$$[g_i \wedge g_j \wedge (\bar{t}_i = \bar{t}_j)] \rightarrow (\tau_i = \tau_j)$$

for all pairs $i < j$ such that the function symbols h_i and h_j are identical.

Programs in NF3 are consistent: if R is inconsistent in a given state, CONS will be false, so no updates in R will be selected to fire.

Lemma 1. Every ASM program Π reduces to an NF3 program Π' such that Π and Π' simulate each other in strict lock-step. Furthermore, if Π is unary then so is Π'.

The proof is obvious.

3.1.4 Fourth Normal Form

Programs in Fourth Normal Form (NF4) have a vocabulary containing arbitrarily many nullary and unary function names, at most one binary function name, and no function names of arity greater than 2.

Given an abstract state program Π, constructing its NF4 translate Π' is straightforward: first, a new binary function name *Pair* is added to the vocabulary; second, every function name f of arity greater than 1 is replaced by the unary function name f'. Terms $f(t_1, t_2, \ldots, t_{r-1}, t_r)$ (for $r > 1$) are correspondingly replaced by terms $f'(Pair(t_1, Pair(t_2, \ldots, Pair(t_{r-1}, t_r)) \cdots)$. Thus, the abstract state machine Π which has functions of arbitrary arity is simulated by the abstract state machine Π' which has only unary functions, with the exception of the binary function $Pair$.

Lemma 2. An arbitrary ASM program Π reduces to an NF4 program Π' such that Π and Π' simulate each other in strict lock-step.

4 Simulating Schönhage machines by Unary Abstract State Machines

Theorem 1. For every Schönhage machine $\mathcal{A}$ there exists an abstract state machine $\mathcal{B}$ that strictly lock-step simulates $\mathcal{A}$.

Before proving Theorem 1, we provide some necessary infrastructure, as well as the mapping ϕ required by the definition of lock-step simulation. Because of the flexible and adaptable nature of the abstract state machine paradigm, this argument is much simpler than its inverse. $\mathcal{B}$ can be seen as a formalization of $\mathcal{A}$ in the abstract state machine context. We begin by providing the vocabulary of $\mathcal{B}$.

We first observe that the initial state of a Schönhage machine contains the program, a pointer alphabet, a center node, an input string, and an empty output string. We simulate this as closely as possible in the abstract state machine model.

Input and output are simulated in a natural way by the ASM input/output conventions.

We regard the Schönhage program as an ordered list of instructions; for the purpose of a close simulation we reuse elements in the universe of InputPositions to index instructions. (We assume that the length of the input is greater than the number of instructions.) We also need a distinguished element CurInst which holds the index of the current instruction.

The universe Nodes is initially empty; it has a distinguished element Center which is initially undef. This universe will be filled as the Schönhage program creates new nodes. For every direction δ in Δ of the Schönhage machine, there is a unary function $\delta : \mathsf{Nodes} \rightarrow \mathsf{Nodes}$.

Finally, Mode is a distinguished natural number which encodes the phase of execution of the simulation; we abbreviate 0 by Initial, 1 by Working, and 2 by Halt.

Note that all functions described here are nullary (i.e. distinguished elements) or unary.

4.1 Simulating Schönhage Programs

In this section we describe how to translate a Schönhage program into a simulating abstract state machine program. We occasionally use the notation $x.f$ for $f(x)$.

We first specify the initial state of the abstract state machine. Mode must be Initial and CurInst is set to 0. Output is undef; Bit$(n) \in \{0,1\}$ for all $n \in$ InputPositions. InputString is set to 0.

Each Schönhage instruction is translated as a transition rule guarded by a test of the index of the rule (i.e. the value of CurInst). We first give the special case of CurInst $= 0$.

```
if CurInst = 0 and Mode = Initial then
        import y
                Nodes(y) := true
                Center := y
                δ_1(y) := y
                        ⋮
                δ_m(y) := y
        endimport
        CurInst := 1
        Mode := Working
endif
```

Before the simulation proper begins, we import a Center node and set the pointer values appropriately, assuming a pointer alphabet $\Delta = \{\delta_1, \ldots, \delta_m\}$.

