Semantics-to-syntax analyses of algorithms

Yuri Gurevich

Microsoft Research, Redmond, WA, USA

The real question at issue is "What are the possible processes which can be carried out in computing a number?"

Turing

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

Archimedes

Abstract. Alan Turing pioneered semantics-to-syntax analysis of algorithms. It is a kind of analysis where you start with a large semantically defined species of algorithms, and you finish up with a syntactic artifact, typically a computation model, that characterizes the species. The task of analyzing a large species of algorithms seems daunting if not impossible. 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 Turing's analysis of human-executable computation, Kolmogorov's analysis of sequential bit-level computation, Gandy's analysis of a species of machine computation, and our own analysis of sequential computation.

1 Introduction

This article is a much revised and extended version of our Foundational Analyses of Computation [17].

1.1 Terminology

Species of algorithms For the sake of brevity we introduce the term *species of algorithms* to mean a class of algorithms given by semantical constraints. We are primarily interested in large species like sequential algorithms or analog algorithms.

Q[1]: Contrary to biological species, yours are not necessarily disjoint. In fact, one of your species may include another as a subspecies.

A: This is true. For example, the species of sequential-time algorithms, that execute step after step, includes the species of sequential algorithms, with steps of bounded complexity.

Q: The semantic-constraint requirement seems vague.

A: It is vague. It's purpose is just to distinguish the analyses of algorithms that we focus upon here from other analyses of algorithms in the literature.

Analyses of algorithms We are interested in the semantics-to-syntax analyses of algorithms like that of Turing. You study a species of algorithms. The original definition of the species may have been vague, and you try to explicate it which may narrow the species in the process. And you finish up with a syntactic artifact, like a particular kind of machines that execute all and only the algorithms of the (possibly narrowed) species in consideration.

Q: Did Turing's analysis cover all algorithms or only those of a particular species.

A: There are limitations on algorithms covered by Turing's analysis, and we address some of them in §2.

Q: In computer science, they teach the analysis of algorithms. I guess that isn't semantics-to-syntax analysis.

A: The analysis of algorithms in such a course is normally the analysis of known algorithms, say known sorting algorithms or known string algorithms. A significant part of that is resource analysis: estimate how much time or space or some other resource is used by an algorithm. It all is important and useful, but it isn't a semantics-to-syntax analysis.

Algorithms and computations By default, in this paper, computations are algorithmic. This is the traditional meaning of the term

[1] Q is our inquisitive friend Quisani, and A is the author.

computation in logic and computer science. However, a wider meaning of the term is not uncommon these days. For an interesting and somewhat extreme example see [29] where computations are viewed as "processes generating knowledge."

The concepts of algorithms and computations are closely related. Whatever algorithms are syntactically, semantically they specify computations. For our purposes, the analysis of algorithms and the analysis of computation are one and the same.

Q: One algorithm may produce many different computations.

A: That is why we defined species as collections of algorithms rather than computations.

Algorithms and computable functions In mathematical logic, theory of algorithms is primarily the theory of recursive functions. 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. Note in this connection that, for some useful algorithms, non-termination is a blessing, rather than a curse. Consider for example an algorithm that opens and closes the gates of a railroad crossing.

Sequential algorithms

Q: You made a distinction between sequential and sequential-time computations. Give me an example of a sequential-time algorithm that isn't sequential.

A: Consider the problem of evaluating a finite logic circuit in the form of a tree. Each leaf is assigned 0 or 1, and each internal node is endowed with a Boolean operation to be applied to the Boolean values of the children nodes. Define the height of a leaf to be 0, and the height of an internal node to be 1 plus the maximum of the heights of its children. At step 1, the nodes of height 1 fire (i.e. apply their Boolean operations). At step 2, the nodes of height 2 fire. And so on, until the top node fires and produces the final Boolean value.

Q: So the nodes of any given height fire in parallel. Presumably sequential algorithms don't do parallel operations.

A: Well, bounded parallelism is permissible in sequential algorithms. For example, a typical Turing machine can do up to three operations in parallel: change the control state, modify the active-cell symbol, and move the tape.

