WHAT IS AN ALGORITHM? (REVISED)

YURI GUREVICH

Abstract. We put the title problem and Church's thesis into a proper perspective, and we address some common misconceptions about Turing's analysis of computation. In addition, we comment on two approaches to the title problem, one well known among philosophers and another well known among logicians.

Foreword

This paper originated from the June 2011 talk "What is an Algorithm?" that the author gave in Krakow at the Studia Logica conference on Church's Thesis: Logic, Mind and Nature. The talk eventually gave rise to a two-part exposition on the notion of algorithm [10] and foundational analyses of computation [11]. This paper is a slight update of [10].

1. Introduction

Two articles in a recent book [13] offer different answers to the title problem. According to article [24],

> Moschovakis has proposed a mathematical modeling of the notion of algorithm — a set-theoretic "definition" of algorithms, much like the "definition" of real numbers as Dedekind cuts on the rationals or that of random variables as measurable functions on a probability space.... A characteristic feature of this approach is the adoption of a very abstract notion of algorithm that takes *recursion* as a primitive operation and is so wide as to admit "non-implementable" algorithms.

The other article [27] is written by Wilfried Sieg.

> Church's and Turing's theses dogmatically assert that an informal notion of effective calculability is adequately captured by a particular mathematical concept of computability. I present an analysis of calculability that

> . . . dispenses with theses. . . . The analysis leads to axioms for discrete dynamical systems (representing human and machine computations) and allows the reduction of models of these axioms to Turing machines.

In this paper we briefly discuss a number of related issues.
In §2, we discuss whether it is possible at all to define algorithms.
In §3, we discuss a related question whether Church's thesis can be proven.
In §4 we address some common misconceptions about Turing's analysis of computations.
In §5 we comment on the analysis of calculability presented by Sieg (but developed originally by Robin Gandy) and on Sieg's claim to dispense with Church's and Turing's theses.
In §6 we discuss what kind of entities algorithms are.
In §7 we comment on Moschovakis's recursion-based definition of algorithms.

Remark. There is also a question why bother to define algorithms. That is a different issue to be addressed elsewhere. Let us just say this. Understanding what algorithms are has practical applications, to software specification and model-based testing in particular. It also has theoretical application, e.g. to the algorithmic completeness of computation models.

Acknowledgments. Many thanks to Andreas Blass for numerous illuminating discussions, and to Serge Grigorieff and Oron Shagrir for useful suggestion.

2. Can the notion of algorithm be rigorously defined?

Our answer to the title question of this section is yes and no. Let us explain.

2.1. **The negative answer.**

In our opinion, the notion of algorithm cannot be rigorously defined in full generality, at least for the time being. The reason is that the notion is expanding.

Concerning the analogy of algorithms to real numbers, mentioned in the beginning of our introduction, Andreas Blass suggested a better analogy: algorithms to numbers. Many kinds of numbers have been introduced throughout history: natural numbers, integers, rationals, reals, complex numbers, quaternions, finite and infinite cardinals, finite and infinite ordinals, etc. Similarly many kinds of algorithms have been introduced. In addition to classical sequential algorithms, in use from antiquity, we have now parallel, interactive, distributed, real-time, analog, hybrid, quantum, etc. algorithms. New kinds of numbers and algorithms may be introduced and most probably will be. The notions of numbers and algorithms have not crystallized (and maybe never will) to support rigorous definitions.

2.2. **The positive answer.** However the problem of rigorous definition of algorithms is not hopeless. Not at all. Large and important strata of algorithms have crystallized and became amenable to rigorous definitions. This applies first of all to classical sequential algorithms, essentially the only algorithms in use from antiquity to the 1950s.

"Algorithms compute in steps of bounded complexity", wrote Andrei Kolmogorov in 1953 [19]. This is a good informal definition of classical sequential algorithms. An axiomatic definition of classical sequential algorithms has been given in [16] and explained in the companion paper [11].

The axiomatic definition of classical sequential algorithm was extended to synchronous parallel algorithms in [3] and to interactive sequential algorithms in [6, 7].

3. Classical and physical versions of Church's thesis

Originally Alonzo Church formulated his thesis for total numerical functions [9]; here numerical functions are functions from natural numbers to natural numbers. His student Stephen Cole Kleene extended the thesis to partial functions [18]; in the sequel functions, by default, may be partial. Kleene's version of Church's thesis became standard: Every computable numerical function is expressible in Gödel's recursive calculus. Gödel's recursive calculus is described in [18]; we will return to it in §7.

