The Invariance Thesis

Nachum Dershowitz and Evgenia Falkovich
School of Computer Science, Tel Aviv University, Ramat Aviv, Israel
Email: nachum.dershowitz@cs.tau.ac.il, jenny.falkovich@gmail.com

The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency.

—William Henry Gates III

***Abstract*—We demonstrate that the programs of any classical (sequential, non-interactive) computation model or programming language that satisfies natural postulates of effectiveness (which specialize Gurevich's Sequential Postulates)—regardless of the data structures it employs—can be simulated by a random access machine (RAM) with only constant factor overhead. In essence, the effectiveness postulates assert the following: states can be represented as logical structures; transitions depend on a fixed finite set of terms (those referred to in the algorithm); basic operations can be programmed from constructors; and transitions commute with isomorphisms. Complexity for any domain is measured in terms of constructor operations. It follows that algorithmic lower bounds for the RAM model hold (up to a constant factor determined by the algorithm in question) for any and all effective classical models of computation, whatever its control structures and data structures. This substantiates the Invariance Thesis (a polynomial-time version of the "extended" Church-Turing Thesis), namely that every effective classical algorithm can be polynomially simulated by a Turing machine.**

I. Introduction

In 1936, Turing [27] invented a theoretical computational model, the Turing machines, and proved that they compute exactly the same functions over the natural numbers (appropriately represented) as do the partial-recursive functions and the lambda calculus. His deep insight was that computation, however complex, can be decomposed into simple atomic steps, consisting of single-step motions and the testing and writing of individual symbols. In 1971, Hartmanis [17] and Cook and Reckhow [5] developed the random-access register machine (RAM) model for the purpose of measuring computational complexity of computer algorithms. This theoretical model is close in spirit to the design of modern (von Neumann architecture) computers and serves as a more realistic measure of (asymptotic) time and space resource usage than do Turing's machines. The question addressed here is to what extent RAMs are in fact the ideal model for measuring algorithmic complexity.

In his handbook survey on computational models, van Emde Boas writes:

> Register-based machines have become the standard machine model for the analysis of concrete algorithms. [29, p. 22]

This work was carried out in partial fulfillment of the requirements for the Ph.D. degree of the second author.

> If it can be shown that reasonable machines simulate each other within polynomial-time bounded overhead, it follows that the particular choice of a model in the definition of feasibility is irrelevant, as long as one remains within the realm of reasonable machine models. [29, p. 4]
>
> I firmly believe that complexity theory, as presently practiced, is based on the following assumption, held to be self evident:
>
> **Invariance Thesis**: There exists a standard class of machine models, which includes among others all variants of Turing Machines [and] all variants of RAM's.... Machine models in this class simulate each other with Polynomially bounded overhead in time, and constant factor overhead in space. [28, p. 2] (cf. [29, p. 5])

Indeed, it is widely believed that all effective classical (that is, deterministic, non-parallel, non-analog, non-interactive) models are polynomially-equivalent with regard to the number of steps required to compute. For example, it is well-known that multitape Turing machines (TMs) require quadratic time to simulate RAMs [5] and that single-tape Turing machines require quadratic time to simulate multitape ones [18]. It remains conceivable, however, that there exists some sort of model that is more sophisticated than RAMs, one that allows for even more time-wise efficient algorithms, yet ought still be considered "reasonable".

As will be shown, RAMs are not just polynomially as good as any reasonable alternative—as suggested by the Invariance Thesis; rather, RAMs actually provide *optimal* complexity, regardless of what control structures are available in the programming model and what data structures are employed by the algorithm. Specifically, we show that any algorithm of *any* "effective" computation model (or programming language) can be simulated with only constant-factor slowdown by a RAM equipped with addition and subtraction. For this, we will need a formal, domain-independent definition of effectiveness.

As we will be counting RAM operations on an *arithmetic* model, we need to bear in mind that each operation is applied to a natural number, which will be represented by a logarithmic number of bits. The complexity of the algorithm will be measured in terms of constructor operations for its data structures, as we'll see. This, then, is the precise statement of the provable thesis:

Theorem 1 (Time Invariance Theorem)**.** *Any effective algorithm using no more than $T(n)$ constructor/destructor/equality operations for inputs of size n can be simulated by an arithmetic RAM in order $T(n)$ steps, with a word size that may grow to order $\log \max\{T(n), n\}$ bits.*

Remark 1. *The* constant *multiplicative factor in this theorem and elsewhere in this paper depends on the algorithm being simulated, or—more precisely—on the maximum number of constructor operations performed in a single step. This is because an algorithm, in general, can perform any bounded number of operations in a single step.*

We proceed to substantiate the strong (constant-factor) invariance thesis in the following manner:

1) We require a generic, datatype-independent notion of algorithm for this claim. Accordingly, we subscribe to the axiomatic characterization of classical *algorithms* over *arbitrary* domains of [16], which posits that an algorithm is a transition system whose states can be any (logical) structure and for which there is some finite description of (isomorphism-respecting) transitions (Sect. II-A).
2) We are only concerned with effective algorithms—as opposed, say, to conceptual algorithms like Gaussian elimination over reals. Accordingly, we adopt the formalization of *effective* algorithms over arbitrary domains developed in [2], [9], [23], [3], insisting that initial states have a uniform finite description (Sect. II-B). This is the broadest class of models of computation for which the (classical) Church-Turing Thesis provably holds.
3) One does not normally want to treat complex operations like multiplication or list-reversal as unit cost. Analogous to single-cell Turing-machine operations and memory-cell access for RAMs, we propose to measure the complexity of algorithms over arbitrary effective data-structures in terms of the number of applications of constructors or destructors, equality tests, and stored-value lookups (Sect. III, Definition 3). A *basic* algorithm is one that employs only these basic operations (Sect. II-C).
4) Basic algorithms may be emulated step-for-step with a powerful extension of Schönhage's Storage Modification Machines (which we call EMMs) (see Sect. IV), extending results of [16] and [10] (Sect. V, Lemma 1). Furthermore, we show how each step of an EMM can be simulated by a constant number of RAM steps, operating on words of logarithmic size (Sect. V, Lemma 2).

