A Formalization and Proof of the Extended Church-Turing Thesis

Nachum Dershowitz
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
nachum.dershowitz@cs.tau.ac.il

Evgenia Falkovich*
School of Computer Science
Tel Aviv University
Tel Aviv, Israel
jenny.falkovich@gmail.com

We prove the *Extended Church-Turing Thesis*: Every effective algorithm can be efficiently simulated by a Turing machine. This is accomplished by emulating an effective algorithm via an abstract state machine, and simulating such an abstract state machine by a random access machine with only linear overhead and logarithmic word size.

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

1 Introduction

The Church-Turing Thesis [10, Thesis I†] asserts that all effectively computable numeric functions are recursive and, likewise, that they can be computed by a Turing machine, or—more precisely—can be simulated under some representation by a Turing machine. As Kleene [11, p. 493] explains,

> 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 claim has recently been axiomatized and proven [4, 7].

The *extended* thesis adds the belief that the overhead in such a Turing machine simulation is only polynomial. One formulation of this extended thesis is as follows:

> The so-called "Extended" Church-Turing Thesis: ... any function naturally to be regarded as efficiently computable is efficiently computable by a Turing machine. (Scott Aaronson [1])

We demonstrate the validity of this extended thesis for all (sequential, deterministic, non-interactive) effective models over arbitrary constructive domains in the following manner:

1. For a generic, datatype-independent notion of algorithm, we adopt the axiomatic characterization of ("sequential") *algorithms* over arbitrary domains due to Gurevich [9] (Section 2.1, Definition 1).
2. To restrict our attention to effective algorithms only—as opposed, for instance, to conceptual algorithms like Gaussian elimination over all the reals—we adopt the formalization of *effective* algorithms over arbitrary domains from [4] (Section 2.3, Definition 4).

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

To appear in EPTCS.

© N. Dershowitz & E. Falkovich
This work is licensed under the
Creative Commons Attribution License.

3. Because we will be dealing with models of computation that work over different domains (such as strings for Turing machines and numbers for random access machines), and will, therefore, need to represent data values of one model in another, we will require the notion of *simulation* of algorithms developed in [3] (Section 3.1, Definition 6).
4. Algorithms, in and of themselves, are indifferent to the actual identity and internal nature of the elements of their domain. The same algorithm will work equally with different representations, as for example, testing primality of numbers whether given as decimal digits, as Roman numerals, or in Neolithic tally notation. So that we may speak of the computation of a function over a given representation, we consider *implementations*, by which we mean algorithms operating over a specific domain (Section 2.2, Definition 2).
5. For counting steps and measuring complexity, we limit operations to basic ones: testing equality of domain elements, application of constructors, and lookup of stored values. We would not, for example, normally want to treat multiplication as a unit-cost operation. A *basic* implementation is one that only employs these basic operations (Section 2.3, Definition 3).
6. We emulate effective algorithms step-by-step by *abstract state machines (ASMs)* [9] in the precise manner of [2] (Section 4.1, Definition 11).
7. We represent arbitrary domain elements of effective algorithms by *minimal* constructor-based graph structures (dags). We call these graph structures *tangles* (Sections 4.2–4.3; cf. [17]).
8. We show that each step of a basic ASM implementation can be simulated in a constant number of random-access (pointer) machine (RAM) steps (Section 4.4, Lemma 23).
9. Domain elements (Platonic numbers, say) have no innate "size". So as not to underestimate complexity by padding input, we measure the *size* of input as the number of vertices in the minimal tangle representation (Section 3.2, Definition 8).
10. As multitape Turing machines (TMs) simulate RAMs in quadratic time [6], the thesis follows (Section 5).

Summarizing the effectiveness postulates under which we will be working, as detailed in the next section: an effective algorithm is a state-transition system, whose states can be any (logical) structure, and for which there is some finite description of initial states and of (isomorphism-respecting) transitions.

A "physical" version of the extended thesis—as expressed in the following quotations—holds to the extent that physical models of computation adhere to these effectiveness postulates.

> The Extended Church-Turing Thesis states ... that time on all "reasonable" machine models is related by a polynomial. (Ian Parberry [13])
>
> 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 [18])

2 Algorithms

We are interested in comparing the time complexity of algorithms implemented in different effective models of computation, models that may take advantage of arbitrary data structures. The belief that all feasible models make relatively similar demands on resources when executing algorithms underlies the study of complexity theory.

2.1 Algorithms

First of all, an algorithm, in its classic sense, is a time-sequential state-transition system, whose transitions are partial functions on its states. This entails that each state is self-contained and that the next state, if any, is determined by the prior one. The necessary information in states can be captured using logical structures, and an algorithm is expected to be independent of the choice of representation and to produce no unexpected elements. Furthermore, an algorithm must, by its very nature, possess a finite description. These considerations lead to the following definition.

Definition 1 (Algorithm [9]). A *classical algorithm* is a (deterministic) state-transition system, satisfying the following three postulates:

I. It is comprised of a set[1] S of *states*, a subset $S_0 \subseteq S$ of *initial* states, and a partial *transition* function $\tau : S \rightharpoonup S$ from states to states. We write $X \leadsto_A X'$ when $X' = \tau(X)$. States for which there is no transition are *terminal*.

II. All states in S are (first-order) structures over the same finite vocabulary F, and X and $\tau(X)$ share the same domain for every $X \in S$. For convenience, we treat relations as truth-valued functions and refer to structures as *algebras*, and let t_X denote the value of term t as interpreted in state X.[2] The sets of states, initial states, and terminal states are each closed under isomorphism. Moreover, transitions respect isomorphisms. Specifically, if X and Y are isomorphic, then either both are terminal or else $\tau(X)$ and $\tau(Y)$ are isomorphic via the same isomorphism.

