A Formalization and Proof of the Extended Church-Turing Thesis

—Extended Abstract—

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, representing data as a minimal term graph.

1 Introduction

The Church-Turing Thesis asserts that all effectively computable numeric functions are recursive and, likewise, they can be computed by a Turing machine, or—more precisely—can be simulated under some representation by a Turing machine. This claim has recently been axiomatized and proven [3, 6]. The "extended" thesis adds the belief that the overhead in such a simulation is polynomial. One formulation of this extended thesis is as follows:

> The Extended Church-Turing Thesis states ... that time on all "reasonable" machine models is related by a polynomial. (Ian Parberry [9])

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

1. We adopt the axiomatic characterization of (sequential) *algorithms* over arbitrary domains due to Gurevich [8] (Section 2, Definition 1).
2. We adopt the formalization of *effective* algorithms over arbitrary domains from [3] (Section 2, Definition 4).
3. We adopt the definition of *simulation* of algorithms in different models of computation given in [2].
4. We consider *implementations*, which are algorithms operating over a specific domain (Section 2, Definition 2).
5. We represent domain elements by their minimal constructor-based graph (dag) representation; cf. [11] (Section 3).
6. We measure the size of input as the number of vertices in a constructor-based representation (Section 3, Definition 6).

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

Submitted to:
DCM 2011

7. We emulate effective algorithms step-by-step by abstract state machines [8] in the precise manner of [1] (Section 4, Section 4).
8. Each basic implementation step can be simulated in a linear number of random-access machine (RAM) steps (Section 5, Theorem 15).
9. Input states to the simulation can be encoded in linearly many RAM steps (Section 5, Theorem 14).
10. As multitape Turing machines simulate RAMs in quadratic time [4], the thesis follows (Section 6).

2 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 ensures that each state is self-contained and that the next state, if any, is determined. 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 should possess a finite description.

Definition 1 (Algorithm [8])**.** 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. 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 any $X \in S$. 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 also 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 $\tau(X) \setminus X = \tau(Y) \setminus Y$.

For detailed support for this characterization of algorithms, see [8, 6]. Clearly, we are only interested here in deterministic algorithms. We use 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.

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* $S_D \subseteq S_0$ are those initial states whose domain is D; its transition function τ is likewise restricted.

[1]Or class—it doesn't matter.

[2]All "terms" in this paper are ground (i.e. variable-free).

So we may view implementations as computing a function over its domain.

In the following, we will always assume a predefined subset $I \cup \{z\}$ of the critical terms, called *inputs* and *output*, respectively. Input states may differ only on input values and input values must cover the whole domain. Then we may speak of an algorithm A with terminating run $X_0 \rightsquigarrow_A \cdots \rightsquigarrow_A X_N$ as computing $A(y^1_{X_0}, \ldots, y^k_{X_0}) = z_{X_N}$. 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.

The postulates in Definition 1 limit transitions to be effective, in the sense of being programmable, as we just saw, but they place no constraints on the contents of initial states. In particular, initial states may contain infinite, uncomputable data. To preclude that, we will need an additional assumption.

Definition 3 (Basic)**.** We call an algebra X over vocabulary F and with domain D *basic* if $F = K \uplus J$, D is isomorphic to the Herbrand universe (free term algebra) over K, the *constructors* of X, and $t_X = s_X \neq$ UNDEF for at most a finite number of terms t and s over $K \uplus J$, for some pervasive constant value UNDEF. An implementation is *basic* if all its initial states are basic with respect to the same constructors.

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 ε and unary constructors a($\cdot$), b($\cdot$), etc. The positive integers in binary notation may be 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.

Definition 4 (Effectiveness [3])**.** Let X be an algebra over vocabulary F and domain D. We call X *effective* over $F = K \uplus C$ if K constructs D and each of the operations in C can be computed by an effective implementation over K. In other words, C is a set of effective oracles, obtained by bootstrapping from basic implementations. An *effective implementation* is a classical algorithm restricted to initial states that are all effective. over the same partitioned vocabulary $F = K \uplus C$.

Clearly, the properties of being basic or effective are closed under the transition of algorithm (this follows from Postulate III). Hence, any reachable state of basic (effective) implementation is also basic (effective, respectively).

3 Complexity

Complexity of an algorithm is classically measured as a number of single steps required by execution, relative to the size of the initial data. This requires an interpretation of the notions "initial data size" and "single step". By a "step", we usually mean a single step of some well-defined theoretical computational model, like a Turing machine or RAM, implementing an algorithm over a chosen representation of the domain.

An effective implementation may simulate an effective algorithm over a chosen representation of domain, but it still cannot count for a faithful measure of a single step, since its states are allowed to contain infinite non-trivial information as an oracle; unlike a basic implementation.

