THE INVARIANCE THESIS

NACHUM DERSHOWITZ AND EVGENIA FALKOVICH-DERZHAVETZ

School of Computer Science, Tel Aviv University, Ramat Aviv, Israel
e-mail address: nachum@tau.ac.il

School of Computer Science, Tel Aviv University, Ramat Aviv, Israel
e-mail address: jenny.derzhavetz@gmail.com

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 postulates of algorithmicity and effectiveness 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); all atomic 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 any algorithmic lower bounds found for the RAM model also hold (up to a constant factor determined by the algorithm in question) for any and all effective classical models of computation, whatever their control structures and data structures. This substantiates the Invariance Thesis of van Emde Boas, namely that every effective classical algorithm can be polynomially simulated by a RAM. Specifically, we show that the overhead is only a linear factor in either time or space (and a constant factor in the other dimension).

> The enormous *number* of animals in the world depends of their varied structure & complexity: — hence as the forms became complicated, they opened *fresh* means of adding to their complexity.... If we begin with the simplest forms & suppose them to have changed, their very changes tend to give rise to others.
>
> — Charles Darwin, *Fourth Notebook on Transmutation of Species* (1839)

1. Introduction

In 1936, Turing [47] invented a theoretical computational model, the Turing machines, and proved that its programs compute exactly the same functions over the natural numbers (appropriately represented as strings) as do the partial-recursive functions (over Platonic

2012 ACM CCS: [**Theory of computation**]: Models of computation; Computational complexity.

Key words and phrases: Invariance Thesis, Church-Turing Thesis, Complexity, Random access machines, Abstract state machines, Pointer machines, Constructors.

This work was carried out in partial fulfillment of the requirements for the Ph.D. degree of the second author [18].

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

numbers) and the lambda calculus (over Church numerals). Church [9] had already postulated that recursive functions capture all of effective computation. Turing's (and Post's [38]) 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, thereby supporting the contention that effectiveness had been characterized. The belief that all effective computations can be carried out in this manner was called the *Church-Turing Thesis* by Kleene [31, p. 232].

The *Sequential Computation Thesis* [22, p. 96] goes further, relating—as it does—to the relative complexity of computations using different models:

> Fortunately, all reasonable sequential computers have related execution times. Not only can they all simulate each other, but the time losses associated with the simulations are not excessive. In fact, any algorithm which executes in polynomial time on one computer can be run in polynomial time on any other computer. Thus it makes sense to talk about polynomial time algorithms *independently* of any particular computer. A theory of feasible algorithms based on polynomial time is machine independent.
>
> The belief that all reasonable sequential computers which will ever be dreamed of have polynomially related execution times is called the *sequential computation thesis*. This thesis may be compared to the Church-Turing thesis.... It is a stronger form of that thesis, because it claims not only that the computable problems are the same for all computers, but also that the feasibly computable problems are the same for all computers.
>
> The major evidence for the sequential computation thesis is that it holds for all current computers and for all known reasonable definitions of sequential computation.

Indeed, it is widely believed that all (non-degenerate) effective classical "sequential" (that is, deterministic, non-parallel, non-analog, non-interactive) models are polynomially-equivalent with regard to the number of steps required to compute. "Degenerate" is meant to exclude models that are inherently inefficient, like counter (Minsky) machines operating with tally numbers (a.k.a. Hilbert strokes).

In the 1960s, Shepherdson and Sturgis [45] developed the random-access register machine (RAM) model. This theoretical model is close in spirit to the design of contemporary, von Neumann architecture computers. Hartmanis [29] and Cook and Reckhow [10] applied the RAM model to the measurement of computational complexity of computer algorithms, since it serves as a more realistic measure of (asymptotic) time and space resource usage than do Turing's more "primitive" machines. It is well-known that multitape Turing machines (TMs) require quadratic time to simulate RAMs [10] and that single-tape Turing machines require quadratic time to simulate multitape ones [30].

The question addressed here is to what extent RAMs are in fact the correct model for measuring algorithmic complexity. The answer will be that they are in fact nearly perfect, a belief expressed by van Emde Boas in his handbook survey on computational models:

> Register-based machines have become the standard machine model for the analysis of concrete algorithms. [49, p. 22]
>
> 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. [49, 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. [48, p. 2] (cf. [49, p. 5])

The thesis notwithstanding, it remains conceivable 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". We shall prove that this is not the case.

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 (and quadratic space) by a RAM equipped with addition and subtraction. This will necessitate a formal, domain-independent definition of effectiveness.

Though we will be counting RAM operations on an *arithmetic* model (using a "uniform" time measure [49]), we ought to bear in mind that each operation is applied to a natural number, normally represented by a logarithmic number of digits. Likewise, the complexity of the algorithm, which may entail operations over data structures other than strings or numbers, will be measured in terms of unit-cost constructor operations for its data structures, as we will see. This, then, is the precise statement of the provable thesis (Theorem 5.5 below):

Time Invariance Theorem. *Any effective algorithm (in the sense to be axiomatized in Section 2) using no more than $T(n)$ constructor, destructor, and equality operations for inputs of size n can be simulated by an arithmetic RAM that also runs in order $T(n)$ steps.*

Note that the (implicit) *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, as axiomatized below, can, in general, perform any bounded number of operations in a single one of its steps. Exactly which operations are counted will be explained in Section 2.2.

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

(1) One requires a generic, datatype-independent notion of algorithm for this claim. Accordingly, we subscribe to the axiomatic characterization of classical algorithms over *arbitrary* domains of [28], 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 (Section 2.1, Postulate I).