The translations of the instructions depend on their index and their type (i.e. `new`, `set`, `output`, etc.) We consider the ith instruction and present schemata for each type of instruction.

`input` λ_0, λ_1

```
if CurInst = i and Mode = Working then
        if Bit(InputString) = undef then
                CurInst := Succ(CurInst)
        elseif Bit(InputString) = 0 then
                CurInst := Lambda0
        else CurInst := Lambda1
        endif
        InputString := Succ(InputString)
endif
```

The `input` command reads a bit from the input string. If the bit is undef, then the instruction is effectively skipped and control moves to the next instruction. If the bit is either a 0 or 1, control is transferred to the instruction numbered Lambda0 or Lambda1 respectively, where these are the indices of the instructions with labels λ_0 and λ_1.

`output` β

```
if CurInst = i and Mode = Working then
    Output := Beta
    CurInst := Succ(CurInst)
endif
```

This simply emits the appropriate output bit. In order to ensure that Ouput is undef for all other (non-output) instructions, we need a rule to explicitly accomplish this.

```
if CurInst ≠ i_1 and ... and CurInst ≠ i_k then
    if Output ≠ undef then
        Output := undef
    endif
endif
```

Suppose $i_1, \ldots, i_k$ are all the indices of output instructions. This rule tests the index of the current instruction and sets Output to undef if the current instruction is not an output instruction.

`goto` λ

```
if CurInst = i and Mode = Working then
    CurInst := Lambda
endif
```

This simply causes control to be transferred to the instruction indexed by Lambda (which is the index of the instruction labeled by λ).

`halt`

```
if CurInst = i and Mode = Working then
    Mode := Final
endif
```

When Mode is set to Final, no further rules will fire, as all are guarded either by Mode = Initial or Mode = Working.

`new` $\square$	`new` $\mathtt{w}_1 \ldots \mathtt{w}_k$ $(k \geq 1)$
if CurInst = i and Mode = Working	if CurInst = i and Mode = Working
then	then
import y	import y
Nodes(y) := True	Nodes(y) := True
Center := y	β(PARENT) := y
$\delta_1(y)$:= Center	$\delta_1(y)$:= δ_1(PARENT)
$\vdots$	$\vdots$
$\delta_m(y)$:= Center	$\delta_m(y)$:= δ_m(PARENT)
endimport	endimport
CurInst := Succ(CurInst)	CurInst := Succ(CurInst)
endif	endif

PARENT abbreviates $\mathtt{w}_1 \ldots \mathtt{w}_{k-1}$.

To add a new element, we import element y from Reserve into Nodes. If the parameter of the `new` instruction is empty, we make this new node the Center and set all pointers from this node to the old Center. Otherwise, we set the β pointer from the node designated by PARENT to point to the new element y; in addition, every possible pointer from the new element y is set to point to the element previously referred to by PARENT.

Remark. Normally (that is, without being bound by our obligation to use only unary functions and restricted abstract state machine syntax), an abstract state machinist would view elements of Δ as elements of the superuniverse and write the last portion of this rule as

var $\delta \in \Delta$
    Neighbor(δ,y) := PARENT
endvar

where Neighbor is a binary function on elements of the superuniverse.

`set` $\square$ `to` $\mathtt{v}_1\mathtt{v}_2 \ldots \mathtt{v}_j$	`set` $\mathtt{w}_1\mathtt{w}_2 \ldots \mathtt{w}_k$ `to` $\mathtt{v}_1\mathtt{v}_2 \ldots \mathtt{v}_j$ $(k \geq 1)$
if CurInst = i and Mode = Working	if CurInst = i and Mode = Working
then	then
Center := Center.v_1.$\mathsf{v}_2 \ldots \mathsf{v}_j$	w_k(Center.w_1.$\mathsf{w}_2 \ldots \mathsf{w}_{k-1}$) :=
CurInst := Succ(CurInst)	Center.v_1.$\mathsf{v}_2 \ldots \mathsf{v}_j$
endif	CurInst := Succ(CurInst)
	endif

The `set` instruction changes function w_k at the point $w_{k-1}(w_{k-2}(\ldots(w_1(\mathsf{Center}))\ldots)$ to point to the element designated by $v_j(v_{j-1}(v_{j-2}(\ldots v_1(\mathsf{Center})\ldots)))$; if `W` is empty, this renames the Center node to the element designated by $v_j(v_{j-1}(v_{j-2}(\ldots v_1(\mathsf{Center})\ldots)))$.

