Some Philosophical Remarks on the Current Definitions of Algorithms

Philippos Papayannopoulos

▶ **To cite this version:**

Philippos Papayannopoulos. Some Philosophical Remarks on the Current Definitions of Algorithms. 13th Panhellenic Logic Symposium, Jul 2022, Volos, Greece. hal-03763964

HAL Id: hal-03763964
https://hal.science/hal-03763964

Submitted on 30 Aug 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Some Philosophical Remarks on the Current Definitions of Algorithms

Philippos Papayannopoulos

IHPST (UMR 8590), CNRS and Université Paris 1 Panthéon-Sorbonne, Paris, France
fpapagia@uwo.ca

Abstract

There has been a resurgent interest in formalizing the notion of 'algorithm'. In this paper, I discuss the relation between algorithms and computations, point to some tensions inherent in our informal concept of an algorithm, and discuss some trade-offs between competing desiderata for any proposed formal definition.

1 Introduction

The idea of an algorithmic procedure is almost as old as mathematics itself (see, e.g., [3]). Nevertheless, despite the long-standing prevalence of algorithmic methods in mathematics, attempts to formalize the concept of an 'algorithm' *itself* are relatively recent, and they are mostly a spin-off from the impressive conceptual advancements in understanding how to demarcate the computable functions. As is well known, this understanding came about as the result of seminal work in the 1930s, by Turing, Gödel, Church, Kleene, Rosser, Herbrand, and others.

The formalisms of Church, Rosser and Kleene (λ-calculus), Gödel and Herbrand (general recursion) and Kleene (μ-recursion) were soon proved equivalent and turned out later to capture what we now consider the correct class of number-theoretic computable functions. However, from a conceptual point of view, these particular formalisms lacked convincing power regarding their completeness, for there seemed to be no compelling reason why (e.g.) the general recursive or the λ-definable functions would include all and only those functions that can be calculated by purely mechanical means. A great reluctance to accept Church's thesis (in this form) was expressed by Gödel's famous comment to Church that such approaches were "thoroughly unsatisfactory". The situation changed radically when Turing's [19] analysis came along, which focused on the *process* of computation itself, by breaking it down into its conceptual constituents; this provided a low-level analysis of what can (and cannot) ultimately be achieved by purely mechanical and elementary steps, carried out by an (idealized) human agent. Turing's analysis was widely conceived as conclusive, and the Church-*Turing* thesis (CTT) became a universally accepted foundation for computer science, and especially computability and complexity.

The fact that Turing's analysis focused on the *process* of computation, together with the (seemingly innocuous) tacit assumption that what is meant by a "mechanical process of computing a function" (aka "effective procedure") coincides with what is meant by "execution of an algorithm" led to the widely held view that the CTT and the Turing Machine (TM) formalism explicate the notion of algorithm. As a result, this view has become part of the folklore of logic and computer science (CS).[1] However, Turing does not mention 'algorithms' at all in [19], and while Church [4] does use the term, he is not concerned with the process of computation itself (only with the extension of the concept of 'computable function'). In actual fact, then, the 1930s developments did not concern the (intentional) concept of algorithm per se but solely the demarcation of the class of computable functions (which is an extensional concern).

[1] See, e.g., [16, 3,102] and [11, 246] for two examples of this view being clearly articulated.

When did the interest in the idea of algorithm itself come along? Markov's [12, 13] seem to be among the very first works that claimed to define '*algorithms*'. But Markov's definition was a narrow one (not too different from a Turing program), unable to capture the informal notion in its generality. But, without an intentionally good definition of algorithms, it was still a conceptual possibility that one could follow a procedure that is much more permissible than Markov's —yet would still seem algorithmic— and get to compute a function that's beyond the class of partial recursive ones. To rule out such scenarios Kolmogorov and Uspenskii (K&U) set out to give the first full-fledged formal definition of algorithms, in a work so influential [10] that some of the ideas it introduced are still found today (even implicit) in almost every work in the area. But what is exactly the relation between algorithms and computable functions?