Alternatively, the thesis can be formulated in terms of Turing machines: Every computable function from binary strings to binary strings is computable by a Turing machine [30]. That form of Church's thesis is known as Turing's thesis.

The notion of computable numerical function and the notion of computable function from binary strings to binary strings are informal. Presumably computable functions are computed by algorithms. What kind of algorithms?

One natural answer is that the algorithms in question are classical sequential algorithms, the kind of algorithms that Church and Turing (and Gödel and other experts) had in mind at the time. This answer leads to the classical version of Church's thesis.

Another natural answer is that the notion of algorithm in question is the unrestricted notion discussed in §2.1. This interpretation gives rise to the so-called physical version of Church's thesis.

Nachum Dershowitz and this author used the axiomatic definition of classical algorithms, mentioned in §2.2, to derive the classical Church's thesis from first principles [8]. A similar derivation of the physical Church's thesis should wait until, if ever, the notion of algorithm in its full generality becomes amenable to formalization.

4. Remarks on Turing's analysis of computation

Turing's analysis of computation [30] was a stroke of genius. The analysis is extremely famous, and yet it is often misunderstood.

People often think that every computable function is Turing computable. This is not so even if we restrict attention to classical sequential algorithms. Here are some counter-examples. Consider Euclid's algorithm for computing the greatest common divisor $d = \gcd(a, b)$ of two natural numbers a, b.

```
let $M = \max(a, b)$,  $m = \min(a, b)$
while $M > m$ do
    $M, m := \max(M - m, m), \min(M - m, m)$
$d := M$.
```

The gcd function on natural numbers is of course Turing computable, but Euclid's algorithm was also applied — in theory and in practice — to the lengths of segments of a straight line, which gives rise to a computable partial function (the algorithm does not terminate if the two given lengths are incommensurable) that is not Turing computable because you cannot place an arbitrary length on the Turing tape. More generally, the functions computed by ruler-and-compass algorithms are not Turing computable. And let us emphasize that ruler and compass were practical tools in ancient Greece and that a number of ruler-and-compass algorithms were practical algorithms.

It is common in mathematics to consider algorithms that work on abstract objects. The functions computed by these algorithms may not be Turing computable. One example is Gaussian elimination. Here is another example: a bisection algorithm that, given a real $\varepsilon > 0$ and a continuous function f on a real segment $[a, b]$ with $f(a) < 0 < f(b)$, computes a point $c \in [a, b]$ with $|f(c)| < \varepsilon$.

```
while $|f((a+b)/2)| \geq \varepsilon$ do
    if $f((a+b)/2) < 0$ then $a := (a+b)/2$ else $b := (a+b)/2$
$c := (a+b)/2$
```

One can argue that these functions are not truly computable, that in practice we can only approximate them. But then there are analog computers *in practical use* that work in real time and compute functions that are not Turing computable.

Turing himself would not be surprised by our examples. He explicitly restricted his analysis to "symbolic" (symbol-pushing, digital) algorithms. He implicitly restricted his analysis to classical sequential algorithms, essentially the only algorithms in his time.

Remark. It turned out easier to axiomatize all classical sequential algorithms [16], whether symbolic or not, including the ruler-and-compass algorithms, Gaussian elimination and the bisection algorithm (but excluding analog algorithms which are not sequential in our sense).

What about quantum algorithms? Do they compute functions that are not Turing computable? Erich Grädel and Antje Nowack showed that the quantum computing models in the literature can be faithfully simulated by parallel abstract state machines [15]. And it is easy to check that functions computed by parallel abstract state machines are Turing computable.

There is also a rather common misunderstanding that Turing defined the notion of algorithm, albeit restricted to symbolic classical sequential algorithms. Let us restrict attention to such algorithms for a moment. Suppose that your computation model (e.g. a programming language) is Turing complete. Does it mean that the model allows you to express all algorithms? Not necessarily. Turing machines simulate faithfully only the input-output behavior of algorithms. But there may be much more to an algorithm than its input-output behavior. Turing completeness does not mean algorithmic completeness. It means only that, for every Turing computable function f, the language allows you to program an algorithm that computes f.