Q: It seems that sequential algorithms are sequential-time algorithms with bounded parallelism.

A: No, things are a bit more complicated. A sequential-time algorithm may be interactive, even intra-step interactive, while it is usually assumed that a sequential algorithm does not interact with its environment, certainly not while performing a step.

Q: I find the term sequential algorithm not very cogent.

A: The term is not cogent but traditional. Elsewhere we used alternative terms: small step algorithms [1], classical algorithms [9], classical sequential algorithms [16]. The term sequential time was coined in [15].

Q: You mentioned that the steps of a sequential algorithm are of bounded complexity. What do you mean?

A: We will confront this question in §5.

Q: I presume a sequential algorithm can be nondeterministic.

A: Many authors agree with you, but we think that true sequential algorithms are deterministic, and so did Turing: "When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator." Nondeterminism involves intra-step interaction [15, §9], because an external intervention is needed to resolve nondeterminism; otherwise the algorithm hangs helplessly. In this connection, recall Yogi Berra's famous nondeterministic algorithm: "When you come to a fork in the road, take it!"

Q: Then why do many authors allow sequential algorithms to be nondeterministic.

A: Because, for many purposes, it is convenient to hide the nondeterminism resolving mechanism.

Q: I guess you'll be talking plenty about interaction in the rest of the paper.

A: No, not in this paper. But we analyzed interactive algorithms elsewhere [3, 5].

1.2 What's in the paper?

We review selected semantics-to-syntax analyses of species of algorithms in the literature. In each case, we try to identify the species and explicate the fulcrum that made the analysis possible.

Q: The fulcrum? The literal meaning of the word is the support point about which a lever turns, as in the Archimedes's "Give me a fulcrum, and I shall move the world." But that isn't what you mean.

A: We use the term as a metaphor. Suppose that you study a large species of algorithms, and you ask yourself whether there are syntactic means to describe the species. The problem seems overwhelming, impossible to solve. There are so many vastly different algorithms there. You don't even know where to start. Yet you persist and continue to think about the problem. Now imagine that one day an idea occurs to you to look at the problem differently, from another angle so to speak. And suddenly the problem looks more feasible though not necessarily easy. It is that idea that we metaphorically call a fulcrum.

Q: Your metaphorical use of the term seems by no means restricted to the semantics-to-syntax analysis of algorithms.

A: That's right. Here's an example from another domain. In the 1960s people tried various ways to organize large quantities of data. There are so many different kinds of data, and they seem to require different presentations. But eventually it occurred to Edgar Frank Codd to view a database as a relational structure of mathematical logic [7]. Relational database theory was born. The rest is history.

In §2, we review Alan Turing's celebrated analysis of computation in his 1936 paper [25].

Q: Was it the first semantics-to-syntax analysis?

A: No, Alonzo Church's analysis [6] preceded that of Turing. There was also analysis of recursion that culminated with Kurt Gödel's definition of recursive numerical[2] functions [18]. These classical analyses are richly covered in the logic literature, and Turing's analysis is the deepest and by far the most convincing.

In §3, we discuss Andrey Kolmogorov's analysis of computation of the 1950s [19, 20].

Q: We spoke about Kolmogorov machines earlier [13]. They compute the same numerical functions as Turing machines do, and there had been other computation models of that kind introduced roughly at the same time or earlier. In particular, Emil Post introduced his machines already in 1936 [23]. Kolmogorov's compatriot Andrey Markov worked on his normal algorithms in the early 1950s [22]. Why do you single out Kolmogorov's analysis?

A: As far as we know, among the analyses of algorithms of that post-Turing period, only Kolmogorov's analysis seems to be a true semantics-to-syntax analysis. By the way, even though Kolmogorov machines do not compute more numerical functions than Turing machines do, they implement more algorithms.

Q: That raises the question when two algorithms are the same.

A: We addressed the question in [4].

In §4, we review Robin Gandy's analysis of machine computations [10]. In §5, we review our own analysis if sequential algorithms.

Q: You review your own analysis? Why?

A: Well, most of the semantics-to-syntax analyses of species of algorithms in the literature are devoted to sequential algorithms, and we think, rightly or wrongly, that our analysis of sequential algorithms is definitive. Also we use this opportunity to spell out, for the first time, our fulcrum.

