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)

February 5, 1997

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 [Sch70] (and expanded them in [Sch80]) as a general model of computation. Although developed independently, Schönhage’s model generalizes an earlier model presented by Kolmogorov in [Kol53] and explained in [KU63]. In both cases, the goal was to provide a machine model flexible enough to simulate the operation of arbitrary sequential algorithms in “in real time.” The notion of real-time simulation is defined by Schönhage in [Sch80].

In this paper we confirm the thesis in [BG94] 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. [Gur93] 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 $A$ is a sequence $(A_i: i \in \Lambda)$ where $\Lambda$ is a nonempty initial sequence of $\mathbb{N}$. $A_0$ is an initial state; if $\Lambda$ 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.

*Partially supported by NSF and ONR grants.
Definition. A machine $B$ simulates a machine $A$ in lock-step with lag factor $c$ if there exists a mapping $\phi$ from the states of $A$ to the states of $B$ such that for every run $\{A_i : i \in \Lambda\}$ of $A$, there is a run $B_0, B_1, \ldots$ of $B$ and a function $J : \Lambda \rightarrow \mathbb{N}$ (from indices to indices) such that

1. $J(0) = 0$. Moreover $A_0$ and $B_0$ contain the same input and neither yield any output.
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 $\beta$ 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 $\beta$.
4. If $0 < i < \Lambda$ then $J(i) - J(i-1) \leq c$.
5. If $\Lambda$ 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 $\psi$ from states of $B$ to states of $A$, so that $\phi$ is replaced by a multivalued function $\psi^{-1}$.

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

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

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 [Sch80]) consists of a dynamic data structure (called a $\Delta$-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 $\Delta$; 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, $\Delta$ is a finite set (alphabet) of directions. The $\Delta$-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 $\Delta$. Thus each $\delta \in \Delta$ defines a mapping $p_\delta$ from $X$ to $X$; $p_\delta(b)$ is the node found at the end of the edge starting at $b$ labeled by $\delta$. See Figure 1 for an example.

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

A state of a Schönhage machine is given by the remaining input, the accumulated output, a current instruction, and a $\Delta$-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
$\Delta$-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 $\Delta$. The common instructions are input, output,
go to, 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 $\lambda_0, \lambda_1$. A symbol $\beta \in \{0, 1\}$ is read from the input string. If
$\beta = 0$, control is transferred to the statement labeled $\lambda_0$; if $\beta = 1$, control is transferred to $\lambda_1$. If the input
string is empty, the input instruction is skipped.

The output instruction takes the form output $\beta$. Intuitively, $\beta$ is emitted during the environment during the
execution of this instruction.

The goto instruction takes the form goto $\lambda$, and transfers control to the statement labeled by $\lambda$.

The halt instruction causes the program to halt. The simulation 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 we think of $W$ as having the
form $U \delta$, then new $U \delta$ causes the $\delta$-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. If
$W$ is the empty string, this has the effect of creating a new center node $a$, with all pointers from the old $a$
pointing to the new $a$. 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.

The set instruction takes the form set $W$ to $V$. This causes a pointer redirection. Again, if we think of
$W$ as $U \delta$, this causes the $\delta$-pointer from $U$ to be directed to the node indicated by $V$, and no other pointers
are changed. If $W$ is the empty string, then this has the effect of renaming $a$ as the node indicated by $V$.
Finally, the if instruction may take either the form if \( U = V \) then \( \sigma \) or if \( U \neq V \) then \( \sigma \). \( \sigma \) 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 [Gur94]) 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 undefined which allow us to deal with relations and partial functions, where \( f(\bar{a}) = \text{undefined} \) 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 $\{true, false\}$. 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 $\{x : U(x)\}$. The universe $\text{Bool} = \{true, false\}$ 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 $f(a) = \text{undef}$ otherwise. We use intuitive notation like $f : U \rightarrow V$, $f : U_1 \times U_2 \rightarrow V$, and $f : V$. The last means that the nullary function (or distinguished element) $f$ belongs to $V$.