Having agreed on the right way to measure complexity of effective algorithms over arbitrary data structures (item 3 above), we could have gone on to show directly how to simulate any basic, effective algorithm by means of multidimensional RAMs [24]. Instead, we choose to build (item 4) on the little-known result of [10], linking a version of EMMs with certain simple abstract state machines, which we will call GASMs. (This work long predates the formalization of effectiveness in [2], [9].) By showing that RAMs simulate EMMs efficiently (which is simpler than showing that they emulate RAMs) and that GASMs emulate any basic implementation of an effective algorithm, the four models (RAM $\geq$ EMM $\geq$ GASM $\geq$ effective algorithm) are chained together and the desired invariance result is obtained in a strong sense—without undue complications.

The simulation based on EMMs requires quadratic space. In earlier, preliminary work [7], we reported on a suboptimal (quadratic, rather than linear), but direct (albeit convoluted and inefficient), RAM simulation of effective algorithms. We use that method in Section VI, in combination with garbage collection, to reduce the space required by simulation. This latter version of the Invariance Thesis is precisely the original formulation, within linear overhead of space, but it comes at the expense of a quadratic, rather than linear, overhead of time.

In [12], the lambda calculus was extended with one-step reduction of primitive operations, and it was shown that any effective computational model can be simulated by this calculus with only constant factor overhead in time. The catch is—as the authors indicate—that individual steps can themselves be quite complex.

II. Background

We are interested in comparing the complexity of algorithms implemented in different effective models of computation, models that may take advantage of arbitrary data structures, not just numbers (as for recursive functions and RAMs) and strings (for Turing machines). For that, we need to formalize what an effective algorithm is in a *domain-independent* fashion. We do this in two parts: first, we present generic axioms for algorithms; second, we restrict attention to algorithms that may be deemed effective. We will summarize the rationale for these axioms so the reader may judge their generality for measuring the complexity of arbitrary effective sequential algorithms. With these requirements in hand, we can then show (in the following sections) how any effective algorithm can be simulated by a RAM.

Gurevich [15] formalized which features exactly characterize a classical algorithm in its most abstract and generic manifestation. The adjective "classical" is meant to clarify that in the current study we are leaving aside new-fangled forms of algorithm, such as probabilistic, parallel, or interactive ones. For detailed support of this axiomatic characterization of algorithms and relevant citations from the founders of computability theory, see [16], [9], [6]. Boker and Dershowitz [2] formalized an additional axiom, which allows one to restrict attention to the family of classical algorithms that are "effective".

A. Classical Algorithms

As Kleene [19, p. 493] explained:

> The notion of an "effective calculation procedure" or "algorithm" (for which I believe Church's thesis) involves its being possible to convey a complete description of the effective procedure or algorithm

by a finite communication, in advance of performing computations in accordance with it.

This desideratum is provided by the following postulate of algorithmicity of Gurevich [16]. The main intuition is that, for transitions to be effective, it must be possible to describe the effect of transitions in terms of the information in the current state.

Postulate 1. *An* algorithm *is posited to be a state-transition system comprising a set (or class) of* states *and a partial* transition *function from state to* next *state. States may be seen as (first-order) logical structures over some (finite) vocabulary, closed under isomorphism (of structures). Transitions preserve the domain (universe) of states, and, furthermore, isomorphic states are either both terminal (have no transition) or else their next states are isomorphic (via the same isomorphism). And transitions are governed by a finite, input-independent set of* critical *(ground) terms over the vocabulary such that, whenever two states assign the same values to those terms, either both are terminal or else whatever changes (in the interpretations of operators) a transition makes to one, it also makes to the other.*

States being structures, they include not only assignments for programming "variables", but also the "graph" of those functions that the algorithm can apply. We may view a state over F with domain (universe) D as storing a (presumably infinite) set of *location-value* pairs $f(a_1, \ldots, a_n) \mapsto b$, for all $f \in F$ and $a_1, \ldots, a_n \in D$ and some $b \in D$. So, by "changes", we mean $\tau(x) \setminus x$, where τ is the transition function, which gives the set of changed location-value pairs. We treat relations as truth-valued functions and treat all states, for now, as input states.

As in this study we are only interested in classical deterministic algorithms, transitions are functional. By "classical" we also mean to ignore interaction and unbounded parallelism. The above characterization excludes "hypercomputational" formalisms, such as [13], [22], in which the result of a computation—or the continuation of a computation—may depend on (the limit of) an infinite sequence of preceding (finite or infinitesimal) steps. Likewise, processes in which states evolve continuously (as in analog processes, like the position of a bouncing ball), rather than discretely, are eschewed.

States as structures make it possible to consider all data structures sans encodings. In this sense, algorithms are generic. The structures are "first-order" in syntax, though domains may include sequences, or sets, or other higher-order objects, in which case, the state would provide operations for dealing with those objects. Thus states with infinitary operations, like the supremum of infinitely many objects, are precluded. Closure under isomorphism ensures that the algorithm can operate on the chosen level of abstraction and that states' internal representation of data is invisible to the algorithm. This means that the behavior of an *algorithm*, as opposed to its "implementation" as an idealized C program, say, cannot depend on the memory address of some variable. If an algorithm does depend on such matters, then its full description must also include specifics of memory allocation. The critical terms are those locations in the current state named by the algorithm. Their finiteness precludes programs of infinite size (like an infinite table lookup) or which are input-dependent.

B. Effective Algorithms

For an algorithm to be effective, it must be possible, not only to describe transitions finitely, but also to fully describe its initial states, that starting subset of the algorithm's states containing input values. Only a state that can be described finitely may be deemed effective.

Postulate 2. *An algorithm is* effective *if all its initial states are effective (in the sense of Definition 1 below) and are all identical except for inputs.*

To handle inputs, we postulate some subset of the critical terms, namely the *input terms*, for which every possible combination of domain values occurs in some initial state, and such that all initial states agree on all terms over the vocabulary of the algorithm except these.

In general, an algorithm's domain might be uncountable—as in Gaussian elimination over the reals, but, when we speak of "effective" algorithms, we are only interested in that countable part of the domain that can be described effectively. Thus, we may as well restrict our discussion to countable domains and assume that every domain element can be described by a term in the algebra of the states of the algorithm. Furthermore, a state's operations could easily require an infinite table lookup. Thus, the initial state of an algorithm may contain ineffective infinite information, in which case the algorithm could not be deemed effective, so we need to place finiteness restrictions on the initial states of algorithms. Another problem is that the same domain element might be accessible via several terms, generating non-trivial relations, which might hide non-computable information.