[2] Here and below a numerical function is a function $f(x_1, \ldots, x_j)$, possibly partial, of finite arity j, where the arguments x_i range over natural numbers, and the values of f — when defined — are natural numbers.

Finally, in §6, we discuss limitations of semantics-to-syntax analyses.

2 Turing

Alan Turing analyzed computation in his 1936 paper "On Computable Numbers, with an Application to the Entscheidungsproblem" [25]. All unattributed quotations in this section are from that paper.

The Entscheidungsproblem is the problem of determining whether a given first-order formula is valid. 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."

> `Q`: Hmm, input strings of a Turing machine are finite which does not suffice to represent real numbers.
>
> `A`: Turing himself did not insist that input strings are finite.

2.1 Turing's species of algorithms

Turing's intention might have been to consider all algorithms. But algorithms of the time were sequential, and the computers were humans[3]. So Turing analyzed sequential algorithms performed by idealized human computers. In the process he explicated some inherent constraints of the species and imposed — explicitly or implicitly — some additional constraints. Here are the more prominent constraints of Turing's analysis.

[3] "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 [11, p. 12]).

Digital Computation is digital (or symbolic, symbol-pushing). "Computing is normally done by writing certain symbols on paper."

Q: Is the digital constraint really a constraint?

A: These days we are so accustomed to digital computations that the digital constraint may not look like a constraint. But it is. Non-digital computations have been performed by humans from ancient times. Think of ruler-and-compass computations or of Euclid's algorithm for lengths [16, §3].

Sequential time Computation splits into a sequence of steps. "Let us imagine the operations performed by the computer to be split up into [a sequence of] 'simple operations'."

Elementary steps The simple operations mentioned above "are so elementary that it is not easy to imagine them further divided."

Q: The elementary steps are elementary indeed but, as you mentioned, they involve parallel actions: changing the control state, modifying the active-cell, moving the tape. There are multi-tape Turing machines where more actions are performed in parallel.

A: This is true, but the parallelism remains bounded.

Interaction with the environment

Q: Earlier you mentioned an important constraint on sequential algorithms which, I guess, applies to Turing's analysis: the computation does not interact with the environment. Of course the environment supplies inputs and presumably consumes the outputs but the computation itself is self-contained. It is determined by the algorithm and the initial state. No oracle is consulted, and nobody interferes with the computation.

A: Actually[4] Turing introduced nondeterministic machines already in [25] and oracle machines in [26].

[4] This was pointed out to us by the anonymous referee.

Q: I didn't know that. Come to think of it, oracle machines make good sense in the context of human computing. The human computer may consult various tables, may ask an assistant to perform an auxiliary computation, etc. But I don't see how nondeterministic algorithms come up in the analysis of human computing.

A: It wasn't the analysis of human computing that brought Turing to nondeterminism. "For some purposes we might use machines (choice machines or *c*-machines) whose motion is only partially determined by the configuration .... When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator. This would be the case if we were using machines to deal with axiomatic systems."

Q: I see, a nondeterministic machine can deal gracefully with axiomatic systems; it can guess and then verify a proof. You argued already that nondeterministic algorithms are really interactive. But can one really use a choice machine for meaningful interaction?

A: Well, a choice machine may be programmed to play chess. The moves of the other player can be entered as choices "made by an external operator."

2.2 Turing's fulcrum

How can one analyze the great and diverse variety of computations performed by human computers. Amazingly Turing found a way to do that. We believe that his fulcrum was as follows. *Ignore what a human computer has in mind and concentrate on what the computer does and what the observable behavior of the computer is.* In other words, Turing treated the idealized human computer as an operating system of sorts.

One may argue that Turing did not ignore the computer's mind. He spoke about the state of mind of the human computer explicitly and repeatedly. Here is an example. "The behaviour of the computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at that moment." But Turing postulated 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'."

