Foundational Analyses of Computation

Yuri Gurevich
Microsoft Research, Redmond, Washington, USA

Technical Report
MSR-TR-2012-14, February 2012

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Foundational Analyses of Computation

Yuri Gurevich
Microsoft Research, Redmond, Washington, USA

Give me a fulcrum, and I shall move the world.
—Archimedes

Abstract

How can one possibly analyze computation in general? The task seems daunting if not impossible. There are too many different kinds of computation, and the notion of general computation seems too amorphous. As in quicksand, one needs a rescue point, a fulcrum. In computation analysis, a fulcrum is a particular viewpoint on computation that clarifies and simplifies things to the point that analysis become possible.

We review from that point of view the few foundational analyses of general computation in the literature: Turing's analysis of human computations, Gandy's analysis of mechanical computations, Kolmogorov's analysis of bit-level computation, and our own analysis of computation on the arbitrary abstraction level.

1 Introduction

Algorithms and computations are closely related concepts. Syntactically algorithms are programs (or recipes) but semantically they specify computations. And the only computations that we consider here are algorithmic (also known as mechanical). In this paper, we abstract from the syntax of algorithms, so that analysis of algorithms and analysis of computation are one and the same.

Turing's analysis of algorithms was provoked by the Entscheidungsproblem, the problem whether the validity of first-order formulas is computable.

Logicians have been interested in what functions are computable, and Turing's analysis is often seen from that point of view. But there may be much more to an algorithm than its input-output behavior. In general algorithms perform tasks, and computing functions is a rather special class of tasks.

Here we concentrate on foundational analyses of algorithms/computations, not on what functions are computable.

Acknowledgements

Many thanks to Andreas Blass and Oron Shagrir for useful comments.

2 Turing

Alan Turing analyzed computation in his 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem" [21]. The validity relation on first-order formulas can be naturally represented as a real number, and the Entscheidungsproblem becomes whether this particular real number is computable. "Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique" [21, p. 230].

How could Turing analyze computation in such generality? The world of algorithms is large and diverse. Explicitly or implicitly, he imposed some constraints on the computations in consideration. And he found a fulcrum. We start with the fulcrum. There were no computers in Turing's time[1] but that does not seem to make Turing's task much simpler. Humans are hard to analyze. Amazingly Turing found a way to do just that: Ignore how the algorithm is given, ignore what human computers have in their minds, and concentrate on what the computers do, what their observable behavior is. That is his fulcrum.

[1] "Numerical calculation in 1936 was carried out by human beings; they used mechanical aids for performing standard arithmetical operations, but these aids were not programmable" (Gandy [8, p. 12]).

One may argue that Turing did not ignore the mind. He speaks about the state of mind of the human computer explicitly and repeatedly. For example, he says that "[t]he behaviour of the computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at that moment" [21, p. 250]. But Turing postulates that "the number of states of mind which need be taken into account is finite." The computer just remembers the current state of mind, and even that is not necessary: "we avoid introducing the 'state of mind' by considering a more physical and definite counterpart of it. It is always possible for the computer to break off from his work, to go away and forget all about it, and later to come back and go on with it. If he does this he must leave a note of instructions (written in some standard form) explaining how the work is to be continued. This note is the counterpart of the 'state of mind'."

Turing introduced abstract computing machines that became known as Turing machines (and constructed a universal Turing machine). He defined a real number to be computable "if its decimal can be written down by a [Turing] machine" [21, p. 230]. His thesis was that Turing computable numbers "include all numbers which could naturally be regarded as computable" (Turing [21, p. 230]). He used the thesis to prove the undecidability of the Entscheidungsproblem. To convince the reader of his thesis, Turing used three arguments.

Reasonableness: He gave examples of large classes of real numbers which are [Turing] computable.

Robustness He gave another explicit definition of computability and proved it is equivalent to the original one "in case the new definition has a greater intuitive appeal." The robustness argument was strengthened in the appendix where, after learning about Church's explicit definition of computability [6], he proved the equivalence of their definitions.

Appeal to Intuition He analyzed computation appealing directly to intuition.

The first two arguments are important but insufficient. There are other reasonable and robust classes of computable real numbers, e.g. the class of primitive recursive real numbers. The direct appeal to intuition is crucial.

While Turing's analysis is very general, his algorithms are subject to some constraints. Here are some of them.