For illustration consider Turing machines with one tape that may be multidimensional. The model is obviously Turing complete. On any such machine, the problem of palindrome recognition requires $\Theta(n^2/\log n)$ time [2]. But the problem is trivially solvable in linear time on a Turing machine with two one-dimensional tapes. For a deeper dive into algorithmic completeness, see [32, §3].

5. Gandy's analysis of mechanisms

Robin Gandy argues in [12] that "Turing's analysis of computation by a human being does not apply directly to mechanical devices," that humans compute sequentially but machines can perform parallel computations. In this connection, Gandy analyzed computations by mechanical devices and introduced (what we call now) Gandy machines.

> A set-theoretic form of description for discrete deterministic machines is elaborated and four principles (or constraints) are enunciated, which, it is argued, any such machine must satisfy. . . . It is proved that if a device satisfies the principles then its successive states form a [Turing] computable sequence. [12, p. 123]

Note "successive states". Gandy machines work in sequential time. This type of parallelism is called synchronous. In the rest of this section, parallelism will by default be synchronous.

Gandy pioneered the use of axioms in the analysis of computation. His four principles are supposed to be satisfied by all discrete deterministic machines. Contrast this with Turing's analysis. While Turing's analysis is more convincing, it is hard to isolate first principles that, in Turing's opinion, are satisfied by all symbolic sequential computations.

Wilfried Sieg adopted Gandy's approach and reworked Gandy's axioms to an extent; see [28] and references there. For our purposes here, there is no essential difference between Gandy's original axioms and Sieg's versions of the axioms.

Critical remarks. In a 2002 article [25], Oron Shagrir suggests that "there is an ambiguity regarding the types of machines that Gandy was postulating". He offers three interpretations: "Gandy machines as physical machines", "Gandy machines as finite physical machines", and "Gandy machines as a mathematical notion". Shagrir concludes that none of the three interpretations "provides the basis for claiming that Gandy characterized finite machine computation." This agrees with our own analysis.

What real-world devices satisfy Gandy's axioms? Probably very few do. One big problem is the presumed form of the computation states: a hereditarily finite set[1]. There are precious few examples in the papers of Gandy and Sieg, and none of the examples is a real-world device. The most prominent example in Gandy's paper is the cellular automaton known as Conway's game of life. As a mathematical machine the cellular automaton satisfies Gandy's axioms. As a physical machine it does not because Gandy requires state transitions to be synchronous.

The cellular automaton in question can grow without any bound; a real-world materialization of it would not stay synchronous.

What algorithms satisfy Gandy's axioms? Literally speaking, typical parallel or even sequential algorithms do not satisfy the axioms because their states aren't hereditarily finite sets. Consider for example a factorial algorithm. The state of the algorithm is naturally infinite and consists of natural numbers. There is of course a Gandy machine that simulates the factorial algorithm. Note that, in addition to simulating the factorial algorithm, the simulating machine may be forced to construct set representations of additional numbers.

It was probably Gandy's intention all along that computation states (or at least their active parts) can be encoded with (rather than literally be) hereditarily finite sets. In our view, Gandy's axioms are really used just to define another parallel computation model. (By the way, it is our ambition in [3] that parallel algorithms, on their natural abstraction levels, literally satisfy our axioms.)

How does Gandy's parallel computation model compare to other parallel computation models? There are numerous models of synchronous parallel computation models in the literature, e.g. parallel random access machines, circuits, alternating Turing machines, first-order logic with the least fixed-point operator, and parallel abstract state machines. What are the advantages, if any, of Gandy's model over the other models? We challenge the adherents of Gandy's approach to address this question. Gandy's model seems quite awkward for programming or specifying algorithms. Since computation states are hereditarily finite sets, state transitions have to be given in terms of hereditarily finite sets.

Dispensing with the Church and Turing theses. Sieg claims to present "an analysis of calculability that ... dispenses with theses." The claim makes no sense to this author. To begin with, Sieg's analysis is not new. It is Gandy's analysis with some technical modifications. Is Gandy's analysis so convincing that we should forget Church's arguments and Turing's analysis? We don't think so. Gandy's analysis is ambitious, and we admire his courage. But formalizing machine computations

[1]A hereditarily finite set S can be described by means of a finite directed graph G. Every vertex x of G is a finite set, and the union of all these finite sets is finite. A pair (x, y) of vertices forms an edge if and only if $x \in y$ so that the graph is acyclic. S is the top vertex if G; there are no edges coming out from S, and for every other vertex x there is a path from x to S.

in full generality seems as intractable as formalizing algorithms in full generality.