`if` $\mathtt{u}_1\mathtt{u}_2\ldots\mathtt{u}_k = \mathtt{v}_1\mathtt{v}_2\ldots\mathtt{v}_j$ `then` σ

```
if CurInst = i and Mode = Working then
        if Center.u1u2...uk = Center.v1v2...vj then
                Rσ
        else
                CurInst := Succ(CurInst)
        endif
endif
```

`if` $\mathtt{u}_1\mathtt{u}_2\ldots\mathtt{u}_k \neq \mathtt{v}_1\mathtt{v}_2\ldots\mathtt{v}_j$ `then` σ

```
if CurInst = i and Mode = Working then
        if Center.u1u2...uk ≠ Center.v1v2...vj then
                Rσ
        else
                CurInst := Succ(CurInst)
        endif
endif
```

R_σ is the abstract state machine update (without the guards of CurInst and Mode) corresponding to the Schönhage machine instruction σ (which is of one of the previous types of instructions). The nodes indicated by U and V are compared, and σ is executed in the appropriate circumstance.

4.2 State Mapping

Per the definition of lock-step simulation, we define ϕ to be a mapping of Schönhage machine states to abstract state machines states such that:

1. If the string $\mathtt{w}_1\ldots\mathtt{w}_k$ designates an element x in A then in $\phi(A)$ $\mathsf{Center}.\mathsf{w}_1\ldots\mathsf{w}_k$ evaluates to x.
2. If k is the index of the current instruction in A, then CurInst evaluates to k in $\phi(A)$.

3. If A contains input x, then $\phi(A)$ contains input x.
4. If the bit β is emitted during the transition from state A, β is emitted during the transition from $\phi(A)$. Otherwise no output is emitted.
5. If A is in the initial (respectively halting) state, Mode equals Initial (respectively Final) in $\phi(A)$; otherwise Mode = Working in $\phi(A)$.

In particular we consider an initial state A of the Schönhage machine $\mathcal{A}$. In the corresponding state $B = \phi(A)$ we have Mode = Initial; CurInst = 0; Output = undef; Bit$(n) \in \{0, 1\}$ for all $n \in$ InputPositions; InputString = 0. In particular every element of A is an element of B. When $\mathcal{A}$ creates a new element a, $\mathcal{B}$ imports some a' to represent a.

This creates a natural one-to-one mapping of elements of any state A of $\mathcal{A}$ to elements of the corresponding state of $\mathcal{B}$. For simplicity, and wlog, we may identify a' with a.

4.3 Proof of Theorem 1

Lemma 3. Let $A_0, A_1, \ldots$ be a run of a Schönhage program $\mathcal{A}$. Let $\mathcal{B}$ be the abstract state machine translate of $\mathcal{A}$. Let $B_0, B_1, \ldots$ be a run of $\mathcal{B}$ such that $B_0 = \phi(A_0)$. Then for every i, $B_i = \phi(A_i)$.

Proof. We prove Lemma 3 by induction on the index of the sequence of Schönhage machine states. The base case follows from our assumptions about the initial state of the simulating abstract state machine.

Now we must demonstrate that if $B_{k-1} = \phi(A_{k-1})$ then $B_k = \phi(A_k)$, where A_k is obtained from A_{k-1} by executing one instruction of $\mathcal{A}$. This follows relatively intuitively from the semantics discussed in §3.1 above, but we consider each case in detail.

We consider each possible type of Schönhage machine instruction.

`input` λ_0, λ_1. This simply shifts control in a manner dependent on the value of the state's input string. In the abstract state machine, this is simulated by testing the value of a bit in the InputString and updating CurInst appropriately.

`output` β. This causes the bit β to be emitted, and control shifts to the next instruction in the program. In $\mathcal{A}$, the output bit is emitted, then control is incremented.

`goto` λ. This causes control to shift to the statement labeled λ. In A, CurInst is set to the index of the appropriate instruction.

`halt`. This stops execution. In $\mathcal{A}$, Mode is set to Final, which prevents any transitions from firing.