(2) We are only concerned with effective algorithms—as opposed, say, to conceptual algorithms like Gaussian elimination over reals. Therefore, we adopt the formalization of *effective* algorithms over arbitrary domains developed in [5, 15, 6], insisting that initial states have a uniform finite description (Section 2.2, Postulate II). 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 though they are of unit cost. Analogous to single-cell Turing-machine operations and memory-cell access for RAMs, we should measure the complexity of algorithms over arbitrary effective data-structures in terms of the number of "atomic" operations on data, rather than the number of more sophisticated operations. Specifically, we propose to count applications of constructors or destructors, and equality tests (Section 3, Definition 3.2). More complex effective operations can *always* be programmed in terms of these basic, constructor operations. (This is a direct consequence of the formulation of effectiveness.) A *basic* algorithm, then, is an effective algorithm that employs only basic operations (Section 2.3, Definition 2.2). Let EFF denote these effective algorithms, expressed in terms of basic operations only.

(4) To establish invariance, we first show (Section 5, Lemma 5.2) that basic algorithms (EFFs) may be emulated step-for-step by what we will call *extended pointer machines (EPMs)*, a powerful extension of Schönhage's Storage Modification Machines [44] (Section 4.3). Then we show how each execution step of such a pointer machine can be simulated in a constant number of RAM steps, operating on words of logarithmic size (Section 5, Lemma 5.3).

Having agreed on the right way to measure complexity of effective algorithms over arbitrary data structures (item 3 above), we build (item 4 above) on [16] (work predating the formalization of effectiveness in [5, 15]), linking a similar variant of pointer machines with (not necessarily effective) *abstract state machines (ASMs)*. By showing that RAMs simulate extended pointer machines efficiently (which is simpler than showing directly that they emulate abstract state machines),[1] and that *basic* abstract state machines (BSMs) emulate any basic implementation of an effective algorithm, the four models (RAM $\geq$ EPM $\geq$ BSM $\geq$ EFF) are chained together and the desired invariance result is obtained in a strong sense—without undue complications. This simulation, based on pointer machines, requires quadratic space. An alternative method of simulation (from [13]) in combination with garbage collection, can be used to reduce the space required by simulation. This latter version of the Invariance Thesis is precisely the original formulation of van Emde Boas [48], within linear space overhead, but it comes at the expense of a quadratic, rather than linear, overhead in time.

We begin in the next section with a review of the connection between algorithms, effectiveness, and abstract state machines—borrowed in the main from [28] and [5]. The question of how to measure complexity generically is addressed in Section 3. Section 4 introduces the machine models that play a rôle, and is followed by the main section, Section 5, containing the simulations and major results. Section 6 outlines the alternative space-saving simulation. We conclude with a brief recapitulation and discussion.

In [19], 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 constant factor overhead in number of steps. The catch is—as the authors indicate—that individual steps can themselves be quite complex.

[1] Regarding pointer machines, van Boas [49] avers that they provide "an interesting theoretical model, but ... its attractiveness as a fundamental model for complexity theory is questionable". The results herein, tightly connecting pointer machines and RAMs, may help increase the potential attractiveness of the former.

2. 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 [27] 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. We present this characterization with minimal motivation; for more detailed support of this axiomatic characterization of algorithms and relevant citations from the founders of computability theory, see [28, 15, 11]. Boker and Dershowitz [5] formalized an additional axiom, which allows one to restrict attention to the family of classical algorithms that are "effective" in a generic sense; this is crucial for our analysis of complexity.

2.1. **Classical Algorithms.**

An algorithm must be describable in finite terms, so that it can be communicated to others. As Kleene [32, p. 493] has 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 postulate of algorithmicity of Gurevich [28] given below. 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. The key observations are:

- A state should contain *all* the relevant information, apart from the algorithm itself, needed to determine the next steps. For example, the "instantaneous description" of a Turing machine computation is just what is needed to pick up a machine's computation from where it has been left off; see [47]. Similarly, the "continuation" of a Lisp program contains all the state information needed to resume its computation. First-order structures suffice to model all salient features of states. This belief is justified by the vast experience of mathematicians and scientists who have faithfully and transparently presented every kind of static mathematical or scientific reality as a logical structure. Compare [39, pp. 420–429].
- The values of programming variables, in and of themselves, are meaningless to an algorithm, which is implementation independent. Rather, it is relationships between values that matter to the algorithm. It follows that an algorithm should work equally well in isomorphic worlds. Compare [20, p. 128]. An algorithm can—indeed, can only—determine relations between values stored in a state via terms in its vocabulary and equalities (and disequalities) between their values.

- Algorithms are expressed by means of finite texts, making reference to only finitely many terms and relations among them. See, for example, [32, p. 493]. So there must be a fixed bound on the amount of work performed by a algorithm in a single step. Compare [33].