2.3 On Turing's results and argumentation

Turing introduced abstract computing machines that became known as Turing machines, and he constructed a universal Turing machine. A real number is Turing computable "if its decimal can be written down by a [Turing] machine." His thesis was that Turing computable numbers "include all numbers which could naturally be regarded as computable." He used the thesis to prove the undecidability of the Entscheidungsproblem. To convince the reader of his thesis, Turing used three arguments [25, §9].

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

Robustness He sketched an alternative definition of computability "in case the new definition has a greater intuitive appeal" and proved it equivalent to his original definition. The alternative definition is based on provability in a finitely axiomatizable fragment of the first-order theory of arithmetic. The robustness argument was strengthened in the appendix where, after learning about Church's definition of computability [6], he proved the equivalence of their definitions.

Appeal to Intuition He analyzed computation appealing to intuition directly.

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.

2.4 Two critical quotes

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" [12]. It is hard for me to believe that this really was Turing's goal. Anyway, Gödel continues thus. "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" [12].

Do you understand that? Apparently Gödel thought that gifted mathematicians may eventually find a sophisticated decision procedure for the Entscheidungsproblem that is not mechanical. But if gifted mathematicians are able to reliably execute the procedure, they should be able to figure out how to program it, and then the procedure is mechanical.

A: Maybe Gödel was just pointing out that, in solving instances of Entscheidungsproblem, human creativity would outperform any mechanical procedure. Turing would surely agree with that.

Q: Let me change the topic. Here is another interesting 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 definitions are equally awkward and unwieldy. In this respect, general recursiveness is superior" (Sol Feferman, [11, p. 6]). Do you buy that?

A: Indeed, the recursive approach has been dominant in mathematical logic. It is different though in computer science where Turing's approach dominates. Turing's machine model enabled computational complexity theory and even influenced the early design of digital computers. Church's λ-calculus has been influential in programming language theory.

3 Kolmogorov

Andrey Kolmogorov's analysis is reflected in a 1953 talk to the Moscow Mathematical Society [19] and in a paper [20] with his student Vladimir Uspensky[5]. Kolmogorov's approach isn't as known as it deserves to be. In this connection, here are some relevant references: [1, 13, 27, 28].

3.1 Kolmogorov's species of algorithms

Like Turing, Kolmogorov might have intended to analyze all algorithms. The algorithms of his time still were sequential. In the 1953 talk, Kolmogorov 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 bounded size. The bound does not depend on the state or the input size, only on the algorithm itself. 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.

Implicitly Kolmogorov presumes also that the algorithm does not interact with its environment, so that a computation is a sequence $S_0, S_1, S_2, \ldots$ of states, possibly infinite, where every $S_{n+1} = S_n^*$

> Q: The second stipulation does not seem convincing to me. For example, a sequential algorithm may multiply and divide integers in one step. Such transformations do not look direct and immediate in some absolute sense.
>
> A: Kolmogorov restricts attention to sequential algorithms working on the lowest level of abstraction, on the level of single bits.

[5] Uspensky told us that the summary [19] of the 1953 talk was written by him after several unsuccessful attempts to make Kolmogorov to write a summary.

3.2 Kolmogorov's fulcrum

Kolmogorov's ideas gave rise to a new computation-machine model different from Turing's model [20]. Instead of a linear tape, a Kolmogorov (or Kolmogorov-Uspensky) 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 types of vertices and a fixed number of types of edges. We speculated in [13] 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 [27, p. 396].

We do not know much about the analysis that led Kolmogorov and Uspensky from the stipulations above to their machine model. "As Kolmogorov believed," wrote Uspensky [27, 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 non-isomorphic 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* [21]. That seems to be Kolmogorov's fulcrum. In particular, the edges of the state graph of a Kolmogorov machine reflect physical closeness of computation elements. One difficulty with this approach is that there may be no finite bound on the dimensionality of the computation space [15, footnote 1].

> Q: I would think that Kolmogorov's analysis lent support to the Church-Turing thesis.
>
> A: It did, to the extent that it was independent from Turing's analysis. We discuss the issue in greater detail in [9, §1.2].
>
> Q: You mentioned that Turing's machine model enabled computational complexity theory. Was the Kolmogorov-Uspensky

machine model useful beyond confirming the Church-Turing thesis?

A: Very much so; please see [1, §3].