Basic implementations provide an underlying model for effective ones (and thus are a faithful measure of a single step):

Proposition 5. *Let $P = \langle \tau, S, S_0 \rangle$ be an effective implementation over $K \uplus C$. Then there exists a basic implementation simulating P over some vocabulary $K \uplus J$.*

The proof uses the notion of *simulation* defined in [2] and standard programming techniques of internalizing operations by bootstrapping.

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 a "pseudo-complexity" measure), but, rather, the number of basic decimal-digit operations, as would be counted in the basic simulation of the effective implementation.

With a notion of single step in hand, we are only left to define a suitable notion of input size. Let $P = \langle \tau, S, S_0 \rangle$ be an effective implementation with constructors K. Recall from Definition 3 that the domain of each $X \in S_0$ is identified with the Herbrand universe over 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. The more conservative way is to count the minimal number of constructors required to access it, which we propose to do. For example, we want the size of $f(c,c)$ to be 2, not 3.

Definition 6 (Size)**.** The *(compact) size* of a term t over vocabulary K is $\|t\| := |\{s : s \text{ is a subterm of } t\}$.

Still another issue to consider is this: a domain may be constructible by infinitely many different finite sets of constructors, which affects the measurement of size. 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 use tally (unary) notation or any other representation. Consider now that somebody states that she has an effective implementation over $\mathbb{N}$, 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 with blind eyes, but require justification.

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 vocabularies.

Definition 7 (Valid Size)**.** Let A be an effective implementation over domain D. A function $f : D \to \mathbf{N}$ is a *valid size* for elements of D if there is an effective implementation B over D such that A and B are computationally equivalent (each simulating the other) via some bijection ρ, such that $f(x) = |\rho(x)|$ for all $x \in D$.

4 Abstract State Machines

Abstract state machines (ASMs) [7, 8, 5] provide a perfect language for descriptions of algorithmic transition functions. They consist of generalized assignment statements $f(s^1, \ldots, s^k) := u$, conditional tests **if** C **then** P or **if** C **then** P **else** Q, where C is a Boolean combination of equations between terms, and parallel composition. A program as such defines a single transition; it is executed repeatedly, as a unit, until no assignments have their conditions are enabled. If no assignments are enabled, then there is no next state.

A triplet $\langle \mathscr{M}, S, S_0 \rangle$ is called *abstract state machine (ASM)* if S_0 are initial states and S are states of an ASM program $\mathscr{M}$, such that $\langle \mathscr{M}, S, S_0 \rangle$ satisfy the conditions for being an algorithm given in Definition 1. Every algorithm is emulated step-by step, state-by-state by an ASM.

Theorem 8 ([8])**.** *Let $\langle \tau, S, S_0 \rangle$ be an algorithm over vocabulary F. Then there exists an ASM $\langle \mathscr{M}, S, S_0 \rangle$ over the same vocabulary, such that $\tau = \mathscr{M} \restriction_S$, with the terms (and subterms) appearing in the ASM program serving as critical terms.*

Definition 9 (ESM)**.** An *effective state machine (ESM)* is an effective implementation of an ASM $\mathscr{M}$.

Constructors are part and parcel of the states, though they need not appear in an ESM program.

5 Simulation

We know from [3, Theorem 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 is to prove that it can be done at polynomial cost. We describe a RAM algorithm satisfying these conditions. The result will then follow from the standard poly-time (cubic) connection between TMs and RAMs. First, we need to choose an appropriate RAM representation for our domain of terms.

For term t, we denote the minimal graph representing it by $\widetilde{t}$ and the quantity of RAM memory required to store it by $|\widetilde{t}|$. These memory cells will each contain a small constant, indicating a vertex label or a pointer, corresponding to an edge in the graph. Note that since $\widetilde{t}$ is minimal, it does not contain repeated factors. To prevent repeated factors not just in one term, but in the whole state, we merge the individual term graphs into one big graph and call the resulting "jungle" a *tangle* (see [10]). The tangle will maintain the constructor-term values of all the critical terms of the algorithm. 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 dag G has three vertices, labeled f,g,c. Two edges point from f to c and the other two from g to c. Our two terms may be represented as pointers to the appropriate vertex: $G(t)$ refers to the f vertex and $G(s)$ to g, where we are using the notation $G(t)$ to refer to the vertex in G that represents the term t.

