Algorithms: A Quest for Absolute Definitions

Andreas Blass* Yuri Gurevich†

Abstract

What is an algorithm? The interest in this foundational problem is not only theoretical; applications include specification, validation and verification of software and hardware systems. We describe the quest to understand and define the notion of algorithm. We start with the Church-Turing thesis and contrast Church's and Turing's approaches, and we finish with some recent investigations.

Contents

*Partially supported by NSF grant DMS–0070723 and by a grant from Microsoft Research.
Address: Mathematics Department, University of Michigan, Ann Arbor, MI 48109–1109.
†Microsoft Research, One Microsoft Way, Redmond, WA 98052.

1 Introduction

In 1936, Alonzo Church published a bold conjecture that only recursive functions are computable [10]. A few months later, independently of Church, Alan Turing published a powerful speculative proof of a similar conjecture: every computable real number is computable by the Turing machine [54]. Kurt Gödel found Church's thesis "thoroughly unsatisfactory" but later was convinced by Turing's argument. Later yet he worried about a possible flaw in Turing's argument. In Section 2 we recount briefly this fascinating story, provide references where the reader can find additional details, and give remarks of our own.

By now, there is overwhelming experimental evidence in favor of the Church-Turing thesis. Furthermore, it is often assumed that the Church-Turing thesis settled the problem of what an algorithm is. That isn't so. The thesis clarifies the notion of computable function. And there is more, much more to an algorithm than the function it computes. The thesis was a great step toward understanding algorithms, but it did not solve the problem what an algorithm is.

Further progress in foundations of algorithms was achieved by Kolmogorov and his student Uspensky in the 1950s [39, 40]. The Kolmogorov machine with its

reconfigurable "tape" has a certain advantage over the Turing machine. The notion of pointer machine was an improvement of the notion of Kolmogorov machine. These issues are discussed in Section 3

This paper started as a write-up of the talk that the second author gave at the Kolmogorov Centennial conference in June 2003 in Moscow. The talk raised several related issues: physics and computation, polynomial time versions of the Turing thesis, recursion and algorithms. These issues are very briefly discussed in Section 4.

In 1991, the second author published the definition of sequential abstract state machines (ASMs, called evolving algebras at the time) [23]. In 2000, he published a definition of sequential algorithms derived from first principles [27]. In the same paper he proved that every sequential algorithm A is behaviorally equivalent to some sequential ASM B. In particular, B simulates A step for step. In Section 5 we outline the approach of [27].

In 1995, the second author published the definition of parallel and distributed abstract state machines [25]. The Foundations of Software Engineering group at Microsoft Research developed an industrial strength specification language AsmL that allows one to write and execute parallel and distributed abstract state machines [2]. In 2001, the present authors published a definition of parallel algorithms derived from first principles as well as a proof that every parallel algorithm is equivalent to a parallel ASM [7]. Section 6 is a quick discussion of parallel algorithms.

The problem of defining distributed algorithms from first principles is open. In Section 7 we discuss a few related issues.

Finally let us note that foundational studies go beyond satisfying our curiosity. Turing machines with their honest counting of steps enabled computational complexity theory. Kolmogorov machines and pointer machines enabled better complexity measures. Abstract state machines enable precise executable specifications of software systems though this story is only starting to unfold [1, 2, 9].

2 The Church-Turing thesis

2.1 Church + Turing

The celebrated Church-Turing thesis [10, 54] captured the notion of computable function. Every computable function from natural numbers to natural numbers is recursive and computable, in principle, by the Turing machine. The thesis has been richly confirmed in practice. Speaking in 1946 at the Princeton Bicentennial Conference, Gödel said this [19, article 1946]:

> Tarski has stressed in his lecture (and I think justly) the great importance of the concept of general recursiveness (or Turing's computability). It seems to me that this importance is largely due to the fact that with this concept one has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen. In all other cases treated previously, such as demonstrability or definability, one has been able to define them only relative to the given language, and for each individual language it is clear that the one thus obtained is not the one looked for. For the concept of computability, however, although it is merely a special kind of demonstrability or decidability, the situation is different. By a kind of miracle it is not necessary to distinguish orders, and the diagonal procedure does not lead outside the defined notion.

2.2 Turing – Church

It became common to speak about the Church-Turing thesis. In fact the contributions of Church and Turing are different, and the difference between them is of importance to us here. Church's thesis was a bold hypothesis about the set of computable functions. Turing analyzed what can happen during a computation and thus arrived at his thesis.

Church's Thesis The notion of an effectively calculable function from natural numbers to natural numbers should be identified with that of a recursive function.

Church had in mind total functions [10]. Later Kleene improved on Church's thesis by extending it to partial functions [32]. The fascinating history of the thesis is recounted in [14]; see also [51].

Originally Church hypothesized that every effectively calculable function from natural numbers to natural numbers is definable in his lambda calculus. Gödel didn't buy that. In 1935, Church wrote to Kleene about his conversation with Gödel [14, Page 9].

> In discussion [*sic*] with him the notion of lambda-definability, it developed that there was no good definition of effective calculability. My proposal that lambda-definability be taken as a definition of it he regarded as thoroughly unsatisfactory. I replied that if he would propose any definition of effective calculability which seemed even partially satisfactory I would undertake to prove that it was included in lambda-definability. His only idea at the time was that it might be possible, in

> terms of effective calculability as an undefined notion, to state a set of axioms which would embody the generally accepted properties of this notion, and to do something on this basis.

Church continued:

> Evidently it occurred to him later that Herbrand's definition of recursiveness, which has no regard to effective calculability, could be modified in the direction of effective calculability, and he made this proposal in his lectures. At that time he did specifically raise the question of the connection between recursiveness in this new sense and effective calculability, but said he did not think that the two ideas could be satisfactorily identified "except heuristically".

The lectures of Gödel mentioned by Church were given at the Institute for Advanced Study in Princeton from February through May 1934. In a February 15, 1965, letter to Martin Davis, Gödel wrote [14, page 8]:

> However, I was, at the time of these lectures [1934], not at all convinced that my concept of recursion comprises all possible recursions.

Soon after Gödel's lectures, Church and Kleene proved that the Herbrand-Gödel notion of general recursivity is equivalent to lambda definability (as far as total functions are concerned), and Church became sufficiently convinced of the correctness of his thesis to publish it. But Gödel remained unconvinced.

Indeed, why should one believe that lambda definability captures the notion of computability? The fact that lambda definability is equivalent to general recursivity, and to various other formalizations of computability that quickly followed Church's paper, proves only that Church's notion of lambda definability is very robust.

To see that a mathematical definition captures the notion of computability, one needs an analysis of the latter. This is what Turing provided to justify his thesis.

Turing's Thesis Let Σ be a finite alphabet. A partial function from strings over Σ to strings over Σ is effectively calculable if and only if it is computable by a Turing machine.

Remark 2.1 Turing designed his machine to compute real numbers but the version of the Turing machine that became popular works with strings in a fixed alphabet. Hence our formulation of Turing's thesis.