4 Gandy

Gandy analyzed computation in his 1980 paper "Church's Thesis and Principles for Mechanisms" [10]. In this section, by default, quotations are from that paper.

4.1 Gandy's species of algorithms

The computers of Gandy's time were machines, or "mechanical devices", rather than humans, and that is Gandy's departure point.

> "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.*"

Since mechanical devices can perform parallel actions, Thesis M "must take parallel working into account." But the species of all mechanical devices is too hard to analyze, and Gandy proceeds to narrow it to a species of mechanical devices that he is going to analyze.

> "(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.*"

Principle I asserts in particular that, for any mechanical device, the states can be described by hereditarily finite sets[6] and there is a transition function F such that, if x describes an initial state, then $Fx, F(Fx), \ldots$ describe the subsequent states. Gandy wants "the form of description to be sufficiently abstract to apply uniformly to mechanical, electrical or merely notional devices," so the term mechanical device is treated liberally.

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

[6] 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 $z \in t$.

$t \subseteq \mathrm{TC}(x)$ 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]."

4.2 Gandy's fulcrum

It seems to us that Gandy's fulcrum is his Principle I. It translates the bewildering world of mechanical devices into the familiar set-theoretic framework.

4.3 Comments

Gandy pioneered the axiomatic approach in the area of semantics-to-syntax analyses of algorithms. He put forward the ambitious Thesis M asserting that a numerical function can be calculated by a machine only if it is Turing computable. In our view, Gandy's decision to narrow the thesis is perfectly justified; see §6 in this connection.

But his narrowing of Thesis M is rather severe. A computing machine may have various features that Turing's analysis rules out, e.g. asynchronous parallel actions, analog computations. Gandy allows only synchronous parallelism, that is sequential-time parallelism. His species is a subspecies of the species of synchronous parallel algorithms (which was analyzed later, already in the new century, in [2]).

One reason that Gandy's species is a subspecies of synchronous parallel algorithm is Principle I. Hereditarily finite sets are finite. Taking into account that Gandy's machines do not interact with the environment, this excludes many useful algorithms. For example it excludes a simple algorithm that consumes a stream of numbers keeping track of the maximal number seen so far.

> `Q`: Isn't this algorithm inherently interactive?
>
> `A`: In a sense yes. But it is a common abstraction in programming to pretend that the whole input stream is given in the beginning. The abstraction is realizable in the Turing machine model.

Also, the principle of local causality (Principle IV) does not apply to all synchronous parallel algorithms. Gandy himself mentions one counterexample, namely Markov's normal algorithms [22]. 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.

Our additional critical remarks of Gandy's analysis are found in [16, §4]. (Wilfried Sieg adopted Gandy's approach and simplified Gandy's axioms, see [24] and references there, but — as far as our critique is concerned — the improvements do not make much difference.)

> Q: If Turing thought of synchronous parallelism, he could have claimed that, without loss of generality, the parallel actions performed during one step can be executed sequentially, one after another.
>
> A: Gandy complains that the claim seems obvious to people. One should be careful though about executing parallel actions sequentially. Consider for example a parallel assignment $x, y := y, x$. If you start with $x := y$, you'd better save the value of x so that $y := x$ can be performed as intended. In any case, the claim can be proved in every model of synchronous parallelism in the literature, including the most general model of [2].

5 Sequential algorithms

5.1 Motivation

By the 1980s, there were plenty of computers and software. A problem arose how to specify software. The most popular theoretical approaches to this problem were declarative. And indeed, declarative specifications (or specs) tend to be of higher abstraction level and easier to understand than executable specs. But executable specs have their own advantages. You can "play" with them: run them, test, debug.

Q: If your spec is declarative then, in principle, you can verify it mathematically.

A: That is true, and sometimes you have to verify your spec mathematically; 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, especially because software evolves. In most cases, it is virtually impossible to keep a declarative spec in sync with the implementation. 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 spec has to be low-level and detailed? This leads to a foundational problem whether any algorithm can be specified, in an executable way, on its intrinsic level of abstraction.

Q: A natural-language spec would not do as it is not executable.