6. What kind of entities are algorithms?

One point of view is that the question about algorithm entities is of no importance. We quoted already in §1 that "Moschovakis has proposed . . . a set-theoretic 'definition' of algorithms, much like the 'definition' of real numbers as Dedekind cuts" [24]. The quotation marks around the word definition make good sense. There is another familiar definition of real numbers, as Cauchy sequences. Dedekind cuts and Cauchy sequences are different entities, yet the two definitions are equivalent for most mathematical purposes. The question of importance in mathematics is not what kind of entities real numbers are but what structure they form. Either definition allows one to establish that real numbers form a complete Archimedean ordered field.

The analogy in the quotation is clear: concentrate on mathematical properties of algorithms rather than on what kind of entities they are. The analogy makes good sense but it is far from perfect because much more is known about algorithm entities than real-number entities. Let us sketch another point of view on algorithm entities.

Consider algorithms that compute in sequential time. This includes sequential algorithms as well as synchronous parallel algorithms. A sequential-time algorithm is a state transition system that starts in an initial state and transits from one state to the next until, if ever, it halts or breaks. The very first postulate in our axiomatizations of sequential and synchronous parallel algorithms [16, 3] is that the algorithms in question are sequential-time.

The question arises what kind of entities states are. In our view, rather common in computer science, algorithms are not humans or devices; they are abstract entities. According to the second postulate in the axiomatizations of sequential and synchronous parallel algorithms, the states are (first-order) structures, up to isomorphism. This admittedly involves a degree of mathematical modeling and even arbitrariness. A particular form of structures is used; why this particular form? But this is a minor detail. Structures are faithful representations of states, and that is all that matters for our purposes. It is convenient to declare that states *are* structures, up to isomorphism; but there is no need to do so.

The point of view that sequential-time algorithms are state transition systems extends naturally to other classes of algorithms. In particular, a sequential-time interactive algorithm (until now we considered non-interactive algorithms) is a state transition system where a state transition may be accompanied by sending and receiving messages. A distributed algorithm is an ensemble of communicating sequential-time interactive algorithms.

7. Moschovakis's recursion-based approach

We recall the basics of recursion-based approaches, touch upon Moschovakis's approach, and present some critical remarks.

7.1. **Recursion-based approaches.** It is often convenient to specify a function by means of recursive equations. Typically the equations define a monotone operator, and semantics is given by means of the least fixed point of the operator. For example, equations

$$\exp(x+1,0) = 1$$

$$\exp(x,y+1) = \begin{cases} 0 & \text{if } x = 0 \\ x \times \exp(x,y) & \text{if } x > 0 \end{cases}$$

specify the exponentiation function $\exp(x,y) = x^y$ on natural numbers[2]. The two equations define a monotone operator on the expansions of the standard arithmetical structure with partial binary function exp. Accordingly the following process gives meaning to the exponentiation function exp. Initially function exp is nowhere defined. Apply the equations once, obtaining $\exp(x,0) = 1$ for every $x > 0$ and $\exp(0,y) = 0$ for all $y > 00$; then apply the equations again, obtaining additionally $\exp(x,1) = x$ for all x, and so on. After ω steps (where ω is the first infinite ordinal), you reach a fixed point; now $\exp(x,y)$ is defined for all x,y except $x = y = 0$. Often the evolution toward the fixed point involves not only the target function (that you intend to compute) but also some auxiliary functions.

In 1934, Gödel formulated a recursion-based calculus of partial numerical functions. Gödel's calculus can be seen as a specification language where a specification of a numerical function f is a system of recursive equations that, taking into account some global conventions, gives us a particular (possibly inefficient) way to compute f.

Recursive specifications of functions have much appeal. They are declarative, abstract from computation details and often are concise. Recursive specification of functions have been rather popular with logicians and computer scientists. Logicians developed a deep theory of recursion [29]. Computer scientists developed many recursion-based languages, starting with John McCarthy's LISP [20].

[2]It is presumed here that the function x^y is partial, with 0^0 being undefined.

7.2. **Moschovakis's approach and thesis.** Moschovakis's approach belongs to the recursion-based framework. The key ideas of the approach appeared already in the 1984 article [21].