Several ways of overcoming these potential problems and capturing the notion that initial states have a finite description, thereby characterizing effectiveness, have been suggested. One alternative [23] characterizes an effective (initial) state as one for which there is a (semi-) decision procedure for equality of terms in the state. That is, there is Turing machine for determining whether a state interprets two terms (given as strings) as the same domain value. A second alternative [9] requires that there exist an (arbitrary) injection from the chosen domain of the algorithm into the natural numbers such that the given base functions (in initial states) are all tracked (under that injection) by partial-recursive functions. These two approaches are somewhat circular: the first relies on Turing-machine computability and the second on recursive functions.

A more objective approach [2] takes its cue from constructor domains. Constructors provide a way to give a unique name for any domain element, and the domain can be identified with the Herbrand universe (free-term algebra) over constructors. Destructors provide an inverse operation for constructors. For every constructor c of arity k, we may have destructors $c_1, \ldots, c_k$ to extract each of its arguments [$c_i(c(x_1, \ldots, x_i, \ldots, x_k)) =$

x], plus c_0, which returns an indicator that the root constructor (of a value) is c. Constructors and destructors are the usual way of thinking of domain values of effective computational models. For example, strings over an alphabet {a,b,...} are constructed from a scalar (nullary) constructor $\varepsilon()$ and unary constructors a$(\cdot)$, b$(\cdot)$, while destructors may read and remove the last letter. Natural numbers in unary (tally) notation are normally constructed from (unary) successor and (scalar) zero, with predecessor as destructor. The positive integers in binary notation are constructed out of (the scalar) ε and (unary) digits 0 and 1, with the constructed string understood as the binary number obtained by prepending the digit 1. The destructors are the last-digit and but-last-digit operations. For Lisp's nested lists (s-expressions): the constructors are (scalar) `nil` (the empty list) and (binary) `cons` (which adds an element to the head of a list); the destructors are `car` (first element of list) and `cdr` (rest of list). To construct 0-1-2 trees, we would have three constructors, $A()$, $B(\cdot)$, and $C(\cdot,\cdot)$, for nodes of out-degree 0 (leaves), 1 (unary), and 2 (binary), respectively. Destructors may choose a child subtree, and also return the degree of the last-added (root) node.

We may assume that domains include two distinct truth values and another distinct default value, and—furthermore—that we have (scalar) constructors, TRUE, FALSE, and UNDEF, for all three. Boolean operations are effective finite tables, so we may presume them.

Definition 1 (Effective State [2])**.** *A state is* effective *if its domain is isomorphic to a free constructor algebra and its operations all fall into one of the following categories: those free constructors and their corresponding destructors and equality; (infinitely) defined operations that can themselves be computed effectively with those same constructors (perhaps using a richer vocabulary); and finitely many other defined location-values (not* UNDEF*).*

So, initial states may include constructor operations, which are certainly effective (they are just a naming convention for domain values). Given free constructors, equality of domain values and destructors are also effective (see [2]). Obviously, an initial state should also be allowed to include some input. Without loss of effectiveness, we can allow any finite amount of non-trivial data, provided that—except for the input values—all initial states are the same. Otherwise, initial states could hide uncomputable outputs. Moreover, initial states can have effective operations. The circularity of this definition of effectiveness "bottoms-out" with operations that are programmable directly from the constructors. We are presuming that constructors are present in states—even if the algorithm avoids their direct use.

As all three characterizations of effectiveness have been shown to lead to one and the same class of effective functions (up to isomorphism) for any computational model over any domain [3], it is credible that the intended generic notion of effectiveness has indeed been captured.

C. Basic Algorithms

Definition 2. *An effective algorithm is* basic *if its initial states have no infinitely-defined operations.*

Basic algorithms are clearly effective, since they operate over finite data only. But they are not expressive enough to emulate all effective functions step-for-step, since the latter may have direct access to bigger operations, such as multiplication. Therefore, we have allowed an effective state to be equipped with "effective oracles", which can be obtained by bootstrapping from a basic algorithm with the chosen constructors. Still, when measuring time complexity, we will want to charge more than unit cost for such programmed operations. To get that, we take advantage of the fact that every effective algorithm can have its defined operations "inlined", yielding a basic algorithm.

Finiteness of critical terms, together with commutativity with isomorphism, guarantees that only finitely many locations can be affected by one transition [16, Lemma 6.1]. That assures that, whenever a state is effective, so is the next state (when there is a next state). This justifies our definition of an effective *algorithm* as having effective *initial* states.

III. Measuring Complexity

The common approach measures (asymptotic) complexity as the (maximum) number of operations relative to input size. As we want to count atomic operations, not arbitrarily complex operations, we should count constructor operations. So we have a choice: to count all the operations executed by an effective algorithm, or to count the transition steps of its corresponding basic algorithm. We take the latter route. To measure the time needed for the execution of a basic algorithm, we use—for the time being—the "uniform measure" [29, pp. 10–11], under which every transition is counted as a one time unit. Later, we will address the question of what cost to assign to each transition step.

To handle arbitrary data types, the only sensible and honest way is to define the size of a domain element to be the number of basic operations required to build it:

Definition 3 (Size)**.** *The* size *of a domain element is the minimal number of constructor operations required to name that value.*

The size $|n|$ of a unary number n, represented as $s^n(0)$, is $n+1$. The size of n in binary is $\lceil \lg n \rceil$; for example, $|5| = 3$, the length of $0(1(\varepsilon))$, the initial 1(for the string 101) being understood. The size of Turing-machine strings is (one more than) the length of its tape, since string constructors are unary (see the basic Turing-machine implementation in [2]). The size of the tree $C(B(A(),A()),B(A(),A()),B(A(),A()))$ is only 3, because subtrees can be reused, and the whole tree can be specified by

$$C(s,s,s) \text{ WHERE } s = B(r,r), r = A()\,.$$

An effective algorithm is allowed to access effective oracles (e.g. multiplication) in its initial states, which however are

required to be programmable (i.e. algorithmically describable) by a basic algorithm, that is, using constructors and destructors only (usually with a larger vocabulary). In other words, by bootstrapping an effective algorithm, we get a basic one, which is the right one to consider for measuring complexity.

