The Logic in Computer Science Column

by

Yuri Gurevich

Computer Science and Engineering
University of Michigan, Ann Arbor, MI 48109, USA
gurevich@umich.edu

UNCONSTRAINED CHURCH-TURING THESIS CANNOT POSSIBLY BE TRUE

Yuri Gurevich

Abstract

The Church-Turing thesis asserts that if a partial strings-to-strings function is effectively computable then it is computable by a Turing machine.

In the 1930s, when Church and Turing worked on their versions of the thesis, there was a robust notion of algorithm. These traditional algorithms are known also as classical or sequential. In the original thesis, effectively computable meant computable by an effective classical algorithm. Based on an earlier axiomatization of classical algorithms, the original thesis was proven in 2008.

Since the 1930s, the notion of algorithm has changed dramatically. New species of algorithms have been and are being introduced. We argue that the generalization of the original thesis, where effectively computable means computable by an effective algorithm of any species, cannot possibly be true.

1 Introduction

This is an expanded and dialogized version of an October 21, 2018 talk at Microsoft Research Redmond. The characters A and Q are the author and his former student Quisani.

Q: There is a discussion about the Church-Turing thesis at the Computer Science Theory StackExchange [9]. It involves your paper [3] with Nachum Dershowitz where you prove the thesis. Peter Shor is skeptical about it:

> "The Dershowitz-Gurevich paper says nothing about probabilistic or quantum computation. It does write down a set of axioms about computation, and prove the Church-Turing thesis assuming those axioms. However, we're left with justifying these axioms. Neither probabilistic nor quantum computation is covered by these axioms (they admit this for probabilistic computation, and do not mention quantum computation at all), so it's quite clear to me these axioms are actually false in the real world, even though the Church-Turing thesis is probably true."

What do you say?

A: The Church-Turing thesis asserts that if a string function is effectively computable then it is computable by a Turing machine. Here a string function is a partial function from strings in a finite alphabet to strings in a finite alphabet.

In the 1930s, when Church and Turing worked on their versions of the thesis, there was a robust notion of algorithm. These traditional algorithms are known also as classical or sequential algorithms. It is this notion of algorithm which is axiomatized in [6]. In the original thesis, the effective computability of a string function means that it is computable by an effective classical algorithm. It is that original thesis which is proven in the Dershowitz-Gurevich paper [3].

Since the 1930s, many new species of algorithms have been introduced, and the notion of algorithm continues to evolve [7]. Apparently Peter Shor thinks that we pretend to prove the unconstrained version of the thesis, for the algorithms of all species, and that the unconstrained thesis is true.

Q: But surely the validity of the thesis is not restricted to the classical algorithms.

A: I believe that the thesis can be proven for a number of well-understood species of algorithms, in particular for algorithms in the quantum circuit model. But the unconstrained version of the thesis cannot possibly be true.

Q: Please explain.

2 The original Church-Turing Thesis

A: Let me quickly revisit the original thesis; details and relevant references are found at [3]. I will address the unconstrained version later.

2.1 Classical algorithms

The 1930s notion of algorithm was robust. People recognized algorithms when they saw them. These algorithms compute in steps, one step after another, and the steps are of bounded complexity [8]. Various names are used today for those algorithms: traditional, classical, sequential.

Q: None of the three names seems perfect to me. Tradition changes with time. "Classical" may mean merely not quantum. "Sequential" seems consistent with unbounded complexity of steps.

A: This is true. To distinguish between bounded and unbounded complexity of steps, we spoke about small-step and wide-step algorithms in [2]. But even that neglects the distinction between classical algorithms and algorithms interacting with their environment as well as the distinction between classical and learning algorithms.

Terminology 1. Classical algorithms *are algorithms in the sense of the 1930s (rather than merely non-quantum algorithms).*

Classical algorithms were analyzed and axiomatized in [6]. The analysis and axiomatization were refined in [1], mostly because the original analysis abstracted from details of intra-step computation.

2.2 The thesis

There are many equivalent formulations of the Church-Turing thesis. The Dershowitz-Gurevich article [3] was published in a logic journal. There, respecting logic tradition, we formulated the thesis in terms of partial recursive functions. Here, respecting computer science tradition, we formulate the thesis in terms of Turing machines.

Terminology 2.

- A *string function* is a partial function from strings in a finite alphabet to strings in a finite alphabet.
- A string function is *Turing computable* if it is computable by some Turing machine.