Symbolic Computation is symbolic (or digital, symbol-pushing).

Sequential time Computation splits into a sequence of steps.

Bounded work Only bounded work is performed at any one step.

Isolated Computation is self-contained. No oracle is consulted, and nobody interferes with the computation either during a computation step or in between steps. The whole computation of the algorithm is determined by the initial state.

2.1 Discussion

`Q`[2] Did Turing really impose the symbolic constraint?

`A:` Yes, he did. "Computing is normally done by writing certain symbols on paper," writes Turing [21, p. 249], and he analyses only such computations.

`Q:` Is this really a constraint?

`A:` These days we are so accustomed to digital computations that the symbolic constraint may not look like a constraint. But it is. Non-symbolic computations have been performed by humans from ancient times [13, §3].

`Q:` I came across a surprising remark of Gödel that Turing's argument "is supposed to show that mental procedures cannot go beyond mechanical procedures" [9]. I believe that Turing's goal was to analyze mechanical procedures. Since such procedures were executed by humans in his time, he had to analyze human execution of mechanical procedures; there was no other way.

`A:` We may never know what goal was in Turing's head; let's hear Gödel's argument.

`Q:` "What Turing disregards completely is the fact that mind, in its use, is not static, but constantly developing, i.e., that we understand abstract terms more and more precisely as we go on using them, and that more and more abstract terms enter the sphere of our understanding. There may exist systematic methods of actualizing this development, which could form part of the procedure" (Gödel, [9]).

[2] `Q` is my inquisitive friend Quisani, and `A` is the author.

A: Gödel raises a possibility that there exists a sophisticated decision procedure for the Entscheidungsproblem that can be executed by gifted mathematicians.

Q: Hmm, if gifted mathematicians can reliably execute a procedure, they should be able to figure out how to program it, and then the procedure is mechanical.

A: Well, it is hard to delimit human creativity. Certainly Turing did not do that.

Q: And didn't intend to, I am sure. But let me change the topic. You said nothing about Church's arguments in favor of his definition of computability.

A: Church had strong arguments that his definition of reasonable and robust. In particular, he and his student Kleene proved that a numerical function is expressible in Church's λ-calculus if and only if it is expressible in Gödel's recursive calculus. Church's thesis was that [Gödel's] recursive functions include all numerical functions that are "effectively calculable".

Q: Here is another quote. "For the actual development of the (abstract) theory of computation, where one must build up a stock of particular functions and establish various closure conditions, both Church's and Turing's definition are equally awkward and unwieldy. In this respect, general recursiveness is superior" (Sol Feferman, [8, p. 6]). Do you buy that?

A: Indeed, the recursive approach has been dominant in mathematical logic, but Turing's approach dominates in computer science and it influenced the early design of digital computers.

3 Kolmogorov

Andrei Kolmogorov analyzed computation in abstraction from the computer. Kolmogorov's fulcrum seems to be the idea that computations, independently from the computer, satisfy nontrivial constraints. In a 1953 talk to the Moscow Mathematical Society [14], he stipulated that every algorithmic process satisfies the following constraints.

Sequentiality An algorithmic process splits into steps whose complexity is bounded in advance.

Elementary steps Each step consists of a direct and unmediated transformation of the current state S to the next state S^*.

Locality Each state S has an active part of size bounded in advance. The direct and unmediated transformation of S to S^* is based only on the information about the active part of S and applies only to the active part.

These ideas gave rise to a new computation model developed by Kolmogorov and his student Vladimir Uspensky [15]. Instead of a linear tape, a Kolmogorov machine has a graph of bounded degree (so that there is a bound on the number of edges attached to any vertex), with a fixed number of the types of vertices and a fixed number of the types of edges. We speculated in [10] that "the thesis of Kolmogorov and Uspensky is that every computation, performing only one restricted local action at a time, can be viewed as (not only being simulated by, but actually being) the computation of an appropriate KU machine." Uspensky agreed [22, p. 396].

We do not know what analysis, if any, allowed Kolmogorov and Uspensky to arrive from the constraints above at the particular architecture of Kolmogorov machines. "As Kolmogorov believed," wrote Uspensky [22, p. 395], "each state of every algorithmic process ... is an entity of the following structure. This entity consists of elements and connections; the total number of them is finite. Each connection has a fixed number of elements connected. Each element belongs to some type; each connection also belongs to some type. For every given algorithm the total number of element types and the total number of connection types are bounded." In that approach, the number of nonisomorphic active zones is finite (because of a bound on the size of the active zones), so that the state transition can be described by a finite program.