Turing analyzed a computation performed by a human computer. He made a number of simplifying without-loss-of-generality assumptions. Here are some of them. The computer writes on graph paper; furthermore, the usual graph paper can be replaced with a tape divided into squares. The computer uses only a finite number of symbols, a single symbol in a square. "The behavior of the computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at that moment". There is a bound on the number of symbols observed at any one moment. "We will also suppose that the number of states of mind which need to be taken into account is finite... If we admitted an infinity of states of mind, some of them will be 'arbitrarily close' and will be confused". He ends up with a Turing machine simulating the original computation. Essentially Turing derived his thesis from more or less obvious first principles though he didn't state those first principles carefully.

"It seems that only after Turing's formulation appeared," writes Kleene in [33, Page 61], "did Gödel accept Church's thesis, which had then become the Church-Turing thesis." "Turing's arguments," he adds in [34, Page 48], "eventually persuaded him."

Church's lambda calculus was destined to play an important role in programming theory. The mathematically elegant Herbrand-Gödel-Kleene notion of partial recursive functions served as a springboard for many developments in recursion theory. The Turing machine gave us honest step counting and became eventually the foundation of complexity theory.

2.3 Remarks on Turing's analysis

Very quickly the Church-Turing thesis acquired the status of a widely shared belief. Meantime Gödel grew skeptical of at least one aspect of Turing's analysis. In a remark published after his death, Gödel writes this [19, article 1972a, page 306].

> *A philosophical error in Turing's work.* Turing in his [54, page 250], gives an argument which is supposed to show that mental procedures cannot go beyond mechanical procedures. However, this argument is inconclusive. 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. Therefore, although at each stage the number and precision of the abstract terms at our disposal may be *finite*, both (and therefore, also Turing's number of *distinguishable states of mind*) may *converge toward infinity* in the

course of the application of the procedure.

Gödel was extremely careful in his published work. It is not clear whether the remark in question was intended for publication as is. In any case, the question whether mental procedures can go beyond mechanical procedures is beyond the scope of this paper, which focuses on algorithms. Furthermore, as far as we can see, Turing did not intend to show that mental procedures cannot go beyond mechanical procedures. The expression "state of mind" was just a useful metaphor that could be and in fact was eliminated: "we avoid introducing the 'state of mind' by considering a more physical and definite counterpart of it" [54, Page 253].

But let us consider the possibility that Gödel didn't speak about biology either, that he continued to use Turing's metaphor and worried that Turing's analysis does not apply to some algorithms. Can an algorithm learn from its own experience, become more sophisticated and thus compute a real number that is not computable by the Turing machine? Note that the learning process in question is highly unusual because it involves no interaction with the environment. (On the other hand, it is hard to stop brains from interacting with the environment.) Gödel gives two examples "illustrating the situation", both aimed at logicians.

> Note that something like this indeed seems to happen in the process of forming stronger and stronger axioms of infinity in set theory. This process, however, today is far from being sufficiently understood to form a well-defined procedure. It must be admitted that the construction of a well-defined procedure which could actually be carried out (and would yield a non-recursive number-theoretic function) would require a substantial advance in our understanding of the basic concepts of mathematics. Another example illustrating the situation is the process of systematically constructing, by their distinguished sequences $\alpha_n \to \alpha$, all recursive ordinals α of the second number-class.

The logic community has not been swayed. "I think it is pie in the sky!" wrote Kleene [34, page 51]. Here is a more expansive reaction of his [34, page 50].

> But, as I have said, our idea of an algorithm has been such that, in over two thousand years of examples, it has separated cases when mathematicians have agreed that a given procedure constitutes an algorithm from cases in which it does not. Thus algorithms have been procedures that mathematicians can describe completely to one another *in advance* of their application for various choices of the arguments. How could someone describe completely to me *in a finite interview* a process for finding the values of a number-theoretic function, the execution of which process for various arguments would be keyed to more than the

> *finite* subset of our mental states that would have developed by the end of the interview, though the total number of mental states might converge to infinity if we were immortal? Thus Gödel's remarks do not shake my belief in the Church-Turing thesis ...

If Gödel's remarks are intended to attack the Church-Turing thesis, then the attack is a long shot indeed. On the other hand, we disagree with Kleene that the notion of algorithm is that well understood. In fact the notion of algorithm is richer these days than it was in Turing's days. And there are algorithms, of modern and classical varieties, not covered directly by Turing's analysis, for example, algorithms that interact with their environments, algorithms whose inputs are abstract structures, and geometric or, more generally, non-discrete algorithms. We look briefly at the three examples just mentioned.

Interactive algorithms This is a broad class. It includes randomized algorithms; you need the environment to provide random bits. It includes asynchronous algorithms; the environment influences action timing. It includes nondeterministic algorithms as well [27, section 9.1]. Clearly, interactive algorithms are not covered by Turing's analysis. And indeed an interactive algorithm can compute a non-recursive function. (The nondeterministic Turing machines, defined in computation theory courses, are known to compute only partial recursive functions. But a particular computation of such a machine cannot in general be simulated by a deterministic Turing machine.)

Computing with abstract structures Consider the following algorithm P that, given a finite connected graph $G = (V, E)$ with a distinguished vertex s, computes the maximum distance of any vertex from s.

A $S := \{s\}$ and $r := 0$.

B If $S = V$ then halt and output r.

C If $S \neq V$ then $S := S \cup \{y : \exists x\,(x \in S \wedge E(x, y))\}$ and $r := r + 1$.

D Go to **B**.

P is a parallel algorithm. Following Turing's analysis we have to break the assignment $S := \{y : \exists x\,(x \in S \wedge E(x, y))\}$ into small tasks of bounded complexity, e.g. by going systematically though every $x \in S$ and every neighbor y of x. But how will the algorithm go through all $x \in S$? The graph G is not ordered. A nondeterministic algorithm can pick an arbitrary vertex and declare it the first

vertex, pick one of the remaining vertices and declare it the second vertex, etc. But a deterministic algorithm cannot do that.

Algorithms like P are not covered directly by Turing's analysis. But there is an easy patch if you don't care about resources and use parallelism. Let n be the number of vertices. In parallel, the desired algorithm orders the vertices in all $n!$ possible ways and then carries on all $n!$ computations.

Non-discrete computations Turing dealt with discrete computations. His analysis does not apply directly e.g. to the classical, geometrical ruler-and-compass algorithms. The particular case of ruler-and-compass algorithms can be taken care of; such algorithms do not allow you to compute a non-recursive function [36]. In general, however, it is not clear how to extend Turing's analysis to non-discrete algorithms.

3 Kolmogorov machines and pointer machines

The problem of the absolute definition of algorithm was attacked again in 1953 by Andrei N. Kolmogorov; see the one-page abstract [39] of his March 17, 1953, talk at the Moscow Mathematical Society. Kolmogorov spelled out his intuitive ideas about algorithms. For brevity, we express them in our own words (rather than translate literally).

- An algorithmic process splits into steps whose complexity is bounded in advance, i.e., the bound is independent of the input and the current state of the computation.
- Each step consists of a direct and immediate transformation of the current state.
- This transformation applies only to the active part of the state and does not alter the remainder of the state.
- The size of the active part is bounded in advance.
- The process runs until either the next step is impossible or a signal says the solution has been reached.