These considerations lead to the following tripartite axiom:

Postulate I (Axiom of Algorithmicity)**.**

(i) *An* algorithm *is a state-transition system, comprising a set (or class) S of* states*, a subset S_0 of which are* initial*, and a partial* transition *function $\tau : S \rightharpoonup S$ from (*nonterminal*) state to* next *state.*

(ii) *States may be viewed as (first-order) logical structures over some (finite) vocabulary F, closed under isomorphism (if $x \in S$, then $\iota x \in S$ for all isomorphisms ι; similarly for initial states S_0 and* terminal *states $S_\ddagger = S \setminus \mathrm{Dom}\ \tau$ not in the domain of definition of τ). Transitions preserve the domain (universe) of states (*$\mathrm{Dom}\ \tau(x) = \mathrm{Dom}\ x$ *for all $x \in S \setminus S_\ddagger$). Furthermore, isomorphic states are either both terminal or else their next states are isomorphic ($\tau(x) \cong_\iota \tau(\iota x)$).*

(iii) *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 critical 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 of values to programming "variables", but also the "graph" of those functions that the algorithm can apply. (We treat relations as truth-valued functions.) We may view a state over vocabulary F with domain 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 for 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. The last requirement (iii) of the postulate means, then, that $\tau(x) \setminus x = \tau(y) \setminus y$ whenever $x, y \in S$ give the same interpretation to all critical terms.

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 "Zeno machines" [21, 40], 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 first-order 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 of the above postulate 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.

The significance of the above postulate lies in its comprehensiveness. It formalizes which features exactly characterize (classical) algorithms in their most abstract and generic manifestation. Indeed, a careful analysis of the notion of algorithm in [28] and an examination of the intent of the founders of the field of computability in [5] demonstrate that the demands of this postulate are in fact true of all ordinary, sequential algorithms, the kind envisioned by the pioneers of the field. All models of effective, sequential computation (including Turing machines, counter machines, pointer machines, etc.) satisfy the postulate, as do idealized algorithms for computing with real numbers, or for geometric constructions with compass and straightedge (see [41] for examples of the latter).

2.2. **Effective Algorithms.** The Church-Turing Thesis [31, Thesis I†] asserts that standard models capture effective computation. Specifically:

> **Church's Thesis.** Every effectively computable (partial) numeric function is (partial) recursive.
>
> **Turing's Thesis.** Every effectively computable (partial) string function can be computed by a (not necessarily halting) Turing machine.

For models of computation that operate over arbitrary data structures, however, these two standard notions of what constitutes effectiveness may not be directly applicable. Instead, they are understood as operating over some sort of encodings of those structures. As Montague asserts [35, pp. 430–431]:

> Turing's notion of computability applies directly only to functions on and to the set of natural numbers. Even its extension to functions defined on (and with values in) another denumerable set S cannot be accomplished in a completely unobjectionable way. One would be inclined to choose a one-to-one correspondence between S and the set of natural numbers, and to call a function f on S computable if the function of natural numbers induced by f under this correspondence is computable in Turing's sense. But the notion so obtained depends on what correspondence between S and the set of natural numbers is chosen; the sets of computable functions on S correlated with two such correspondences will in general differ. The natural procedure is to restrict consideration to those correspondences which are in some sense "effective", and hence to characterize a computable function on S as a function f such that, for some effective correspondence between S and the set of natural numbers, the function induced by f under this correspondence is computable in Turing's sense. But the notion of effectiveness remains to be analyzed, and would indeed seem to coincide with computability.

As we are manifestly interested in measuring the complexity of algorithms that deal with graphs and other data structures, we are in need of a generic notion of effectiveness that gets around the circularity in the notion of correspondence (representation) pointed out by Montague.

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 restrict 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, the operations of the initial states of an arbitrary algorithm could contain uneffective infinite information,

with an infinite table lookup, in which case the algorithm could not be regarded as effective. Another problem is that the same domain element might be accessible via several terms, generating nontrivial relations, relations that might hide uncomputable information.

The moral is that we need to place finiteness limitations on the operations included in initial states of algorithms. For an algorithm over arbitrary data structures to be considered effective, we simply require that it be possible to fully and finitely describe its initial states, that starting subset of the algorithm's states containing input values, operations and all. This, in addition to being able to describe transitions finitely (per Postulate I(iii)). Only a state that can be described finitely may be deemed effective.

Three ways of formalizing the notion that initial states have a finite description—thereby characterizing effectiveness—have been suggested in recent literature. One alternative [42] characterizes an effective (initial) state as one for which there is a 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 [15] builds on the popular notion of *effective algebra* [46] and demands 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. This way, an algorithm is effective if there is an injection $\rho : D \to \mathbb{N}$ for each domain D of its states, such that the (partial) function $\rho(f) : \mathbb{N} \rightharpoonup \mathbb{N}$ is (partial) recursive for every operation f of its initial states. These two formalizations are somewhat circular: the first relies on Turing-machine computability to define generic computability and the second relies on mappings to recursive functions for that purpose.

The third alternative [5], insisting that initial states contain virtually nothing more than constructors, is truly domain independent. It is this definition that we adopt below. All the same, it turns out that all three characterizations of effectiveness lead to one and the same class of effective functions (up to isomorphism) for any computational model over any domain [6]. In particular, partial-recursion for natural numbers and Turing machines for strings form maximal effective models with this constructor-based definition, too [5]. And every effective algorithm can be simulated (under some injective representation) by some machine [5] and by some recursive function [15]. In this sense, the Church-Turing thesis can be said to have been formalized and proved from first principles. Moreover, all three notions can be generalized to model *relative* effectiveness [12].

Definition 2.1 (Effective State [5])**.** *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 (excluding those having the* default *value,* UNDEF*).*

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.

Postulate II (Axiom of Effectiveness)**.** *An algorithm is* effective *if all its initial states are effective (in the sense of Definition 2.1) and are all identical except for inputs.*

Constructors provide a way to give a unique name for any domain element. 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 $[c_i(c(x_1, \ldots, x_i, \ldots, x_k)) = x_i]$, 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.