Leonid Levin told us that Kolmogorov thought of computation as a physical process developing in space and time, that the edges of Kolmogorov machine reflect physical closeness of computation elements [16]. But then, as we mentioned in [12], the dimensionality of the space may grow with the input size.

Kolmogorov's analysis has not been well known. In this connection, let us point out these references: [1, 10, 22, 23].

4 Gandy

Gandy analyzed computation in his 1980 paper "Church's Thesis and Principles for Mechanisms" [7]. In this section, by default, quotations are from

that paper.

> Turing's analysis of computation by a human being does not apply directly to mechanical devices ... Our chief purpose is to analyze mechanical processes and so to provide arguments for ...
> **Thesis M.** *What can be calculated by a machine is computable.*

Contrary to human computers, a machine can perform parallel actions. Kolmogorov machines are fine "but at each step only a bounded portion of the whole state [of a Kolmogorov machine] is changed." Thesis M "must take parallel working into account." A question arises what machines are.

> (1) In the first place I exclude from consideration devices which are *essentially* analogue machines. ... I shall distinguish between "mechanical devices" and "physical devices" and consider only the former. The only physical presuppositions made about mechanical devices ... are that there is a lower bound on the linear dimensions of every atomic part of the device and that there is an upper bound (the velocity of light) on the speed of propagation of changes.
> (2) Secondly we suppose that the progress of calculation by a mechanical device may be described in discrete terms, so that the devices considered are, in a loose sense, digital computers.
> (3) Lastly we suppose that the device is deterministic; that is, the subsequent behaviour of the device is uniquely determined once a complete description of its initial state is given.
> After these clarifications we can summarize our argument for a more definite version of Thesis M in the following way.
>
> **Thesis P.** *A discrete deterministic mechanical device satisfies principles I–IV below.*

Later, discussing how to describe computation states, Gandy says that he wants "the form of description to be sufficiently abstract to apply uniformly to mechanical, electrical or merely notional devices." After all the clarifications, it is not clear what Gandy's notion of machine is; see [18] in this connection. Gandy does presume that machine computations are sequential-time and isolated; these are two of the four constraints in §2. Sequential time parallelism is known as synchronous.

Principles I-IV are precise though require too many definitions to be stated precisely here. The four principles entail Gandy's main theorem: "What can be calculated by a machine is computable."

Principle I asserts in particular that, for any machine, the states can be described by hereditarily finite sets[3]and there is a transition function F such that, if x describes an initial state, then $Fx, F(Fx), \ldots$ describe the subsequent states. Principles II are III are technical restrictions on the state descriptions and the transition function respectively. Principle IV generalizes Kolmogorov's locality constraint to parallel computations.

> We now come to the most important of our principles. In Turing's analysis the requirement that the action depend only on a bounded portion of the record was based on a human limitation. We replace this by a physical limitation [Principle IV] which we call the *principle of local causation*. Its justification lies in the finite velocity of propagation of effects and signals: contemporary physics rejects the possibility of instantaneous action at a distance.

A preliminary version of Principle IV gives a good idea about the intentions behind the principle.

> **Principle IV** (Preliminary version). The next state, Fx, of a machine can be reassembled from its restrictions to overlapping "regions" s and these restrictions are locally caused. That is, for each region s of Fx there is a causal neighborhood $t \subseteq \mathrm{TC}(x)$ of bounded size such that $Fx \restriction s$ [the restriction of Fx to s] depends only on $x \restriction t$ [the restriction of x to t].

4.1 Comments

It isn't clear to us what Gandy's fulcrum was and even whether he had a fruitful viewpoint on machine computations. We recently [13] criticized Gandy's approach. Here we add just a few remarks.

The only parallelism that Gandy considers is synchronous. That is restrictive. Nowadays asynchronous machine computations are common.

[3]A set x is *hereditarily finite* if its transitive closure $\mathrm{TC}(x)$ is finite. Here $\mathrm{TC}(x)$ is the least set t such that $x \in t$ and such that $z \in y \in t$ implies $z \in t$.