Now we can formulate the Church-Turing thesis succinctly. Here is a generic version of the thesis which leaves open what is meant by effective computability.

Thesis 0 (Generic Church-Turing thesis)**.** *If a string function is effectively computable then it is Turing computable.*

The approriate version of the original/classical thesis is this:

Thesis 1 (Classical Church-Turing thesis)**.** *If a string function is computed by an effective classical algorithm then it is Turing computable.*

Q: What does it mean exactly that an algorithm computes a string function?

A: Without loss of generality, we can define this in a way convenient for our purposes. An algorithm A computes a string function f if

- inputs of A are strings x in the input alphabet of f,
- if f is defined at x then the computation of A on x eventually converges and outputs $f(x)$,

- if f is not defined at x then the computation of A on x produces an error message or diverges, i.e., goes on forever.

Notice that this definition abstracts from limited resources. In the real world, a computation of an algorithm A on input x may break because we ran out of time or money or something else.

Q: What does it mean that an algorithm A is effective?

A: An algorithm A is effective if, given sufficient resources, the computation of A on any input x can be carried out in the real world.

Q: Show me some noneffective algorithms.

2.3 Noneffective classical algorithms

A: One example is Euclid's algorithm for lengths. You know Euclid's algorithm for natural numbers; given two natural numbers, the algorithm computes their greatest common divisor. Euclid used a similar algorithm for lengths; today we can think of lengths as nonnegative real numbers. Given two lengths, the algorithm finds their greatest common divisor if the two lengths are commensurable and diverges otherwise.

Another example is Gauss Elimination algorithm for real numbers.

Q: In both cases, reals can be approximated by rationals as closely as desired, and the computation on rationals can be carried out effectively.

A: This is true though the approximating algorithm will be much more involved, and there are some subtleties. For example, two reals may or may not be commensurable while any two rationals are commensurable. Besides, noneffective classical algorithms may be more abstract. For example, Gauss Elimination works over every field.

Q: Neither of the two noneffective algorithms computes a string function.

A: Oracle algorithms, which compute string functions, may be and often are non-effective. In particular, a Turing machine with an appropriate oracle solves the halting problem for oracle-free Turing machines.

Q: Using oracles looks like cheating.

A: But it may be useful. Turing used oracles machines already in 1939 [11].

2.4 Proving the original thesis

Q: Your proof of the thesis appeared only on 2008 [3]. How come that the thesis wasn't proven earlier?

A: One reason for that could be that it is easier to axiomatize all classical algorithms rather than only effective ones. The proof of the thesis builds on the axiomatization of classical algorithms in [6]

Q: But people could think of all classical algorithms earlier on.

A: It was natural to restrict attention to effective algorithms. Turing for example ignores noneffective algorithms completely in his thesis paper [10].

With time, software grew more involved, and software specifications started to use oracles and even work with genuine reals.

Q: Did you axiomatize algorithms with an eye on proving the Church-Turing thesis?

A: No, not at all. I introduced abstract state machines (originally called evolving algebras) and posited a thesis that every algorithm is an abstract state machine [5]. The purpose of the axiomatization in [6] was to prove the new thesis for classical algorithms.

Later, Nachum Dershowitz and I extended that axiomatization with an axiom saying essentially that there is no funny stuff in the initial state of the algorithm. This allowed us to derive the Church-Turing thesis [3].

3 Unconstrained Church-Turing thesis

A: Let's formulate the unconstrained thesis more explicitly.

Thesis 2 (Unconstrained Church-Turing thesis)**.** *If a string function is computed by any effective algorithm whatsoever then it is Turing computable.*

Now I am ready to posit my antithesis.

Antithesis. *The unconstrained thesis cannot possibly be true.*

Q: How do you justify the Antithesis?

A: Let me give you three arguments.

3.1 A moving target

My first argument is related to the evolution of the notion of algorithm. The notion of algorithm keeps evolving and getting more liberal [7]. This makes it a moving target.

In that sense it is analogous to the notion of number. We have already many species of numbers, e.g.,

- integers, rationals and reals,
- complex numbers and algebraic numbers,
- quaternions, octonions, sedenions,
- ordinal numbers and cardinal numbers,
- non-standard numbers, introduced by Abraham Robinson, and surreal numbers, introduced by John Conway.

And surely new species of numbers will be introduced. One should be careful about claiming that a property is common to all species of numbers.