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

The ASM model [Gur94] 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 $\text{InputPositions}$ with distinguished elements $0$ and $\text{Last}$, and unary functions $\text{Succ}$ and $\text{Bit}$. $0$, $\text{Last}$, and $\text{Succ}$ give an ordering on the elements of the universe; $\text{Succ}(\text{Last}) = \text{undef}$. We may abbreviate $\text{Succ}(0)$ by $1$, $\text{Succ}(1)$ by $2$, etc. $\text{Bit}$ maps $\text{InputPositions}$ to $\{0, 1\}$ (where $0$ and $1$ are elements of $\text{InputPositions}$; we assume $\text{InputPositions}$ contains at least these two elements). The input string itself is represented by the distinguished element $\text{InputString}$ (this represents the “current position” in the input string; thus $\text{Bit}(\text{InputString})$ represents the current bit in the input string).

We represent output with a nullary function $\text{Output}$. Intuitively, if $(A_i : i \in \Lambda)$ is a run, $\text{Output} = \beta \neq \text{undef}$ at $A_i$, and $i + 1 \in \Lambda$, then $\beta$ is emitted to the environment during the transition from $A_i$ to $A_{i+1}$. We assume $\text{Output} = \text{undef}$ 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 $R_1 \ldots R_k$ endblock

(ii) if $g_0$ then $R_0$

else $g_1$ then $R_1$

;

else $g_k$ then $R_k$

endif

(iii) import $v$ $R_1$ 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.

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

\[
\begin{align*}
\text{if } g_0 \text{ then } R_0 \\
\text{elseif } g_1 \text{ then } R_1 \\
\text{elseif } g_2 \text{ then } R_2 \\
\text{endif}
\end{align*}
\]

as

\[
\begin{align*}
\text{if } g_0 \text{ then } R_0 \text{ endif} \\
\text{if not } g_0 \text{ and } g_1 \text{ then } R_1 \text{ endif} \\
\text{if not } g_0 \text{ and not } g_1 \text{ and } g_2 \text{ then } R_2 \text{ endif}
\end{align*}
\]

3.1.1 First Normal Form

An arbitrary abstract state program may be put in NF1

\[
\text{import } v_1, \ldots, v_k \\
R \\
\text{endimport}
\]

by rewriting

\[
\begin{align*}
\text{if } g_0 \text{ then} \\
\text{import } v \\
R \\
\text{endimport} \\
\text{endif}
\end{align*}
\]

as

\[
\begin{align*}
\text{import } v \\
\text{if } g_0 \text{ then} \\
R \\
\text{endif}
\end{align*}
\]

and rewriting

\[
\begin{align*}
\text{import } v \\
R \\
\text{endimport} \\
\text{as} \\
\text{import } v, w \\
R \\
S \\
\text{endimport}
\end{align*}
\]

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

\begin{verbatim}
import v_1, \ldots, v_k
  if g_1 then R_1
  \ldots
  if g_n then R_n
endimport
\end{verbatim}

where \(R_1, \ldots, R_n\) are updates. We may translate programs in NF1 to programs in NF2 by rewriting

\begin{verbatim}
if g then
  if h then as
    R
  endif
endif
\end{verbatim}

and rewriting

\begin{verbatim}
if g then R_1, \ldots, R_n endif as
\end{verbatim}

\begin{verbatim}
if g then R_1 endif
if g then R_2 endif
\ldots
if g then R_n endif
\end{verbatim}

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

\begin{verbatim}
import v_1, \ldots, v_k
  if g_1 then R_1 endif
  \ldots
  if g_i then R_i endif
    R_{i+1}
  \ldots
    R_{i+m}
endimport
\end{verbatim}

where \(R_1, \ldots, R_{i+m}\) are updates. We then rewrite this as

\begin{verbatim}
import v_1, \ldots, v_k
  if g_1 then R_1 endif
  \ldots
  if g_i then R_i endif
    if true then R_{i+1}
  \ldots
\end{verbatim}
if true then $R_{+m}$
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

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

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

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 II, constructing its NF4 translate II' 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 $f'(\text{Pair}(t_1, \text{Pair}(t_2, \ldots, \text{Pair}(t_{r-1}, t_r)) \cdots))$. Thus, the abstract state machine II which has functions of arbitrary arity is simulated by the abstract state machine II' which has only unary functions, with the exception of the binary function Pair.