The principle of local causality does not apply to all synchronous parallel algorithms. Gandy himself mentions one counterexample, namely Markov's normal algorithms [17]. The principle fails in the circuit model of parallel computation, the oldest model of parallel computation in computer theory. The reason is that the model allows gates to have unbounded fan-in. We illustrate this on the example of a first-order formula $\forall x R(x)$ where $R(x)$ is atomic. The formula gives rise to a collection of circuits C_n of depth 1. Circuit C_n has n input gates, and any unary relation R on $\{1, \ldots, n\}$ provides an input for C_n. Circuit C_n computes the truth value of the formula $\forall x R(x)$ in one step, and the value depends on the whole input. Ironically, it is easy to construct a Turing machine that simulates sequentially these parallel computations.

Hereditarily finite sets are finite. The finiteness constraint is understandable taking into account that Gandy's goal was to confirm Church's thesis. However, taking into account that Gandy's machines are isolated (and thus non-interactive), the finiteness constraint excludes some useful algorithms. For example it excludes a simple algorithm that consumes a stream of numbers keeping track of the maximum of the numbers seen so far. The finiteness constraint is not necessarily satisfied by Turing machines. In particular, a Turing machine can execute the stream algorithm above if the whole stream is written on its initial tape.

We accept that computation states can be described in set theoretic terms. A problem arises how to make the transition function work with such a description. Many of Gandy's technical problems are related to this problem, and indeed describing algorithms in Gandy's terms is rather challenging.

This said, let us emphasize that Gandy pioneered the axiomatic approach in foundational analysis of algorithms. He bravely attacked the hard problem of a general analysis of machine computations. Wilfried Sieg adopted Gandy's approach and reworked Gandy's axioms, see [23] and references there, but he did not clarify or justify Gandy's fulcrum. The problem of a general analysis of machine computations is wide open. In our view, the notion of machine computation is evolving and will be evolving for the foreseeable future; think of quantum computers for example. The notion has not matured enough to lend itself to formal analysis.

5 Analyzing computations on their native levels of abstraction

5.1 Motivation

By the 1980s, there were plenty of computers and software. A problem arose how to specify software. The most popular approaches to this problem were denotational semantics and algebraic specifications. Both approaches were proudly declarative. The declarative character of specifications was supposed to be an advantage. Indeed, declarative specifications tend to be more comprehensible, higher-level (that is of higher level of abstraction) and cleaner than operational, executable specifications, which is great. But executable specifications have their own advantages. You can"play" with them: run them, test, debug. In principle, you can verify properties of a declarative or executable spec mathematically, and sometimes you have to, and there are better and better tools to do that. In practice though, mathematical verification is out of the question in an overwhelming majority of cases, and the possibility to test specs is indispensable. Declarative specifications are static while software evolves. In most cases, it is virtually impossible to keep a declarative spec and an implementation in sync. In the case of an executable spec, you can test whether the implementation conforms to the spec (or, if the spec was reverse-engineered from an implementation, whether the spec is consistent with the implementation).

A question arises whether an executable specification have to be low-level and detailed? This leads to a theoretical, even foundational problem. Is there an executable specification of any algorithm A on the level of abstraction of A itself? For example, imagine that you conceived a wonderful algorithm. How would you specify it succinctly in an executable way? A natural-language explanation would not do as it is not executable. Besides, such an explanation may introduce ambiguities and misunderstanding. You can program your algorithm in a conventional programming language but this will surely introduce lower-level details.

Turing and Kolmogorov machines are executable but low-level. Consider for example Turing-machine implementations of these two versions of Euclid's algorithm for the greatest common divisor of two natural numbers: the original version where you advance by means of differences, and a faster (and higher-level) version where you advance by means of divisions. The

chances are that divisions were reduced to differences in the Turing machine implementation, and the distinction of the abstraction levels disappeared.

Can we generalize Turing and Kolmogorov machines in order to solve the foundational problem in question? The answer turns out to be positive, at least for sequential algorithms [12], synchronous parallel algorithms [2], and interactive algorithms [3, 4].