`new W` (where `W` = $\mathtt{w}_1, \ldots, \mathtt{w}_k$). This instruction makes three changes to the Δ-structure. First, a new node, y, is added to the structure. If `W` is empty, this node becomes the center, and all pointers from the new node point to the old center. Otherwise, the pointer labeled $\mathtt{w}_k$ from the node reached by following $\mathtt{w}_1, \ldots, \mathtt{w}_{k-1}$ from a is directed to y, and all pointers δ from y are directed to the node formerly reached by $\mathtt{w}_1 \ldots \mathtt{w}_k$. In $\mathcal{A}$, the import constructor brings a new element y into the universe of *Nodes*, the function w_k is updated, and $\delta(y)$ for every δ is set to the former value of $\delta(\mathsf{Center}.w_1 \ldots w_k)$. And, of course, control in both the Schönhage program and abstract state machine program is incremented.

`set W to V`. If `W` is empty, the center is renamed to be the node $\mathtt{v}_1, \ldots, \mathtt{v}_j$; otherwise, this causes the pointer $\mathtt{w}_k$ from the node $\mathtt{w}_1, \ldots, \mathtt{w}_{k-1}$ to be redirected to the node $\mathtt{v}_1, \ldots, \mathtt{v}_j$, and control is incremented. In $\mathcal{A}$, either

$$\mathsf{Center}.w_1.w_2 \ldots w_{k-1} := \mathsf{Center}.v_1.v_2 \ldots v_j$$

or

$$w_k(\mathsf{Center}.w_1.w_2 \ldots w_{k-1}) := \mathsf{Center}.v_1.v_2 \ldots v_j$$

is fired, and CurInst is updated.

`if` $U = [\neq] V$ `then` σ. These two statements test the nodes found by traversing `U` and `V`; if they are equal [not equal], then σ is executed; otherwise control passes to the next instruction. In $\mathcal{A}$, the guard testing the equality of the two terms represented by `U` and `V` is evaluated; if it is true, then R_σ is executed; otherwise CurInst is incremented.

This demonstrates that each transition from $\phi(A_{k-1})$ to $\phi(A_k)$ can be achieved by firing $\mathcal{B}$ at $\phi(A_{k-1})$, and thus that $\phi(A_i) = B_i$. □

From Lemma 3 we may conclude that the given ϕ with $J(i) = i$ fulfills the definition of lock-step simulation with lag factor 1; thus Theorem 1 is proved.

5 Simulating Unary Abstract State Machines by Schönhage Machines

Theorem 2. For every unary abstract state machine $\mathcal{A}$ there exists a Schönhage machine $\mathcal{B}$ that strictly lock-step simulates $\mathcal{A}$.

For the purposes of proving Theorem 2, we present a methodology for converting a given unary abstract state machine with input into an equivalent Schönhage machine. We then provide a mapping of abstract state machine states to Schönhage machine states, and proceed with the proof. We restrict the input of the abstract state machine to binary sequences to match the input of Schönhage machines (note that other forms of input may be encoded as binary sequences). Intuitively, our approach will be to use the pointers in the data structure of the Schönhage machine to represent the values of the functions in the abstract state machine. We simplify our task somewhat by assuming that the abstract state program being simulated is in NF3 (i.e. is consistent), but some extra effort is required to account for the evaluation of guards — these are expressions that involve binary (Boolean) functions, so there is no clean way of expressing them using the inherently unary Schönhage machine constructs.

First, we describe the pointer alphabet of the simulating Schönhage machine. The elements of this alphabet depend on three kinds of functions in the abstract state machine: static nullary functions, dynamic nullary functions (i.e. constants), and unary functions. The nullary functions are used to name elements in the abstract state machine; these will be translated into the Schönhage machine Δ-structure as pointers emanating directly from the center node (note that this construction obligates us to refrain from moving the center node at any point during the simulation). Unary functions will be translated as pointers directed from elements to elements (that is, from node to node in the Schönhage machine). The destination node represents the value of the unary function when applied to the element represented by the source node.