Similarly, we have already many species of algorithms, e.g.,

- sequential and parallel algorithms,
- nondeterministic algorithms,
- real-time and analog algorithms,
- randomized and probabilistic algorithms,
- distributed algorithms,
- quantum algorithms,
- biology-inspired algorithms,
- learning algorithms.

And surely new species of algorithms will be introduced. One should be careful about claiming that a property is common to all species of algorithms.

Q: I cannot think of any intrinsic property of all numbers. Some but not all numbers are quantities, some but not all numbers represent orderings. Yet, as far as I know, addition and multiplication are defined for all species of numbers. This property seems to survive the introduction of new species of numbers; it may be common to all species of numbers, present and future. By analogy, there should be properties common to all species of algorithms, present and future. It is possible a priori that the validity of the Church-Turing thesis is such a property.

A: I will argue that this is not the case.

3.2 Engineering

Classical algorithms are mathematical objects. Large real-world algorithms of today are engineering systems. Typically they perform tasks and provide services, but sometimes they compute string functions as well. My second argument in favor of the Antithesis is that, in the case of large real-world algorithms, the Church-Turing thesis is sort of trivially true and therefore uninteresting.

Consider for example a popular industrial compiler for some common programming language, e.g. C++, which has been written by many people. Typically such a compiler runs on numerous platforms, but for simplicity let's fix a platform. The compiler computes a string function: In comes a source code, and out goes an object code or an error message.

Q: But are compilers algorithms?

A: Semantically, any software product is an algorithm, in my opinion. But notice that the generic Thesis 0 does not use the term algorithm. It is about effective computability. We could reformulate the unconstrained Thesis 2 by replacing "algorithm" with a term that sounds more inclusive, e.g. "computing system."

Q: A question arises whether the Church-Turing thesis holds for real-world algorithms — or computing systems — like compilers.

A: Any real-world compiler accepts only finitely many source programs. It doesn't accept source programs which are too long or too involved. The function computed by the compiler is finite and therefore recursive.

Q: This is disappointing. The thesis is true but uninteresting. Can we abstract from limited resources in this case?

A: Any popular industrial compiler is updated from time to time. Some bugs are fixed, and the new version may accept some source programs which had not been accepted earlier. Assume that the compiler will be updated forever and that there are infinitely many source programs P such that some version of the compiler accepts P.

Q: For our purposes, there is an ambiguity problem with such a continuously developing compiler. It does not compute a single-valued string function. Different versions may treat the same source program differently.

A: Furnish every application of (any version of) the compiler with a unique identity. Formally, the identity is a part of compiler input, and this way we solve the ambiguity problem. But the compilation process does not use the identity.

Q: The resulting string function does not seem to be Turing computable which challenges the Church-Turing thesis. But people may disagree that a continuously developing compiler is an algorithm or even a computing system.

A: This brings me to my third argument.

3.3 Changing attitude

Let me start with another example and then formulate my third argument in favor of the Antithesis.

Consider Google Translate [4] and fix some source language, say English, and some target language, say Russian. An English text (a query) is translated into Russian. I presume that every application of Google Translate is furnished with a unique identity. Such an application can be seen as a pair (X, Y) where X is a so-called unique query, i.e. an English text with the unique identity, and Y is the resulting translation to Russian. All such pairs (X, Y) form a function which I will call GT. The abstraction of unlimited resources renders GT infinite.

Q: I do not like when you apply the abstraction of unlimited resources to real-world systems. Companies come and go, and so do their tools. But at least in this case the abstraction looks more natural than in the compiler case. Even though Google Translate is continuously learning and thus continuously changing, or maybe because of this, it is more naturally perceived as one entity than a sequence of compiler versions.

A: Do you think that GT is Turing computable?

Q: Surely not. Let's suppose that a Turing machine T computes GT. Then T "knows" how English and Russian will develop, in particular what English slang will emerge and how it will be translated to Russian with its own new slang. This is absurd.

A: Would you consider GT effectively computable?

Q: Hmm, GT is certainly computable in practice. As a frequent user of Google Translate, I know that it works. Furthermore, it works fast, almost in no time. The translation may be poor but this is beside the point.

If effective algorithms are algorithms that work in practice then Google Translate is an effective algorithm. I am somewhat bothered that Google Translate is so different from algorithms of my college days. It is being trained on huge data. Its program keeps changing. What do you think?

A: My opinion is that practically computable functions like GT are effectively computable. My third argument in favor of the Antithesis is that this opinion will become more and more common. There is an informative analogy between the following two questions.