Following Kolmogorov, we consider computation in abstraction from the computer. Following Gandy, we use an axiomatic approach. The fulcrum for the sequential case is this. Every algorithm A has its native level of abstraction. On that level, the states can be faithfully represented by first-order structures of a fixed vocabulary in such a way that state transitions become just sets of assignments. The fulcrums for the parallel and interactive cases are built on this fulcrum. Here we restrict attention to sequential algorithms and do not cover parallel and interactive ones. Sequential algorithms are also known as classical as they had been virtually the only algorithms from time immemorial to the 1950s. The three stipulations of Kolmogorov in §3 give a great informal description of sequential algorithms. In the rest of this section, algorithms are by default sequential.

5.2 Constraints

Sequential Time *Any algorithm A is associated with a nonempty collection $\mathcal{S}(A)$ of states, a subcollection $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ of initial states and a (possibly partial) state transition map $\tau_A : \mathcal{S}(A) \longrightarrow \mathcal{S}(A)$.*

`Q:` Your algorithm A is deterministic: $\tau_A(X)$ is determined by state X. Why not to make τ_A multi-valued?

`A:` In our view, this would involve intra-step (within a single step) interaction with the environment [12, §9]. Intra-step interactive algorithms are analyzed in [3, 4]. Note in this connection that we do not rule out inter-step interaction with the environment. In other words, the environment can intervene between the steps of the algorithm A. If the intervention results in a legitimate state of A, the algorithm A continues to run. So, in general, the steps of A are interleaved with those of the environment, and thus the behavior of A is not necessarily determined by the initial step.

Recall that a first-order structure X is a nonempty set (the base set of X) with relations and operations; the vocabulary of X consists of the names of those relations and operations. For example, if the vocabulary of X consists of one binary relation then X is a directed graph.

Abstract State *The states of an algorithm A can be faithfully represented by first-order structures of the same finite vocabulary, which we call the vocabulary of A, in such a way that*

- *τ_A does not change the base set of a state,*
- *collections $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms, and*
- *any isomorphism from a state X to a state Y is also an isomorphism from $\tau_A(X)$ to $\tau_A(Y)$.*

`Q:` You claim that first-order structures are sufficiently general to faithfully represent the states of any algorithm?

`A:` I have been making that claim from the 1980s. The collective experience of computer science seems to corroborate the claim.

`Q:` But maybe the notion of first-order structure is too broad. Consider for example, natural numbers with the usual arithmetic relations and operations plus the unary relation $T(n)$ that is true if and only if the Turing machine number n halts on the empty tape. Starting with such a structure, a simple algorithm solves the halting problem.

`A:` Suppose, more generally, that T is produced by some process, not necessarily algorithmic. For example, T is the result of some measurement or coin flipping. How would you rule out your particular version of T?

`Q:` OK, but I have another question about the postulate. That base-set preservation sounds restrictive. A graph algorithm may extend the graph with new nodes.

`A:` And where will the algorithm take those nodes? From some reserve? Make that reserve a part of your initial state.

`Q:` Now, why should the collection of states be closed under isomorphisms, and why should the state transition respect isomorphisms?

`A:` Every algorithm works at its native level of abstraction. Irrelevant details should not matter. Consider a graph algorithm for example. In an implementation, nodes may be integer numbers, but the algorithm cannot examine whether a node is even or odd or which of the nodes is greater. These are implementation details irrelevant to the graph algorithm. And if the algorithm does take advantage of the integer representation of nodes then its vocabulary should reflect the relevant part of arithmetic.

According to Kolmogorov's informal definition of sequential algorithms, there is a bound on the amount of work done during any one step. But how to measure step complexity or the work done during one step? Fortunately the abstract state constraint helps.

Note that, according to the sequential-time constraint, the next state $\tau_A(X)$ of an algorithm A depends only on the current state X of A. The executor does not need to remember any history (even the current position in the program); all that is reflected in the state. If the executor is human and writes something on scratch paper, that paper should be a part of the computation state.

In order to change the given state X into $\tau_A(X)$, the algorithm A explores a portion of X and then performs the necessary changes of the values of the predicates and operations of X. According to Kolmogorov's informal definition, the explored portion, the "active zone", is bounded. And the change from X to $\tau_A(X)$, let us call it $\Delta_A(X)$, depends only on the results of exploration. Formally, $\Delta_A(X)$ can be defined as the collection of equations $F(\bar{a}) = b$ where b is a new value of a vocabulary function F at point $\bar{a}$.

But how does the algorithm know what to explore and what to change? That information is normally supplied by the program, and it should be applicable to all the states. In the light of the abstract state constraint, it should be given symbolically, in terms of the vocabulary of A.