Because we are essentially simulating the execution of a parallel machine by a sequential machine, we need to augment the vocabulary of the Schönhage machine. Specifically, for every function name f in the abstract state machine vocabulary, we include an additional pointer "shadow" f' in the pointer alphabet of the Schönhage machine. These shadow pointers will be used to accumulate updates which will then be applied after all guards have been evaluated.

We include extra nodes which are reached from the center by pointers `True` and `False`. The center node itself corresponds to undef in the abstract state machine. We also add a finite number of additional pointers $\mathtt{New}_1, \ldots, \mathtt{New}_k$ to be used in simulating import in the abstract state machine, where k is the number of variables imported at the beginning of the NF3 abstract state program.

The initial state of an ASM consists of a superuniverse, interpretations on function names in the vocabulary, a universe of InputPositions, and Output = undef. To simulate this, we assume that the initial state of the simulating Schönhage machine contains an input string and an empty output string, and a Δ-structure that reflects the superuniverse and the initial vocabulary interpretation. Specifically, for every element x of the superuniverse such that $f_k(f_0.f_1 \ldots f_{k-1}) = x$ for some $f_0, \ldots, f_k$, $\mathtt{f}_0\mathtt{f}_1 \ldots \mathtt{f}_k$ and $\mathtt{f}_0\mathtt{f}_1 \ldots \mathtt{f}'_k$ designate x in the Δ-structure.

5.1 Execution

We first describe how each abstract state machine rule is converted into a Schönhage program fragment, then discuss how these fragments are combined to simulate the entire abstract state machine.

5.1.1 Updates

Since the maximum arity allowed in our case is 1, each update instruction has the form $f_k(f_0.f_1 \ldots f_{k-1}) := g_0.g_1 \ldots g_l$, where f_0 and g_0 are nullary functions.

Because we need to separate the tasks of evaluating and updating in the simulation, we translate each update to Schönhage machine code as $\texttt{set f}_0\texttt{f}_1 \ldots \texttt{f}'_k$ `to` $\texttt{g}_0\texttt{g}_1 \ldots \texttt{g}_l$ (abbreviated `set F' to G` where clear from context), where $\texttt{f}_i$ and $\texttt{g}_i$ are the Schönhage machine pointers corresponding to the appropriate abstract state machine functions, and $\texttt{f}'_k$ is the shadow of $\texttt{f}_k$. We then later include code that copies the relevant values of $\texttt{f}'$ to $\texttt{f}$.

5.1.2 Importing Elements

New elements in an abstract state machine can be brought into a universe by using the import constructor; since we are assuming the program is in NF3, the constructor will be of the form

import $v_1 \ldots v_k$
    R
end import

This is translated into the Schönhage program fragment

new $\texttt{New}_1$
⋮
new $\texttt{New}_k$
R'

where R' is the translation of R into Schönhage program code with every occurrence of v_i replaced by $\texttt{New}_i$.

5.1.3 Conditional Constructors and Guards

As we are assuming NF3, we need only describe how to translate conditional constructors of the form

if g then R endif

into Schönhage machine code. We let R' denote the sequence of Schönhage machine instructions which simulates R.

If g is a Boolean term of the form $f(t)$, where f is a unary relation and t is a term composed of unary functions. We simply translate g as $F =$ `True`, where, as before, F is the is the word corresponding to the Boolean term. Thus, we translate if g then R as

```
    if F = True then goto 𝓛
    goto 𝓛'
𝓛 : R'
𝓛' :
```

where $\mathcal{L}$ and $\mathcal{L}'$ are labels, and $\mathcal{L}'$ labels the statement following the conditional rule.

Otherwise, the translation is more complicated. We illustrate it with an example, where g is $(\neg g_a \mathsf{or} g_b)\mathsf{and}(t_1 = t_2)$ and g_a, g_b, t_1, and t_2 are unary terms.

```
     if G_a = False then goto 𝓛1
     if G_b = True then goto 𝓛1
     goto 𝓛3
𝓛1 : if T1 = T2 then goto 𝓛2
     goto 𝓛3
𝓛2 : R'
𝓛3 :
```

where G_a, G_b, $\mathtt{T}_1$, and $\mathtt{T}_2$ are the Schönhage translations of g_a, g_b, t_1, and t_2.