In addition to these intuitive ideas, Kolmogorov gave a one-paragraph sketch of a new computation model. The ideas of [39] were developed in the article [40] written by Kolmogorov together with his student Vladimir A. Uspensky. The Kolmogorov machine model can be thought of as a generalization of the Turing machine model

where the tape is a directed graph of bounded in-degree and bounded out-degree. The vertices of the graph correspond to Turing's squares; each vertex has a color chosen from a fixed finite palette of vertex colors; one of the vertices is the current computation center. Each edge has a color chosen from a fixed finite palette of edge colors; distinct edges from the same node have different colors. The program has this form: replace the vicinity U of a fixed radius around the central node by a new vicinity W that depends on the isomorphism type of the digraph U with the colors and the distinguished central vertex. Contrary to Turing's tape whose topology is fixed, Kolmogorov's "tape" is reconfigurable.

Remark 3.1 We took liberties in describing Kolmogorov machines. Kolmogorov and Uspensky require that the tape graph is symmetric — for every edge (x, y) there is an edge (y, x). The more liberal model is a bit easer to describe. And the symmetry requirement is inessential in the following sense: any machine of either kind can be step-for-step simulated by a machine of the other kind.

Like Turing machines, Kolmogorov machines compute functions from strings to strings; we skip the description of the input and output conventions. In the footnote to the article title, Kolmogorov and Uspensky write that they just wanted to analyze the existing definitions of the notions of computable functions and algorithms and to convince themselves that there is no hidden way to extend the notion of computable function. Indeed, Kolmogorov machines compute exactly Turing computable functions. It seems, however, that they were more ambitious. Here is a somewhat liberal translation from [40, page 16].

> To simplify the description of algorithms, we introduced some conventions that are not intrinsic to the general idea, but it seems to us that the generality of the proposed definition remains plausible in spite of the conventions. It seems plausible to us that an arbitrary algorithmic process satisfies our definition of algorithms. We would like to emphasize that we are talking not about a reduction of an arbitrary algorithm to an algorithm in the sense of our definition but that every algorithm essentially satisfies the proposed definition.

In this connection the second author formulated a Kolmogorov-Uspensky thesis [22, page 227]: "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 concurred [55, page 396].

Kolmogorov's approach proved to be fruitful. It led to a more realistic complexity theory. For example, given a string x, a Kolmogorov machine can build a binary tree over x and then move fast about x. Leonid Levin used a universal Kolmogorov machine to construct his algorithm for NP problems that is optimal

up to a multiplicative constant [41, 22]. The up-to-a-multiplicative-constant form is not believed to be achievable for the multitape Turing machine model popular in theoretical computer science. Similarly, the class of functions computable in *nearly linear* time $n(\log n)^{O(1)}$ on Kolmogorov machines remains the same if Kolmogorov machines are replaced e.g. by various random access computers in the literature; it is not believed, however, that the usual multitape Turing machines have the same power [29].

Kolmogorov machines allow one to do reasonable computations in reasonable time. This may have provoked Kolmogorov to ask new questions. "Kolmogorov ran a complexity seminar in the 50s or early 60s," wrote Leonid Levin, a student of Kolmogorov, to us [42]. "He asked if common tasks, like integer multiplication, require necessarily as much time as used by common algorithms, in this case quadratic time. Unexpectedly, Karatsuba reduced the power to $\log_2(3)$ [31]." (Readers interested in fast integer multiplication are referred to [38].)

It is not clear to us how Kolmogorov thought of the tape graph. One hypothesis is that edges reflect physical closeness. This hypothesis collides with the fact that our physical space is finite-dimensional. As one of us remarked earlier [27, page 81], "In a finite-dimensional Euclidean space, the volume of a sphere of radius n is bounded by a polynomial of n. Accordingly, one might expect a polynomial bound on the number of vertices in any vicinity of radius n (in the graph theoretic sense) of any state of a given KU machine, but in fact such a vicinity may contain exponentially many vertices."

Another hypothesis is that edges are some kind of channels. This hypothesis too collides with the fact that our physical space is finite-dimensional.

Probably the most natural approach would be to think of informational rather than physical edges. If vertex a contains information about the whereabouts of b, draw an edge from a to b. It is reasonable to assume that the amount of information stored at every single vertex a is bounded, and so the out-degree of the tape graph is bounded. It is also reasonable to allow more and more vertices to have information about b as the computation proceeds, so that the in-degree of the tape graph is unbounded. This brings us to Schönhage machines. These can be seen as Kolmogorov machines (in the version with directed edges) except that only the out-degrees are required to be bounded. The in-degrees can depend on the input and, even for a particular input, can grow during the computation.

"In 1970 the present author introduced a new machine model (cf. [47]) now called *storage modification machine* (SMM)," writes Schönhage in [48], "and posed the intuitive thesis that this model possesses extreme flexibility and should therefore serve as a basis for an adequate notion of time complexity." In article [48], Schönhage gave "a comprehensive presentation of our present knowledge of SMMs". In particular, he proved that SMMs are "real-time equivalent" to

successor RAMs (random access machines whose only arithmetical operation is $n \mapsto n+1$). The following definitions appear in [48, page 491].

Definition 3.2 A machine M' is said to *simulate* another machine M "in real time", denoted $M \xrightarrow{r} M'$, if there is a constant c such that for every input sequence x the following holds: if x causes M to read an input symbol, or to print an output symbol, or to halt at time steps $0 = t_0 < t_1 < \cdots < t_l$, respectively, then x causes M' to act in the very same way with regard to those external actions at time steps $0 = t'_0 < t'_1 < \cdots < t'_l$ where $t'_j - t'_{j-1} \leq c(t_j - t_{j-1})$ for $1 \leq j \leq l$. For machine classes $\mathcal{M}, \mathcal{M}'$ *real time reducibility* $\mathcal{M} \xrightarrow{r} \mathcal{M}'$ is defined by the condition that for each $M \in \mathcal{M}$ there exists an $M' \in \mathcal{M}'$ such that $M \xrightarrow{r} M'$. *Real time equivalence* $\mathcal{M} \xleftrightarrow{r} \mathcal{M}'$ means $\mathcal{M} \xrightarrow{r} \mathcal{M}'$ and $\mathcal{M}' \xrightarrow{r} \mathcal{M}$. $\square$

Dima Grigoriev proved that Turing machines cannot simulate Kolmogorov machines in real time [21].

Schönhage introduced a precise language for programming his machines and complained that the Kolmogorov-Uspensky description of Kolmogorov machines is clumsy. For our purposes here, however, it is simplest to describe Schönhage machines as generalized Kolmogorov machines where the in-degree of the tape graph may be unbounded. It is still an open problem whether Schönhage machines are real time reducible to Kolmogorov machines.

Schönhage states his thesis as follows: "$\mathcal{M} \xrightarrow{r}$ SMM holds for all *atomistic* machine models $\mathcal{M}$."