A: Besides, such a spec may (and almost invariably does) introduce ambiguities and misunderstanding.

Q: You can program the algorithm in a conventional programming language but this will surely introduce lower-level details.

A: Indeed, even higher-level level programming languages tend to introduce details that shouldn't be in the spec.

Turing and Kolmogorov machines are executable but low-level. Consider for example two distinct versions of Euclid's algorithm for the greatest common divisor of two natural numbers: the ancient version where you advance by means of differences, and a modern (and higher-level) version where you advance by means of divisions. The chances are that, in the Turing machine implementation, the distinction disappears.

Can one 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 [15], synchronous parallel algorithms [2], and interactive algorithms [3, 5]. We discuss here only the first of these.

5.2 The species

Let's restrict attention to the species of sequential algorithms but without any restriction on the abstraction level. It could be the Gauss Elimination Procedure for example. Informally, paraphrasing the first stipulation in §3, an algorithm is sequential if it computes in steps whose complexity is bounded across all computations of the algorithm. In the rest of this section, algorithms are by default sequential.

We use the axiomatic method to explicate the species. The first axiom is rather obvious.

Axiom 1 (Sequential Time) *Any algorithm A is associated with a nonempty collection $\mathcal{S}(A)$ of states, a sub-collection $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ of initial states and a (possibly partial) state transition map $\tau_A : \mathcal{S}(A) \longrightarrow \mathcal{S}(A)$.*

Definition 1. Two algorithms are *behaviorally equivalent* if they have the same states, the same initial states and the same transition function.

> Q: I guess the computation of a sequential algorithm A is determined by the initial state.
>
> A: Actually we can afford to be more general and allow the environment to intervene between the steps of the algorithm A, provided that every such intervention results in a legitimate state of A, so that A can continue to run. Thus, in general, the steps of A are interleaved with those of the environment, and the computation of A depends on the environment. But, for a given environment, replacing A with a behaviorally equivalent algorithm results in exactly the same computation.

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.

Axiom 2 (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: We do, and we have been making that claim since the 1980s. The collective experience of computer science seem to corroborate the claim.

Q: The states of real-world algorithms seems to be finite. Do you need infinite structures?

A: The states of real-world algorithms often are infinite. For example, consider the C programming language. A programming language can be viewed as an algorithm that runs a given program on the given data. In C, data structures like multi-sets or trees need to be programmed but arrays are readily available; they pre-exist in the initial state of C. So the states of C are infinite. And of course the states of the Gauss Elimination Procedure are infinite.

Q: OK, I have another question about the axiom. 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 (a possibly abstracted version of) 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: Because an algorithm works at a particular level of abstraction, and lower-level details are abstracted away. Consider a graph algorithm for example. In an implementation, nodes may be integer numbers, but the algorithm can't examine whether a node is even or odd because implementation details are irrelevant. If the algorithm does take advantage of the integer representation of nodes then its vocabulary should reflect the relevant part of arithmetic.

According to the informal definition of sequential algorithms, there is a bound on the complexity of the steps of the algorithm. But how to measure step complexity? The abstract-state axiom allows us to address the problem. 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 of 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. We argue in [15] that an abstracted version of Kolmogorov's locality constraint is valid on any level of abstraction. The "active zone" of state X is bounded. The change from X to $\tau_A(X)$, let us call it $\Delta_A(X)$, depends only on the results of the exploration of the active zone. Formally, $\Delta_A(X)$ can be defined as a collection of assignments $F(\bar{a}) := b$ where F is a vocabulary function.

Q: What about vocabulary relations? Are they necessarily static?

A: We view relations as Boolean-valued functions, so the vocabulary relations may be updatable as well.

Q: How does the algorithm know what to explore and what to change?

A: That information is supplied by the program, and it is applicable to all the states. In the light of the abstract-state axiom, it should be given symbolically, in terms of the vocabulary of A.

Axiom 3 (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.*

Now we are ready to define (sequential) algorithms.

Definition 2. A (sequential) algorithm is any entity that satisfies the sequential-time, abstract-state and bounded-exploration axioms.

Abstract state machines (ASMs) were defined in [14]. Here we restrict attention to sequential ASMs which are undeniably algorithms.