Definition 4 (Complexity)**.** *We measure the complexity of an effective algorithm by the number of basic operations (constructors, destructors, equality) required to perform the computation from initial to final states, relative to the input size.*

In other words, we inline effective sub-algorithms to get a basic one and measure the complexity of the latter.

Consider an effective algorithm `rev` to reverse the top-level elements of a Lisp-like list. The domain consists of all nested lists $\mathcal{L}$; that is, either an empty list $\langle\rangle$, or else a nonempty list of lists: $\langle\langle\rangle\rangle$, $\langle\langle\rangle\langle\rangle\rangle$, $\langle\langle\rangle\langle\rangle\langle\rangle\rangle$, ..., $\langle\langle\langle\rangle\rangle\rangle$, $\langle\langle\langle\rangle\rangle\langle\rangle\rangle$, ..., $\langle\langle\rangle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\rangle$, ... The function $\texttt{rev}\colon \mathcal{L} \to \mathcal{L}$ takes a list $\langle \ell_1 \dots \ell_n \rangle$ and returns $\langle \ell_n \dots \ell_1 \rangle$, with the sublists ℓ_j unchanged. For instance, $\texttt{rev}(\langle\langle\rangle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\rangle)$=$(\langle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\langle\rangle\rangle)$.

Now, `rev` could be a built-in operation of the Lisp model of computation, which in one fell swoop reverses any list. Clearly, constant cost for `rev` is not what is intended; we want to count the number of basic list operations needed to reverse a list of length n. So there is no escape but to take into account how `rev` is implemented internally. Suppose $\texttt{rev}(x)$ is effectively something like this:

$y := x;\ z := \texttt{nil}$
repeat
if $y = \texttt{nil}$
then return z
else [$z := \texttt{cons}(\texttt{car}(y), z);\ y := \texttt{cdr}(y)$]

We want to count the operations executed by this implementation, which is cn for some constant c that is the (maximum) number of (constructor and destructor) operations in a single iteration. Note that any straightforward Turing machine would require many more steps, quadratic in the *size* of the input x, rather than the number of elements at the top level, as in this list-based algorithm. In any RAM implementation, each list is represented by some natural number; what encoding is chosen is immaterial, as long as all operations perform consistently. Regardless of what number is used for the list $\ell = \langle\langle\rangle\langle\langle\rangle\langle\langle\rangle\rangle\rangle\rangle$, $\texttt{car}(\texttt{rev}(\texttt{car}(\texttt{rev}(\ell))))$ should return the number that represents $\langle\langle\rangle\rangle$.

IV. Machine Models

A. Random Access Machines

The RAM machine has access to a finite number of registers, and memory locations indexed by non-negative integers; each register or memory location can hold a non-negative integer. For the definition of RAMs, we take the set of instructions suggested by Cook and Reckhow in [5] and use the classification of RAM machines suggested by Boas in [29].

- For *basic RAMs*, the following operations are considered to take "unit time":
  1) $X \leftarrow C$, where X is a register and C is a constant.
  2) $X \leftarrow [Y]$, where $[Y]$ denotes the contents of the memory location indexed by Y.
  3) $[Y] \leftarrow X$.
  4) TRA m if $X > 0$: Transfer control to the m-th line of the program if $X > 0$.
- *Successor RAMs* are an extension of basic RAMs with successor/predecessor operations:
  5) INC X. Increase the value of register X by 1.
  6) DEC X. Decrease the value of register X by 1.
- *Arithmetic RAMs* are the model originally defined by Cook and Reckhow in [5]; they extend basic RAMs with addition and subtraction:
  5) $X \leftarrow Y + Z$.
  6) $X \leftarrow Y - Z$.
- *Multidimensional RAMs* are an extension of the classical ones, allowing for memory organization in multiple dimensions. The instruction set remains the same, but memory cells are accessed using one address per dimension.

Proposition 1 ([24])**.** *Multidimensional arrays may be organized in the memory of a one-dimensional arithmetic RAM in such a way that a program can access an entry indexed by $[i_1, \dots, i_\ell]$ in a constant number of steps, whether or not it is a first-time access (given that a unit instruction can operate over words of size $O(\log i_1 + \dots + \log i_\ell)$).*

B. Extended Storage Modification Machines

SMMs [25] manipulate a dynamic pointer structure (while reading an input string and writing to output). Their memory takes the form of a dynamic labeled (multi-) graph. Edges are labeled; nodes are named (not necessarily uniquely) by a path to them from a distinguished *focus* node. A machine may add new nodes to the structure and can redirect edges (perhaps rendering some nodes inaccessible in the process).

Let Λ be a finite alphabet of *direction* labels and X be a finite set of nodes with a distinguished *focus*. For each direction $\delta \in \Lambda$, there is a corresponding function over nodes, such that $\delta(x) = y$ exactly when there is a labelled edge $x \xrightarrow{\delta} y$.

For the convenience, we will denote by $\|W\|$ the end-node of path W.

The set of machine instructions, which may be labeled, is as follows:

- **goto** ℓ—Continue with instruction labeled ℓ.
- **new** W—Create a new node at the end of path W; if W is empty, then the new node becomes the focus; if $W = U\delta$ then the edge labeled δ from the node $\|U\|$ is redirected to a new node; all pointers from this new node are directed to the original $\|W\|$.
- $W := V$—Redirect the last edge of path W to point to the end node of path V.
- **if** $V = W$ **then** P—If paths V and W end at the same node, execute P.

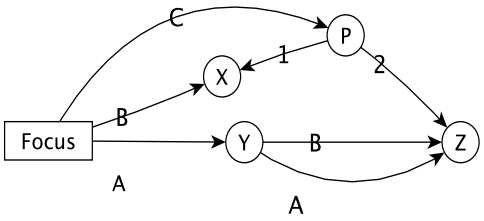
- **if** $V \neq W$ **then** P—If paths V and W end at distinct nodes, execute P.