- Are practically computable functions effectively computable?
- Can machines think?

Here is an instructive quote of Turing [12, §6]:

> "The original question, 'Can machines think?' I believe to be too meaningless to deserve discussion. Nevertheless I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted."

Life is a better opinion-changer than arguments.

3.4 Finale

Q: Let me review your arguments to get a better overall picture. Your first argument in favor of the Antithesis is that the notion of algorithm is a moving target and therefore one should be cautious with universal claims about all algorithms.

Your second argument is that, in the case of large real-world programs, the Church-Turing thesis is at best uninteresting. This erodes the thesis somewhat but does not demolish it.

It is the third argument that is most damaging to the thesis. String functions like the translation function GT are practically computable but not Turing computable. So why you don't claim that the unconstrained thesis is plainly false?

A: My opponents may argue that Google Translate is not really an algorithm because, in addition to a given text in the source language, huge data has been used to train Google Translate, or because Google Translate keeps changing its program. In my view, the continuing progress will render these counterarguments less and less convincing.

Q: Do you expect that it will be recognized eventually that the unconstrained Church-Turing thesis is false?

A: This outcome is possible. Notice, however, that the thesis requires the unlimited-resources abstraction. In the case of Google Translate, this abstraction requires that Google Translate works forever. In the real world, the unlimited-resources abstraction is absurd and, I expect, will be viewed as such. The unconstrained thesis itself will be considered meaningless.

In either case, whether the unconstrained thesis is considered false or meaningless, it is not true, and so the Antithesis holds.

Acknowledgments

Many thanks to Andreas Blass for most useful discussions throughout my work on this dialog.

I am grateful also to my colleagues who took time to comment on the penultimate version of the dialog: Cris Calude, Anuj Dawar, Pierre Lescanne, Leonid Levin, Naphtali Rishe, Alexander Shen, Volodya Vovk.

References

[1] Andreas Blass, Nachum Dershowitz and Yuri Gurevich, "Exact Exploration and Hanging Algorithms," Springer Lecture Notes in Computer Science 6247 140–154 2010

[2] Andreas Blass and Yuri Gurevich, "Algorithms: A Quest for Absolute Definitions," in "Current Trends in Theoretical Computer Science: The Challenge of the New Century; Volume 2: Formal Models and Semantics" (eds. G. Paun et al.) World Scientific 283–311 2004. Reprinted in "Church's Thesis After 70 Years" (eds. A. Olszewski et al.) Ontos Verlag 24–57 2006

[3] Nachum Dershowitz and Yuri Gurevich, "A natural axiomatization of computability and proof of Church's thesis," Bulletin of Symbolic Logic 14:3 299–350 2008

[4] Google Translate, `https://translate.google.com/`, accessed Dec. 13, 2018.

[5] Yuri Gurevich, "Evolving algebra 1993: Lipari guide," in E. Börger (ed.), *Specification and validation methods* 9–36, Oxford University Press 1995. Also, arXiv:1808.06255

[6] Yuri Gurevich, "Sequential Abstract State Machines capture Sequential Algorithms," *ACM Transactions on Computational Logic* 1:1 77–111, 2000

[7] Yuri Gurevich, "What is an Algorithm?" in *SOFSEM 2012: Theory and Practice of Computer Science*, M. Bielikova et al. (eds), Springer Lecture Notes in Computer Science 7147 31–42 2012

[8] Andrei N. Kolmogorov, "On the concept of algorithm," Uspekhi Matematicheskikh Nauk 8:4 175–176 1953 (Russian). English version in Vladimir Uspensky and Alexei Semenov, "Algorithms: Main ideas and applications," Kluwer 1993, pp. 18–19

[9] StackExchange contributors, "What would it mean to disprove Church-Turing thesis?" Theoretical Computer Science StackExchange, Asked August 17, 2010, https://cstheory.stackexchange.com/questions/88/

[10] Alan M. Turing, "On computable numbers, with an application to the Entscheidungsproblem", Proceedings of London Mathematical Society Ser. 2 Vol. 42 230–265 1936, corrections, Ibid, Vol. 43 544–546 1937

[11] Alan M. Turing, "Systems of logic based on ordinals," *Proceedings of London Mathematical Society* Series 2 Vol. 45 161–228 1939

[12] Alan M. Turing, "Computing machinery and intelligence," Mind LIX:236 433–460 1950