III. There exists a fixed finite set T of *critical* terms over F that fully determines the behavior of the algorithm. Viewing any state X over F with domain D as a set of location-value pairs $f(a_1,\ldots,a_n) \mapsto a_0$, where $f \in F$ and $a_0,a_1,\ldots,a_n \in D$, this means that whenever states X and Y *agree* on T, in the sense that $t_X = t_Y$ for every critical term $t \in T$, either both are terminal states or else the state changes are the same: $\tau(X) \setminus X = \tau(Y) \setminus Y$.

A *run* of an algorithm A is a finite or infinite sequence $X_0 \leadsto_A X_1 \leadsto_A \cdots \leadsto_A$, where $X_0 \in S_0$ is an initial state of A.

For detailed support of this axiomatic characterization of algorithms, see [9, 7]. Clearly, we are only interested here in deterministic algorithms. We used the adjective "classical" to clarify that, in the current study, we are leaving aside new-fangled forms of algorithm, such as probabilistic, parallel or interactive algorithms.

2.2 Implementations

A classical algorithm may be thought of as a class of "implementations", each computing some (partial) function over its state space. An implementation is determined by the choice of representation for the values over which the algorithm operates, which is reflected in a choice of domain.

Definition 2 (Implementation). An *implementation* is an algorithm $\langle \tau, S, S_0 \rangle$ restricted to a specific domain D. Its states are those states $S \upharpoonright D$ with domain D; its *input states* $S_D = S_0 \upharpoonright D$ are those initial states whose domain is D; its transition function τ is likewise restricted.

So that we may view an implementation as computing a partial function over its domain, we require that there exist a subset I of the critical terms, called *input terms*, such that all input states agree on all terms over F except for these, and such that the values of the input terms in input states cover the whole

[1] Or class—it matters not.
[2] All "terms" in this paper are ground (that is, variable-free).

domain. That is, we require $\{(y^1_X, \ldots, y^\ell_X) : X \in S_D\} = D^\ell$, where $I = \{y^1, \ldots, y^\ell\}$. There should also be a critical term z to hold the output value for terminal states. Then we may speak of an algorithm A with terminating run $X_0 \rightsquigarrow_A \cdots \rightsquigarrow_A X_N$ as computing $A(y^1_{X_0}, \ldots, y^\ell_{X_0}) = z_{X_N}$. The function being computed is undefined for those inputs for which the run does not terminate. (The presumption that an implementation accepts any value from its domain as a valid input is not a limitation, because the outcome of an implementation on undesired inputs is of no consequence.)

2.3 Effectiveness

The postulates in Definition 1 limit transitions to be effective, in the sense of being amenable to finite description (see Theorem 13 below), but they place no constraints on the contents of initial states. In particular, initial states may contain infinite, uncomputable data. To preclude that and ensure that an algorithm is effective, we will need an additional assumption. For an initial state to be "effective", it must either be "basic", or all its operations must be programmable from basic states.

Definition 3 (Basic)**.**

(a) We call an algebra X over vocabulary F and with domain D *basic* if $F = K \uplus C$, and the following conditions hold:

- The domain D is isomorphic to the Herbrand universe (free term algebra) over K. This means that there is exactly one term over K taking on each domain value. We call K the *constructors* of X.
- All the operators in C have some fixed pervasive constant value UNDEF.

(b) An implementation is *basic* if all its initial states are basic with respect to the same constructors.

(c) Invariably, states are equipped with equality and Boolean operations, and K includes (nullary) constructors for domain values TRUE, FALSE, and UNDEF.

Constructors are the usual way of thinking of the domain values of computational models. For example, strings over an alphabet {a,b,...} are constructed from a nullary constructor $\varepsilon()$ and unary constructors a($\cdot$), b($\cdot$), etc. The positive integers in binary notation are constructed out of the nullary ε and unary 0 and 1, with the constructed string understood as the binary number obtained by prepending the digit 1. To construct 0-1-2 trees, we would have three constructors, $k_0()$, $k_1(\cdot)$, and $k_2(\cdot,\cdot)$, for nodes of outdegree 0 (leaves), 1 (unary), and 2 (binary), respectively.

Definition 4 (Effectiveness [4])**.**

(a) We call an algebra X over vocabulary F and with domain D *constructive* if $F = K \uplus C$, and the following conditions hold:

- The operations K construct D, as in the previous definition.
- Each of the operations in C can be computed by some effective implementation over K.

(b) An *effective implementation* is a classical algorithm restricted to initial states that are all constructive over the same partitioned vocabulary $F = K \uplus C$.

In other words, C is a set of effective "oracles", obtained by bootstrapping from basic implementations.

Clearly, basic implementations are effective, as are implementations with only finitely many arguments for which operations in C take on values other than UNDEF. Moreover, if there is a basic algorithm with the same domain and constructors that computes some function, then that function can be the interpretation given by initial states to one of the symbols in C.

For example, 0 and successor are constructors for the naturals. With them alone, one can design a basic algorithm for addition. Multiplication is also effective, since there is an algorithm for multiplication with $+$ in C interpreted as addition.

In [4], equality of constructor values is not given for free, and must be programmed—in linear time—as part of C (as shown therein). For now, we will assume we always have it and discuss its cost later.

If $X_0 \in S_0$ is basic or constructive, then it has a finite description: the finitely-many operators in C can each be described by an effective algorithm. (In the pathological situation where an algorithm modifies its constructors, we should imagine that C contains shadow implementations of the constructors K and it is the latter that are wrecked.) Thus, any reachable state of an effective implementation also has a finite description, since—as a consequence of Postulate III—each step of the implementation can only change a bounded number of values of the operations in F.