For example, the instruction **if** AA $\neq$ AB **then** AA $:=$ B has the effect shown in Figure 1

We extend the syntax of SMMs with an "inverse" operation, and refer to them as *EMMs*:

- **remember**$\langle W, V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ will remember node $\|W\|$ as the one with edges of type δ_i pointing to nodes $\|V_i\|$, for $i = 1, \ldots, k$. In case of collision with previous applications of **remember**, previously stored values are forgotten.
- **lookup**$\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$:
  - The machine will set an edge labeled $\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ to point from the focus to a node X if X was the last one remembered as the node with edges δ_i pointing to nodes $\|V_i\|$.
  - If there is no appropriate node remembered by the machine, then an edge labeled $\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ outgoing from the focus will be removed, if such exists.

For example, performing **remember**$\langle C, B, AA, 1, 2 \rangle$ on the topology



the machine will remember the node P as the one with edge of type 1 outgoing to node X and edge of type 2 outgoing to Z. Assume the edges labeled C and 1 have been removed and we request **lookup**$\langle B, AB, 1, 2 \rangle$. The outcome is shown in Figure 2

Note that, despite the fact that P is not reachable from the focus and that its outgoing edges were changed, the machine still remembers it as the one that satisfies the requirement of **lookup**, since it was the last one remembered as such. The set of edge labels is fixed for any one program, even though these compound labels may be nested.

V. RAM Simulation of Basic Algorithms

As explained in Sect. III, we should measure the complexity of effective algorithms in terms of *basic* operations. Thus, we need to show how RAMs simulate basic algorithms. As an intermediary device, we make use of the extension of Schönhage's Storage Modification Machines, EMMs, described in the previous section. We prove that constructor-based algorithms can be simulated by an EMM, which, in turn can be simulated by an arithmetic multidimensional RAM. And all this with negligible overhead. (One could avoid EMMs altogether and show directly how to simulate basic algorithms by multidimensional RAMs, but that would engender the expense of a great deal of technical detail.)

Lemma 1. *Any basic algorithm can be simulated by an EMM with at most constant factor overhead in the number of transitions.*

Proof: A similar claim was proved in [10, Lemma 1] for a different set of algorithms—we'll call them GASMs—and a different extension of SMMs. We prove that GASMs emulate our basic algorithms step-for-step and that our EMMs simulate GASMs with constant-factor overhead in the number of steps.

GASMs satisfy the axiom of algorithmicity, with an added ability to access (**import**) at a single transition a bounded number of fresh (as yet unused) elements. Without further restrictions on the domain, available oracles and **import** behavior, this class obviously contains also non-effective algorithms, like Euclidian geometry algorithms working over the space of reals or algorithms with access to a Turing halting oracle. Also unrestricted and thus unpredictable behavior of **import** cannot be considered effective. On the other hand, our basic algorithms access domain elements by invoking constructors. So a GASM emulating it will use the same vocabulary as the emulated basic algorithm, and each time a basic algorithm wants to access an element via constructors, a GASM will import a new domain element for that, if that is a first-time access.

It was proved in [10] that any GASM can be simulated for only constant-factor overhead by another GASM whose vocabulary has only nullary (scalar) and unary function symbols plus, optionally, a unique binary symbol used for ordered pairing of elements. The idea behind that is simple: a function of arity n is considered a unary function working over ordered n-tuples. Those n-tuples may be created by $n-1$ applications of pairings. Since the vocabulary of algorithms is finite and depends on algorithm only, the process requires only a bounded number of pairings. They prove that a GASM as above and without pairing can be simulated by a classical SMM. For simulation of pairing, they introduce an extension rule, **create**. An application of **create** V, W, for paths V, W, provides the machine with access to a node representing the ordered pair $\langle \|V\|, \|W\| \rangle$ ($\|V\|$ being the end node of V), in other words, a node with edges labeled `1st` and `2nd` pointing to nodes $\|V\|$ and $\|W\|$, respectively. A machine will use the existing node when possible or else will create a new one (and that despite the fact that the desired node might be inaccessible from the *focus*). To avoid nondeterminism, it is required that any call to pairing be via the **create** rule.

Obviously, **create** may be simulated by our **remember** and **lookup** extensions. Just replace each appearance of **create** by a formal program computing the following:

> Use **lookup**. If nothing is found, use **new** to create the desired node and then use **remember** on it.

Also, any change of edges outgoing that node should be **remember**ed (EMMs with our extension can, in general, **remember** any change it performs). ■

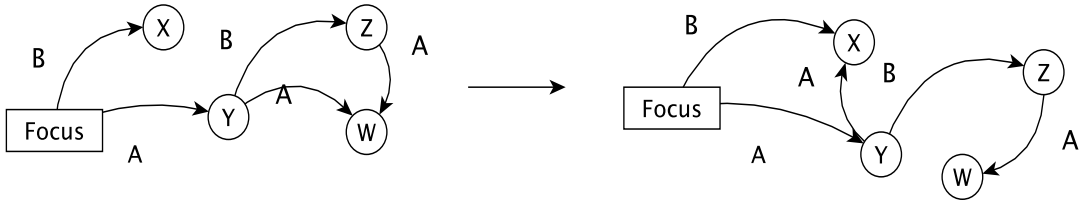


Fig. 1. The result of an assignment.

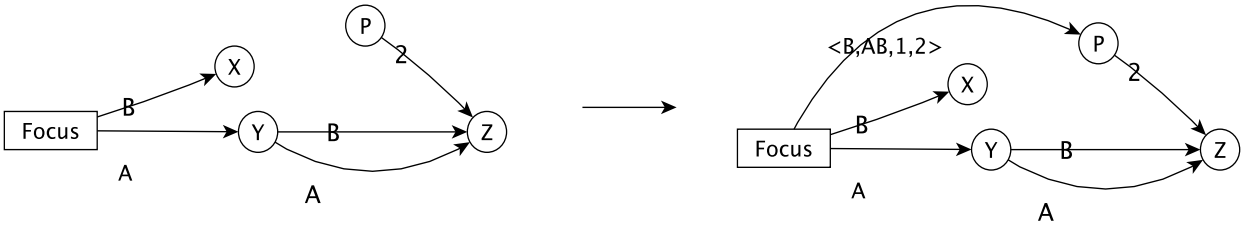


Fig. 2. The result of a lookup.

Lemma 2. *EMMs can be simulated by multidimensional successor RAMs with at most constant factor overhead in the number of transitions.*

Proof: We give each node $x \in X$ a unique integer identifier $\widehat{x}$. When a new node is created, its identifier will be the successor of the largest previously used integer. In this way, a graph with n nodes uses identifiers $1, \ldots, n$. An identifier for a new node is created using successor.