Proposition 10. *For any tangle G of terms over a finite vocabulary, we have $|E(G)| = O(|V(G)|)$.*

Let $\langle \mathscr{M}, S, S_0\rangle$ be a basic ESM over vocabulary $F = K \uplus J$, with input terms $I \subseteq J$, and critical terms $T = \{t^1, \dots, t^m\}$, including all their subterms, ordered from *small to big*. Also, let $X_0 \rightsquigarrow_{\mathscr{M}} X_1 \rightsquigarrow_{\mathscr{M}} \cdots$ be some run of $\mathscr{M}$, for which we let $\widetilde{T_i}$ denote the tangle of the domain values $\{t_{X_i} : t \in T\}$ of the critical terms in the i-th state X_i. For $\bar{t}$, a finite sequence or set of terms, we use the abbreviation $\|\bar{t}\| = \sum_{s \in \bar{t}} \|s\|$.

One transition of ESM involves a bounded number of comparisons of the values of critical terms. The cost for each is constant:

Proposition 11. *Let $\widetilde{T}$ be a critical tangle and let s and t be critical terms in T. Therefore, the question whether $\widetilde{t} = \widetilde{s}$, is decidable in constant number of RAM-operations of logarithmic word size.*

One transition of an ESM involves a bounded number of assignments. The cost of each assignment is linear:

Proposition 12. *Let $t = f(\bar{t})$ be a term over vocabulary K. Then $\widetilde{t}$ can be constructed using $O(\|\bar{t}\|)$ RAM-operations of logarithmic word size.*

Combining the previous propositions together, we may conclude:

Proposition 13. *The critical tangles grow by a constant amount in each step. So, $|\widetilde{T_i}| = O(|\widetilde{T_0}| + i)$.*

Theorem 14 (Initial States)**.** *Given term-graphs $\widetilde{I}$ for the inputs I in an initial state X_0, Algorithm 1 constructs the initial critical tangle $\widetilde{T_0}$ in $O(\|I\|)$ steps.*

Algorithm 1

```
• for $i = 1, \dots, m$
    – let $t^i = f(s^1, \dots, s^\ell)$
    – if all $s^j$ are defined, then
        ∗ if $f \in K$, create $f(s^1, \dots, s^\ell)$, as described in Proposition 12
        ∗ if $f \notin K$, then
            · if found $r^1, \dots, r^\ell, r \in \widetilde{T}$ such that $\widetilde{r^j} = \widetilde{s^j}$ for all $j$ and $r = f(r^1, \dots, r^\ell)$ is defined,
              then copy the content of $r$ to $t^i$
```

Theorem 15 (Transitions)**.** *Algorithm 2 computes the successor tangle $\widetilde{T}_{i+1}$ from $\widetilde{T}_i$ in time linear in $|\widetilde{T}_i|$.*

Algorithm 2

```
• for each critical term $t \in T$ create a new pointer $t'$ to point to its new value
• for each possible assignment in the ESM, do the following:
    – if all guards evaluate to TRUE, then
    – for the enabled assignment $f(s^1,\ldots,s^\ell) := s$ do
        ∗ if $s$ is UNDEF then $f'(s^1,\ldots,s^\ell)$ is also UNDEF
        ∗ if some $s^i$ is UNDEF then $f'(s^1,\ldots,s^\ell)$ is also UNDEF
        ∗ otherwise, if $f \in K$ then
            · set $f'(s^1,\ldots,s^\ell)$ to point to the graph constructed as described in Proposition 12
        ∗ whereas, if $f \notin K$, then if found $r^1,\ldots r^\ell, r \in \widetilde{T}$ such that $\widetilde{r}^j = \widetilde{s}^j$ for all $j$ and $r = f(r^1,\ldots,r^\ell)$ is defined, then
            · if $f'(s^1,\ldots,s^\ell)$ is not UNDEF, set $f'(s^1,\ldots,s^\ell)$ to point to a copy of $\widetilde{r}$
```

The result we set out to achieve now follows.

Theorem 16 (Simulating ESMs)**.** *Any effective implementation with complexity $T(n)$, with respect to a valid size measure, can be simulated by a RAM in order $n + nT(n) + T(n)^2$ steps, with a word size that grows to order $\log T(n)$.*

6 Summary

We have shown—as has been conjectured—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. Specifically, we have shown that any algorithm running on an effective sequential model can be simulated, independent of the problem, by a single-tape Turing machine with a quintic overhead: quadratic for the RAM simulation and another cubic for a TM simulation of the RAM [4].

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 maximally 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.

It remains to be seen whether it may be possible to improve the complexity of the simulation.

References

[1] 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:10.1007/978-3-642-15205-4_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).

[2] 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.

[3] 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).

[4] 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).

[5] Nachum Dershowitz (2012): *The Generic Model of Computation*. In: *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2012, Zurich, Switzerland)*, Electronic Proceedings in Theoretical Computer Science. Available at `http://nachum.org/papers/Generic.pdf` (viewed July 13, 2012).

[6] 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).

[7] 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).

[8] 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).

[9] 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.

[10] 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).

[11] 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).