Two alternative characterizations of effectiveness, reflecting classical notions involving Turing machines [15] and recursive functions [7], respectively, were shown to be equivalent to this one in [5].

3 Complexity

One usually measures the complexity of an algorithm as the number of individual computation steps required for execution, relative to the size of the input data. This demands a definition of the notions, "data size" and "individual step". By a "step", one usually means a single step of some well-defined theoretical computational model, like a Turing machine or RAM, implementing the algorithm via a chosen representation of domain. Despite the fact that Postulate III imposes a fixed bound on the number of operations executed in any single step, the length of a run of an effective implementation need not correspond to a faithful counting of steps. That is because states are allowed to contain infinite non-trivial operations as predefined oracles in initial states; a single application of such a complex operation should entail more than the implicit unit cost. Hence, we will need to work with a lower-level model, consisting of "basic implementations", and show how it too can simulate any effective implementation.

3.1 Representations

We aim to prove that any effective implementation can be simulated by a basic one. Basic implementations will provide us with a faithful underlying model for measuring steps. But to do this, let us first recall the notion of simulation defined in [3].

Definition 5 (Representation)**.**

1. A *representation* of domain D within domain E is a total injective function $\rho : D \rightarrowtail E$. Representations naturally extend to functions:

$$\rho(f)(x^1,\ldots,x^\ell) = \rho[f(\rho^{-1}(x^1),\ldots,\rho^{-1}(x^\ell))],$$

whenever $x^i \in \operatorname{Im}\rho$ for all i.

2. Let X and Y be algebras over vocabularies F and G with domains D and E, respectively. We say that X is *mimicked* by Y via representation ρ, denoted $X \preceq_\rho Y$, if Y contains all the information that X does (and maybe more), that is, for every $f \in F$, there is a $g \in G$ such that $g_Y = \rho(f_X)\restriction_{\operatorname{Im}\rho}$.

Definition 6 (Simulation)**.**

1. Let A and B be two implementations over vocabularies F and G with domains D and E, respectively. We say that A is *simulated* by B via representation ρ, denoted $A \preceq_\rho B$, if B can perform any computation that A can (and maybe more). In other words, for any run $X_0 \leadsto_A X_1 \leadsto_A \cdots$ of A there exists a run $Y_0 \leadsto_B Y_1 \leadsto_B \cdots$ of B and an increasing sequence $\{j_i\} \subseteq \mathbb{N}$, such that $X_i \preceq_\rho Y_i$ for all i.
2. We say that A is *emulated* by B via ρ if for any run $X_0 \leadsto_A X_1 \leadsto_A \cdots$ of A there exists a run $Y_0 \leadsto_B Y_1 \leadsto_B \cdots$ of B, such that $X_i \preceq_\rho Y_{j_i}$ for all i.
3. We say that A and B are *computationally equivalent* if they emulate each other and possess exactly the same runs up to renaming and isomorphism. Formally, there should be a bijective renaming $\rho : F \rightarrowtail\!\!\!\!\twoheadrightarrow G$ and a bijective representation $\rho : D \rightarrowtail\!\!\!\!\twoheadrightarrow E$ such that $X_0 \leadsto_A X_1 \leadsto_A \cdots$ is a run of A if and only if $\rho(X_0) \leadsto_B \rho(X_1) \leadsto_B \cdots$ is a run of B, where $\rho(X)$ assigns $\rho(f)(\rho(\bar{a})) \mapsto \rho(b)$ when X assigns $f(\bar{a}) \mapsto b$.

By standard programming techniques, bootstrapped operations (subroutine calls) can be internalized:

Proposition 7. *For every effective implementation P over a vocabulary $K \uplus C$, there is a basic implementation simulating P over some vocabulary $K \uplus C'$.*

Thus, we may conclude that any effective implementation possesses a basic implementation, which is the faithful underlying model by which to count individual computation steps.

For example, if an effective implementation includes decimal multiplication among its bootstrapped operations, then we do not want to count multiplication as a single operation (which would give us a "pseudo-complexity" measure), but, rather, we need to count the number of basic decimal-digit operations, as would be tallied in the basic simulation of this effective implementation.

3.2 Input Size

There are effective algorithms for all manner of data, numbers, matrices, text, trees, graphs, and so on. We need some objective means to measure the size of inputs to any such algorithm.

Recall from Definition 3 that the domain of each initial state of an effective implementation is identified with the Herbrand universe over a given set of constructors K. Thus, domain elements may be represented as terms over the constructors K. Now, we need to measure the "size" of input values y, represented as constructor terms. The standard way to do this would be to count the number of symbols $|y|$ in the constructor term for y. A more conservative way is to count the minimal number of constructors required to access it, which is what we propose to do. This measure is proportional to the number of basic steps that would be required to create that value.

Definition 8 (Size)**.** The *(compact) size* $\|t\|$ of a term t over vocabulary K is as follows:

$$\|t\| = |\{s : s \text{ is a subterm of } t\}|.$$

For example, $|f(c,d)| = 3$, whereas $\|f(c,c)\| = 2$.

One may wish to add a logarithmic factor to this measure to account for the need to point to the shared subterms. We defer discussion of this issue until later.

So now we have a way to measure the domain elements of effective implementations. But there is one more issue to consider: The same domain may possess infinitely many constructors and, thus, infinitely many representations. For example, consider the complexity of implementations over $\mathbb{N}$. We are accustomed to say that the size of $n \in \mathbb{N}$ is $\lg n$, relying on the binary representation of natural numbers. This, despite the fact that the implementation itself may make use of tally notation or any other representation.