2 Algorithms and computation: A marriage made in heaven?

Computability is a semantic notion. A function is computable if it is such that its values can be identified by a process of computation; that is, by following a mechanical procedure. But the process of computation is syntactic and symbolic. In carrying out a computation, an agent (human or otherwise) deals with concrete entities (symbols on paper, physical voltages, etc.). Insofar as algorithms are understood as specifying mathematical computations, then, they specify procedures over symbols. Shapiro echoes exactly this view:

> Mechanical devices engaged in computation and humans following algorithms[..] do not encounter numbers themselves, but rather physical objects such as ink marks on paper. Since strings are the relevant abstract forms of these physical objects, *algorithms should be understood as procedures for the manipulation of strings*, not numbers. ([18, p.14]; emphasis added)

Thus, on a view that sees algorithms as specifying actual computations, algorithms are procedures for manipulating symbols and, hence, synonymous to effective procedures. They are tightly interlocked with the representations of the data they operate upon. Given some vocabulary and a representation of the input by strings of symbols from this vocabulary, an algorithm is a stepwise procedure for combinatorily manipulating these symbols and obtaining a result, which is a representation of the computed function's output. Since a schoolchild in ancient Rome would be taught a different combinatory sequence of steps for multiplying two 3-digit integers from a schoolchild in ancient Greece (owing to the different notation systems), the two children would have mastered *different* algorithms for obtaining the product of two numbers (and the same holds for multiplying two integers today in, say, decimal and binary notations).

This presupposition is clearly seen embedded in Markov's as well as in K&U's approach to defining algorithms: "Without fixing a standard way of writing numbers, to speak of the algorithm computing [the value of a function from its input] would make no sense." [10, fn.2]. What is more, if one goes further and simply *identifies* algorithms to Turing programs, then the above presupposition becomes also reflected in the dominant contemporary approach to real computability, in terms of Type-2 Turing Machines (TTE) [21]. As some key results in this area indicate, when Turing computations are extended to uncountable domains (such as $\mathbb{R}$) computability of functions acquires a strong dependence on the employed representations.[2]

The strong association of algorithms with computation has placed the concept of 'algorithm' at the heart of computer science. Since it is indeed a dominant view that *algorithms specify*

[2] For example, there is no (Type-2) Turing machine —hence no algorithm either— that computes the function: $g(x) = 3x$ ($g : \mathbb{R} \to \mathbb{R}$) on the decimal representation. Yet, the same function is (almost trivially) computable on the base-3 representation.

computations, statements about specific properties of algorithms (including existence) are staples in areas such as computability and complexity theory. Consider, e.g., assertions like "there is no algorithm that decides the validity for any first-order sentence" or "if $\mathbf{P} \neq \mathbf{NP}$, there is no algorithm that solves the Boolean satisfiability problem in polynomial time."

3 The third in the marriage: Mathematical practice

Despite the well-received view of algorithms and computations as being tightly interlocked, one quickly notices non-negligible conceptual problems. The view of algorithms as specifying computations does not quite square with prominent uses of the term in certain areas of mathematical practice. To wit, recall that algorithms have also been the subject-matter of the long-standing field of numerical analysis. Numerical algorithms concern continuous problems, and their purposes, very often, include identifying (exactly or approximately) solutions of (systems of) equations, guiding linear interpolation, etc. Typical examples include the bisection method, least-squares fitting, Gaussian elimination, Newton's method, and many others.

It does not seem natural to say that algorithms like the aforementioned specify computations (i.e., syntactic procedures) in the same sense that we saw in the previous section. They definitely do not specify exact sequences of steps to the smallest detail, in the sense that any specified sequence would have to change, had the employed notation (or representation) changed as well. Rather, numerical algorithms are developed and analyzed without any consideration of notation or representation, and they are naturally thought of as each one possessing a natural structure and identity of its own; a structure and identity that are invariable under changes in how data are represented and in what the exact order of operations is.[3] This attitude toward the fundamental idea of an 'algorithm' is an explicit motivation behind the Blum-Shub-Smale (BSS) model of real computability [2] as well as Moschovakis's foundational approach to algorithms (e.g., [14, 15]). As L. Blum puts it:

> We want a model of computation which is more natural for describing algorithms of numerical analysis, such as Newton's method [..] Translating to bit operations would wipe out the natural structure of this algorithm. [1, 1028]

Indeed, attending to the long-standing study of numerical algorithms and their purposes ([3], [6]) shows that it would be stretching a point to say that (e.g.) Newton's method is a mechanical procedure for pushing symbols around, in the sense found in works on algorithms like [13], [10], [9], and others. In stark contrast to the multiplication example from above —i.e., different algorithms for different notations— Newton's algorithm arguably remains the same (abstract) entity, regardless of what notation is used or what the exact order of operations is.