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 necessarily also effective, as demonstrated in [5]. 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, too.

Without loss of effectiveness, we can allow any finite amount of nontrivial data in initial states, provided that—except for the input values—all initial states are the same. (Otherwise, initial states could hide uncomputable outputs.) Moreover, initial states can incorporate any collection of operations that are themselves effective. The circularity of this definition of effectiveness "bottoms-out" with those operations that are programmable directly from the constructors. We may presume that constructors are present in states—even if the algorithm in question avoids their direct use.

2.3. **Basic Algorithms.** We will be counting atomic, constructor operations.

Definition 2.2 (Basic Algorithm)**.** *An effective algorithm is* basic *if its initial states have no infinitely-defined operations other than constructor/destructor operations—not even effective ones.*

Basic algorithms are clearly effective, since they operate over finite data only. They are not expressive enough to emulate all effective functions step-for-step, however, since the latter may have direct access to larger operations, such as multiplication. For that reason, we have allowed an effective state to be equipped with arbitrary "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 down to that level, we take advantage of the fact that (by induction on the definition of effectiveness) any effective algorithm can have its defined operations "in-lined", 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 [28, Lemma 6.1]. That assures that, whenever a state is effective, so is the next state (when there is a next state), as it may be described in terms of the prior state and finitely many updates. This justifies our definition of an effective *algorithm* as having effective *initial* states.

3. 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 each and every constructor operation. So we have a choice: count each basic operation executed by an effective algorithm, or count the transition steps of its corresponding basic algorithm, since each step involves only a bounded number of operations. 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" [49, pp. 10–11], under which every transition is counted as a one time unit. One can refine the cost assigned to each basic operation of a transition step to take the size of its arguments into account, as will be discussed later.

To handle arbitrary data types, the 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 3.1 (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 $\lceil \lg n + 1 \rceil$; 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 [5]). 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, as we said, 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 3.2 (Complexity)**.** *The complexity of an effective algorithm is measured by the number of basic steps (involving constructors, destructors, equality) required to perform the computation from initial to final state, relative to input size.*

In other words, we inline effective sub-algorithms to get a basic one and measure the time 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 $\mathbb{L}$; 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 `rev`: $\mathbb{L} \to \mathbb{L}$ 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, `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 `rev`(x) is effectively something like this:

$y := x$
$z :=$ `nil`
repeat
if $y =$ `nil`
  then return z
  else $z :=$ `cons`(`car`$(y), z)$
    $y :=$ `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/destructor operations (`cons`, `car`, `cdr`, =) performed in a single iteration (in this case 4). 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.

4. Machine Models

Three models play major rôles in our simulation of basic algorithms: the random access machine (RAM), our extended pointer machine (EPM), and the basic abstract state machine (BSM).

4.1. **Random Access Machines.** The RAM machine has access to a finite number of registers plus infinitely-many memory locations indexed by nonnegative integers; each register or memory location can hold a nonnegative integer. For the definition of RAMs, we take the set of instructions suggested by Cook and Reckhow in [10] and use the classification of RAM machines given in [49].

- For *primitive 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) GOTO m if $X > 0$: Transfer control to the m-th line of the program if $X > 0$.
- *Successor RAMs* are an extension of primitive 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 [10]; they extend primitive RAMs with addition and subtraction:
  (5) $X \leftarrow Y + Z$.
  (6) $X \leftarrow Y - Z$.

Other variants allow even more sophisticated instructions, such as multiplication and division.

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. We may also have multiple memories, in which case each gets a distinct name.

Proposition 4.1 ([43])**.** *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* $[i_1, \ldots, i_\ell]$ *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* $O(\log i_1 + \ldots + \log i_\ell)$*).*

4.2. **Storage Modification Machines.** Schönhage's Storage Modification Machines (SMMs) [44] are a type of pointer machine (see [2] for a survey), which manipulate a dynamic pointer structure (while reading an input string and writing to output). Their memory takes the form of a labeled (multi-) graph. Duplicate outgoing edge labels are disallowed. A *path* is a sequence of labels beginning at a distinguished *focus* node. Nodes may be identified (not necessarily uniquely) by paths leading to them. A machine, when running, may add new nodes to the structure and redirect edges, perhaps rendering some existing nodes inaccessible in the process. Schönhage's machines are a generalization of Kolmogorov's machines [34], in that the graph's edges are directed, and only the out-degree of vertices is bounded. Kolmogorov machines, in turn, generalize Turing machines and are more powerful in terms of real-time computability [24]. See [26].

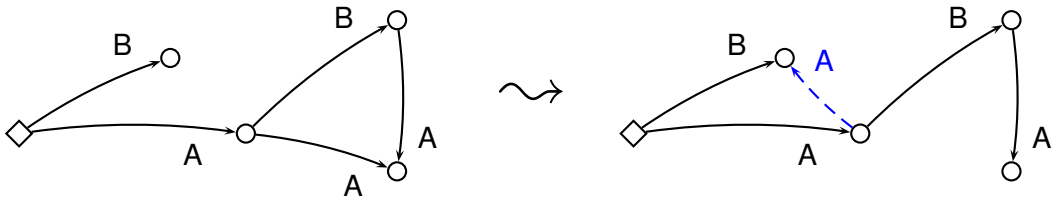
Let Δ be a finite alphabet of *direction* labels and X be a finite set of nodes with a distinguished *focus*. From each node in X there may be outgoing edges $\xrightarrow{\delta}$ labeled with directions $\delta \in \Delta$, but no more than one edge per direction. For convenience, we denote by $W^!$ the end-node of path W; when W is empty, the focus is intended.