5.1.4 Programs

If we let $\mathcal{B}'$ be the Schönhage program we get by applying the above translations to the ASM program $\mathcal{A}$, then the Schönhage program $\mathcal{B}$ equivalent to $\mathcal{A}$ has the form

```
𝓛: if Mode = Final halt
   𝓑'
   UPDATE
   goto 𝓛
```

where UPDATE is a sequence of set instructions of the form set $\mathtt{f}_0\mathtt{f}_1 \ldots \mathtt{f}_k$ to $\mathtt{f}_0\mathtt{f}_1 \ldots \mathtt{f}'_k$, with one such instruction for every $\mathtt{f}_0\mathtt{f}_1 \ldots \mathtt{f}'_k$ that appears in $\mathcal{B}'$. Thus the Schönhage machine states at which instruction 1 is about to be executed correspond naturally to ASM states; we refer to these Schönhage states as *restart* states.

5.2 State Mapping

We define ϕ to be a mapping of ASM states to Schönhage machine states such that:

1. If $f_k(f_0.f_1 \ldots f_{k-1}) = x$ in A_i then $\mathtt{f}_0\mathtt{f}_1 \ldots \mathtt{f}_k$ designates x in $\phi(A_i)$.
2. The index of the instruction about to be executed in $\phi(A_i)$ is 1.
3. If A contains input x then $\phi(A)$ contains input x.
4. If the bit β is emitted between states A_{i-1} and A_i, then exactly β is emitted between states $\phi(A_{i-1})$ and $\phi(A_i)$. Otherwise no output is emitted between $\phi(A_{i-1})$ and $\phi(A_i)$.
5. If $\mathsf{Mode} = \mathsf{Initial}$ (respectively, $\mathsf{Mode} = \mathsf{Final}$) in A_i, then $\phi(A_i)$ is an initial (respectively, halting) state; otherwise $\phi(A_i)$ is neither initial nor halting.

As before, we observe that we may identify elements of states of $\mathcal{A}$ with elements of the corresponding states of $\mathcal{B}$.

5.3 Proof of Theorem 2

Lemma 4. Let $A_0, A_1, \ldots$ be a run of an abstract state machine $\mathcal{A}$. Let $\mathcal{B}$ be the Schönhage machine translate of $\mathcal{A}$. Let $B_0, B_1, \ldots$ be the run of $\mathcal{B}$ such that $B_0 = \phi(A_0)$. Let $S_0, S_1, \ldots$ (where $S_0 = B_0$) be the subsequence of restart states of the run of $\mathcal{B}$. Then for every i, $S_i = \phi(A_i)$.

Proof. Since the abstract state program is consistent, we may assume wlog that the updates selected to fire at some state A_i affect distinct locations (although more than one update may update a given location to the same value), so we may consider them independently of each other.

We proceed by induction on i, the index of the abstract state machine state.

The base case (S_0) follows by assumption.

To show the induction, we must show that the Schönhage machine correctly simulates the firing of the update set of A_i. Specifically, we must show that if the update $\mathsf{f}(\mathsf{t}) := \mathsf{t}_0$ is fired in state A_{k-1}, then the updates `set T′ to T`$_0$ and `set T to T′` are executed between S_{k-1} and S_k. We observe that as the updates `set T to T′` are not guarded, it suffices to demonstrate the execution of `set T′ to T`$_0$.

We proceed by structural induction on the transition rule in which the update occurs.

If the update is fired in an update rule, then the translation is direct and the Schönhage machine instruction `set T′ to T`$_0$ is executed.

If the update occurs within the scope of the import rule, then there are two cases. If the update does not refer to some v_i, then this reduces to the base case. If it does refer to v_i, then the simulated update will be fired with every occurrence of v_i replaced by New_i for the appropriate i. We must check to see that the abstract state machine import is simulated correctly: when an element is imported in an abstract state machine, it is (1) distinct from any elements that already exist and (2) distinct from any elements that might be imported simultaneously. When this is simulated by a Schönhage machine, condition (1) is met by the semantics of the Schönhage machine `new` instruction: the instruction `new` $\mathtt{New}_i$ creates a new node which is reached from the center node by following the $\mathtt{New}_i$ pointer (earlier nodes designated by $\mathtt{New}_i$ are chained from the newest node). Condition (2) is met by the pointer vocabulary; one $\mathtt{New}_i$ pointer exists for each variable imported.