One might try to remedy this apparent discrepancy between the two understandings of algorithms just described by saying that numerical algorithms still specify computations, albeit at some "higher level" of abstraction. That is, they still report mechanical procedures, but by abstracting away from any particular details of the process (such details can *always* be filled in later). But, as usual, the devil hides in the details: consider that a good many numerical algorithms specify as *essential* steps comparisons between real numbers; think, for example, the bisection algorithm. This creates a conceptual gap between numerical algorithms and actual mechanical computations, because comparing two reals is in fact not effectively decidable. More precisely, comparisons (and identity) are not decidable by a (Type-2) TM [21]; they are

[3] This holds also for CS algorithms; think, e.g., of the MERGESORT algorithm. But, for reasons that will only briefly be touched upon here, the case for numerical algorithms is stronger, because the existence of their natural structure is orthogonal to whether they are realizable by a TM (which is not the case with CS algorithms).

only (negatively) semi-decidable. As a result, if we adopt an extended version of the CTT (an '*Uncountable-CTT*') to the effect that "the effectively computable real-valued functions are exactly the functions that are computable by a Type-2 TM" —which is a very natural assumption—, then most numerical algorithms will turn out not to be effectively computable. This indicates that the link between algorithms and effective procedures/mechanical computations has to be severed. But now a fundamental question arises: how, and to what extend, should the above considerations be taken into account by any attempt to define algorithms?

4 A tension in definitions: inclusive or specialized?

Mathematical definitions often face challenges imposed by a strong tension between generality and inclusiveness on the one hand and domain-specific fecundity on the other. Regarding algorithms, we have, on the one hand, the desire to include under some unified formal concept both algorithms over countable and uncountable domains; in particular, both effective procedures and numerical algorithms; but, ideally, the definition might also go some distance toward subsuming additional related notions such as parallel algorithms and geometric constructions, so that a uniform study of all these notions could become possible. This desideratum for inclusiveness pushes in the direction of a formal concept that is as abstract as possible. More specifically, we would like to have a formal explicatum of 'algorithms' such that: (a) the identity of any algorithm is not essentially dependent on the representations of the data it operates upon or on the finest-grained details of its evolution (so it is not essentially affected by implementation details); (b) it lends itself to a spectrum of primitive operations of variant strengths (a desideratum that is best served by a model/structure/level-theoretic view of algorithms, since in that case a step can be any primitive operation defined as such by the model/structure/level itself); (c) it retains its applicability to particular domains, so it does not contradict fundamental results of the more specific instances of the same concept (i.e., it should preserve basic theorems of computability theory or of numerical analysis). The formal approaches by Blum et al. [2], Gurevich (e.g., [7, 8]), and Moschovakis (e.g., [14, 15]) all satisfy the first two conditions, for they all offer formal concepts that are representation-invariant and level-relative (though not necessarily effective), trying purposely to capture algorithms that go beyond Turing programs, while TTE is mainly the only model with a wide scope (it applies to both countable and uncountable domains) which satisfies (c) (though it does not satisfy a and b).

On the other hand, the desire for the formal counterpart to be such that it lends itself to interesting relations with well-entrenched concepts pushes in the direction of a formal concept with a significantly narrower domain of application than in the previous case. More specifically, we would like to have a formal explicatum of algorithms such that (a′) it retains as much as possible the intentional character of the informal processes it purports to formalize; (b′) more importantly, it would feature in deep theorems and connect to other well-established concepts, such as those from complexity theory. Formal approaches like K&U machines [10], (ordinary or Type-2) TMs [21], and BSS machines [2] are successful models in these regards but, predictably, each one meets only one of the two conditions and in a particular domain. As I discuss next, K&U fares better in (a′) and applies to discrete algorithms, while BSS and (ordinary or Type-2) TMs fare better in (b′) and apply solely to either discrete or real algorithms but not both.

5 How the existing definitions meet the different needs

A main upshot of the above discussion is that it seems a Herculean task to find a formal explicatum of 'algorithms' that respects all our intuitions and expectations together. This is

not an unusual situation in logic and mathematics. In many other cases, however, a certain formal concept among the rivals (capturing *some* of what we consider the essential features of the intuitive idea) catches on and becomes the "orthodoxy", on account of being successful in providing interesting results ('continuity' being a case in point). But in the case of algorithms most formal frameworks have been fruitful already, even though some of them are genuinely incompatible (e.g., TTE vs. BSS).

The way I see the situation, then, is that 'algorithm' is a cluster concept; this means that in order to be able to give preference to any particular framework, the community first needs to have decided on which intuitions and goals to prioritize. In what follows, I will consider some of the (conflicting) *intuitions* and possible different *goals*. Undoubtedly, there are many intuitions about algorithms that most mathematicians would agree on. Here I will focus only on those that I think practitioners might rather disagree.