> If, by Church's Thesis, the precise mathematical notion of *recursive function* captures the intuitive notion of *computable function*, then the precise, mathematical notion of *recursion* ... should model adequately the mathematical properties of the intuitive notion of *algorithm*. [21, p. 291]

The article contains interesting mathematical results. In particular, giving recursive equations for the mergesort algorithm, Moschovakis proves that at most $n \cdot \log_2(n)$ comparisons are required to sort n elements. For our purposes here, the following philosophical remark of Moschovakis is important. Discussing Euclid's algorithm for the greatest common divisor of two natural numbers, Moschovakis says:

> Following the drift of the discussion, we might be expected at this point to simply identify the Euclidean algorithm with the functional `gcd`. We will not go quite that far, because the time-honored intuitive concept of algorithm carries many linguistic and intensional connotations (some of them tied up with *implementations*) with which we have not concerned ourselves. Instead we will make the weaker (and almost trivial) claim that *the functional* ***`gcd`*** *embodies all the essential mathematical properties of the Euclidean algorithm.* [21, p. 295]

Contrast this with a later view of Moschovakis:

> When algorithms are defined rigorously in Computer Science literature (which only happens rarely), they are generally identified with abstract machines, mathematical models of computers. ... My aims here are to argue that this does not square with our intuitions about algorithms and the way we interpret and apply results about them; to promote the problem of defining algorithms correctly; and to describe briefly a plausible solution, by which algorithms are recursive definitions while machines model implementations, a special kind of algorithms. [22, p. 919].

The main technical notion in Moschovakis's approach is that of *recursor* which is a generalization of function specification in Gödel's calculus. A recent definition of recursor is found in [24, p. 95]. The semantics of a recursor is given by means of the least fixed point of an appropriate monotone operator. In some cases, the least fixed point is not achieved in $\leq \omega$ steps; then the recursor is *infinitary* and cannot be implemented by finite computations. For illustration, see "the infinitary Gentzen algorithm" in [23]. Moschovakis formulates this thesis[23, p. 4]:

Thesis (Moschovakis)**.** *The theory of algorithms is the theory of recursive equations.*

7.3. **Critical remarks.** We have great respect to recursion theory in general and to Moschovakis's contributions to recursion theory in particular, but we reject Moschovakis's thesis. The theory of recursion and that of algorithms have a large intersection but are very different. This reminds us the following pronouncement of Richard Feynman:

> We must, incidentally, make it clear from the beginning that if a thing is not a science, it is not necessarily bad. For example, love is not a science. So, if something is said not to be a science, it does not mean that there is something wrong with it; it just means that it is not a science [14, p. 3-1].

"Every analogy limps" according to a popular Hebrew saying, and we wish avoid any misunderstanding. Recursion theory is science, excellent science. It is just not the theory of algorithms.

The issue of recursors vs. algorithms deserves a systematic analysis. In the meantime, here are somewhat disparate critical remarks.

Algorithms as abstract machines. The identification of algorithms with abstract machines squares with our intuition about algorithms. Moreover, that intuition inspired the theory of abstract state machines. We suspect, at the time of writing article [22], Moschovakis was exposed to and had in mind only machine models of low abstraction level, like Turing machines or Random Access Machines. Abstract machines may be vastly higher level and may even be recursive [4].

Recursors vs. algorithms. In §7.2, we contrasted earlier and later views of Moschovakis on his thesis. We think that Moschovakis was right the first time around when he refrained from identifying (what he later called) recursors with algorithms "because the time-honored intuitive concept of algorithm carries many linguistic and intensional connotations" which are contrary to such identification.

Consider the system of two recursive equations (and thus a recursor) for the exponentiation in the beginning of this section. Is it an algorithm or not? The recursor certainly looks like an algorithm, and in many functional programming languages, this recursor would be a legitimate program (modulo syntactic details of no importance to us here). Typically $\exp(x^y)$ would be interpreted as a function call and, for example, the evaluation of 3^2 would proceed thus:

$$3^2 = 3 \cdot 3^1 = 3 \cdot (3 \cdot 3^0) = 3 \cdot (3 \cdot 1)) = 3 \cdot 3 = 9.$$

But the recursor theory is different. The meaning of a recursor is given by the least fixed point construction, and there is *nothing else*. In the case of the exponentiation recursor, the only "computation" is the process that we described above: start with the nowhere defined exp function, compute $\exp(x^0)$ for *all* $x > 0$, compute x^1 for *all* x, etc. What should we do in order to compute 3^2? Should we wait until the "computation" of exp is completed and then apply exp, or should we wait only to the end of stage 3 when all x^2 are computed? The recursor theory says nothing about that.