If the update occurs within the scope of a conditional rule, then it is guarded by some guard g_j. By inspection, we see that when a guard g_j is true in A_{i-1}, then the Schönhage machine translation of the guarded rule is executed between $\phi(A_{i-1})$ and $\phi(A)$.

Finally, if it occurs in a block rule, we know that no other updates in that block affect the same location (by consistency), so this reduces to the base case.

Thus, exactly those updates fired in A_{k-1} are executed in the Schönhage machine simulation between states S_{k-1} and S_k. Therefore $S_i = \phi(A_i)$. □

From Lemma 4 we may conclude that the given ϕ and a J which maps indices of ASM states to indices of Schönhage restart states fulfill the definition of lock-step simulation, where c is a finite number determined by the number of instructions in the program $\mathcal{B}$. Thus Theorem 2 is proved.

6 Extended Schönhage Machines

We now discuss extending the Schönhage machine model by adding a pairing function. In this section, abstract state machines are arbitrary (that is, not necessarily unary) sequential abstract state machines without external functions.

6.1 Extended Syntax

In contrast to the usual pairing function found in set theory, we regard the pairing function as one that encodes pairs of elements in one set with an element in another set. We will code the pairing function as a collection of auxiliary nodes with outgoing edges `First` and `Second`. These edges are permanent, in the sense that once they are created, they cannot be modified by the Schönhage machine control. The semantics is intuitive — each such node represents an ordered pair of elements indicated by the `First` and `Second` pointers. Using such a pairing function in composition with itself, we can let one node in the Schönhage machine structure represent a tuple (of arbitrary arity) of elements in the abstract state machine; e.g. (X, Y, Z) may be represented by the pairing of the pair $\langle X, Y \rangle$ and Z. Thus, while functions continue to be represented by pointers from one node to another, the nodes themselves will represent not just single-element arguments but tuples of arguments.

We must extend the syntax of Schönhage programs to incorporate the pairing function. The new command

$$\texttt{create}\ \langle \mathtt{u}_1\mathtt{u}_2\ldots\mathtt{u}_i,\mathtt{v}_1\mathtt{v}_2\ldots\mathtt{v}_j\rangle$$

creates a new node that represents the ordered pair of values represented by $\mathtt{U} = \mathtt{u}_1\ldots\mathtt{u}_i$ and $\mathtt{V} = \mathtt{v}_1\ldots\mathtt{v}_j$. This node is denoted $\langle \mathtt{U},\mathtt{V}\rangle$. This command is similar to the `new` command in that it brings a new node into the nodeset; it differs from the `new` command in that it automatically sets the pointers `First` and `Second` to point from node $\langle \mathtt{U},\mathtt{V}\rangle$ to nodes U,V, respectively — and these edges can not be altered by the `set` command. Additionally, if such a node $\langle \mathtt{U},\mathtt{V}\rangle$ already exists, Δ is not altered.

For example, to create a node representing the tuple (X, Y, Z), write

create $\langle X, Y\rangle$
create $\langle\langle X, Y\rangle, Z\rangle$

The node representing the value of f at the location specified by (X, Y, Z) is the node $\langle\langle X, Y\rangle, Z\rangle f$.

We can now show that Schönhage machines extended by a pairing function in such a way are equivalent to sequential abstract state machines containing functions of arbitrary arity.

6.2 Simulating Extended Schönhage Machines by Abstract State Machines

Theorem 3. For every extended Schönhage machine $\mathcal{A}$ there exists an abstract state machine $\mathcal{B}$ which lock-step simulates $\mathcal{A}$.

Representing the pairing function in abstract state machine code is done with a binary function Pair, and unary functions First and Second.

An extended Schönhage machine statement create $\langle \mathtt{u}_1\ldots\ \mathtt{u}_i,\mathtt{v}_1\ldots\ \mathtt{v}_j\rangle$ can be simulated by the rule

```
if Pair(Center.u1 ... ui,Center.v1 ... vj) = undef then
   import v
      Pair(Center.u1 ... ui,Center.v1 ... vj) := v
      First(v) := Center.u1 ... ui
      Second(v) := Center.v1 ... vj
   endimport
endif
```

Wlog we can identify the imported element with the pair node.