Schönhage writes that Donald E. Knuth "brought to his attention that the SMM model coincides with a special type of 'linking automata' briefly explained in volume one of his book (cf. [37, pages 462-463]) in 1968 already. Now he suggests calling them 'pointer machines' which, in fact, seems to be the adequate name for these automata." Note that Kolmogorov machines also modify their storage. But the name "pointer machine" fits Knuth-Schönhage machines better than it fits Kolmogorov machines.

A successor RAM is a nice example of a pointer machine. Its tape graph consists of natural numbers and a couple of special registers. Each special register has only one pointer, which points to a natural number that is intuitively the content of the register. Every natural number n has only a pointer to $n+1$, a pointer to another natural number that is intuitively the content of register n, and a pointer to every special register.

The notion of pointer machine seems an improvement over the notion of Kolmogorov machine to us (and of course the notion of Kolmogorov machine was an improvement over the notion of Turing machine). And the notion of pointer machine proved to be useful in the analysis of the time complexity of algorithms. In

that sense it was successful. It is less clear how much of an advance all these developments were from the point of view of absolute definitions. The pointer machine reflected the computer architecture of real computers of the time. (The modern tendency is to make computers with several CPUs, central processing units, that run asynchronously.)

Remark 3.3 In an influential 1979 article, Tarjan used the term "pointer machine" in a wider sense [53]. This wider notion of pointer machines has become better known in computer science than the older notion.

4 Related issues

We mention a few issues touched upon in the talk that was the precursor of this paper. It is beyond the scope of this paper to develop these issues in any depth.

4.1 Physics and computations

What kind of computations can be carried out in our physical universe? We are not talking about what functions are computable. The question is what algorithms are physically executable. We don't expect a definitive answer soon, if ever. It is important, however, to put things into perspective. Many computer science concerns are above the level of physics. It would be great if quantum physics allowed us to factor numbers fast, but this probably will not greatly influence programming language theory.

Here are some interesting references.

- Robin Gandy attempted to derive Turing's thesis from a number of "principles for mechanisms" [17]. Wilfried Sieg continues this line of research [52].
- David Deutsch [15] designed a universal quantum computer that is supposed to be able to simulate the behavior of any finite physical system. Gandy's approach is criticized in [16, pages 280–281]. Deutsch's approach and quantum computers in general are criticized in [43, Section 2].
- Charles H. Bennett and Rolf Landauer pose in [3] important problems related to the fundamental physical limits of computation.
- Marian Boykan Pour-El and Ian Richards [45] investigate the extent to which computability is preserved by fundamental constructions of analysis, such as those used in classical and quantum theories of physics.

4.2 Polynomial time Turing's thesis

There are several versions of the polynomial time Turing's thesis discussed in theoretical computer science. For simplicity, we restrict attention to decision problems.

To justify the interest in the class P of problems solvable in polynomial time by a Turing machine, it is often declared that a problem is feasible (= practically solvable) if and only if it is in P. Complexity theory tells us that there are P problems unsolvable in time n^{1000}. A more reasonable thesis is that a "natural problem" is feasible if and only if it is in P. At the 1991 Annual Meeting of the Association of Symbolic Logic, Steve Cook argued in favor of that thesis, and the second author argued against it. Some of the arguments can be found in [11] and [24] respectively.

A related but different version of the polynomial time Turing thesis is that a problem is in P if it can be solved in polynomial time at all, by any means. The presumed reason is that any polynomial time computation can be polynomial time simulated by a Turing machine (so that the computation time of the Turing machine is bounded by a polynomial of the computation time of the given computing device). Indeed, most "reasonable" computation models are known to be polytime equivalent to the Turing machine. "As to the objection that Turing machines predated all of these models," says Steve Cook [12], "I would reply that models based on RAMs are inspired by real computers, rather than Turing machines."

Quantum computer models can factor arbitrary integers in polynomial time [50], and it is not believed that quantum computers can be polynomial time simulated by Turing machines. For the believers in quantum computers, it is more natural to speak about probabilistic Turing machines. We quote from [4].

> Just as the theory of computability has its foundations in the Church-Turing thesis, computational complexity theory rests upon a modern strengthening of this thesis, which asserts that any "reasonable" model of computation can be efficiently simulated on a probabilistic Turing Machine (an efficient simulation is one whose running time is bounded by some polynomial in the running time of the simulated machine). Here, we take reasonable to mean in principle physically realizable.

Turing's analysis does not automatically justify any of these new theses. (Nor does it justify, for example, the thesis that polynomial time interactive Turing machines capture polynomial time interactive algorithms.) Can any of the theses discussed above be derived from first principles? One can analyze Turing's original justification of his thesis and see whether all the reductions used by Turing are polynomial time reductions. But one has to worry also about algorithms not covered directly by Turing's analysis.

4.3 Recursion

According to Yiannis Moschovakis, an algorithm is a "recursor", a monotone operator over partial functions whose least fixed point includes (as one component) the function that the algorithm computes [44]. He proposes a particular language for defining recursors. A definition may use various givens: functions or recursors.

Moschovakis gives few examples and they are all small ones. The approach does not seem to scale to algorithms interacting with an unknown environment. A posteriori the approach applies to well understood classes of algorithms. Consider for example non-interactive sequential or parallel abstract state machines (ASMs) discussed below in Sections 5 and 6. Such an ASM has a program for doing a single step. There is an implicit iteration loop: repeat the step until, if ever, the computation terminates. Consider an operator that, given an initial segment of a computation, augments it by another step (unless the computation has terminated). This operator can be seen as a recursor. Of course the recursion advocates may not like such a recursor because they prefer stateless ways.

We are not aware of any way to derive from first principles the thesis that algorithms are recursors.

5 Formalization of sequential algorithms

Is it possible to capture (= formalize) sequential algorithms on their natural levels of abstraction? Furthermore, is there one machine model that captures all sequential algorithms on their natural levels of abstraction? According to [27], the answer to both questions is yes. We outline the approach of [27] and put forward a slight but useful generalization.

As a running example of a sequential algorithm, we use a version Euc of Euclid's algorithm that, given two natural numbers, computes their greatest common divisor d.

```
1. Set a = Input1, b = Input2.
2. If a = 0 then set d = b and go to 1
   else set a, b = b mod a, a respectively and go to 2.
```

Initially Euc waits for the user to provide natural numbers Input1 and Input2. The assignment on the last line is simultaneous. If, for instance, $a = 6$ and $b = 9$ in the current state then $a = 3$ and $b = 6$ in the next state.

5.1 Sequential Time Postulate

A sequential algorithm can be viewed as a finite or infinite state automaton.