For each $\delta \in \Lambda$ we define an array, also named δ, and write $\delta[i] = j$ if the contents of the ith entry of array δ is j. (All the arrays can be stored together in one multidimensional array.) To simulate the state of an SMM, these arrays will have the following property:

> For all nodes $x, y \in X$, we have $\delta[\widehat{x}] = \widehat{y}$ if and only if $x \xrightarrow{\delta} y$.

A constant focus will contain the identifier of the focus. With this construction, one can find the end point of an edge out of node x labeled δ via a simple query for the value of $\delta[\widehat{x}]$. This can be done in one RAM operation. And an end node of a path of length k can be found in k RAM operations.

Since an SMM program is described finitely, the paths it may query have length bounded by the length of this description. We may conclude from this that, whenever a RAM needs to investigate a path, this path has bounded length and thus the investigation can be performed in a bounded number of RAM operations. It is easy to see that any SMM instruction can be performed using only a bounded number of RAM instructions, corresponding to following and updating edges in the graph.

The extended instruction of EMMs, however, requires the ability to compute the source node of a path. To simulate this, we will have to extend our construction as well. Let $\Lambda = \{\delta_1, \ldots, \delta_k\}$. We define a k-dimensional array A and utilize it to simulate the extension in the following way:

- An application of **remember**$\langle W, V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ will be simulated by $A\left[\widehat{\|V_1\|}, \ldots, \widehat{\|V_k\|}\right] := \widehat{\|W\|}$. If an edge labeled δ_i is missing in the description of a command, then the array index will be 0.
- The application of **lookup**$\langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ will therefore be (remember that an empty path stands for the *focus* node): $\delta\left[\widehat{\|\epsilon\|}\right] := A\left[\widehat{\|V_1\|}, \ldots, \widehat{\|V_k\|}\right]$, with $\delta = \langle V_1, \ldots, V_k, \delta_1, \ldots, \delta_k \rangle$ and with 0 for missing labels.

All the above operations use only assignments and comparisons and may be implemented with only basic RAM commands. A multidimensional RAM with multiple arrays can be easily implemented using one array of large enough dimension. ■

Corollary 1. *Any basic algorithm can be simulated by a multidimensional successor RAM with only a constant factor in the number of transitions.*

Proof: To measure the complexity of effective algorithm we "bootstrap" it to a basic one, and then measure the complexity of the latter (see Definition 4). It follows from Lemma 1 that any basic algorithm may be simulated by an EMM, which, by Lemma 2, is doable with a multidimensional successor RAM. ■

Everything is in place now to prove our primary result, the Invariance Theorem, which states that every basic algorithm can be simulated by a RAM with the same order of number of steps.

Proof of Theorem 1: It follows from Corollary 1 that any basic algorithm may be simulated by a multidimensional successor RAM, which, in turn, can be simulated by an ordinary arithmetic RAM, by Proposition 1.

Until now, we charged one time unit per transition step, be it a basic algorithm, an EMM, or a RAM. If we desire to

count basic operations—constructors and destructors—which is natural for basic algorithms, the overhead will be only a constant factor, since each transition of a basic algorithm may be fully described by a bounded number of invocations of basic operations, where the bound depends on the algorithm only (the maximum number of certain operations) and not on the inputs.

The numbers manipulated by the RAM can grow proportionately to the length of the computation, because there are a bounded number of new domain elements introduced in any one step. So the word length of the RAM is logarithmic in the computation length, plus the length of input if the algorithm is sublinear. ■

For a RAM, one might, in fact, wish to charge according to the length of the numbers stored in its arrays [5]. For basic algorithms, counting lookup of values of defined functions and testing equality of domain elements of arbitrary size as atomic operations may also be considered unrealistic. Just as it is common to charge a logarithmic cost for memory access ($\lg x$ to access $f(x)$), it would make sense to place a logarithmic charge $\lg x + \lg y$ on an equality test $x = y$.

Corollary 2. *Basic algorithms (that is, algorithms with basic initial states) and (ordinary) storage modification machines (SMMs) simulate each other to within a constant factor.*

Proof: This follows from the fact that an arithmetic RAM is equivalent time-wise to an SMM, up to a constant factor [25], and that basic algorithms can easily emulate RAMs. ■

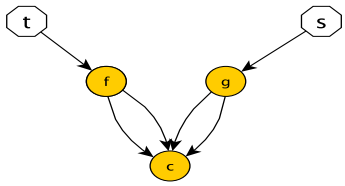
VI. Less Space but More Time

Let $X_1 \rightsquigarrow \cdots \rightsquigarrow X_n$ be a run of a basic algorithm and let a be an element in domain of this run. We say that a is *active* at X_i, for some i, if there is a critical term t whose value over X_i is a; that is *activated* at X_i if there is some $j \leq i$ such that a is active at X_j; and that it is *accessible* at X_i if it can be obtained by a finite sequence of constructor operations on active elements. It was demonstrated in [23] that an element that is accessible at X_i may become inaccessible at X_{i+1}. Inaccessible values should be recycled, much like a Turing machine does not preserve prior values of its tape. The *space usage* of X_i is the minimal number of constructor applications required to construct all active accessible values of X_i. The *space complexity* of a run is the maximal space usage among all states in the run.

For term t, we denote the minimal graph representing it by $G(t)$. Its nodes will each contain a small constant, indicating a vertex label or a pointer, corresponding to an edge in the graph. Note that, since $G(t)$ is minimal, it does not contain repeated factors.

To prevent repeated factors, not just in one term but in the whole state, we merge the individual term graphs (see [21]) into one big graph and call the resulting "jungle" of terms, a *tangle*. The tangle will be used to maintain the constructor-term values of all the critical terms of the algorithm. See [7].

Consider, for example, the natural way to merge terms $t = f(c, c)$ and $s = g(c, c)$, where c is a constant. The resulting directed acyclic graph G has three vertices, labeled f, g, and c. Two edges point from f to c and the other two from g to c. Our two terms may be represented as pointers to the appropriate vertex: $G(t)$ refers to the f vertex and $G(s)$ to g, where we are using the notation $G(t)$ to also refer to the vertex in G that represents the term t. The tangle looks like this:



On account of the above considerations, tangles are very convenient for distinguishing accessible elements from the inaccessible.