It is not our goal to make the recursor theory look ridiculous. In fact we agree that recursors are useful for mathematical analysis of algorithms. We just see no good reason to identify them with algorithms. Paraphrasing Richard Feynman, if thing is not an algorithm, it is not necessarily bad.

The abstraction level of imperative algorithms. It seems to us that many recursor theorists underestimate the abstraction capabilities of imperative programming. An imperative program for an algorithm, and in particular an abstract state machine, can be as abstract as the algorithm itself. We addressed this point in [4]. Here let us just quickly say this. Yes, an algorithm comes with a program for executing the algorithm. But this does not mean that the program necessarily addresses low-level computational details. Every algorithm operates on its natural level of abstraction. This level may be very low but it may be arbitrarily high.

Declarative specifications. Recursion is appealing. A part of the appeal comes from the declarative nature of recursion. That declarative nature is by itself a limitation for software specification; note that every piece of software is an algorithm. Declarative specification of software was very popular in the 1980s and 1990s, but it was discredited to a large extent later. As software is developed, it evolves. A book with a declarative specification quickly becomes obsolete. If specification is not executable, you cannot experiment with it.

Recursion is but one aspect of an algorithm. The theory of algorithms does not reduce to recursion. For one thing, there are clever data structures. For many linear-time algorithms, for example, it is crucially important that an algorithm does not manipulate large objects directly; instead it manipulates only pointers to those objects. Such aspects of complexity analysis are normally below the abstraction level of recursors.

Distributed algorithms. The recursor approach does not extend to distributed algorithms, and the number of useful distributed algorithms is large and growing.

Monotonicity limitation. Here is something that the recursor theory should be able to cover but doesn't. The current recursor theory is limited to recursors with semantics given by the least fixed point of a monotone operator. That is a serious limitation.

For a simple example consider Datalog with negation [1]. The operator defined by a Datalog-with-negation program is not monotone but it is inflationary, and semantics is given by the inflationary fixed point [17].

For illustration, here is a Datalog-with-negation program computing the complement C of the transitive closure T of a nonempty binary relation R on a finite domain [1, Example 3.3].

$$\begin{aligned}
&T(x,y) \leftarrow R(x,y)\\
&T(x,y) \leftarrow R(x,z), T(z,y)\\
&U(x,y) \leftarrow T(x,y)\\
&V(x,y) \leftarrow T(x,y), R(x',z'), T(z',y'), \neg T(x',y')\\
&C(x,y) \leftarrow \neg T(x,y), U(x',y'), \neg V(x',y')
\end{aligned}$$

Explanation. At every step all rules are fired. By the first two rules, the computation of T proceeds in the usual way. Since the domain is finite, the computation of T completes after some number k of steps. The pairs of T are stored in U with a delay of one step, so the computation of U completes after $k+1$ steps. The computation of V is identical to that of U, except that at the step $k+1$, when U is completed, the last batch of pairs from T is not stored in V. The final rule is idle during the first k steps but on step $k+1$ it stores the complement of T into C.

More examples, please. It would be useful to have more examples of recursors of interest to computer scientists. All current examples of relevance seem to be present already in the 1984 article [21].

References

[1] Serge Abiteboul and Victor Vianu, "Datalog extensions for database queries and updates", J. of Computer and System Sciences 43 (1991) 62–124.

[2] Therese Biedl, Jonathan F. Buss, Erik D. Demaine, Martin L. Demaine, Mohammadtaghi Hajiaghayi, Tomáš Vinař, "Palindrome recognition using a multidemensional tape", Theoretical Computer Science 302 (2003) 475–480.

[3] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," ACM Transactions on Computational Logic 4:4 (2003) 578–651. Correction and extension, same journal, 9:3 (2008) article 19.

[4] Andreas Blass and Yuri Gurevich, "Algorithms vs. machines", Bull. European Association for Theoretical Computer Science 77 (2002), 96–118.