Consider now that somebody states that he has an effective implementation over $\mathbb{N}$, while working under the supposition that the size of n ought to be measured by $\log\log n$. Should this be legal? We neither allow nor reject such statements blindly, but require justification. This requires our preceding notion of equivalent implementations.

Definition 9 (Valid Size)**.** Let A be an effective implementation over domain D. A function $f : D \to \mathbb{N}$ is a *valid size* measure for elements of D if there is an effective implementation B over D such that A and B are computationally equivalent via some bijection ρ, such that $f(x) = \|\rho(x)\|$ for all $x \in D$.

Switching representations of the domain, one actually changes the vocabulary and thus the whole description of the implementation. Still we want to recognize the result as being the "same" implementation, doing the same job, even over the different vocabulary.

3.3 Complexity

It is standard to consider the application of a constructor to existing objects, so as to form a new object, to be an atomic operation of unit cost. Examples include successor for recursive functions, CONS for Lisp, and adding a letter to a string for models like Turing machines. Similarly, destructors (projections) like predecessor, CAR and CDR, and removing a letter are often considered atomic.

Though equality is not always considered atomic (in recursive functions, for instance), determining which constructor was used to create a given object is. For example, testing for zero is an atomic operation for recursive functions, as is testing for an empty list in Lisp, and determining which letter of the alphabet is at the end of a string. In pointer-based languages, testing equality of pointers is atomic, but equality of graph structures is not.

In what follows, we apply analogous considerations to our generic constructor framework.

Definition 10 (Time Complexity)**.** Let A be an effective implementation over domain D. A function $T : \mathbb{N} \to \mathbb{N}$ is a *time complexity* function for A if there exists a valid size $f : D \to \mathbb{N}$ and a basic implementation simulating A such that $T(n)$ is the maximal number of steps in a run of B with initial data $d \in D$ such that $f(d) = n$.

This definition counts steps, though a step may involve more than one action. But, thanks to Postulate III, algorithms can only do a bounded amount of work per step, so the number of steps is within a linear factor of the number of operations performed in a run. One may wish to charge a logarithmic factor per operation, since they may be applied to arbitrarily large values; we will return to this point later.

A notion of space complexity is a bit more subtle. We postpone a detailed treatment for subsequent work.

4 Simulation

We know from [4, Thm. 3] that, for any effective model, there is a string-representation of its domain under which each effective implementation has a Turing machine that simulates it, and—by the same token—there are RAM simulations. Our goal now is to prove that, given a basic implementation P with complexity $T(n)$, there exists a RAM M such that

- P is simulated by M (in the sense of Definition 6) via some representation ρ;

- $T_M(n) = O(T(n)^k)$ for some $k \in \mathbb{N}$,

where T_M measures M's steps. In what follows, we will describe a RAM algorithm satisfying these conditions. We will view the machine's memory as forming a graph (pointer) structure. The desired result will then follow from the standard poly-time (cubic) connection between TMs and RAMs.

First, we explain what an ASM program looks like. Then we will need to choose an appropriate RAM representation for its domain of terms.

4.1 Abstract State Machines

Abstract state machines (ASMs) [8, 9] provide a language for descriptions of algorithmic transition functions.

Definition 11 (Program [8])**.** *ASM programs*, over a vocabulary F, are composed of assignments and conditionals.

- A generalized assignment statement $f(s^1,\ldots,s^\ell) := u$ involves terms $u, s^1,\ldots,s^\ell$ over F. Applying it to a state X changes the interpretation that the state gives to f at the point $(s^1_X,\ldots,t^\ell_X)$ to be u_X. The result is an algebra X' such that $t_{X'} = (t[f(s^1,\ldots,s^\ell) := u])_X$ for any term t over F, where $t[s := u]$ denotes the term obtained from t by simultaneous replacement of all occurrences of the subterm s in t by u.
- Program statements may be prefaced by a conditional test, **if** c **then** p or **if** c **then** p **else** q, where c is a Boolean combination of equalities between terms. Only those branches of conditional statements whose condition evaluates to TRUE are executed.
- Furthermore, statements may be composed in parallel.
- The program, as such, defines a single transition, which is executed repeatedly, as a unit, until no assignments are enabled. If no assignments are enabled, then there is no next state.

Denote by Alg_F the class of all algebras over F. An ASM-program $\mathscr{A}$ defines a partial transition function $\mathscr{A} : \mathrm{Alg}_F \rightharpoonup \mathrm{Alg}_F$ such that $\mathscr{A}(X)$ is the result of applying $\mathscr{A}$ to X.

Definition 12 (ASM)**.** Let $\mathscr{A}$ be an ASM program over F and let $S_0 \subseteq S \subseteq \mathrm{Alg}_F$. Then, $A = \langle \mathscr{A}, S, S_0 \rangle$ is called an *abstract state machine (ASM)* if it is an algorithm in the sense of Definition 1.

Every algorithm is emulated step-by step, state-by-state by an ASM.

Theorem 13 ([9])**.** *Let $\langle \tau, S, S_0 \rangle$ be an algorithm over vocabulary F. Then there exists an ASM $\langle \mathscr{A}, S, S_0 \rangle$ over the same vocabulary, such that $\tau = \mathscr{A} \upharpoonright_S$, with the terms (and subterms) appearing in the ASM program serving as critical terms.*

Remark 14. *An ASM can have conflicting assignments, that is, inconsistent values being assigned to the same location, a situation called a "clash". In the prevalent ASM semantics, a clash would cause the algorithm to abort. But the above theorem guarantees the existence of a clash-free ASM emulating any clash-free algorithm. So, we may henceforth safely ignore any possibility of such clashes.*

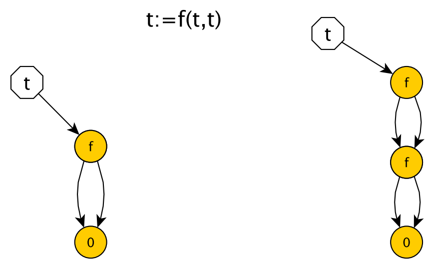
Definition 15 (ESM)**.** An *effective state machine (ESM)* is an effective implementation of an ASM. It is *basic* when the implementation is.