Symbolic vs. Abstract: Is the identity of an algorithm relative to the vocabulary of symbols it operates upon? For example, by changing from a decimal to a binary notation, would we have different algorithms of (say) multiplication or one algorithm with different implementations? At its heart, the question concerns the extent to which the precise sequence of steps bears on the identity of an algorithm. Based on common informal characterizations of algorithms in logic texts (commonly to the effect that "an algorithm is *a precise, step-by-step procedure...*") any difference in the exact sequence of steps (caused by the different notations) would give rise to a different algorithm. But based on the (also) common practice of assigning specific names (e.g., 'Euclid's algorithm') and properties (e.g., asymptotic running costs) to various algorithms, "small variations" in steps should not affect the algorithm's identity. To give an example of what is at stake: in sequentially executing a MERGESORT, does the algorithm change if we stipulate that the left-most possible merge operation is to be executed first (and the right-most one second) or if we stipulate the opposite? Intuitively, we might want to say that it is always the same algorithm, which is just implemented differently; so algorithms are abstract objects in a sense. But, then, such an abstract notion with no additional constraints may be too broad to underpin algorithmic analyses, for, it may allow of "algorithms" that trivially accomplish complicated tasks within just one step.[4] This is because, in practice, the way to exclude such unrestricted cases is by assuming that the algorithms that are suitable for underpinning complexity analyses are those that are easily couched in some formal model of computation ([5]) from the first machine class (fn.5). But this leads us back to granting conceptual priority to machine models, i.e., entities that have sensitive dependence on notational choices. The formal concepts of (Type-2) TMs, K&U machines, ASMs (Gurevich), BSS machines (Blum et al.) and recursors (Moschovakis) tackle these questions differently. But there seems to be no way of ranking our preferences for these concepts on the basis of how well they address the above questions, unless one has already decided on answers to the above questions pre-formally.

Absolute vs. Relative: Are algorithms absolute entities, whose existence is a yes-or-no matter, or relevant with respect to some structure/model/level of abstraction, whose existence is dependent on the defined primitive operations over the stipulated entities in the structure's/model's/level's universe? While a choice on this matter may have no significant bearing on algorithms over countable domains, it does make a difference in the case of uncountable domains, for the latter approach may give rise to algorithmic steps that even involve infinitary labor on a symbolic configuration within one step —e.g., a complete operation between two irrational numbers— and to functions that *would* be deemed algorithmically computable with-

[4] Consider a TRIVIALSORT(B) algorithm for sorting a list B, whose sole instruction reads: "Return sort(B)". The unique step of this algorithm is effective (since effective sorting algorithms exist) and the algorithm is very efficient (running time is $O(1)$). Clearly this is an undesirable "algorithm" for purposes of algorithmic analysis, for, if accepted, it would lead astray our analyses of the complexity of sorting tasks. The example is from [5].

out being effectively computable (an example is the floor function). A related dilemma has to do with the notion of an algorithmic step, and whether any such step is required to be "local" in the sense of some pre-fixed suitable metric or just in relation to the stipulated primitive operations within the given structure/model/level in which the algorithm lives. TMs (ordinary and Type-2) and K&U machines can be seen as formalizing an "absolute" view of algorithms and steps, while BSS, ASMs and recursors can be seen as capturing a "relative" view.

Turning now to trade-offs between *goals*, it seems difficult to single out a formal explicatum on hopes that it would be responsive to all the linguistic and technical practices in mathematics and computer science. An important issue is complexity theory. Can we single out a formal explicatum that would underlie a unified complexity theory for both computer science and numerical analysis? To answer, consider that in computational and mathematical practice we grant (discrete and numerical) algorithms intrinsic asymptotic running time costs. As Dean [5] notes for the discrete case (but also holds for the numerical one), such asymptotic costs must be preserved by any particular machine model that aspires to formalize these algorithms. Ordinary TMs (or equivalent models) satisfy this condition for the discrete case. Therefore, such models support a rich theory of classical complexity and a network of powerful theorems. But the TM model is too narrow to express algorithms in their generality, and it violates the first two inclusiveness desiderata from above (a and b).[5] And when it comes to computations over uncountable domains, although the TTE-framework offers also a relevant complexity theory for real computation [21], it is however too "low-level" to be naturally used by practitioners [17]. Recall after all that Type-2 TMs cannot compute comparisons between reals, which are staples in numerical algorithms. Finally, the BSS formalism, which accepts highly-idealized TMs that operate on exact real numbers as unanalyzed entities in an algebra (so it permits comparisons in a single step) does provide a rich complexity theory for numerical analysis (so it satisfies a, b, and c′). But a BSS machine is far too powerful to be a first machine class, so it cannot be a formal concept that relates to the concepts of classical complexity theory for discrete problems.