Postulate 1 (Sequential Time) *A sequential algorithm A is associated with*

- *a nonempty set $\mathcal{S}(A)$ whose members are called* states *of A,*
- *a nonempty*[1] *subset $\mathcal{I}(A)$ of $\mathcal{S}(A)$ whose members are called* initial states *of A, and*
- *a map $\tau_A : \mathcal{S}(A) \longrightarrow \mathcal{S}(A)$ called the* one-step transformation *of A.*

The postulate ignores final states [27, section 3.3.2]. We are interested in runs where the steps of the algorithm are interleaved with the steps of the environment. A step of the environment consists in changing the current state of the algorithm to any other state. In particular it can change the "final" state to a non-final state. To make the one-step transformation total, assume that the algorithm performs an idle step in the "final" states. Clearly Euc is a sequential time algorithm. The environment of Euc includes the user who provides input numbers (and is expected to take note of the answers).

This sequential-time postulate allows us to define a fine notion of behavioral equivalence.

Definition 5.1 Two sequential time algorithms are behaviorally equivalent if they have the same states, the same initial states and the same one-step transformation.

The behavioral equivalence is too fine for many purposes but it is necessary for the following.

Corollary 5.2 *If algorithms A and B are behaviorally equivalent then B step-for-step simulates A in any environment.*

The step-for-step character of simulation is important. Consider a typical distributed system. The agents are sequential-time but the system is not. The system guarantees the atomicity of any single step of any agent but not of a sequence of agent's steps. Let A be the algorithm executed by one of the agents. If the simulating algorithm B makes two steps to simulate one step of A then another agent can sneak in between the two steps of B and spoil the simulation.

[1] In [27], $\mathcal{I}(A)$ and $\mathcal{S}(A)$ were not required to be nonempty. But an algorithm without an initial state couldn't be run, so is it really an algorithm? We therefore add "nonempty" to the postulate here.

5.2 Small-step algorithms

An object that satisfies the sequential-time postulate doesn't have to be an algorithm. In addition we should require that there is a program for the one-step transformation. This requirement is hard to capture directly. It will follow from other requirements in the approach of [27].

Further, a sequential-time algorithm is not necessarily a sequential algorithm. For example, the algorithm P in subsection 2.3 is not sequential. The property that distinguishes sequential algorithms among all sequential-time algorithms is that the steps are of bounded complexity. The algorithms analyzed by Turing in [54] were sequential:

> The behavior of the computer at any moment is determined by the symbols which he is observing and his 'state of mind' at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite.

The algorithms analyzed by Kolmogorov in [39] are also sequential: "An algorithmic process is divided into separate steps of limited complexity."

These days there is a tendency to use the term "sequential algorithm" in the wider sense of the contrary of the notion of a distributed algorithm. That is, "sequential" often means what we have called "sequential-time". So we use the term "small-step algorithm" as a synonym for the term "sequential algorithms" in its traditional meaning.

5.3 Abstract State Postulate

How does one capture the restriction that the steps of a small-step algorithms are of bounded complexity? How does one measure the complexity of a single-step computation? Actually we prefer to think of bounded work instead of bounded complexity. The work that a small-step algorithm performs at any single step is bounded, and the bound depends only on the algorithm and does not depend on input. This complexity-to-work reformulation does not make the problem easier of course. How does one measure the work that the algorithm does during a step? The algorithm-as-a-state-automaton point of view is too simplistic to address the problem. We need to know more about what the states are. Fortunately this question can be answered.

Postulate 2 (Abstract State)

- *States of a sequential algorithm A are first-order structures.*
- *All states of A have the same vocabulary.*
- *The one-step transformation τ_A does not change the base set of any state.*
- *$\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms. Further, any isomorphism from a state X onto a state Y is also an isomorphism from $\tau_A(X)$ onto $\tau_A(Y)$.*

The notion of first-order structure is well-known in mathematical logic [49]. We use the following conventions:

- Every vocabulary contains the following *logic symbols:* the equality sign, the nullary relation symbols `true` and `false`, and the usual Boolean connectives.
- Every vocabulary contains the nullary function symbol `undef`.
- Some vocabulary symbols may be marked *static*. The remaining symbols are marked *external* or *dynamic* or both[2]. All logic symbols are static.
- In every structure, `true` is distinct from `false` and `undef`, the equality sign has its standard meaning, and the Boolean connectives have their standard meanings on Boolean arguments.

The symbols `true` and `false` allow us to treat relation symbols as special function symbols. The symbol `undef` allows us to deal with partial functions; recall that first-order structures have only total functions. The static functions (that is the interpretations of the static function symbols) do not change during the computation. The algorithm can change only the dynamic functions. The environment can change only the external functions.

It is easy to see that higher-order structures are also first-order structures (though higher-order logics are richer than first-order logic). We refer to [27] for justification of the abstract-state postulate. Let us just note that the experience of the ASM community confirms that first-order structures suffice to describe any static mathematical situation [1].

It is often said that a state is given by the values of its variables. We take this literally. Any state of a sequential algorithm should be uniquely determined (in

[2]This useful classification, used in [23, 25] and in ASM applications, was omitted in [27] because it wasn't necessary there. The omission allowed the following pathology in the case when there is a finite bound on the size of the states of A. The one-step transformation may change the values of `true` and `false` and modify appropriately the interpretations of the equality relation and the Boolean connectives.

the space of all states of the algorithm) by the interpretations of the dynamic and external function symbols.

What is the vocabulary (of the states) of Euc? In addition to the logic symbols, it contains the nullary function symbols `0, a, b, d, Input1, Input2` and the binary function symbol `mod`. But what about labels `1` and `2`? Euc has an implicit program counter. We have some freedom in making it explicit. One possibility is to introduce a Boolean variable, that is a nullary relational symbol, `initialize` that takes value `true` exactly in those states where Euc consumes inputs. The only dynamic symbols are `a, b, d, initialize`, and the only external symbols are `Input1, Input2`.

5.4 Bounded Exploration Postulate and the definition of sequential algorithms

Let A be an algorithm of vocabulary Υ and let X be a state of A. A *location* ℓ of X is given by a dynamic function symbol f in Υ of some arity j and a j-tuple $\bar{a} = (a_1, \ldots, a_j)$ of elements of X. The *content* of ℓ is the value $f(\bar{a})$.

An *(atomic) update* of X is given by a location ℓ and an element b of X and denoted simply (ℓ, b). It is the action of replacing the current content a of ℓ with b.

By the abstract-state postulate, the one-step transformation preserves the set of locations, so the state X and the state $X' = \tau_A(X)$ have the same locations. It follows that X' is obtained from X by executing the following set of updates:

$$\Delta(X) = \{(\ell, b) : \quad b = Content_{X'}(\ell) \neq Content_X(\ell)\}$$

If A is Euc and X is the state where $a = 6$ and $b = 9$ then $\Delta(X) = \{(a, 3), (b, 6)\}$. If Y is a state of A where $a = b = 3$ then $\Delta(Y) = \{(a, 0)\}$.

Now we are ready to formulate the final postulate. Let X, Y be arbitrary states of the algorithm A.

Postulate 3 (Bounded Exploration) *There exists a finite set T of terms in the vocabulary of A such that $\Delta(X) = \Delta(Y)$ whenever every term $t \in T$ has the same value in X and Y.*

In the case of Euc, the term set $\{\texttt{true}, \texttt{false}, 0, a, b, d, b \ mod \ a, \texttt{initialize}\}$ is a bounded-exploration witness.