What is crucial for us is the observation that every basic implementation is emulated by such an ESM, with the above instruction set. Note that the constructors themselves need not appear in an ESM program, though they are part and parcel of its states.

4.2 Tangles

One's first inclination might be to represent terms as strings in a model like Turing machines. To see the problem resulting from such a naïve representation, consider a simple ESM, consisting of one unconditional assignment, $t := f(t,t)$. It is easy to see that k iterations would multiply the length of the string representing t by 2^k. Thus, we get an exponential growth of data size, which would prevent poly-time simulation.

Clearly, this exponential blowup is due to the ability of an ESM to create several copies of existing data in one transition, in contradistinction to a TM, regardless of the size of this data. To prevent the consequent unwanted growth, we must avoid creating multiple copies of the same subterms, using pointers to shared locations instead:



This structure is a fully collapsed "term graph", which we call a *tangle*. For more on term graphs—that is, rooted, labelled, ordered, directed acyclic graphs, see [14].

Let $\mathbb{G}(K)$ be all tangles over vocabulary K, for constructors K.

4.3 States

Throughout this section, let X be a constructive algebra over vocabulary $F = K \uplus C$, with domain D and constructors K (as in Definition 4). We describe the state of a RAM over the same signature F, but with domain $\mathbb{G}(K)$, simulating X via the injection $\rho : D \rightarrowtail \mathbb{G}(K)$ that maps elements of D to the tangle of their unique constructor term.

Tangles will be implemented in RAMs by using matrices of pointers, where pointers are just natural numbers in $\mathbb{N}$. The pointer structure of a domain element is unique, but the numerical value of the pointers themselves will vary from run to run. For any particular run of an algorithm, let $\iota : D \rightarrowtail \mathbb{N}$ give the value of the pointer to the appropriate tangle constructed in that run.

We view (the graph of) an arity-ℓ function f as an infinite ℓ-dimensional matrix. If $f(d^1,\ldots,d^\ell) = d^0$, then the value of the matrix's entry indexed by $[d^1,\ldots,d^\ell]$ is d^0. Actually, the indices are not domain elements, per se, but numerical values of pointers to the appropriate tangles. In other words, we require that f and ι commute: $f(\iota(d^1),\ldots,\iota(d^\ell)) = \iota(f(d^1,\ldots,d^\ell))$. Note that constants are nullary functions and thus are associated with a matrix of dimension 0, with one unique entry, accessed via the constant name, directly, without recourse to indices. The benefit of this representation is described in the following proposition.

Proposition 16 ([16]). *Multidimensional arrays can be organized in the memory of a RAM in such a way that one can access an entry indexed by $[i^1,\ldots,i^\ell]$ in $O(\log i^1 + \ldots + \log i^\ell)$ RAM-time, whether it is a first-time access or a subsequent one.*

The trick is to segment the memory into big chunks in such a way that one can compute squares of indices, hence, cell locations, without need for multiplication; see [16] for details. One can also have more than one array in such a RAM.

We start by describing properties of the domain, as constructed in a RAM.

Proposition 17. *Let $\widetilde{D}$ be a finite set of matrices with natural-number values and indices, such that there exists a bijection between matrices and constructors, satisfying the following:*

1. *Each constant has an integer value associated with it.*
2. *Entry values constitute a proper subset of index values (each domain element can be constructed from the constants).*
3. *No repeated entry values (unique representation).*

Then there exists some injection $\iota : D \rightarrowtail \mathbb{N}$, such that $\widetilde{D}$ represents D over it. We will say that $\widetilde{D}$ mimics *D.*

We continue with the description of state simulation.

Proposition 18. *Let $\widetilde{X}$ be a state of RAM such that the following holds:*

1. *State $\widetilde{X}$ has an ℓ-dimensional matrix for each arity-ℓ function in the vocabulary of X.*
2. *The set $\widetilde{D}$ of matrices associated with the constructors satisfies the requirements of Proposition 17.*
3. *For any nonconstructor ℓ-ary function g there exists a ℓ-dimensional matrix in $\widetilde{X}$, also named g, such that the following holds: whenever $g(d^1,\dots,d^\ell) = d^0$ in X and $d^0,\dots,d^\ell$ are represented by natural numbers $c^0,\dots,c^\ell$ using set $\mathscr{D}$ of matrices in $\widetilde{X}$, then the value of the entry in g indexed by $d^1,\dots,d^\ell$ is d^0.*

Then $\widetilde{X}$ mimics X.

Whenever we say that $\widetilde{X}$ mimics a constructive algebra X, we mean it in the sense of this proposition.

Note that a basic state (Definition 3) possesses only finite non-trivial information and thus refers to only finitely many domain elements. Thus, it should be mimicked by a RAM state with a finite description, as well.

Proposition 19. *Assume that X is a basic algebra. Let $\widetilde{X}$ be a finite state of RAM such that the following hold:*

1. *There exists a state $\widetilde{X}'$ simulating X in a sense of Proposition 18.*
2. *State $\widetilde{X}$ shares with $\widetilde{X}'$ nonconstructor matrices ($\widetilde{X}$ mimics all non-trivial data).*
3. *Values appearing in nonconstructor matrices of $\widetilde{X}$ appear also in domain matrices.*

Then $\widetilde{X}$ mimics X.