The set of machine instructions, each of which may carry an instruction label, 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 going out of the penultimate node $U^!$ is redirected to the new node; all pointers from this new node are directed to the original $W^!$.
- $W\delta := V$: Redirect the last edge δ of path $W\delta$ to point via direction δ to the end node $V^!$ of path V.
- **if** $V = W$ **then** P: If paths V and W end at the same node ($V^! = W^!$), then execute P.
- **if** $V \neq W$ **then** P: If paths V and W end at distinct nodes ($V^! \neq W^!$), then execute P.
- **halt**: Stop now.

A *program* is a sequence of these instructions.

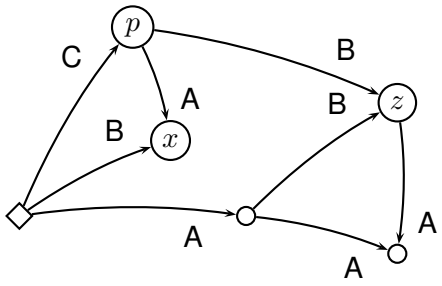
For example, the instruction **if** $\mathsf{AA} \neq \mathsf{AB}$ **then** $\mathsf{AA} := \mathsf{B}$ (on graphs with directions $\Delta = \{\mathsf{A}, \mathsf{B}, \ldots\}$) has the following effect (the focus is drawn as a diamond and the new edge is dashed):



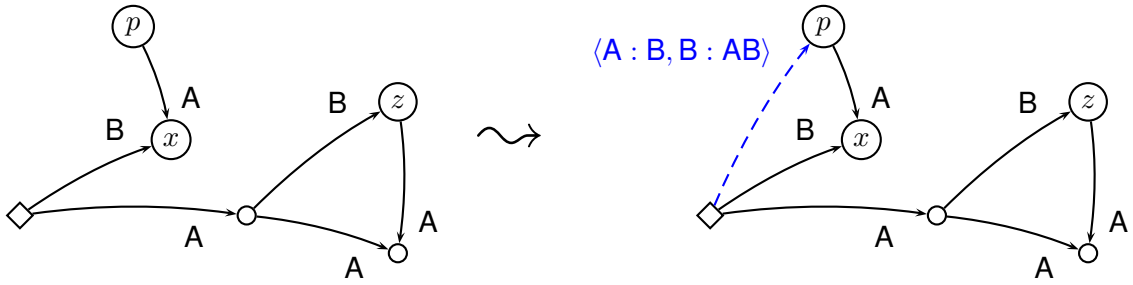
4.3. **Extended Pointer Machines.** We need to extend the syntax of Schönhage's machines with an "inverse" operation, so as to resurrect "lost" connections; we refer to this enhanced model as *extended pointer machines (EPMs)*:

- **remember**$\langle W \mid \delta_1 : V_1, \ \ldots \ , \delta_k : V_k\rangle$: Remember end node $W^!$ as the one with edges of type δ_i pointing to nodes $V_i^!$, for $i = 1, \ldots, k$. Any previous applications of **remember** for the same node and directions are forgotten. ***Nota bene:*** *Once remembered, the information is maintained regardless of changes to the structure.*
- **lookup**$\langle \delta_1 : V_1, \ \ldots \ , \delta_k : V_k\rangle$: Suppose X was the last node remembered as the one with edges δ_i pointing to nodes $V_1^!, \ldots, V_k^!$ via directions $\delta_1, \ldots, \delta_k$, respectively. The set an edge labeled with the tuple $\langle \delta_1 : V_1, \ \ldots \ , \delta_k : V_k\rangle$ to point from the focus to node X. If, however, there is no appropriate node already memorized by the machine, then any edge labeled $\langle \delta_1 : V_1, \ \ldots \ , \delta_k : V_k\rangle$ from the focus will be removed, if such exists.

For example, performing **remember**$\langle \mathsf{C} \mid \mathsf{A} : \mathsf{B}, \mathsf{B} : \mathsf{AB}\rangle$ on the topology



the machine will remember the node marked p as the one with edge of type A outgoing to node x (at the end of path B) and edge of type B outgoing to z (at the end of path AB). Assume, now, that the edge labeled C and the outgoing edge B to z have been removed, and then we request **lookup**$\langle \mathsf{A} : \mathsf{B}, \mathsf{B} : \mathsf{AB}\rangle$. The outcome is as follows:



Despite the fact that p was no longer reachable from the focus and that its outgoing edges were changed, the machine still remembers it as the one that satisfies the requirements of the **lookup**, since it was the last one **remember**ed as such. Note that the set of edge labels is fixed for any one program, even though these compound labels may be nested.

5. RAM Simulation of Effective Algorithms

As already explained in Section 3, we should measure the complexity of effective algorithms in terms of *basic* operations. Thus, we need to show how RAMs simulate basic algorithms.

As intermediary devices, we make use of the extension of Schönhage's machines just described, as well as BSMs. We prove that constructor-based algorithms may be expressed as BSMs, which can be simulated by extended pointer machines, and which, in turn can be simulated by arithmetic multidimensional RAMs. Arrays are used for labeled edges and to remember back pointers. All this with negligible overhead.[2]

5.1. **BSMs Describe EFFs.** The first thing we need is a way to formally capture the step-by-step behavior of basic, effective algorithms.