Definition 5.3 A *sequential algorithm* is an object A that satisfies the sequential-time, abstract-state and bounded-exploration postulates.

5.5 Sequential ASMs and the characterization theorem

The notion of a sequential ASM rule of a vocabulary Υ is defined by induction. In the following definition, all function symbols (including relation symbols) are in Υ and all terms are first-order terms.

Definition 5.4 If f is a j-ary dynamic function symbol and $t_0, \ldots, t_j$ are first-order terms then the following is a rule:

$f(t_1, \ldots, t_j) := t_0$.

Let φ be a Boolean-valued term, that is φ has the form $f(t_1, \ldots, t_j)$ where f is a relation symbol. If P_1, P_2 are rules then so is

`if` φ `then` P_1 `else` P_2.

If P_1, P_2 are rules then so is

```
do in-parallel
   P1
   P2
```

The semantics of rules is pretty obvious but we have to decide what happens if the constituents of the `do in-parallel` rule produce contradictory updates. In that case the execution is aborted. For a more formal definition, we refer the reader to [27]. Syntactically, a sequential ASM program is just a rule; but the rule determines only single steps of the program and is supposed to be iterated. Every sequential ASM program P gives rise to a map $\tau_P(X) = Y$ where X, Y are first-order Υ-structures.

Definition 5.5 A sequential ASM B of vocabulary Υ is given by a sequential ASM program Π of vocabulary Υ, a nonempty set $\mathcal{S}(B)$ of Υ-structures closed under isomorphisms and under the map τ_Π, a nonempty subset $\mathcal{I}(B) \subseteq \mathcal{S}(B)$ that is closed under isomorphisms, and the map τ_B which is the restriction of τ_Π to $\mathcal{S}(B)$.

Now we are ready to formulate the theorem of this section.

Theorem 5.6 (ASM Characterization of Small-Step Algorithms) *For every sequential algorithm A there is a sequential abstract state machine B behaviorally equivalent to A. In particular, B simulates A step for step.*

If A is our old friend Euc, then the program of the desired ASM B could be this.

```
if initialize then
   do in-parallel
      a := Input1
      b := Input2
      initialize := false
else
   if a = 0 then
      do in-parallel
         d := b
         initialize := true
   else
      do in-parallel
         a := b mod a
         b := a
```

We have discussed only deterministic sequential algorithms. Nondeterminism implicitly appeals to the environment to make the choices that cannot be algorithmically prescribed [27]. Once nondeterminism is available, classical ruler-and-compass constructions can be regarded as nondeterministic ASMs operating on a suitable structure of geometric objects.

A critical examination of [27] is found in [46].

6 Formalization of parallel algorithms

Encouraged by the success in capturing the notion of sequential algorithms in [27], we "attacked" parallel algorithms in [7]. The attack succeeded. We gave an axiomatic definition of parallel algorithms and checked that the known (to us) parallel algorithm models satisfy the axioms. We defined precisely a version of parallel abstract state machines, a variant of the notion of parallel ASMs from [25], and we checked that our parallel ASMs satisfy the definitions of parallel algorithms. And we proved the characterization theorem for parallel algorithms: every parallel algorithm is behaviorally equivalent to a parallel ASM.

The scope of this paper does not allow us to spell out the axiomatization of parallel ASMs, which is more involved than the axiomatization of sequential ASMs described in the previous section. We just explain what kind of parallelism we have in mind, say a few words about the axioms, say a few words about the parallel ASMs, and formulate the characterization theorem. The interested reader is invited to read — critically! — the paper [7]. More scrutiny of that paper is highly desired.

6.1 What parallel algorithms?

The term "parallel algorithm" is used for a number of different notions in the literature. We have in mind sequential-time algorithms that can exhibit unbounded parallelism but only bounded sequentiality within a single step. Bounded sequentiality means that there is an *a priori* bound on the lengths of sequences of events within any one step of the algorithm that must occur in a specified order. To distinguish this notion of parallel algorithms, we call such parallel algorithms *wide-step*. Intuitively the width is the amount of parallelism. The "step" in "wide-step" alludes to sequential time.

Remark 6.1 Wide-step algorithms are also bounded-depth where the depth is intuitively the amount of sequentiality in a single step; this gives rise to a possible alternative name *shallow-step algorithms* for wide-step algorithms. Note that the name "parallel" emphasizes the potential rather than restrictions; in the same spirit, we choose "wide-step" over "shallow-step".

Here is an example of a wide-step algorithm that, given a directed graph $G = (V, E)$, marks the well-founded part of G. Initially no vertex is marked.

```
1. For every vertex x do the following.
   If every vertex y with an edge to x is marked
   then mark x as well.
2. Repeat step 1 until no new vertices are marked.
```

6.2 A few words on the axioms for wide-step algorithms

Adapt the sequential-time postulate, the definition of behavioral equivalence and the abstract-state postulate to parallel algorithms simply by replacing "sequential" with "parallel". The bounded-exploration postulate, on the other hand, specifically describes sequential algorithms. The work that a parallel algorithm performs within a single step can be unbounded. We must drop the bounded-exploration postulate and assume, in its place, an axiom or axioms specifically designed for parallelism.

A key observation is that a parallel computation consists of a number of processes running (not surprisingly) in parallel. The constituent processes can be parallel as well. But if we analyze the computation far enough then we arrive at processes, which we call *proclets*, that satisfy the bounded-exploration postulate. Several postulates describe how the proclets communicate with each other and how they produce updates. And there is a postulate requiring some bound d (depending only on the algorithm) for the amount of sequentiality in the program. The length

of any sequence of events that must occur in a specified order within any one step of the algorithm is at most d.

There are several computation models for wide-step algorithms in the literature. The two most known models are Boolean circuits and PRAMs [35]. (PRAM stands for "Parallel Random Access Machines".) These two models and some other models of wide-step algorithms that occurred to us or to the referees are shown to satisfy the wide-step postulates in [7].

6.3 Wide-step abstract state machines

Parallel abstract state machines were defined in [25]. Various semantical issues were elaborated later in [26]. A simple version of parallel ASMs was explored in [8]; these ASMs can be called BGS ASMs. We describe, up to an isomorphism, an arbitrary state X of a BGS ASM. X is closed under finite sets (every finite set of elements of X constitutes another element of X) and is equipped with the usual set-theoretic operations. Thus X is infinite but a finite part of X contains all the essential information. The number of *atoms* of X, that is elements that are not sets, is finite, and there is a nullary function symbol Atoms interpreted as the set of all atoms. It is easy to write a BGS ASM program that simulates the example parallel algorithm above.

```
forall x ∈ Atoms
   if {y :  y ∈ Atoms :  E(y,x) ∧ ¬(M(y))} = ∅
   then M(x) := true
```

Note that x and y are mathematical variables like the variables of first-order logic. They are not programming variables and cannot be assigned values. In comparison to the case of sequential ASMs, there are two main new features in the syntax of BGS ASMs:

- set-comprehension terms $\{t(x) : x \in r :\ \varphi(x)\}$, and
- `forall` rules.