Theorem 1 ([15]). *For every algorithm A, there exists a sequential ASM that is behaviorally equivalent to A.*

5.3 The fulcrum

Every sequential algorithm 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 can be expressed naturally in the language of the fixed vocabulary.

Q[7]: I can't ask Turing, Kolmogorov or Gandy how they arrived to their fulcrums, but I can ask you that question.

A: In 1982 we moved abruptly from logic to computer science. We tried to understand what was that emerging science about. The notion of algorithm seemed central. Compilers, programming languages, operating systems are all algorithms. As far as sequential algorithms are concerned, the sequential-time axiom was obvious.

Q: How do you view a programming language as an algorithm?

A: It runs a given program on given data.

Q: What about all those levels of abstraction?

A: It had been well understood by 1982 that a real-world computation process can be viewed at different levels of abstraction. The computer was typically electronic, but you could

[7] This discussion is provoked by the anonymous referee who thought that the one sentence above is insufficient for this subsection.

abstract from electronics and view the computation on the level of single bits. More abstractly, you could view the computation on the assembly-language level, on the level of the virtual machine for the programming language of the program, or on the level of the programming language itself. But there is no reason to stop there. The abstraction level of the algorithm that the program executes is typically higher yet.

Besides, we became interested in software specifications and took the position that software specs are high-level algorithms. The existing analyses of algorithms did not deal with the intrinsic levels of abstraction of algorithms.

Q: Is the intrinsic abstraction level of any algorithm well determined?

A: This is one of the critical questions. It was our impression that the abstraction level of an algorithm is determined by that of its states and by the operations that the algorithm executes in one step. That led us eventually to the abstract-state axiom and to abstract state machines originally called evolving algebras [14].

Q: Was your logic experience of any use?

A: Yes, it was particularly useful to know how ubiquitous first-order structures were. Any static mathematical object that we knew could be faithfully represented by a first-order structure, and the states of an algorithm are static mathematical objects.

Q: When did you realize that sequential algorithms could be axiomatized?

A: This took years. We mentioned above that, in a semantics-to-syntax analysis, a fulcrum makes the analysis more feasible but not necessarily easy. The sequential-time and abstract-state axioms are implicit in the definition of abstract state machines [14]. We had not thought of them as axioms yet. Our thesis, restricted to sequential algorithms, was that sequential abstract state machines are able to simulate arbitrary sequential algorithms "in a direct and essentially coding-free way" [14]. The thesis was successfully tested in numerous applications of abstract state machines, and our confidence in

the thesis as well as the desire to derive it from "first principles" grew. The problem was solved in [15].

6 Final remarks

In [16] we argue that "the notion of algorithm cannot be rigorously defined in full generality, at least for the time being." The reason is that the notion is expanding. In addition to sequential algorithms, in use from antiquity, we have now parallel algorithms, interactive algorithms, distributed algorithms, real-time algorithms, analog algorithms, hybrid algorithms, quantum algorithms, etc. New kinds of algorithms may be introduced and most probably will be. Will the notion of algorithms ever crystallize to support rigorous definitions? We doubt that.

"However the problem of rigorous definition of algorithms is not hopeless. Not at all. Large and important strata of algorithms have crystallized and became amenable to rigorous definitions" [16]. In §5, we explained the axiomatic definition of sequential algorithms. That axiomatic definition was extended to synchronous parallel algorithms in [2] and to interactive sequential algorithms in [3, 5].

The axiomatic definition of sequential algorithms was also used in to derive Church's thesis from the three axioms plus an additional Arithmetical State axiom which asserts that only basic arithmetical operations are available initially [9].

`Q`: I wonder whether there is any difference between the species of all algorithms and that of machine algorithms.

`A`: This is a good point, though there may be algorithms executed by nature that machines can't do. In any case, our argument that the species of all algorithms can't be formalized applies to the species of machine algorithms. The latter species also evolves and may never crystallize.

Acknowledgements

Many thanks to Andreas Blass, Bob Soare, Oron Shagrir and the anonymous referee for useful comments.