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

4 Simulating Schönhage machines by Unary Abstract State Machines

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

Before proving Theorem 1, we provide some necessary infrastructure, as well as the mapping $\phi$ 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. $B$ can be seen as a formalization of $A$ in the abstract state machine context. We begin by providing the vocabulary of $B$.  

8
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 Curlnst 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 δ : Nodes → 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 Curlnst is set to 0.
Output is undef; Bit(n) ∈ {0, 1} for all n ∈ 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 Curlnst). We first give the special case of Curlnst = 0.

if Curlnst = 0 and Mode = Initial then
  import y
  Nodes(y) := true
  Center := y
  δ₁(y) := y
  ...  
  δₘ(y) := y
  endimport
  Curlnst := 1
  Mode := Working
endif

Before the simulation proper begins, we import a Center node and set the pointer values appropriately,
assuming a pointer alphabet Δ = {δ₁, ..., δₘ}.

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 λ₀, λ₁
if CurlInst = i and Mode = Working then
    if Bit(InputString) = undef then
        CurlInst := Succ(CurlInst)
    elseif Bit(InputString) = 0 then
        CurlInst := Lambda0
    else CurlInst := Lambda1
end if
InputString := Succ(InputString)
end if

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 λ₀ and λ₁.

output β

if CurlInst = i and Mode = Working then
    Output := Beta
    CurlInst := Succ(CurlInst)
end if

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

if CurlInst ≠ i₁ and ... and CurlInst ≠ iₖ then
    if Output ≠ undef then
        Output := undef
    endif
end if

Suppose i₁, ..., iₖ 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 CurlInst = i and Mode = Working then
    CurlInst := Lambda
end if

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 Curlnst = 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 α₁α₂...αₙβ

if Curlnst = i and Mode = Working then
    import y
    Nodes(y) := True
    β(PARENT) := y
    δ₁(y) := δ₁(PARENT)
    ...
    δₘ(y) := δₘ(PARENT)
endimport
    Curlnst := Succ(Curlnst)
endif

PARENT abbreviates Centerα₁...αₙ.

To add a new element, we import element y from Reserve into Nodes, and then 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 δ ∈ Δ
    Neighbor(δ, y) := PARENT
endvar

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

set v₁v₂...vₖ to v₁v₂...vₗ

if Curlnst = i and Mode = Working then
    wₖ(Center.w₁.w₂...wₖ₋₁) := Center.v₁.v₂...vₗ
    Curlnst := Succ(Curlnst)
endif
The set instruction changes function \( w_k \) at the point \( w_{k-1}(w_{k-2}(\ldots(w_1(\text{Center})\ldots)\ldots) \) to point to the element designated by \( v_j(v_{j-1}(v_{j-2}(\ldots v_1(\text{Center})\ldots)) \).

\[
\text{if } u_1u_2\ldots u_k = v_1v_2\ldots v_j \text{ then } \sigma
\]

\[
\text{if } \text{CurlInst} = i \text{ and Mode} = \text{Working} \text{ then}
\]
\[
\text{if } u_1u_2\ldots u_k = v_1v_2\ldots v_j \text{ then }
\]
\[
R_\sigma
\]
\[
\text{else }
\text{CurlInst} := \text{Succ(CurlInst)}
\]
\[
\text{endif}
\]
\[
\text{endif}
\]

\[
\text{if } u_1u_2\ldots u_k \neq v_1v_2\ldots v_j \text{ then } \sigma
\]

\[
\text{if } \text{CurlInst} = i \text{ and Mode} = \text{Working} \text{ then}
\]
\[
\text{if } u_1u_2\ldots u_k \neq v_1v_2\ldots v_j \text{ then }
\]
\[
R_\sigma
\]
\[
\text{else }
\text{CurlInst} := \text{Succ(CurlInst)}
\]
\[
\text{endif}
\]
\[
\text{endif}
\]

