What Is an Algorithm?

Yuri Gurevich

Microsoft Research
gurevich@microsoft.com

Abstract. We attempt to put the title problem and the Church-Turing thesis into a proper perspective and to clarify some common misconceptions related to Turing's analysis of computation. We examine two approaches to the title problem, one well-known among philosophers and another among logicians.

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.

Richard Feynman

1 Introduction

Two articles in a recent book [10] present two approaches to the title problem and offer different answers. Article [19] presents an approach developed by Yiannis Moschovakis. "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" [19]. The article starts thus.

> In the sequence of articles . . . , 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.

We discuss this definition of algorithms in §6.

Article [22] presents an approach originally developed by Robin Gandy, a student of Alan Turing, in a 1980 article [9]. Gandy intended to complement Turing's analysis of human computers with an analysis of computation by mechanical devices. He came up with an axiomatically defined class of computation devices, later named Gandy machines. The approach was adopted by Wilfried Sieg. In article [22], Sieg uses Gandy machines to "dispense with [Church's and Turing's] theses". The article starts thus.

> 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.

We discuss this axiomatization of discrete dynamical systems and dispensing with the theses in §4.

In §2, we discuss whether it is possible at all to define algorithms. (There is also a question why bother to define algorithms. Well, understanding what algorithms are should — and does — have practical applications, to software specification and model-based testing in particular, as well as theoretical application, like semantics of software or algorithmic completeness of computation models. But that is a different issue to be addressed elsewhere.)

In §3, we discuss and clarify a couple of misconceptions related to Turing's analysis of computations. In §4 we discuss Gandy machines. In §5, we discuss what kind of entities algorithms are; this discussion is closely related to §6 where we discuss Moschovakis's recursor theory.

This article can be seen as a companion to our older article [5].

2 Can the Notion of Algorithm Be Rigorously Defined?

Two articles [19] and [22], mentioned in §1, give different answers to the question in the title of this article. The two answers are not at all equivalent. A question arises whether the notion of algorithm can be defined at all. The answer is yes and no. Let us explain.

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 §1, Andreas Blass suggested a better analogy: algorithms to numbers. Many kinds of numbers have been introduced throughout history: positive integers, natural numbers, rationals, reals, complex numbers, quaternions, infinite cardinals, 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. The notions of numbers and algorithms have not crystallized (and maybe never will) to support rigorous definitions.

The Positive Answer

But the problem of rigorous definition of algorithms is not hopeless. Not at all. Some strata of algorithms have matured enough to support rigorous definitions.

This applies to classical (or classical sequential or just 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 [14]. This is a good informal definition of sequential algorithms.

An axiomatic definition of sequential algorithms have been given in [12]. That definition was used to derive the Church-Turing thesis from first principles in [8]. The derivation presumes that, at the time, Church and Turing (and Gödel and other experts) had in mind only sequential algorithms, which we believe they did. The axiomatic definition was extended to synchronous parallel algorithms in [3] and to interactive sequential algorithms in [6,7].

The Status of the Church-Turing Thesis

As far as the input-output relation is concerned, synchronous parallel algorithms and interactive sequential algorithms can be simulated by Turing machines. This gives additional confirmation of the Church-Turing thesis.

None of the other known kinds of algorithms seem to threaten the thesis but the thesis has not been dispensed with and probably never will be. The question whether some algorithm of a currently unknown kind would allow us to compute a function from natural numbers to natural numbers that is not Turing computable remains open, possibly forever. And even if we had a satisfactory axiomatic definition of algorithms in full generality, the thesis would not be dispensed with. It would be just reduced to the first principles enshrined in the axioms.

3 Remarks on Turing’s Analysis of Computation

Turing’s analysis of computation [24] was a stroke of genius. The analysis is extremely famous, and yet it is often misunderstood.

Some people think that every computable function, total or partial, can be computed by a Turing machine. This is not so, and 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 the 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 incommensurate) 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.

Of course Turing would not be surprised by our examples. He explicitly restricted his analysis to “symbolic” (symbol-pushing, digital) algorithms. He implicitly restricted his analysis to sequential algorithms, essentially the only algorithms in his time. It is interesting that it turned out easier to axiomatize all sequential algorithms [12], 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 checked that the quantum computing models in the literature can be faithfully simulated by parallel abstract state machines [11]. And, as we mentioned above, functions computed by parallel ASMs are Turing computable.

There is also a rather common misunderstanding that Turing defined the notion of algorithm, albeit restricted to symbolic 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 algorithms than their 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 [26, §3].