Theorem 2 (Space Invariance Theorem)**.** *Any effective algorithm with time complexity (constructor operations) $T(n)$ and space complexity $S(n)$ can be simulated by an arithmetic RAM in order $nT(n) + T(n)^2$ steps and order $S(n)$ space, with a word size that grows to order $\log S(n)$.*

Proof: In [7], we described how, as an intermediate device, one can simulate basic algorithms using tangles. It is pretty clear how to construct a tangle for a finite set of domain elements. A tangle that simulates state X_i, then, is a tangle for all activated elements of X_i, which we denote by $G(X_i)$. For each critical term t, we keep a pointer, called also t, which points to the value of t in $G(X_i)$. In [7, Thm. 16], it was shown that a basic algorithm with time complexity $T(n)$ can be simulated by a RAM that implements tangles with time complexity $nT(n) + T(n)^2$, using words of size $\log T(n)$.

To obtain the desired result, we need to show that the simulation can be performed with some sort of garbage collection so that each $G(X_i)$ is a tangle of only accessible elements of X_i. Since tangles are acyclic, all we need to do is to maintain a reference count for each node, incrementing it when a new pointer to the node is made and decrementing when a pointer is removed. Whenever the count goes to zero, the node may be added to the free list so that it can be recycled. Only when the free list is empty would a new node be allocated, upping the space usage.

Each of the $T(n)$ constructor operations now comes with a bounded number of additions and subtractions of reference counters by the arithmetic RAM. There is also the cost of collecting free nodes; indeed, it could be that almost all nodes are recycled in a single step. But amortized, this adds a small constant factor to the time spent by the RAM, since for each creation of a node, of which there are at most $T(n)+n$, there can be at most one freeing up of it. The space overhead is one counter per node, which may double the required space. ■

VII. Discussion

We have shown (Theorem 1) that any algorithm running on any effective classical model of computation can always be simulated by an arithmetic RAM with minimal (linear) overhead and with words of at most logarithmic size, counting constructor operations for effective algorithms. So lower bounds for the RAM model are (up to a constant factor) lower bounds in an *absolute* sense. We have also seen (Theorem 2) that space complexity may be preserved with only a quadratic increase in time.

It follows that to outperform any RAM, an alternative model must violate one of our postulates. This can be for a number of reasons:

- It is not a discrete-time state-transition system—examples include various analog models of computation, like Shannon's General Purpose Analog Computer (GPAC) [26]. See the discussion in [30].
- States cannot be finitely represented as first-order logical structures, all over the same vocabulary—for example models allowing infinitary operations, like the supremum of infinitely many objects.
- The number of updates produced by a single transition is not universally bounded—examples are parallel and distributive models (and probably quantum algorithms, as is widely believed).
- It has a "non-effective" domain—for example, continuous-space algorithms, as in Euclidian geometry or, alternatively, access to non-programmable oracles, like the halting function for Turing machines.

Summing up, any algorithm running on any effective model of computation can be simulated, independent of the problem, by a single-tape Turing machine with little more than cubic overhead in the number of operations: logarithmic for the RAM simulation of the algorithm and cubic for a Turing machine simulation of the RAM [5]. It follows—as has been conjectured and is widely believed—that every effective algorithm, regardless of what data structures it uses, can be simulated by a Turing machine, with at most polynomial overhead in time complexity. This claim is van Emde Boas's *Invariance Thesis*, also known as the *Extended Church-Turing Thesis* [1]. Every model meeting the natural axioms posed here is "reasonable" and belongs to van Emde Boas's "standard class".

The result herein does not cover large-scale parallel computation, such as quantum computation, as we have posited that there is a fixed bound on the degree of parallelism, with the number of critical terms fixed by the algorithm. It is for this reason that Grover's quantum search algorithm, for instance, is outside its scope, though quantum algorithms have been formalized as abstract state machines in [14]. The question of relative complexity of parallel (and quantum) models is a subject of current research [8].

A physical version of the extended thesis has also been formulated:

> The Extended Church-Turing Thesis states… that time on all "reasonable" machine models is related by a polynomial. (Ian Parberry [20])
>
> The Extended Church-Turing Thesis makes the… assertion that the Turing machine model is also as efficient as any computing device can be. That is, if a function is computable by some hardware device in time $T(n)$ for input of size n, then it is computable by a Turing machine in time $(T(n))^k$ for some fixed k (dependent on the problem). (Andrew Yao [30])

This claim about the real physical world holds, of course, to the extent that physical models of computation adhere to the effectiveness postulates for discrete algorithms elaborated here (cf. [11]). But physical devices need not be discrete. Recent preliminary work on axiomatizing analog and hybrid algorithms, operating in continuous time, and perhaps involving physical components, is in [4].

References

[1] S. Aaronson, "Quantum computing since Democritus," Lecture notes, Fall 2006, available at http://www.scottaaronson.com/democritus/lec4.html (viewed December 31, 2013).

[2] U. Boker and N. Dershowitz, "The Church-Turing thesis over arbitrary domains," in *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, ser. Lecture Notes in Computer Science, A. Avron, N. Dershowitz, and A. Rabinovich, Eds. Berlin: Springer, 2008, vol. 4800, pp. 199–229, available at http://nachum.org/papers/ArbitraryDomains.pdf (viewed December 31, 2013).

[3] ——, "Three paths to effectiveness," in *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, ser. Lecture Notes in Computer Science, A. Blass, N. Dershowitz, and W. Reisig, Eds., vol. 6300. Berlin, Germany: Springer, Aug. 2010, pp. 36–47, available at http://nachum.org/papers/ThreePathsToEffectiveness.pdf (viewed December 31, 2013).

[4] O. Bournez, N. Dershowitz, and E. Falkovich, "Towards an axiomatization of simple analog algorithms," in *Proceedings of the 9th Annual Conference on Theory and Applications of Models of Computation (TAMC 2012, Beijing, China)*, ser. Lecture Notes in Computer Science, M. Agrawal, S. B. Cooper, and A. Li, Eds., vol. 7287. Berlin: Springer, May 2012, pp. 525–536, available at http://nachum.org/papers/SimpleAnalog.pdf (viewed December 31, 2013).