In [6], we introduced the notion of a background of an ASM. BGS ASMs have a set background. The specification language AsmL, mentioned in the introduction, has a rich background that includes a set background, a sequence background, a map background, etc. The background that naturally arises in the analysis of wide-step algorithms is a multiset background. That is the background used in [7].

6.4 The wide-step characterization theorem

Theorem 6.2 (ASM Characterization of Wide-Step Algorithms) *For every parallel algorithm A there is a parallel abstract state machine B behaviorally equivalent to A. In particular, B simulates A step for step.*

Thus, Boolean circuits and PRAMs can be seen as special wide-step ASMs (which does not make then any less valuable). The existing quantum computer models satisfy our postulates as well [20] assuming that the environment provides random bits when needed. The corresponding wide-step ASMs need physical quantum-computer implementation for efficient execution.

7 Toward formalization of distributed algorithms

Distributed abstract state machines were defined in [25]. They are extensively used by the ASM community [1] but the problem of capturing distributed algorithms is open. Here we concentrate on one aspect of this important problem: interaction between a sequential-time agent and the rest of the system as seen by the agent. One may have an impression that this aspect has been covered because all along we studied runs where steps of the algorithm are interleaved with steps made by the environment. But this interleaving mode is not general enough.

If we assume that each agent's steps are atomic, then interleaving mode seems adequate. But a more detailed analysis reveals that even in this case a slight modification is needed. See Subsection 7.1.

But in fact an agent's steps need not be atomic because agents can interact with their environments not only in the inter-step fashion but also in the intra-step fashion. It is common in the AsmL experience that, during a single step, one agent calls on other agents, receives "callbacks", calls again, etc. It is much harder to generalize the two characterization theorems to intra-step interaction.

7.1 Trivial updates in distributed computation

Consider a small-step abstract state machine A. In Section 5, we restricted attention to runs where steps of A are interleaved with steps of the environment. Now turn attention to distributed computing where the agents do not necessarily take turns to compute. Assume that A is the algorithm executed by one of the agents. Recall that an update of a location ℓ of the current state of A is the action of replacing the current content a of ℓ with some content b. Call the update *trivial*

if $a = b$. In Section 5 we could ignore trivial updates. But we have to take them into account now. A trivial update of ℓ matters in a distributed situation when the location ℓ is shared: typically only one agent is allowed to write into a location at any given time, and so even a trivial update by one agent would prevent other agents from writing to the same location at the same time.

Recall that $\Delta(X)$ is the set of nontrivial updates computed by the algorithm A at X during one step. Let $\Delta^+(X)$ be the set of all updates, trivial or not, computed by A at X during the one step. It seems obvious how to generalize Section 5 in order to take care of trivial updates: just strengthen the bounded-exploration postulate by replacing Δ with Δ^+. There is, however, a little problem. Nothing in the current definition of a small-step algorithm A guarantees that there is a $\Delta^+(X)$ map associated with it. ($\Delta(X)$ is definable in terms of X and $\tau_A(X)$.) That is why we started this subsection by assuming that A is an ASM. Euc also has a $\Delta^+(X)$ map: if X is the state where $a = 6$ and $b = 9$ then $\Delta^+(X) = \{(a, 3), (b, 6)\}$, and if Y is a state of A where $a = b = 3$ then $\Delta(Y) = \{(a, 0)\}$ and $\Delta^+(Y) = \{(a, 0), (b, 3)\}$.

To generalize Section 5 in order to take into account trivial updates, do the following.

- Strengthen the abstract-state postulate by assuming that there is a mapping Δ^+ associating a set of updates with every state X of the given algorithm A in such a way that the set of non-trivial updates in $\Delta^+(X)$ is exactly $\Delta(X)$.
- Strengthen the definition of behavioral equivalence of sequential algorithms by requiring that the two algorithms produce the same $\Delta^+(X)$ at every state X.
- Strengthen the bounded exploration postulate by replacing Δ with Δ^+.

It is easy to check that Theorem 5.6, the small-step characterization theorem, remains valid.

Remark 7.1 In a similar way, we refine the definition of wide-step algorithms and strengthen Theorem 6.2, the wide-step characterization theorem.

Remark 7.2 Another generalization of Section 5, to algorithms with the `output` command, is described in [7]. The two generalizations of Section 5 are orthogonal and can be combined. The `output` generalization applies to wide-step algorithms as well.

7.2 Intra-step interacting algorithms

During the execution of a single step, an algorithm may call on its environment to provide various data and services. The AsmL experience showed the importance

of intra-step communication between an algorithm and its environment. AsmL programs routinely call on outside components to perform various jobs.

The idea of such intra-step interaction between an algorithm and its environment is not new to the ASM literature; external functions appear already in the tutorial [23]. In simple cases, one can pretend that intra-step interaction reduces to inter-step interaction, that the environment prepares in advance the appropriate values of the external functions. In general, even if such a reduction is possible, it requires an omniscient environment and is utterly impractical

The current authors are preparing a series of articles extending Theorems 5.6 and 6.2 to intra-step interacting algorithms. In either case, this involves

- axiomatic definitions of intra-step interacting algorithms,
- precise definitions of intra-step interacting abstract state machines,
- the appropriate extension of the notion of behavioral equivalence,
- verification that the ASMs satisfy the definitions of algorithms,
- a proof that every intra-step interacting algorithm is behaviorally equivalent to an intra-step interacting ASM.

Acknowledgment

We thank Steve Cook, John Dawson, Martin Davis, Sol Feferman, Leonid Levin, Victor Pambuccian and Vladimir Uspensky for helping us with references. We thank John Dawson, Sol Feferman, Erich Grädel, Leonid Levin, Victor Pambuccian and Dean Rosenzweig for commenting, on very short notice (because of a tight deadline), on the draft of this paper.

References

[1] ASM Michigan Webpage, `http://www.eecs.umich.edu/gasm/`, maintained by James K. Huggins.

[2] The AsmL webpage, `http://research.microsoft.com/foundations/AsmL/.`

[3] Charles H. Bennett and Rolf Landauer, "Fundamental physical limits of computation", Scientific American 253:1 (July 1985), 48–56.

[4] Ethan Bernstein and Umesh Vazirani, "Quantum complexity theory", SIAM Journal on Computing 26 (1997), 1411–1473.

[5] Andreas Blass and Yuri Gurevich, "The linear time hierarchy theorem for abstract state machines and RAMs", *Springer Journal of Universal Computer Science* 3:4 (1997), 247–278.

[6] Andreas Blass and Yuri Gurevich, "Background, reserve, and Gandy machines," Springer Lecture Notes in Computer Science 1862 (2000), 1–17.

[7] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," Technical Report MSR-TR-2001-117. A journal version is scheduled to appear in *ACM Transactions on Computational Logic* 4:4, October 2003.

[8] Andreas Blass, Yuri Gurevich, and Saharon Shelah, "Choiceless polynomial time", *Annals of Pure and Applied Logic* 100 (1999), 141–187.

[9] Egon Börger and Robert Stärk, *Abstract State Machines*, Springer, 2003.