Lemma 5.1. *Any basic algorithm (EFF) can be described by a basic abstract state machine (BSM).*

Proof. Basic ASMs describe the transitions of basic algorithms, since—in general—ASMs capture the behavior of all classical algorithms precisely, using the same operations [28]. □

5.2. **EPMs Simulate BSMs.** The second step is to simulate basic constructor/destructor steps via extended pointer machine commands.

Lemma 5.2. *Any basic abstract state machine (BSM) can be simulated by an extended pointer machine (EPM) with at most constant factor overhead in the number of transitions.*

A similar claim was proved in [16, Lemma 1] for *non-basic* abstract state machines and a comparable extension of pointer machines. In general, an abstract state machine consists of a set of conditional assignments (to locations in the state) that are repeated in parallel, as long as at least one condition is satisfied. This model satisfies the requirement of algorithmicity given in Postulate I. These ASMs have the added ability to access (**import**) a bounded number of fresh (as yet unused) elements in a single transition, paralleling the **new** operation of pointer machines. Without further restrictions on the domain, available oracles and **import** behavior, this class obviously also contains 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 an ASM 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, an ASM will import a new domain element for that, if that is a first-time access.

Proof. We prove that our extended pointer machines simulate ASMs with constant-factor overhead in the number of steps.

It was proved in [16] that any ASM can be simulated with only constant-factor overhead by another ASM whose vocabulary has only nullary (scalar) and unary function symbols—plus a single binary symbol used for ordered pairing of elements. The idea behind that is simple: any function of arity n may be treated instead as a unary function working over ordered n-tuples. Those n-tuples may be created by $n-1$ nested pairings. Since the vocabulary of any algorithm is finite and depends only on the algorithm, the transformation introduces only a bounded number of pairings.

[2]Alternatively, we could have avoided pointer machines altogether and shown directly how to simulate BSMs by (multidimensional) RAMs, but that would have engendered a mass of technical detail.

Furthermore, an ASM as above but without pairing can be simulated by a classical pointer machine. For simulation of pairing, an extension rule, **create**, was introduced in [16]. 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 program doing 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. (A pointer machine with our extension can, in general, **remember** any change it performs.) □

5.3. **RAMs Simulate of EPMs.** The next step is to simulate pointer machines commands by means of RAM operations.

Lemma 5.3. *Extended pointer machines (EPMs) 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 $\delta \in \Delta$ 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 a storage modification machine, 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 a pointer machine 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 storage modification machine 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 pointer machines, however, requires the ability to remember source nodes. To simulate this, we will have to extend our construction as well. Let $\Delta = \{\delta_1, \ldots, \delta_k\}$. We define a k-dimensional array A and utilize it to simulate the extension in the following way:

- An application of **remember**$\langle W \mid \delta_1 : V_1, \ \ldots \ , \delta_k : V_k\rangle$ will be simulated by $A[\widehat{V_1^!}, \ldots, \widehat{V_k^!}] := \widehat{W^!}$. 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**$\langle \delta_1 : V_1, \ \ldots \ , \delta_k : V_k\rangle$ will therefore be $\delta[\widehat{\epsilon^!}] := A[\widehat{V_1^!}, \ldots, \widehat{V_k^!}]$, with $\delta = \langle \delta_1 : V_1, \ \ldots \ , \delta_k : V_k\rangle$ and with 0 for missing labels. (Remember that an empty path stands for the *focus* node.)

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

As explained already, to measure the complexity of effective algorithm, we "bootstrap" it to a basic one, and then measure the complexity of the latter. (See Definition 3.2.)

Corollary 5.4. *Any basic algorithm (EFF) can be simulated by a multidimensional successor RAM with only a constant factor in the number of transitions.*

Proof. It follows from Lemma 5.2 that any basic algorithm may be simulated by an extended pointer machine, which, by Lemma 5.3, is doable with a multidimensional successor RAM. □

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

Theorem 5.5 (Time Invariance Theorem)**.** *Any basic effective algorithm using no more than $T(n)$ constructor, destructor, and equality operations for inputs of size n can be simulated by an arithmetic RAM in $O(T(n))$ steps, with a word size that may grow to order* $\log \max\{T(n), n\}$ *bits.*

Proof. It follows from Corollary 5.4 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 4.1.

Each transition of a basic algorithm may be fully described by a bounded number of invocations of basic constructor and destructor operations. This 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 $T(n)$, 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. This gives the stated bound on word size of $\log \max\{T(n), n\}$ bits. □

For a RAM, one might, in fact, wish also to take into account the length of the numbers stored in its arrays [10]. 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 $x = y$.

Theorem 5.6. *Basic algorithms (EFFs) and storage modification machines (SMMs) simulate each other to within a logarithmic factor.*

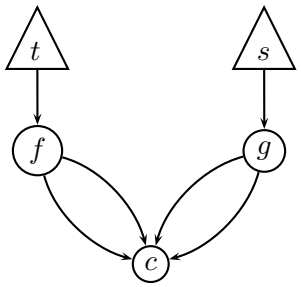
Proof. It is known that any SMM is equivalent (up to a constant factor) time-wise to an arithmetic RAM [44]. Our main theorem tells us that an arithmetic RAM, in turn, can simulate a basic algorithm with constant factor difference in the number of steps. For the other direction, basic algorithms can easily simulate arithmetic RAMs: primitive RAM operations require only assignments and conditionals, while addition and subtraction require logarithmic-many constructor/destructor basic operations to manipulate the bits. □

6. Less Space, More Time

Let $X_1 \rightsquigarrow \cdots \rightsquigarrow X_n$ be a run of a basic algorithm, and let a be an element in the 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; we say that it 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 noted in [42] 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 state 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 X_i in the run.