The upshot is that formal concepts that turn out to be successful in theorem-generation (in particular, those that support a rich complexity theory in some particular domain) achieve this goal at the expense of generality, violating either (a) or (b) or even (c). On the other hand, Gurevich's and Moschovakis's frameworks fare much better at the generality desideratum. But, as Dean [5, p.54] notes, their achieved generality comes at the cost of severing the foundational link between the practice of informal algorithmic analysis (concerning discrete algorithms) and the complexity costs of first class machine models.

6 Conclusions

I have proposed that there is no unambiguous and uniform way in which the concept of algorithm functions in mathematical and computational practice. Consequently, there is no unique informal concept that could serve as the yardstick by which we evaluate the success of the formal concepts purporting to explicate it. The inherent tensions in our long-time use of algorithms can be alleviated by deliberately sharpening the informal concept *in advance*. And yet there is a number of different ways of trading off inclusiveness against strength of results, which makes it possible that in the end we will have more than one formal notion of 'algorithm' established in the theoretical and practical discourse.

[5] In fact, the situation is worse, because there are additional restrictions for those TMs that found complexity classes. Such machines form an (equivalence) class, called the *first machine class*, which imposes restrictions on the computational power of its members. First machine class models must be powerful enough to handle representation of numbers in binary, but no so powerful as to allow parallel computations with arbitrary branching (see [20]). As it becomes apparent, this pushes even stronger in the direction of a specialized formal concept.

Acknowledgments

This work has been partially supported by the ANR project *The Geometry of Algorithms – GoA* (ANR-20-CE27-0004). I am thankful to Alberto Naibo for many stimulating discussions on algorithms and to the anonymous reviewers of PLS13 for their comments and suggestions.

References

[1] Lenore Blum. Computing over the reals: Where Turing meets Newton. *Notices of the AMS*, 51(9):1024–1034, 2004.

[2] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer Science, 1997.

[3] Jean-Luc Chabert, editor. *A History of Algorithms: From the Pebble to the Microchip*. Springer-Verlag, 1999.

[4] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[5] Walter Dean. Algorithms and the mathematical foundations of computer science. In L. Horsten and P. Welch, editors, *Gödel's Disjunction: The scope and Limits of Mathematical Knowledge*, pages 19–66. Oxford University Press, 2016.

[6] Herman H. Goldstine. *A History of Numerical Analysis from the 16th Through the 19th Century*. Studies in the History of Mathematics and Physical Sciences. Springer-Verlag, 1977.

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

[8] Yuri Gurevich. What is an algorithm? In M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, editors, *SOFSEM: Theory and Practice of Computer Science 7147*, pages 31–42. Springer, Berlin, 2012.

[9] Hans Hermes. *Enumerability, Decidability, Computability: An Introduction to the Theory of Recursive Functions*. Springer-Verlag, 2nd. edition, 1969.

[10] Andrey N. Kolmogorov and Vladimir A. Uspenskii. On the definition of an algorithm. *American Mathematical Society Translations*, 29:217–245, 1963. Translated from the Russian by Elliott Mendelson. Original publication 1958.

[11] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall., 2nd edition, 1998.

[12] Andrey A. Markov. The theory of algorithms. *American Mathematical Society Translations*, (Series 2)(15):1–14, 1960. Original publication 1951.

[13] Andrey A. Markov. *Theory of algorithms*. Israel Program for Scientific Translations, 1962. Translated from the Russian ed. by Jacques J. Schorr-Kon and PST Staff. Original publication 1954.

[14] Yiannis N. Moschovakis. On founding the theory of algorithms. In H. G. Dales and Gianluigi Oliveri, editors, *Truth in Mathematics*, pages 71–104. Clarendon Press, Oxford, 1998.

[15] Yiannis N. Moschovakis. What is an algorithm? In Björn Engquist and Wilfried Schmid, editors, *Mathematics Unlimited — 2001 and Beyond*, pages 929–936. Springer, 2001.

[16] Piergiorgio Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Elsevier, 2nd. edition, 1999.

[17] Philippos Papayannopoulos. Unrealistic models for realistic computations: How idealisations help represent mathematical structures and found scientific computing. *Synthese*, 199:249–283, 2021.

[18] Stewart Shapiro. Acceptable notation. *Notre Dame Journal of Formal Logic*, 23(1):14 – 20, 1982.

[19] Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1936.

[20] Peter van Emde Boas. Machine models and simulations. In *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*, page 1–66. MIT Press, 1990.

[21] Klaus Weihrauch. *Computable Analysis: An Introduction*. Springer Science, 2000.