4 Gandy's Analysis of Mechanisms

Robin Gandy argues in [9] that "Turing's analysis of computation by a human being does not apply directly to mechanical devices." The reason is 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. [9, 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. He came up with four principles (or constraints, or axioms) satisfied, he claimed, by all discrete deterministic machines. Contrast this with Turing's analysis. While Turing's analysis was 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; see [23] and references there.

Critical Remarks

In a 2002 article [20], 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. By the way, for our purposes, there is no difference between Gandy's original axioms and Sieg's versions of the axioms. So we will just speak about Gandy's axioms.

• *What real-world devices satisfy Gandy's axioms?* Probably very few do. One problem is the form of the states of Gandy machines: a collection of hereditary finite sets. Another problem is the requirement that state transitions are synchronous. You, the reader, may say that we have not proved our point. Well, the burden of proof is on the proponents of the approach. And 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. Note that a cellular automaton can grow without any bound. In the real-world, such a cellular automaton would not stay synchronous.

It seems obvious that Gandy abstracts from material and views discrete deterministic machines as algorithms, abstract algorithms. So Gandy's claim can be restated thus: parallel algorithms satisfy the axioms.

• *What algorithms satisfy Gandy's axioms?* Typical parallel or even sequential algorithms do not satisfy the axioms. 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.

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, satisfy our axioms.)

• *How does Gandy's parallel computation model compare to other parallel computation models?* By now, there are numerous models of synchronous parallelism 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? Neither Gandy nor Sieg addressed this question. Gandy's model seems quite awkward for programming or specifying algorithms.

• *Dispensing with the Church-Turing thesis* Gandy proved that his machines can be simulated by Turing machines. This is another confirmation of the Church-Turing thesis. But is it a good ground for dispensing with the thesis? We do not think so, even if we restrict attention to parallel algorithms and forget other kinds of algorithms. By the first two bullets above, Gandy's theorem does not imply that his axioms are satisfied by all discrete mechanical devices or by all parallel algorithms.

5 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" [19]. 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 [12,3] 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.

6 Moschovakis's Recursion-Based Approach

We start with basics. In recursion-based approaches you write recursive equations that specify a function. Typically the equations define a monotone operator, and semantics is given by means of the least fixed point of the operator. For example, equations

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

specify exponentiation $\exp(x,y)=x^y$ on natural numbers. The equations define a monotone operator on extensions of the standard arithmetical structure with partial binary function exp. Accordingly the following process gives meaning to exponentiation. Initially exp is nowhere defined. Apply the equations, obtaining $\exp(x,0)=1$ for every $x>0$; 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 function that you are computing but also some auxiliary functions.

In 1934, Gödel formulated a recursion-based calculus of (in general partial) numerical functions. Gödel's calculus can be seen as a specification language where a specification of a function f is a system of recursive equations that, taking into account some global conventions, suggests a particular (possibly inefficient) way to compute f. Church's thesis (extended to partial functions by Kleene) asserts that every "effectively calculable", that is computable by an algorithm, function on natural numbers is programmable in Gödel's calculus.

Recursive specification of functions has much appeal. It is declarative and abstracts from computation details. It is often concise. There has been much progress since the 1930s. Logicians developed recursion theory. McCarthy created a functional (that is recursion-based) programming language LISP, and many other functional languages followed.

The key ideas of Moschovakis's approach appear already in the 1984 article [16] that seems to be the very first publication on the subject.

> 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*. [16, p. 291]

Moschovakis discusses Euclid's algorithm for the greatest common divisor of two natural numbers. Then he 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.* [16, p. 295]

He gives recursive equations for the mergesort algorithm on a set X and proceeds to prove that at most $n \cdot \log_2(n)$ comparisons are required to sort n elements.

Moschovakis's views have been evolving.

> 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. [17, 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. The most recent published definition of recursor is found in [19, p. 95]. Semantics is given by means of the least fixed point of a 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 abstract machines. For illustration, see "the infinitary Gentzen algorithm" in [18]. Moschovakis formulates this slogan:

The theory of algorithms is the theory of recursive equations. [18, p. 4]

Critical Remarks

• *Recursors vs. algorithms* We think that Moschovakis was right the first time around, in [16, p. 295] when he refrained from identifying (what he later called) recursors with algorithm "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 recursor theorists underestimate the abstraction capabilities of imperative programming. Imperative programs, and in particular abstract state machines, can be as abstract as needed. We addressed this point once [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; and 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. 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 seem below the abstraction level of recursors.

• *Distributed algorithms* The recursor approach does not seem to 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 [13].

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 much useful to have more example of recursors of interest to computer scientists. All current examples of that sort seem to be present already in the 1984 article [16].