For term t, we denote the minimal term-graph (dag) representation of it by $G(t)$, with nodes labeled by function symbols (see [37]). Since $G(t)$ is minimal, it will not contain repeated factors. To avoid repeated factors, not just in one term but in the whole state, we merge individual term graphs into one big graph and call the resulting "jungle" of terms, a *tangle*. The tangle is used to maintain the constructor-term values of all the critical terms of the algorithm. See [13] for details.

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. The tangle looks like this:



with the pointers labeled s and t pointing to the vertices in the graph that represent those terms.

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

Theorem 6.1 (Space Invariance Theorem)**.** *Any basic effective algorithm (EFF) with time complexity (measured in constructor operations) $T(n)$ and space complexity $S(n)$ can be*

simulated by an arithmetic RAM in $O(nT(n) + T(n)^2)$ *steps and* $O(S(n))$ *space, with a word size that grows to order* $\log S(n)$.

Proof. In [13], 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 that points to the value of t in $G(X_i)$. In [13, 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 ongoing 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. □

7. Discussion

In proving the Time Invariance Theorem (Theorem 5.5), we have established 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 in the Space Invariance Theorem (Theorem 6.1) that space complexity may be preserved with a quadratic increase in time.

It follows that to dramatically outperform any RAM, an alternate model must violate one of the postulates of effective algorithms. 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). See the discussion in [50].
- 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. 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 [25], for instance,

is outside its scope, though quantum algorithms have been formalized as abstract state machines in [23].

- It has a "non-effective" domain—for example, continuous-space (real number) algorithms, as in Euclidian geometry or, alternatively, access to non-programmable oracle operations, like the halting function for Turing machines.

Summing up, any algorithm running on any effective (constructor-based) 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 [10]. 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* [49], known also as the *Extended Church-Turing Thesis* [1]. Every model meeting the natural postulates posed here is "reasonable" and belongs to van Emde Boas's "standard class".

It remains to be determined whether our Time and Space Invariance Theorems (Theorems 5.5 and 6.1) can be combined together to show that a RAM can simulate all effective algorithms with only constant factor overhead in both time and space.

For a discussion of the importance of honestly measuring the complexity of algorithms over arbitrary domains, see [7].

The question of relative complexity of parallel models is addressed in recent work [14].

Algorithms that interact with the user or the environment have been formalized in [3, 4], but characterizing their complexity has not been studied.

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 [36])
>
> 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 [50])

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. [17]). 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, may be found in [8].

References

[1] Scott Aaronson. Quantum computing since Democritus. Lecture notes, Fall 2006. Available at `http://www.scottaaronson.com/democritus/lec4.html` (viewed December 5, 2016).

[2] Amir M. Ben-Amram. What is a "pointer machine"? *SIGACT News*, 26(2):88–95, June 1995.

[3] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive small-step algorithms, Part I: Axiomatization. *Logical Methods in Computer Science*, 3(4):paper 3, 2007.

[4] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive small-step algorithms, Part II: Abstract state machines and the characterization theorem. *Logical Methods in Computer Science*, 4(4):paper 4, 2007. Available at `http://arxiv.org/pdf/0707.3789v2` (viewed December 5, 2016).

[5] Udi Boker and Nachum Dershowitz. The Church-Turing thesis over arbitrary domains. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 199–229. Springer, Berlin, 2008.

[6] Udi Boker and Nachum Dershowitz. Three paths to effectiveness. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 36–47, Berlin, Germany, August 2010. Springer.

[7] Udi Boker and Nachum Dershowitz. Honest computability and complexity. In Eugenio Omodeo and Alberto Policriti, editors, *Martin Davis on Computability, Computational Logic, and Mathematical Foundations*, volume 10 of *Outstanding Contributions to Logic Series*, chapter 6, pages 153–175. Springer, 2017.

[8] Olivier Bournez, Nachum Dershowitz, and Pierre Néron. Axiomatizing analog algorithms. In *Computability in Europe 2016: Pursuit of the Universal (CiE) (Paris, France)*, volume 9709 of *Lecture Notes in Computer Science*, pages 215–224, Switzerland, June 2016. Springer-Verlag. Available at `https://arxiv.org/abs/1604.04295` (viewed December 5, 2016).

[9] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[10] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. *Journal of Computer Systems Science*, 7:354–375, 1973.

[11] Nachum Dershowitz. The generic model of computation. In *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2011, Zürich, Switzerland)*, volume 88 of *Electronic Proceedings in Theoretical Computer Science*, pages 59–71, July 2012. Available at `http://dx.doi.org/10.4204/EPTCS.88.5` (viewed December 5, 2016).

[12] Nachum Dershowitz and Evgenia Falkovich. Effectiveness. In Hector Zenil, editor, *A Computable Universe: Understanding & Exploring Nature as Computation*, pages 77–97. World Scientific, Singapore, December 2012.

[13] Nachum Dershowitz and Evgenia 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, July 2011, Zürich, Switzerland)*, volume 88 of *Electronic Proceedings in Theoretical Computer Science*, pages 72–78, July 2012. Available at `http://dx.doi.org/10.4204/EPTCS.88.6` (viewed December 5, 2016).

[14] Nachum Dershowitz and Evgenia Falkovich-Derzhavetz. On the parallel computation thesis. *Logic Journal of the IGPL*, 24(3):346–374, March 2016.