This is what we will mean whenever we mention that $\widetilde{X}$ mimics a basic algebra X.

So now we know how to mimic a basic algebra. To mimic a state of a basic ESM, we will have to store additional information in the direct-access RAM.

Definition 20. Let $\langle \mathscr{A}, S, S_0 \rangle$ be a basic ESM over vocabulary $F = K \uplus C$. Let X be a state of $\mathscr{A}$. We say that a finite state $\widetilde{X}$ of RAM *mimics X* if the following hold:

1. State $\widetilde{X}$ mimics X in the sense of Proposition 19.
2. For each nonconstructor constant $c \in C$, there is a pointer named c that points to the value of c.
3. An additional counter $\textsc{max} \in \mathbb{N}$ contains the maximal value of the pointers to domain elements used so far.

Any time we mention that $\widetilde{X}$ mimics a basic state X, we mean the simulation in the sense of this proposition.

From now on, let $m = \lg \textsc{max}$.

To see why it is enough to have pointers for constants only, and not for the whole set of critical terms, consider the following lemma.

Lemma 21. *Let $\langle \mathscr{A}, S, S_0 \rangle$ be a basic ESM over vocabulary $F = K \uplus C$. Let X be its state and $\widetilde{X}$, a finite state of RAM simulating X. Let t be some term appearing in the description of $\mathscr{A}$. Then the value t_X can be computed over $\widetilde{X}$ in $O(m)$ RAM-time.*

Proof. The proof is based on the fact that, to compute the value of term, it is enough to compute the values of all its subterms. These are bounded by the description of $\mathscr{A}$. And an application to any of the matrices is bounded by $O(m)$, as per Proposition 16, because $\textsc{max}$ is an upper bound on pointer values. □

4.4 Transition

Throughout this section, let $\langle \mathscr{A}, S, S_0 \rangle$ be a basic ESM over vocabulary $F = K \uplus C$, with input terms $I \subseteq J$. Consider some run $X_0 \rightsquigarrow X_1 \rightsquigarrow \cdots$ of $\mathscr{A}$.

Lemma 22 (Building Initial States)**.** *Let X_0 be an initial state of basic ESM $\mathscr{A}$ with input data I. Let $n = \|I\|$. Then there exists a constant c, not dependent on X_0 and I and state $\widetilde{X}_0$ of RAM simulating X_0, using $O(n \log n)$ space such that $|\textsc{max}| \leq n + c$.*

Proof. Recall that $\textsc{max}$ is a variable containing the upper bound of the representations of the domain elements used. Recall from the description of an ESM that the initial state contains input values and some finite initial data, which is the same for all the initial states and does not depend on the input. Recall, also, that each domain element is a term over a constructor vocabulary. To insert a domain term $f(t^1, \ldots, t^\ell)$ in a matrix, one needs a representation for each t^i. So to represent a term, one must represent all its subterms. Thus, a representation of domain term t requires values in $\|t\|$ entries of matrices, that is, $\|t\|$ natural numbers. Representing input values requires at most n natural numbers (some input values may have common subterms, so that the bound is not necessary reached). Also, since the other data encoded in the initial state is finite, relative to the input, we may assume that it requires at most c additional integers. Hence, to mimic the initial state, it is enough to use $n + c$ natural numbers. Taking them to be successive integers starting with zero, we conclude that $|\textsc{max}| \leq n + c$. □

Lemma 23 (Simulating Transitions)**.** *Let $\widetilde{X}_i$ be a RAM state simulating a basic state X_i. Then a RAM state $\widetilde{X}_{i+1}$ simulating a basic state X_{i+1} can be constructed in $O(m)$ RAM-time. Also there exists a constant b (which depends on $\mathscr{A}$ only and not on X_i) such that* $\textsc{max}$ *is increased by at most b (that is, the new counter value is greater then the prior one by at most b).*

Proof. To construct $\widetilde{X}_{i+1}$, we have to simulate the transitions described by an ESM $\mathscr{A}$. To do so, we must simulate a finite number of predefined comparisons of the values of critical terms, and—based on the result—we have to perform a set of assignments of new values (see Definition 11). Since the number of comparisons and assignments has a finite bound fixed by the algorithm, it is enough to estimate the complexity of one comparison and one assignment.

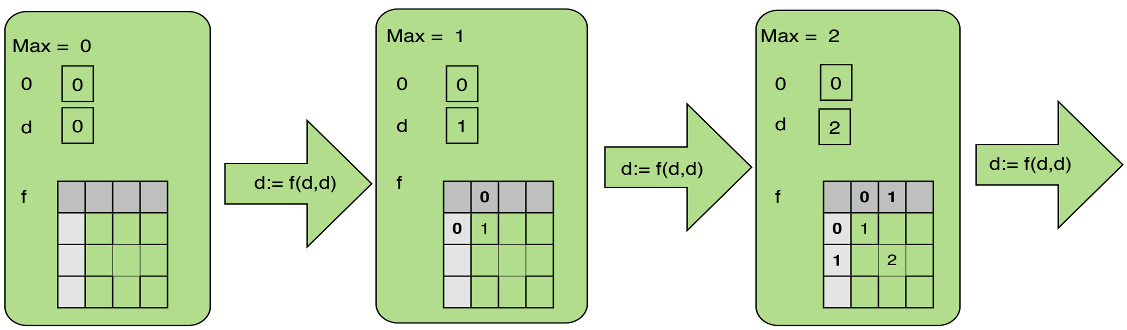
Recall that a comparison is a Boolean operation that requires comparisons of values of pairs of critical terms appearing in the description of $\mathscr{A}$. It thus follows from Lemma 21 that this operation can be performed in $O(m)$ RAM-time.