[5] S. A. Cook and R. A. Reckhow, "Time-bounded random access machines," *Journal of Computer Systems Science*, vol. 7, pp. 354–375, 1973, available at http://www.cs.utoronto.ca/~sacook/homepage/rams.pdf (viewed December 31, 2013).

[6] N. Dershowitz, "The generic model of computation," in *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2012, Zürich, Switzerland)*, ser. Electronic Proceedings in Theoretical Computer Science, 2012, pp. 59–71, available at http://nachum.org/papers/Generic.pdf (viewed December 31, 2013).

[7] N. Dershowitz and E. Falkovich, "A formalization and proof of the Extended Church-Turing Thesis (Extended abstract)," in *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011)*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 88, Zürich, Switzerland, Jul. 2011, pp. 72–78, available at http://nachum.org/papers/ECTT_EPTCS.pdf (viewed December 31, 2013).

[8] ——, "Generic parallel algorithms," Jan. 2014, submitted. Available at http://nachum.org/papers/GenericParallel.pdf (viewed January 20, 2014).

[9] N. Dershowitz and Y. Gurevich, "A natural axiomatization of computability and proof of Church's Thesis," *Bulletin of Symbolic Logic*, vol. 14, no. 3, pp. 299–350, Sep. 2008, available at http://nachum.org/papers/Church.pdf (viewed December 31, 2013).

[10] S. Dexter, P. Doyle, and Y. Gurevich, "Gurevich abstract state machines and Schönhage storage modification machines." *J. UCS*, vol. 1, pp. 279–303, 1997, available at http://jucs.org/jucs_3_4/gurevich_abstract_state_machines/Dexter_S.pdf (viewed December 31, 2013).

[11] G. Dowek, "Around the physical Church-Turing thesis: Cellular automata, formal languages, and the principles of quantum theory," in *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA 2012, A Coruña, Spain)*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer Verlag, 2012, vol. 7183, pp. 21–37.

[12] M. Ferbus-Zanda and S. Grigorieff, "ASMs and operational algorithmic completeness of lambda calculus," in *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, ser. Lecture Notes in Computer Science, A. Blass, N. Dershowitz, and W. Reisig, Eds. Berlin, Germany: Springer, Aug. 2010, vol. 6300, pp. 301–327, available at http://arxiv.org/pdf/1010.2597v1.pdf (viewed December 31, 2013).

[13] E. M. Gold, "Limiting recursion," *J. Symbolic Logic*, vol. 30, no. 1, pp. 28–48, 1965.

[14] E. Grädel and A. Nowack, "Quantum computing and abstract state machines," in *Proceedings of the 10th International Conference on Abstract State Machines: Advances in Theory and Practice (ASM '03; Taormina, Italy)*. Berlin: Springer, 2003, pp. 309–323, available at http://www.logic.rwth-aachen.de/pub/graedel/GrNo-asm03.ps (viewed December 31, 2013).

[15] Y. Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, Ed. Oxford: Oxford University Press, 1995, pp. 9–36, available at http://research.microsoft.com/~gurevich/opera/103.pdf (viewed December 31, 2013).

[16] ——, "Sequential abstract state machines capture sequential algorithms," *ACM Transactions on Computational Logic*, vol. 1, no. 1, pp. 77–111, Jul. 2000, available at http://research.microsoft.com/~gurevich/opera/141.pdf (viewed December 31, 2013).

[17] J. Hartmanis, "Computational complexity of random access stored program machines," *Mathematical Systems Theory*, vol. 5, pp. 232–245, 1971.

[18] F. E. Hennie and R. E. Stearns, "Two-way simulation of multitape Turing machines," *J. of the Association of Computing Machinery*, vol. 13, pp. 533–546, 1966.

[19] S. C. Kleene, "Reflections on Church's thesis," *Notre Dame Journal of Formal Logic*, vol. 28, no. 4, pp. 490–498, 1987.

[20] I. Parberry, "Parallel speedup of sequential machines: A defense of parallel computation thesis," *SIGACT News*, vol. 18, no. 1, pp. 54–67, March 1986.

[21] D. Plump, "Term graph rewriting," in *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds. World Scientific, 1999, vol. volume 2, ch. 1, pp. 3–61, available at http://www.informatik.uni-bremen.de/agbkb/lehre/rbs/texte/Termgraph-rewriting.pdf (viewed December 31, 2013).

[22] H. Putnam, "Trial and error predicates and the solution to a problem of Mostowski," *J. Symbolic Logic*, vol. 30, no. 1, pp. 49–57, 1965.

[23] W. Reisig, "The computable kernel of Abstract State Machines," *Theoretical Computer Science*, vol. 409, no. 1, pp. 126–136, Dec. 2008, draft available at http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2004_hub_tr177.pdf (viewed December 31, 2013).

[24] J. M. Robson, "Random access machines with multi-dimensional memories," *Information Processing Letters*, vol. 34, pp. 265–266, 1990.

[25] A. Schönhage, "Storage modification machines," *SIAM J. Computing*, vol. 9, pp. 490–508, 1980.

[26] C. Shannon, "Mathematical theory of the differential analyzer," *J. Math. Phys.*, vol. 20, pp. 337–354, 1941.

[27] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, pp. 230–265, 1937, corrections in vol. 43 (1937), pp. 544–546. Reprinted in M. Davis (ed.), *The Undecidable*, Raven Press, Hewlett, NY, 1965. Available at http://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf (viewed December 31, 2013).

[28] P. van Emde Boas, "Machine models and simulations (Revised version)," Institute for Language, Logic and Information, University of Amsterdam, Tech. Rep. CT-88-05, Aug. 1988, available as http://www.illc.uva.nl/Research/Reports/CT-1988-05.text.pdf (viewed December 31, 2013).

[29] ——, "Machine models and simulations," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Amsterdam: North-Holland, 1990, vol. A: Algorithms and Complexity, pp. 1–66.

[30] A. C. Yao, "Classical physics and the Church-Turing Thesis," *Journal of the ACM*, vol. 77, pp. 100–105, Jan. 2003, available at http://eccc.hpi-web.de/eccc-reports/2002/TR02-062/Paper.pdf (viewed December 31, 2013).