References

1. Andreas Blass and Yuri Gurevich, "Algorithms: A quest for absolute definitions," in Current Trends in Theoretical Computer Science, World Scientific (eds. G. Paun et al.), 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. on Computational Logic 7:2 (2006) 363–419 (Part I), plus 8:3 (2007), articles 15 and 16 (Parts II and III).
4. Andreas Blass, Nachum Dershowitz and Yuri Gurevich, "When Are Two Algorithms the Same?" Bulletin of Symbolic Logic 15:2 (2009), 145-168.
5. 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).
6. Alonzo Church, "An unsolvable problem of elementary number theory", American Journal of Mathematics 58 (1936), 345–363.
7. Edgar Frank Codd, "Relational model of data for large shared data banks," Communications of the ACM 13:6 (1970), 377–387.
8. Martin Davis (editor), "The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions," Raven Press, Hewlett, NY, 1965, reprinted by Dover Publications, Mineola, NY, 2004.
9. 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.
10. Robin O. Gandy, "Church's thesis and principles for mechanisms", In The Kleene Symposium (eds. J. Barwise et al.), North-Holland, 1980, 123–148.
11. Robin O. Gandy and C.E.M. (Mike) Yates (editors), "Collected works of A.M. Turing: Mathematical logic", Elsevier, 2001.
12. Kurt Gödel, "A philosophical error in Turing's work," in Kurt Gödel: Collected Works," Volume II (eds. S. Feferman et al.), Oxford University Press, 1990, p. 306.
13. Yuri Gurevich, "On Kolmogorov machines and related issues," Bull. of Euro. Assoc. for Theor. Computer Science 35 (1988), 71-82.
14. Yuri Gurevich, "Evolving algebra 1993: Lipari guide," in Specification and Validation Methods (E. Börger, ed.), Oxford Univ. Press (1995), 9–36.
15. Yuri Gurevich, "Sequential abstract state machines capture sequential algorithms," ACM Trans. on Computational Logic 1:2 (2000), 77–111.
16. Yuri Gurevich, "What is an algorithm?" In SOFSEM 2012: Theory and Practice of Computer Science (eds. M. Bielikova et al.), Springer LNCS 7147, 2012. A slight revision will appear in Proc. of the 2011 Studia Logica conference on Church's Thesis: Logic, Mind and Nature.
17. Yuri Gurevich, "Foundational Analyses of Computation", in *How the World Computes* (eds. S. Barry Cooper et al.), Turing Centennial Conference, Springer LNCS 7318 (2012), 264–275.
18. Stephen Cole Kleene, "Introduction to metamathematics," D. Van Nostrand, 1952.
19. Andrey N. Kolmogorov, "On the concept of algorithm", Uspekhi Mat. Nauk 8:4 (1953), 175–176, Russian.
20. Andrey 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.

21. Leonid A. Levin, Private communication, 2003.
22. Andrey 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.
23. Emil L. Post, "Finite combinatorial processes — formulation I," Journal of Symbolic Logic 1 (1936), 103–105.
24. Wilfried Sieg, "On computability," in Handbook of the Philosophy of Mathematics (A. Irvine, ed.), Elsevier, 2009, 535-630.
25. Alan M. Turing, "On computable numbers, with an application to the Entscheidungsproblem", Proceedings of London Mathematical Society, ser. 2, vol. 42 (1936-37), 230–265.
26. Alan M. Turing, "Systems of logic based on ordinals," Proceedings of London Mathematical Society, ser. 2, vol. 45 (1939), 161–228.
27. Vladimir A. Uspensky, "Kolmogorov and mathematical logic," Journal of Symbolic Logic 57:2 (1992), 385–412.
28. Vladimir A. Uspensky and Alexei L. Semenov, Theory of algorithms: Main Discoveries and Applications, Nauka 1987 (Russian), Kluwer 2010 (English).
29. Jiří Wiedermann and Jan van Leeuwen, "Rethinking Computation", in Proc. 6th AISB Symposium on Computing and Philosophy (eds. M. Bishop and Y.J. Erden.), Society for the Study of Artificial Intelligence and the Simulation of Behaviour, Exeter, UK, 2013, pp. 6–10.