An assignment $f(t^1,\dots,t^\ell) := t^0$ causes the value of f at location $[t^1_X,\dots,t^\ell_X]$ to become t^0_X. It follows again from Lemma 21 that computing $[t^1_X,\dots,t^\ell_X]$ is $O(m)$. If t^0_X does not involve constructors, then the same is true for it. In case it does, it may create new domain elements and thus new integers will be required to represent them. Their number is bounded by the number of constructors used in the description of t^0 and thus is constant. Hence, the overall process is again $O(m)$ RAM-time.

Denote by b the number of constructors appearing in the description of $\mathscr{A}$. It follows from the above that MAX is increased by at most b after the overall process. The resulting state $\widetilde{X}_{i+1}$ mimics basic state X_{i+1}. The construction required $O(m)$ RAM-time, as desired. □

We demonstrate the evaluation of a RAM simulating ESM with the following artificial example.

Example 24. Consider a simple ESM defined by the one-line program $d := f(d,d)$ over a vocabulary $\Sigma = \{0, f\cdot,\cdot)\} \cup \{d\}$, where 0 and f are constructors of $\mathscr{A}$ and d is a constant accumulating the result of the computation. Assume that we start with with $d = 0$ and f undefined. Then the transition of RAM will be as follows:



Note that the length of the string representation of a domain element accessed at step i of an ESM run is $O(2^i)$. Thus, any attempt to store the string representation of domain elements would require RAM memory exponential in the number of the ESM's simulated steps. Since the memory requirement is a lower bound on complexity, we conclude that a RAM would require at least an exponential-time overhead to simulate the ESM. We overcome this difficulty by counting accessed elements, but not saving their actual representations as domain terms.

4.5 Complexity

Theorem 25 (Time)**.** *Any effective implementation with time complexity $T(n)$, with respect to a valid size measure, can be simulated by a RAM in order $O(T(n))$ steps, with a word size that grows up to order $\log T(n)$.*

Proof. Let $\langle \tau, S, S_0 \rangle$ be a basic implementation, simulating the given effective one with complexity $T = T(n)$. Let $\langle \mathscr{A}, S, S_0 \rangle$ be a basic ESM simulating it step-for-step. Let X_0 be an initial state with input I of size n. Then a run $X_0 \rightsquigarrow X_1 \rightsquigarrow \cdots$ of $\mathscr{A}$ starting from X_0 has at most T states. It follows from Lemma 22 that there exists a RAM state $\widetilde{X}_0$ simulating X_0, using $O(n\log n)$ space such that $|\text{MAX}| \le n + c$ (where c is some constant defined by $\mathscr{A}$ itself and not dependent on the choice of initial state). The proof continues by induction on the number of steps. Assume RAM state $\widetilde{X}_i$ mimics X_i using $O(n\log n + i\log i)$ space and that $|\text{MAX}| \le n + id + c$. It follows then from Lemma 23 that a RAM state $\widetilde{X}_{i+1}$ simulating X_{+1} can be constructed in $O(\log n + \log i)$ RAM-time with MAX increased by at most d. Thus in $\widetilde{X}_{i+1}$ we have that $|\text{MAX}| \le n + (i+1)d + c$. So we may conclude that we have to simulate T transitions, each of them requiring at most $O(\log n + \log T)$ RAM-time. So the overall simulation will take $O(n \log n + T \log T)$ RAM-time. Assuming that a run is long enough, we may conclude that the cost is $O(T \log T)$ time. The desired result thus follows. □

5 Discussion

We have shown—as has been conjectured and is widely believed—that every effective implementation, regardless of what data structures it uses, can be simulated by a Turing machine, with at most polynomial overhead in time complexity.[3]

To summarize the argument in a nutshell: Any effective algorithm is behaviorally identical to an abstract state machine operating over a domain that is isomorphic to some Herbrand universe, and whose term interpretation provides a valid measure of input size. That machine is also behaviorally identical to one whose domain consists of compact dags, labeled by constructors. Each basic step of such a machine, counting also the individual steps of any subroutines, increases the size of a fixed number of such compact dags by no more than a constant number of edges. Lastly, each machine step can be simulated by a RAM that manipulates those dags in time that is linear in the size of the stored dags.

Specifically, we have shown that any algorithm running on an effective sequential model can be simulated, independent of the problem by a multi-tape Turing machine with little more than cubic overhead: linearithmic for the RAM simulation of the ESM and quadratic for a TM simulation of the RAM [6].

In basic effective implementations, lookup of values of defined functions is an atomic operation, as is testing equality of domain elements of arbitrary size. This may be unrealistic, of course, in which case the overhead of the simulation may be considered lower than we have determined. Just as it is standard 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 equality test to test if $x = y$, that is if x and y point to the same domain value. Though we did not include (unit-cost) destructors in basic implementations, that should be possible, since destructors can be added to the RAM model without undue overhead.

In the simulation we have given, space-usage grows whenever a new domain value is accessed, and that space is never reclaimed. For the simulation to preserve the space complexity of the original algorithm, some version of "garbage collection" would be necessary, with a concomitant increase in the number of steps. Further work is needed to quantify the space-time tradeoff involved.

Our result does not cover large-scale parallel computation, such as quantum computation, as it posits that there is a fixed bound on the degree of parallelism, with the number of critical terms fixed by the algorithm. The question of relative complexity of parallel models is a subject for future research.

References

[1] Scott Aaronson (2006): *Quantum Computing since Democritus*. Lecture notes. Available at `http://www.scottaaronson.com/democritus/lec4.html` (viewed June 3, 2011).