Bounded Exploration *There exists a finite set T of terms (or expressions) in the vocabulary of algorithm A such that $\Delta_A(X) = \Delta_A(Y)$ whenever states X, Y of A coincide over T.*

5.3 Definition and the representation theorem

Now think of the sequential-time constraint as a postulate where $\mathcal{S}(A)$ is just a nonempty collection of things. Think of the abstract-state and bounded-exploration constraints as postulates that clarify/restrict what those things are and how the map τ_A works.

Definition 1 A (sequential) algorithm is any entity that satisfies the sequential-time, abstract-state and bounded-exploration postulate.

Abstract state machines (ASMs) were defined in [11]. Here we restrict attention to sequential ASMs, which are undoubtedly algorithms.

Theorem 1 ([12]) *For every algorithm A, there exists a sequential ASM with the same states and the same state transition function.*

5.4 Deriving Church's thesis

Our postulates do not entail Church's thesis. The reason is that initial states of sequential algorithms may be uncomputable. The halting problem for Turing machines may be encoded in an initial state. Think also of ruler-and-compass algorithms or the Gauss elimination procedure; they satisfy our postulates but cannot be simulated by Turing machines. An arithmetical-state postulate of [5] asserts that only undeniably-computable operations are available in initial states; see details in [5].

Theorem 2 ([5]) *Church's thesis follows from the sequential-time, abstract-state, bounded-exploration and arithmetical-state postulates.*

References

[1] Andreas Blass and Yuri Gurevich, "Algorithms: A quest for absolute definitions," in Current Trends in Theoretical Computer Science, World Scientific (G. Paun et. al, eds.), 2004, 283–311, and in Church's Thesis After 70 Years (A. Olszewski, ed.) Ontos Verlag, 2006, 24–57.

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

[3] 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 and III).

[4] 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).

[5] 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.

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

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

[8] R.O. (Robin) Gandy and C.E.M (Mike) Yates (eds.), "Collected works of A.M. Turing: Mathematical logic", Elsevier, 2001.

[9] Kurt Gödel, "A philosophical error in Turing's work," in Kurt Gödel: Collected Works," Volume II (S. Feferman et. al, eds.), Oxford University Press, 1990, p. 306.

[10] Yuri Gurevich, "On Kolmogorov machines and related issues," Bull. of Euro. Assoc. for Theor. Computer Science 35 (1988), 71-82.

[11] Yuri Gurevich, "Evolving algebra 1993: Lipari guide," in Specification and Validation Methods (E. Börger, ed.), Oxford Univ. Press (1995), 9–36.

[12] Yuri Gurevich, "Sequential abstract state machines capture sequential algorithms," ACM Trans. on Computational Logic 1:2 (2000), 77–111.

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

[14] Andrei N. Kolmogorov, "On the concept of algorithm", Uspekhi Mat. Nauk 8:4 (1953), 175–176, Russian.

[15] Andrei N. Kolmogorov and Vladimir A. Uspensky, "On the definition of algorithm", Uspekhi Mat. Nauk 13:4 (1958), 3–28, Russian. English translation in AMS Translations 29 (1963), 217–245.

[16] Leonid A. Levin, Private communication, 2003.

[17] Andrei A. Markov, "Theory of algorithms," Trans. of the Steklov Institute of Mathematics 42, 1954, Russian. English translation by the Israel Program for Scientific Translations, 1962; also by Kluwer, 2010.

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

[19] Oron Shagrir, "Gödel on Turing on computability," Church's Thesis after 70 years (A. Olszewski et. al, eds.), Ontos-Verlag, 2006, 393–419.

[20] Wilfried Sieg, "On computability," in Handbook of the Philosophy of Mathematics (A. Irvine, ed.), Elsevier, 2009, 535-630.

[21] Alan M. Turing, "On computable numbers, with an application to the Entscheidungsproblem", Proceedings of London Mathematical Society 2:42 (1936), 230–265.

[22] Vladimir A. Uspensky, "Kolmogorov and mathematical logic," Journal of Symbolic Logic 57:2 (1992), 385–412.

[23] Vladimir A. Uspensky and Alexei L. Semenov, Theory of algorithms: Main Discoveries and Applications, Nauka 1987 (Russian), Kluwer 2010 (English).