Proof. Extend the definition of ϕ from section 4.2:

6. If $\langle u_1 \dots u_i, v_1 \dots v_j \rangle = x$ in state A, then $\mathsf{Pair}(\mathsf{Center}.u_1 \dots u_j, \mathsf{Center}.v_1 \dots v_j) = x$ in $\phi(A)$

The bulk of this proof is found in the proof of Lemma 3; we need simply augment it to address the `create` statement.

`create` $\langle$`W,V`$\rangle$. This creates a new node and sets the `First` and `Second` pointers from this node appropriately. In the abstract state machine, this is mimicked exactly; our induction hypothesis guarantees that the abstract state machine translates of `W` and `V` (even if they involve pairing nodes) map appropriately to Schönhage machine nodes. □

6.3 Simulating Abstract State Machines by Extended Schönhage Machines

Theorem 4. For every abstract state machine $\mathcal{A}$ there exists an extended Schönhage machine $\mathcal{B}$ which lock-step simulates $\mathcal{A}$.

As we observed in section 3, arbitrary abstract state machines are strictly lock-step equivalent to NF4 ASMs; thus it suffices to give a simulation of NF4 ASMs by extended Schönhage machines. We assume that the binary abstract state machine function is named Pair as in §3. We translate the abstract state machine program into a Schönhage program as described in section 5, replacing terms $\mathsf{Pair}(t_1, t_2)$ by $\langle T_1, T_2 \rangle$.

Lemma 5. For every ASM program $\mathcal{A}$ in NF4 there exists an extended Schönhage machine $\mathcal{B}$ that lock-step simulates $\mathcal{A}$.

Proof. The proof is similar to the proof of Lemma 4, except that now we have to use the create instruction in order to translate terms. Extend the definition of ϕ from section 5.2:

5. If $\mathsf{Pair}(\mathsf{Center}.u_1 \dots u_j, \mathsf{Center}.v_1 \dots v_j) = x$ in A_i then $\langle u_1 \dots u_i, v_1 \dots v_j \rangle = x$ in $\phi(A_i)$.

We assume that $\mathcal{A}$ is in NF3 as well and consider arbitrary updates as in Lemma 4. If a given update contains no terms of the form $Pair(\bar{t}_1, \bar{t}_2)$ then the proof proceeds as in Lemma 4. If a given update contains one or more pairing terms, then we observe that, by the definition of ϕ and our construction, $\langle T_1, T_2 \rangle$ is an element of the superuniverse and so the proof of Lemma 4 applies to updates involving these elements as well.

Proof. This follows directly from Lemmae 4 and 5. □

References

[Blass and Gurevich] Blass, A., and Gurevich, Y.: "Evolving algebras and linear time hierarchy"; Proc. of IFIP Congress 94, vol. I, Elsevier (1994), 383–390.

[Gurevich 91] Gurevich, Y.: "Evolving algebras: An attempt to discover semantics"; Bulletin of European Assoc. for Theor. Computer Science, no. 43, Feb. 1991, 264–284. A slightly revised version appeared in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292.

[Gurevich 95] Gurevich, Y.: "Evolving algebras 1993: Lipari guide"; Specification and Validation Methods, Ed. E. Boerger, Oxford University Press (1995), 9–36.

[Kolmogorov 53] Kolmogorov, A. N.: "On the notion of an algorithm";Uspekhi Mat. Nauk, 8, 4 (1953), 175–176.

[Kolmogorov and Uspenskii 63] Kolmogorov, A. N. and Uspenskii, V. A.: "On the definition of an algorithm"; AMS Tranlations 2nd Series, 29 (1963), 217–245.

[Schönhage 70] Schönhage, A.: "Universelle Turing Speicherung"; Automatentheorie und Formale Sprachen, Bibliogr. Institut, Mannheim (1970), 369–383.

[Schönhage 80] Schönhage, A.: "Storage modification machines"; SIAM J. Computing, 9 (1980), 490–508.

Acknowledgements

The work of Yuri Gurevich has been partially supported by NSF and ONR grants.