[2] Andreas Blass, Nachum Dershowitz & Yuri Gurevich (2010): *Exact Exploration and Hanging Algorithms*. In: *Proceedings of the 19th EACSL Annual Conferences on Computer Science Logic (Brno, Czech Republic)*, *Lecture Notes in Computer Science* 6247, Springer, Berlin, Germany, pp. 140–154, doi:10.1007/978-3-

[3] We do *not* muster support for the version framed by Leonid Levin [12]: "The Polynomial Overhead Church-Turing Thesis: Any computation that takes t steps on an s-bit device can be simulated by a Turing Machine in $s^{O(1)}t$ steps within $s^{O(1)}$ cells." Presumably, the models we have in mind are what Levin deems "extravagant".

642-15205-4_14. Available at http://nachum.org/papers/HangingAlgorithms.pdf (viewed June 3, 2011); longer version at http://nachum.org/papers/ExactExploration.pdf (viewed May 27, 2011).

[3] Udi Boker & Nachum Dershowitz (2006): *Comparing Computational Power*. *Logic Journal of the IGPL* 14(5), pp. 633–648, doi:10.1007/978-3-540-78127-1.

[4] Udi Boker & Nachum Dershowitz (2008): *The Church-Turing Thesis over Arbitrary Domains*. In Arnon Avron, Nachum Dershowitz & Alexander Rabinovich, editors: *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, *Lecture Notes in Computer Science* 4800, Springer, pp. 199–229, doi:10.1007/978-3-642-15205-4_14. Available at http://nachum.org/papers/ArbitraryDomains.pdf (viewed Aug. 11, 2010).

[5] Udi Boker & Nachum Dershowitz (2010): *Three Paths to Effectiveness*. In Andreas Blass, Nachum Dershowitz & Wolfgang Reisig, editors: *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, *Lecture Notes in Computer Science* 6300, Springer, Berlin, Germany, pp. 36–47, doi:10.1007/978-3-642-15025-8_7. Available at http://nachum.org/papers/ThreePathsToEffectiveness.pdf (viewed Aug. 11, 2010).

[6] Stephen A. Cook & Robert A. Reckhow (1973): *Time-Bounded Random Access Machines*. *Journal of Computer Systems Science* 7, pp. 73–80, doi:10.1145/800152.804898. Available at http://www.cs.berkeley.edu/~christos/classics/Deutsch_quantum_theory.pdf (viewed June 3, 2011).

[7] Nachum Dershowitz & Yuri Gurevich (2008): *A Natural Axiomatization of Computability and Proof of Church's Thesis*. *Bulletin of Symbolic Logic* 14(3), pp. 299–350, doi:10.2178/bsl/1231081370. Available at http://nachum.org/papers/Church.pdf (viewed Apr. 15, 2009).

[8] Yuri Gurevich (1995): *Evolving Algebras 1993: Lipari Guide*. In Egon Börger, editor: *Specification and Validation Methods*, Oxford University Press, pp. 9–36. Available at http://research.microsoft.com/~gurevich/opera/103.pdf (viewed Apr. 15, 2009).

[9] Yuri Gurevich (2000): *Sequential Abstract State Machines Capture Sequential Algorithms*. *ACM Transactions on Computational Logic* 1(1), pp. 77–111, doi:10.1145/343369.343384. Available at http://research.microsoft.com/~gurevich/opera/141.pdf (viewed Apr. 15, 2009).

[10] Stephen C. Kleene (1952): *Introduction to Metamathematics*. D. Van Nostrand, New York, doi:10.1007/978-0-8176-4769-8.

[11] Stephen C. Kleene (1987): *Reflections on Church's Thesis*. *Notre Dame Journal of Formal Logic* 28(4), pp. 490–498, doi:10.1305/ndjfl/1093637645.

[12] Leonid A. Levin (2003): *The Tale of One-Way Functions*, doi:10.1023/A:1023634616182. Available at http://arXiv.org/pdf/cs/0012023v5, (viewed June 3, 2011).

[13] Ian Parberry (1986): *Parallel Speedup of Sequential Machines: A Defense of Parallel Computation Thesis*. *SIGACT News* 18(1), pp. 54–67, doi:10.1145/8312.8317.

[14] Detlef Plump (1999): *Term Graph Rewriting*. In H. Ehrig, G. Engels, H.-J. Kreowski & G. Rozenberg, editors: *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools*, chapter 1, volume 2, World Scientific, pp. 3–61, doi:10.1142/9789812815149. Available at http://www.informatik.uni-bremen.de/agbkb/lehre/rbs/texte/Termgraphrewriting.pdf (viewed June 3, 2011).

[15] Wolfgang Reisig (2008): *The Computable Kernel of Abstract State Machines*. *Theoretical Computer Science* 409, pp. 126–136, doi:10.1016/j.tcs.2008.08.041. Available at http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2004_hub_tr177.pdf (viewed June 3, 2011).

[16] John Michael Robson (1990): *Random Access Machines with Multi-dimensional Memories*. *Information Processing Letters* 34, pp. 265–266, doi:10.1016/0020-0190(90)90133-I.

[17] Comandure Seshadhri, Anil Seth & Somenath Biswas (2007): *RAM Simulation of BGS Model of Abstract-State-Machines*. *Fundamenta Informaticae* 77(1–2), pp. 175–185. Available at http://www.cse.iitk.ac.in/users/sb/papers/asm2ram.pdf (viewed June 3, 2011).

[18] Andrew C. Yao (2003): *Classical Physics and the Church-Turing Thesis*. Journal of the ACM 77, pp. 100–105, doi:10.1142/9789812815149. Available at http://eccc.hpi-web.de/eccc-reports/2002/TR02-062/Paper.pdf (viewed June 3, 2011).