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

### 4.2 State Mapping

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

1. If the string \( w_1 \ldots w_k \) designates an element \( x \) in \( A \) then in \( \phi(A) \text{ Center}.w_1 \ldots w_k \) evaluates to \( x \).
2. If \( k \) is the index of the current instruction in \( A \), then \( \text{CurlInst} \) evaluates to \( k \) in \( \phi(A) \).
3. If \( A \) contains input \( x \), then \( \phi(A) \) contains input \( x \).
4. If the bit \( \beta \) is emitted during the transition from state \( A \), \( \beta \) is emitted during the transition from \( \phi(A) \). Otherwise no output is emitted.

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

This creates a natural one-to-one mapping of elements of any state \( A \) of \( A \) to elements of the corresponding state of \( B \). 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 $A$. Let $B$ be the abstract state machine translate of $A$. Let $B_0, B_1, \ldots$ be a run of $B$ 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 $A$. 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** $\lambda_0, \lambda_1$. 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 Curlnst appropriately.

- **output** $\beta$. This causes the bit $\beta$ to be emitted, and control shifts to the next instruction in the program. In $A$, the output bit is set, then control is incremented.

- **goto** $\lambda$. This causes control to shift to the statement labeled $\lambda$. In $A$, Curlnst is set to the index of the appropriate instruction.

- **halt**. This stops execution. In $A$, Mode is set to Final, which prevents any transitions from firing.

- **new** $W$ (where $W = w_1, \ldots, w_k$). This instruction makes three changes to the $\Delta$-structure. First, a new node, $y$, is added to the structure. Then the pointer labeled $w_k$ from the node reached by following $w_1, \ldots, w_{k-1}$ from $a$ is directed to $y$. Finally, all pointers $\delta$ from $y$ are directed to the node formerly reached by $w_1 \ldots w_k$. In $A$, the import constructor brings a new element $y$ into the universe of Nodes, the function $w_k$ is updated, and $\delta(y)$ for every $\delta$ is set to the former value of $\delta(Center, w_1 \ldots w_k)$. And, of course, control in both the Schönhage program and abstract state machine program is incremented.

- **set** $W$ to $V$. This causes the pointer $w_k$ from the node $w_1, \ldots, w_{k-1}$ to be redirected to the node $v_1, \ldots, v_j$, and control is incremented. In $A$, the update

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

is fired, and Curlnst is updated.

- **if** $U = [\#]V$ then $\sigma$. These two statements test the nodes found by traversing $U$ and $V$; if they are equal [not equal], then $\sigma$ is executed; otherwise control passes to the next instruction. In $A$, the guard testing the equality of the two terms represented by $U$ and $V$ is evaluated; if it is true, then $R_\sigma$ is executed; otherwise Curlnst is incremented.

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

From Lemma 3 we may conclude that the given $\phi$ 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 abstract state machine $A$ there exists a Schönhage machine $B$ that strictly lock-step simulates $A$.

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 \( \Delta \)-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 \texttt{True} and \texttt{False}. The center node itself corresponds to \texttt{undef} in the abstract state machine. We also add a finite number of additional nodes \( \texttt{New}_1, \ldots, \texttt{New}_k \) to be used in simulating importing elements 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 \texttt{InputPositions}, and \texttt{Output} = \texttt{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 \( \Delta \)-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 \( f_0, \ldots, f_k \), \( f_0f_1 \ldots f_k \) and \( f_0f_1 \ldots f_k' \) designate \( x \) in the \( \Delta \)-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 \( f_k(f_0, f_1 \ldots f_{k-1}) := g_0, g_1 \ldots g_k \), 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_0f_1 \ldots f_k' \texttt{ to } g_0g_1 \ldots g_k \texttt{ (abbreviated } \texttt{set } F' \texttt{ to } G \texttt{ where clear from context)}, where \( f_i \) and \( g_i \) are the Schönhage machine pointers corresponding to the appropriate abstract state machine functions, and \( f_k' \) is the shadow of \( f_k \). We then later include code that copies the relevant values of \( f' \) to \( f \).
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

\[
\text{import } v_1 \ldots v_k \\
R \\
\text{end import}
\]