[15] Nachum Dershowitz and Yuri Gurevich. A natural axiomatization of computability and proof of Church's Thesis. *Bulletin of Symbolic Logic*, 14(3):299–350, September 2008.

[16] Scott Dexter, Patrick Doyle, and Yuri Gurevich. Gurevich abstract state machines and Schönhage storage modification machines. *Journal of Universal Computer Science*, 3(4):279–303, 1997.

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

[18] Evgenia Falkovich. *On Generic Computational Models*. PhD thesis, School of Computer Science, Tel Aviv University, Tel Aviv, Israel, 2015.

[19] Marie Ferbus-Zanda and Serge Grigorieff. ASMs and operational algorithmic completeness of lambda calculus. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 301–327. Springer, Berlin, Germany, August 2010. Available at `http://arxiv.org/pdf/1010.2597v1.pdf` (viewed December 5, 2016).

[20] Robin Gandy. Church's thesis and principles for mechanisms. In *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 123–148. North-Holland, 1980.

[21] E. Mark Gold. Limiting recursion. *J. Symbolic Logic*, 30(1):28–48, 1965.

[22] Les Goldschlager and Andrew Martin Lister. *Computer Science: A Modern Introduction*. Prentice-Hall, 1982.

[23] Erich Grädel and Antje 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)*, pages 309–323, Berlin, 2003. Springer.

[24] Dima V. Grigoryev. Kolmogorov algorithms are stronger than Turing machines. *Journal of Soviet Mathematics*, 14:1445–1450, 1980. Russian original 1976.

[25] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (Philadelphia, PA)*, pages 212–219, May 1996.

[26] Yuri Gurevich. On Kolmogorov machines and related issues. *Bulletin of the European Association for Theoretical Computer Science*, pages 71–82, 1988.

[27] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, Oxford, 1995.

[28] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[29] Juris Hartmanis. Computational complexity of random access stored program machines. *Mathematical Systems Theory*, 5:232–245, 1971.

[30] Frederick E. Hennie and Richard E. Stearns. Two-way simulation of multitape Turing machines. *J. of the Association of Computing Machinery*, 13:533–546, 1966.

[31] Stephen C. Kleene. *Mathematical Logic*. Wiley, New York, 1967.

[32] Stephen C. Kleene. Reflections on Church's thesis. *Notre Dame Journal of Formal Logic*, 28(4):490–498, 1987.

[33] Andreĭ N. Kolmogorov. O ponyatii algoritma [on the concept of algorithm] (in Russian). *Uspekhi Matematicheskikh Nauk [Russian Mathematical Surveys]*, 8(4):175–176, 1953. English version: Vladimir A. Uspensky and Alexei L. Semenov, *Algorithms: Main Ideas and Applications*, Kluwer, Norwell, MA, 1993, pp. 18–19.

[34] Andreĭ N. Kolmogorov and Vladimir A. Uspensky. K opredeleniu algoritma (in Russian). *Uspekhi Matematicheskikh Nauk [Russian Mathematical Surveys]*, 13(4):3–28, 1958. English version: On the definition of an algorithm, *American Mathematical Society Translations*, ser. II, vol. 29, 1963, pp. 217–245.

[35] Richard Montague. Towards a general theory of computability. *Synthese*, 12(4):429–438, 1960.

[36] Ian Parberry. Parallel speedup of sequential machines: A defense of parallel computation thesis. *SIGACT News*, 18(1):54–67, March 1986.

[37] Detlef Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, volume volume 2, chapter 1, pages 3–61. World Scientific, 1999.

[38] Emil L. Post. Finite combinatory processes—Formulation 1. *Journal Symbolic Logic*, 1(3):103–105, 1936.

[39] Emil L. Post. Absolutely unsolvable problems and relatively undecidable propositions: Account of an anticipation. In Martin Davis, editor, *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, pages 375–441. Birkhaüser, Boston, MA, 1994. Unpublished paper, 1941.

[40] Hilary Putnam. Trial and error predicates and the solution to a problem of Mostowski. *J. Symbolic Logic*, 30(1):49–57, 1965.

[41] Wolfgang Reisig. On Gurevich's theorem on sequential algorithms. *Acta Informatica*, 39(4):273–305, April 2003.

[42] Wolfgang Reisig. The computable kernel of Abstract State Machines. *Theoretical Computer Science*, 409(1):126–136, December 2008.

[43] John Michael Robson. Random access machines with multi-dimensional memories. *Information Processing Letters*, 34:265–266, 1990.

[44] Arnold Schönhage. Storage modification machines. *SIAM J. Computing*, 9:490–508, 1980.

[45] John C. Shepherdson and Howard E. Sturgis. Computability of recursive functions. *Journal of the Association of Computing Machinery*, 10:217–255, 1963.

[46] Viggo Stoltenberg-Hansen and John V. Tucker. *Effective Algebra*, volume 4 of *Handbook of Logic in Computer Science*, chapter 4, pages 357–526. Oxford University Press, 1995.

[47] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937. Corrections in vol. 43 (1937), pp. 544–546. Reprinted in Martin Davis (ed.), *The Undecidable*, Raven Press, Hewlett, NY, 1965.

[48] Peter van Emde Boas. Machine models and simulations (Revised version). Technical Report CT-88-05, Institute for Language, Logic and Information, University of Amsterdam, August 1988.
[49] Peter van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 1–66. North-Holland, Amsterdam, 1990.
[50] Andrew C. Yao. Classical physics and the Church-Turing Thesis. *Journal of the ACM*, 77:100–105, January 2003.