[10] Alonzo Church, "An unsolvable problem of elementary number theory", *American Journal of Mathematics* 58 (1936), 345–363. Reprinted in [13, 88–107].

[11] Stephen A. Cook, "Computational complexity of higher type functions", Proceedings of 1990 International Congress of Mathematicians, Kyoto, Japan, Springer–Verlag, 1991, 55–69.

[12] Stephen A. Cook, Private communication, 2003.

[13] Martin Davis, "The Undecidable", Raven Press, 1965.

[14] Martin Davis, "Why Gödel didn't have Church's thesis", *Information and Control* 54 (1982), 3–24.

[15] David Deutsch, "Quantum theory, the Church-Turing principle and the universal quantum computer", *Proceedings of the Royal Society, A*, vol. 400 (1985), 97–117.

[16] David Deutsch, Artur Ekert and Rossella Lupaccini, "Machines, logic and quantum physics", The Bulletin of Symbolic Logic 6 (2000), 265–283.

[17] Robin Gandy, "Church's thesis and principles for mechanisms", In J. Barwise, H. J. Keisler, and K. Kunen, Eds., *The Kleene Symposium*, North-Holland, 1980, 123–148.

[18] Robin Gandy, "The confluence of ideas in 1936", in Rolf Herken, Editor, *The universal Turing machine: A half-century story*, Oxford University Press, 1988, 55–111.

[19] Kurt Gödel, "Collected Works", Volume II, Oxford University Press, 1990.

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

[21] Dima Grigoriev, "Kolmogorov algorithms are stronger than Turing machines", *Journal of Soviet Mathematics* 14:5 (1980), 1445–1450.

[22] Yuri Gurevich, "Kolmogorov machines and related issues", in G. Rozenberg and A. Salomaa, Editors, *Current Trends in Theoretical Computer Science*, World Scientific, 1993, 225–234; originally in Bull. EATCS 35 (1988).

[23] Yuri Gurevich, "Evolving algebras: An attempt to discover semantics", in G. Rozenberg and A. Salomaa, Editors, *Current Trends in Theoretical Computer Science*, World Scientific, 1993, 266–292; originally in Bull. EATCS 43 (1991).

[24] Yuri Gurevich, "Feasible Functions", *London Mathematical Society Newsletter,* 206 (June 1993), 6–7.

[25] Yuri Gurevich, "Evolving algebra 1993: Lipari guide", in E. Börger, Editor, *Specification and Validation Methods*, Oxford University Press, 1995, 9–36.

[26] Yuri Gurevich, "May 1997 Draft of the ASM Guide", Technical Report CSE-TR-336-97, EECS Department, University of Michigan, 1997.

[27] Yuri Gurevich, "For every sequential algorithm there is an equivalent sequential abstract state machine", ACM Transactions on Computational Logic, vol. 1, no. 1 2000), 77–111.

[28] Yuri Gurevich and James K. Huggins, "The semantics of the C programming language", Springer Lecture Notes in Computer Science 702 (1993), 274–308.

[29] Yuri Gurevich and Saharon Shelah, "Nearly linear time", Springer Lecture Notes in Computer Science 363 (1989), 108–118.

[30] Yuri Gurevich and Marc Spielmann, "Recursive abstract state machines", *Springer Journal of Universal Computer Science* 3:4 (1997), 233–246.

[31] Anatoly Karatsuba and Yuri Ofman, "Multiplication of multidigit numbers on automata", Soviet Physics Doklady (English translation), 7:7 (1963), 595–596.

[32] Stephen C. Kleene, "On notation for ordinal numbers", Journal of Symbolic Logic 3 (1938), 150–155.

[33] Stephen C. Kleene, "Origins of recursive function theory", *Annals of the History of Computing* 3:1 (January 1981), 52–67.

[34] Stephen C. Kleene, "Turing's analysis of computability, and major applications of it", in Rolf Herken, editor, *The universal Turing machine: A half-century story*, Oxford University Press, 1988, 17–54.

[35] Richard M. Karp and Vijaya Ramachandran, "Parallel algorithms for shared-memory machines," in J. van Leeuwen, Editor, *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, Elsevier and MIT Press (1990), 869–941.

[36] Dono Kijne, "Plane Construction Field Theory", Van Gorcum, Assen, 1956.

[37] Donald E. Knuth, "The Art of Computer Programming", volume 1: "Fundamental Algorithms", *Addison-Wesley*, Reading, MA, 1968.

[38] Donald E. Knuth, "The Art of Computer Programming", volume 2: "Seminumerical Algorithms", *Addison-Wesley*, Reading, MA, 1981.

[39] Andrei N. Kolmogorov, "On the concept of algorithm", *Uspekhi Mat. Nauk* 8:4 (1953), 175–176, Russian. An English translation is found in [56, pages 18–19].

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

[41] Leonid A. Levin, "Universal Search Problems", *Problemy Peredachi Informatsii,* 9:3 (1973), 265–266, Russian. The journal is translated into English under the name *Problems of Information Transmission.*

[42] Leonid A. Levin, Private communication.

[43] Leonid A. Levin, "The Tale of One-Way Functions", *Problemy Peredachi Informatsii,* 39:1 (2003), 92–103, Russian. The journal is translated into English under the name *Problems of Information Transmission.* The English version is available online at http://arXiv.org/abs/cs.CR/0012023.

[44] Yiannis N. Moschovakis, "What is an algorithm?" in B. Engquist and W. Schmid, Editors, *Mathematics Unlimited*, Springer-Verlag (2001) 919–936.

[45] Marian Boykan Pour-El and J. Ian Richards, "Computability in Analysis and Physics" (Perspectives in Mathematical Logic), Springer-Verlag 1989.

[46] Wolfgang Reisig, "On Gurevich's theorem on sequential algorithms", Acta Informatica 39 (2003), 273–305.

[47] Arnold Schönhage, "Universelle Turing Speicherung", in J. Dörr and G. Hotz, Editors, *Automatentheorie und Formale Sprachen*, Bibliogr. Institut, Mannheim, 1970, 369–383. In German.

[48] Arnold Schönhage, "Storage modification machines", *SIAM Journal on Computing* 9 (1980), 490–508.

[49] Joseph R. Shoenfield, "Mathematical Logic", Addison-Wesley 1967.

[50] Peter W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer", *SIAM Journal on Computing* 26:5 (1997), 1484–1509.

[51] Wilfried Sieg, "Step by recursive step: Church's analysis of effective calculability", The Bulletin of Symbolic Logic 3:2 (1997), 154–180.

[52] Wilfried Sieg, "An abstract model for parallel computations: Gandy's thesis", *The Monist* 82:1 (1999), 150–164.

[53] Robert Endre Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets", Journal of Computer and System Sciences 18 (1979), 110–127.

[54] 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, *ibidem*, vol. 43, 544–546. Reprinted in [13, 155–222] and available online at http://www.abelard.org/turpap2/tp2-ie.asp.

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

[56] Vladimir A. Uspensky and Alexei L. Semenov, "Algorithms: Main Ideas and Applications", Kluwer, 1993.