This is translated into the Schönhage program fragment

\[
\text{new } \text{New}_1 \\
\vdots \\
\text{new } \text{New}_k \\
R'
\]

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

5.1.3 Conditional Constructors and Guards

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

\[
\text{if } g \text{ then } R
\]

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

If \( g \) is a simple (i.e. containing no logical connectives) Boolean term, then it has the form \( f(t) \), where \( f \) is a relation and \( t \) is a term composed of unary functions. We simply translate \( g \) as \( F = \text{True} \), where, as before, \( F \) is the is the word corresponding to the Boolean term. Thus, we translate

\[
\text{if } g \text{ then } R \quad \text{as} \quad \text{if } F = \text{True} \text{ then goto } \mathcal{L} \\
\quad \text{goto } \mathcal{L}' \\
\quad \mathcal{L} : R' \\
\quad \mathcal{L}' : 
\]

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

If \( g \) is not a simple Boolean term, then it has one of the forms \( g_a \) and \( g_b, g_a \) or \( g_b \), or \( \text{not } g_a \).

If we have a constructor of the form

\[
\text{if } g_a \text{ and } g_b \text{ then } R_i \quad \text{we replace it by} \quad \text{if } g'_a \text{ then goto } \mathcal{L}_a \\
\quad \text{goto } \mathcal{L}_c \\
\quad \mathcal{L}_a : \text{if } g'_b \text{ then goto } \mathcal{L}_b \\
\quad \text{goto } \mathcal{L}_c \\
\quad \mathcal{L}_b : R'_i \\
\quad \mathcal{L}_c : 
\]
where $g'_a$ and $g'_b$ are the abstract state machine translations of $g_a$ and $g_b$.

Similarly, if we have a constructor of the form

\[
\begin{align*}
\text{if } g_a \text{ or } g_b \text{ then } \sigma & \quad \text{we replace it by} & \\
\text{if } g'_a \text{ then goto } \mathcal{L}_T & & \\
\text{if } g'_b \text{ then goto } \mathcal{L}_T & & \\
& \quad \text{go to } \mathcal{L}_F & \\
\mathcal{L}_T : R'_i & & \\
\mathcal{L}_F : & & 
\end{align*}
\]

And if we have a constructor of the form

\[
\begin{align*}
\text{if not } g_a \text{ then } R_i & \quad \text{we replace it by} & \\
\text{if } g'_a \text{ then goto } \mathcal{L}_F & & \\
& \quad \text{go to } \mathcal{L}_F & \\
\mathcal{L}_F : & & 
\end{align*}
\]

5.1.4 Programs

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

\[
\begin{align*}
\mathcal{L} : \text{if Mode = Final halt} & & \\
& \quad \text{UPDATE} & \\
& \quad \text{go to } \mathcal{L} & \\
\end{align*}
\]

where $\text{UPDATE}$ is a sequence of set instructions of the form set $f_0 f_1 \ldots f_k$ to $f'_0 f'_1 \ldots f'_k$, with one such instruction for every $f_0 f_1 \ldots f_k$ that appears in $B'$. 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 $\phi$ 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 $f_0 f_1 \ldots f_k$ 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 $\beta$ is emitted between states $A_{i-1}$ and $A_i$, then exactly $\beta$ is emitted between states $\phi(A_{i-1})$ and $\phi(A_i)$. Otherwise no output is emitted between $\phi(A_{i-1})$ and $\phi(A_i)$.