[5] Andreas Blass and Yuri Gurevich, "Algorithms: A quest for absolute definitions"; in *Current Trends in Theoretical Computer Science* (eds. G. Paun et al.), World Scientific, 2004, 195–225; also in *Church's Thesis after 70 Years* (eds. A. Olszewski et al.), Ontos Verlag, 2006, 24–57.

[6] Andreas Blass and Yuri Gurevich, "Ordinary interactive small-step algorithms", ACM Trans. Computational Logic 7:2 (2006) 363–419 (Part I), plus 8:3 (2007), articles 15 and 16 (Parts II, III).

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

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

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

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

[11] Yuri Gurevich, "Foundational analyses of computation," in *How the World Computes* (eds. S. Barry Cooper et al.), Turing Centenary Conference Proceedings, Springer LNCS 7318, 264–275.

[12] Robin Gandy, "Church's thesis and principles for mechanisms", In The Kleene Symposium (eds. J. Barwise et al.), North-Holland, 1980, 123–148.

[13] S. Barry Cooper, Benedict Löwe and Andrea Sorbi, editors, "New computational paradigms: Changing conceptions of what is computable", Springer, 2008.

[14] Richard P. Feynman, Robert B. Leighton and Matthew Sands, "The Feynman lectures on physics," Vol. 1, Caltech 1963.

[15] Erich Grädel and Antje Nowack, "Quantum computing and abstract state machines", Springer Lecture Notes in Computer Science 2589 (2003), 309–323.

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

[17] Yuri Gurevich and Saharon Shelah, "Fixed-point extensions of first-order logic", Annals of Pure and Applied Logic 32 (1986) 265–280.

[18] Stephen C. Kleene, *Introduction to metamathematics*, D. Van Nostrand, New York, 1952.

[19] Andrei N. Kolmogorov, “On the concept of algorithm”, Uspekhi Mat. Nauk 8:4 (1953) 175–176, Russian. English translation in [31, p. 8–19].

[20] John McCarthy, “A basis for a mathematical theory of computation”, in *Computer Programming and Formal Systems* (eds. P. Brafford and D. Herschberg), North-Holland, 1963, 33–70.

[21] Yiannis N. Moschovakis, “Abstract recursion as a foundation of the theory of algorithms”, in *Computation and Proof theory* (eds. M.M. Richter et al.), Springer Lecture Notes in Mathematics 1104 (1984), 289–364.

[22] Yiannis N. Moschovakis, “What is an algorithm?”, in *Mathematics Unlimited — 2001 and beyond* (eds. B. Engquist and W. Schmid), Springer, 2001, 919–936.

[23] Yiannis N. Moschovakis, “Algorithms and implementations”, Tarski Lecture 1, March 3, 2008, `http://www.math.ucla.edu/~ynm/lectures/tlect1.pdf`.

[24] Yiannis N. Moschovakis and Vasilis Paschalis, “Elementary algorithms and their implementations”, in [13], 87–118.

[25] Oron Shagrir, “Effective computation by humans and machines”, Minds and Machines 12 (2002) 221–240.

[26] Wilfried Sieg, “Calculations by man & machine: Mathematical presentation”, in Proceedings of the Cracow International Congress of Logic, Methodology and Philosophy of Science, Kluwer, 2002, 245–260.

[27] Wilfried Sieg, “Church without dogma – Axioms for computability”, in [13], 139–152.

[28] Wilfried Sieg, “On computability”, in *Handbook of the Philosophy of Mathematics* (A. Irvine, editor), Elsevier, 2009, 535-630.

[29] Robert I. Soare, “Computability and recursion,” Bulletin of Symbolic Logic 2 (1996), 284-321.

[30] Alan M. Turing, “On computable numbers, with an application to the Entscheidungsproblem”, Proceedings of London Mathematical Society, series 2, vol. 42 (1936–1937), 230–265. Correction, same journal, vol. 43, 544–546.

[31] Vladimir A. Uspensky and Alexei L. Semenov, “Algorithms: Main ideas and applications”, Kluwer, 1993.

[32] Pierre Valarcher, “Autour du concept d’algorithme: intensionalité, complétude algorithmique, programmation, et modèles de calcul, suivi de fonctions boolénnes et cryptographie,” Habilitation à Diriger des Recherches, Université Paris Est Créteil, France, 2010. `http://www.paincourt.net/perso/Publi/hdr.pdf`

Microsoft Research, Redmond, WA, USA