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

References

1. Abiteboul, S., Vianu, V.: Datalog extensions for database queries and updates. J. of Computer and System Sciences 43, 62–124 (1991)
2. Biedl, T., Buss, J.F., Demaine, E.D., Demaine, M.L., Hajiaghayi, M., Vinař, T.: Palindrome recognition using a multidemensional tape. Theoretical Computer Science 302, 475–480 (2003)
3. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Transactions on Computational Logic 4(4), 578–651 (2003); Correction and extension, same journal 9(3) article 19 (2008)
4. Blass, A., Gurevich, Y.: Algorithms vs. machines. Bull. European Association for Theoretical Computer Science 77, 96–118 (2002)
5. Blass, A., Gurevich, Y.: Algorithms: A quest for absolute definitions. In: Current Trends in Theoretical Computer Science, pp. 195–225. World Scientific (2004); also in Olszewski, A., et al. (eds): Church's Thesis after 70 Years, pp. 24–57. Ontos Verlag (2006)
6. Blass, A., Gurevich, Y.: Ordinary interactive small-step algorithms. ACM Trans. Computational Logic (Part I), 7(2), 363–419 (2006); plus 8(3), articles 15 and 16 (Parts II, III) (2007)
7. Blass, A., Gurevich, Y., Rosenzweig, D., Rossman, B.: Interactive small-step algorithms. Logical Methods in Computer Science 3(4), papers 3 and 4 (Part I and Part II) (2007)
8. Dershowitz, N., Gurevich, Y.: A natural axiomatization of computability and proof of Church's thesis. Bull. of Symbolic Logic 14(3), 299–350 (2008)
9. Gandy, R.: Church's thesis and principles for mechanisms. In: Barwise, J., et al. (eds.) The Kleene Symposium, pp. 123–148. North-Holland (1980)
10. Cooper, S., Löwe, B., Sorbi, A. (eds.): New Computational Paradigms: Changing Conceptions of what is Computable. Springer, Heidelberg (2008)
11. Grädel, E., Nowack, A.: Quantum Computing and Abstract State Machines. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 309–323. Springer, Heidelberg (2003)
12. Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms. ACM Transactions on Computational Logic 1(1), 77–111 (2000)
13. Gurevich, Y., Shelah, S.: Fixed-point extensions of first-order logic. Annals of Pure and Applied Logic 32, 265–280 (1986)
14. Kolmogorov, A.N.: On the concept of algorithm. Uspekhi Mat. Nauk 8(4), 175–176 (1953) (in Russian); English translation in [25]
15. McCarthy, J.: A basis for a mathematical theory of computation. In: Brafford, P., Herschberg, D. (eds.) Computer Programming and Formal Systems, pp. 33–70. North-Holland (1963)
16. Moschovakis, Y.N.: Abstract recursion as a foundation of the theory of algorithms. In: Computation and Proof Theory. Lecture Notes in Mathematics, vol. 1104, pp. 289–364. Springer, Heidelberg (1984)

17. Moschovakis, Y.N.: What is an algorithm? In: Engquist, B., Schmid, W. (eds.) Mathematics Unlimited – 2001 and Beyond, pp. 919–936. Springer, Heidelberg (2001)
18. Moschovakis, Y.N.: Algorithms and implementations. Tarski Lecture 1 (2008), `http://www.math.ucla.edu/~ynm/lectures/tlect1.pdf`
19. Moschovakis, Y.N., Paschalis, V.: Elementary algorithms and their implementations. In: [10], pp. 87–118
20. Shagrir, O.: Effective computation by humans and machines. Minds and Machines 12, 221–240 (2002)
21. Sieg, W.: Calculations by man & machine: Mathematical presentation. In: Proceedings of the Cracow International Congress of Logic, Methodology and Philosophy of Science, pp. 245–260. Kluwer (2002)
22. Sieg, W.: Church without dogma – Axioms for computability. In: [10], pp. 139–152
23. Sieg, W.: On Computability. In: Irvine, A. (ed.) Handbook of the Philosophy of Mathematics, pp. 535–630. Elsevier (2009)
24. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of London Mathematical Society, Series 2 42, 230–265 (1936-1937); Correction, same journal 43, 544–546
25. Uspensky, V.A., Semenov, A.L.: Algorithms: Main Ideas and Applications. Kluwer (1993)
26. Valarcher, P.: Habilitation à Diriger des Recherches, Université Paris Est Créteil, LACL (EA 4219), Département d'Informatique, IUT Fontainebleau, France (2010), `http://www.paincourt.net/perso/Publi/hdr.pdf`