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

16
5.3 Proof of Theorem 2

Lemma 4. Let $A_0, A_1, \ldots$ be a run of an abstract state machine $A$. Let $B$ be the Schönhage machine translate of $A$. Let $B_0, B_1, \ldots$ be the run of $B$ 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 $B$. 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 $f(t):=t_0$ 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 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 import instruction: the instruction new $\text{new}_i$ creates a new node which is reached from the center node by following the $\text{new}_i$ pointer (earlier nodes designated by $\text{new}_i$ are chained from the newest node). Condition (2) is met by the pointer vocabulary: one $\text{New}_i$ 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 $\phi(A_i)$.

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 $\phi$ and a $J$ which maps indices of ASM states to indices of Schönhage restart states fulfill the definition of lock-step simulation, where $e$ is a finite number determined by the number of instructions in the program $B$. Thus Theorem 2 is proved.

6 Extended Schönhage Machines

We now discuss extending the Schönhage machine model by adding a pairing function.

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 \((X, Y)\) 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

\[
\text{create } (u_1 u_2 \ldots u_i, v_1 v_2 \ldots v_j)
\]

creates a new node that represents the ordered pair of values represented by \(U = u_1 \ldots u_i\) and \(V = v_1 \ldots v_j\). This node is denoted \((U, V)\). This command is similar to the new command: it brings a new node into the nodeset; additionally, it sets the pointers First and Second to point from node \((U, V)\) to nodes \(U, V\), respectively. If such a node \((U, V)\) already exists, \(\Delta\) is not altered.

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

\[
\begin{align*}
\text{create } (X, Y) \\
\text{create } ((X, Y), Z)
\end{align*}
\]

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

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 \(A\) there exists a sequential abstract state machine \(B\) which lock-step simulates \(A\).

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 \((u_1 \ldots u_i, v_1 \ldots v_j)\) can be simulated by the rule

\[
\begin{align*}
\text{if } \text{Pair} (\text{Center}.u_1 \ldots u_i, \text{Center}.v_1 \ldots v_j) &= \text{undef} \text{ then} \\
& \text{import } v \\
& \quad \text{Pair} (\text{Center}.u_1 \ldots u_i, \text{Center}.v_1 \ldots v_j) := v \\
& \quad \text{First}(v) := \text{Center}.u_1 \ldots u_i \\
& \quad \text{Second}(v) := \text{Center}.v_1 \ldots v_j \\
& \text{endimport}
\end{align*}
\]

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

**Proof.** Extend the definition of \(\phi\) from section 4.2:

5. If \((u_1 \ldots u_i, v_1 \ldots v_j) = x\) in state \(A_i\), then \(\text{Pair} (\text{Center}.u_1 \ldots u_j, \text{Center}.v_1 \ldots v_j) = x\) in \(\phi(A_i)\)

The bulk of this proof is found in the proof of Lemma 3; we need simply augment it to address the create statement.
create \((\mathcal{W}, V)\). This creates a new node and sets the \textbf{First} and \textbf{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. \(\square\)

6.3 Simulating Abstract State Machines by Extended Schönhage Machines

\textbf{Theorem 4.} For every sequential abstract state machine \(A\) there exists an extended Schönhage machine \(B\) which lock-step simulates \(A\).

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 \texttt{Pair} as in \S\ 3. We translate the abstract state machine program into a Schönhage program as described in section 5, replacing terms \texttt{Pair}(\(t_1, t_2\)) by \((T_1, T_2)\).

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

\textbf{Proof.} The proof is similar to the proof of Lemma 4. Extend the definition of \(\phi\) from section 5.2:

5. If \(\texttt{Pair}(\text{Center}. u_1 \ldots u_j, \text{Center}. v_1 \ldots v_j) = x\) in \(A_t\) then \(\langle u_1 \ldots u_i, v_1 \ldots v_j \rangle = x\) in \(\phi(A_t)\).

We assume that \(A\) is in NF3 as well and consider arbitrary updates as in Lemma 4. If a given update contains no terms of the form \(\texttt{Pair}(\vec{t}_1, \vec{t}_2)\) 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 \(\phi\) and our construction \((T_1, T_2) \in U\) and so the proof of Lemma 4 applies to updates involving these elements as well.

\textbf{Proof.} This follows directly from Lemmata 4 and 5. \(\square\)

References


