On Algorithms, Effective Procedures, and Their Definitions

Philippos Papayannopoulos

On Algorithms, Effective Procedures, and Their Definitions*

Philippos Papayannopoulos

fpapagia@uwo.ca

IHPST (UMR 8590), CNRS and Université Paris 1 Panthéon-Sorbonne, Paris, France

Abstract

I examine the classical idea of 'algorithm' as a sequential, step-by-step, deterministic procedure (i.e., the idea of 'algorithm' that was already in use by the 1930s), with respect to the following themes: (a) its relation to the notion of an 'effective procedure', (b) its different roles and uses in logic, computer science, and mathematics (focused on numerical analysis), and (c) its different formal definitions that have been proposed by practitioners in these areas. I argue that the term 'algorithm' has actually been conceptualized and used in contrasting ways between the above areas, and I discuss the challenges and prospects for the community toward adopting a final foundational theory of (classical) 'algorithms'.

Keywords: Definitions of algorithms, Real computation, (Type-2) Turing computability, BSS, Abstract State Machines, Recursors, Numerical analysis, Formalization of mathematical concepts, Computational complexity

1 Introduction

Intuitively, we seem to have a good idea of what an algorithm is. If a competent mathematician or computer scientist was asked to define 'algorithms', her answer would go something like this. Algorithms are step-by-step procedures that are set out in a finite number of instructions; their steps are small, so that they can be carried out by a calculator (human or otherwise) "mechanically", that is without any special knowledge or understanding being required at any step. Each next step is uniquely determined by the current step and just the list of instructions. Call this the *classical* idea of an 'algorithm' (described more precisely in sec.2); or, in Gurevich's words: *algorithms in the sense of the 1930s* (2019). How precise is this intuitive idea really? Can it be more rigorously defined?

*Preprint version. Please cite the published version, which can be found in *Philosophia Mathematica,* Volume 31, Issue 3, 2023, 291–329, https://doi.org/10.1093/philmat/nkad011

1.1 Background and received views

Despite the long-standing prevalence of algorithmic methods in mathematics, attempts to formalize the above idea are relatively recent, and they are mostly a spin-off from the impressive conceptual advancements in our understanding of how to demarcate the computable functions. As it is well known, this understanding came about as the result of seminal work in the 1930s by Gödel, Church, Kleene, Rosser, Turing, Post, and others. The proposed formalisms turned out to be extensionally equivalent, identifying the same class of number-theoretic functions as the algorithmically computable ones. Nevertheless, from a conceptual point of view, the formalisms of Church-Rosser-Kleene and Gödel-Herbrand were unconnected with the notion of 'algorithm' itself, for this notion refers rather to the *process* than to the *result* of a computation. The situation changed when Turing's (1936) work came along, which analyzed the process of computing 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 his name was added next to Church's in a well-known thesis about the class of effectively computable functions:

> **The Church-Turing Thesis** (in the sense of the 1930s) **(CTT)**: *The* effectively *computable functions over the non-negative integers are exactly the Turing-computable functions.*

The fact that Turing's analysis focused on the process of computation, together with the (seemingly innocuous) tacit assumption that what is meant by an 'effective process of computing a function' (aka 'mechanical 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' itself. Indeed, there is now a long tradition in logic and computer science that just identifies 'effective procedures' with 'algorithms'. This tendency is especially manifest in complexity theory, where statements regarding the existence or not of *algorithms* with specific running time abound. Think, for example, statements of the kind: "there is no algorithm that solves the Boolean satisfiability problem in polynomial time" (assuming $\mathbf{P} \neq \mathbf{NP}$); such statements are based on the use of TMs as the underlying model of computation, and hence on the identification of algorithms with TM (i.e., effective) computations (see, e.g., Goldreich 2008, Lewis and Papadimitriou 1998, Sipser 2013, and also sec.3). Since this widely-held view will play a special role in this paper, we will call it *the First Received View* (and later on, the *Symbolic View*) about algorithms:

> **The First Received View about algorithms**: A given procedure is an algorithm iff it is an effective procedure.[1]

[1] Any talk of 'algorithms' in this paper concerns the classical, *deterministic* notion (the sense of the 1930s); see sec.2.

Expectedly, the CTT has been widely taken in logic and computer science as concerning *algorithms*. However, in order to distinguish this statement from the above-mentioned CTT, we will call it the *algorithmic Church-Turing thesis*:[2]

> **The algorithmic Church-Turing Thesis ($\mathbf{CTT_{(alg)}}$)**: *The functions over the non-negative integers that can be computed by following an* algorithm *are exactly the Turing-computable functions.*

The striking extensional equivalence of all the proposed formal concepts that aimed to delineate the effectively computable number-theoretic functions (λ-definable functions, recursive functions, TMs, etc.) helped, in addition, to corroborate a conviction that there must be a clear underlying concept of 'mechanical procedures'. Gödel wrote:

> "If we begin with a vague intuitive concept, how can we find a sharp concept to correspond to it faithfully? The answer is that the sharp concept is there all along, only we did not perceive it clearly at first. ... We had not perceived the sharp concept of mechanical *procedures* before Turing, who brought us to the right perspective. And then we do perceive clearly the sharp concept.
>
> If there is nothing sharp to begin with, it is hard to understand how, in many cases, a vague concept can uniquely determine a sharp one without even the slightest freedom of choice." (quoted in Wang 1997, 232-3; emphasis added)

Combined with the above-mentioned identification of mechanical (effective) procedures with algorithms, this conviction naturally carried over into the notion of 'algorithm' itself. That is, 'algorithms' were also seen as an informal-yet-rather-precise concept.[3] Gurevich (2019, 2.1), for example, wrote recently that "[t]he 1930s notion of algorithm was robust". This sentiment remains so prevalent today that it also deserves a special name; call it *the Second Received View* about algorithms:

> **The Second Received View about algorithms**: There is a unified and robust pre-theoretic idea of 'algorithm' in mathematics and computer science.

1.2 What this paper is about

This paper puts to the test both the aforementioned received views. Are the notions of 'algorithm' and 'effective procedure' the same? Is the classical idea of 'algorithms' —the one that was also at play in the '30s and underlay the development of computability— actually *robust*

[2] Dean (2016a, 20) draws essentially the same distinction between these two versions of CTT.

[3] As just one example (among the many found in the literature) of unequivocal support for this sentiment, consider:

> "The intuitive concept of an algorithm, although it is nonrigorous, is clear to the extent that in practice there are no serious cases when mathematicians disagree in their opinion about whether some concretely given process is an algorithm or not" (Malc'ev, 1970, 18-19).

(or *sharp*)? To both questions I will give a negative answer. First, there exist well-defined stepwise deterministic routines that the mathematical community is willing to accept as proper algorithms, but which are not effective[4] (assuming a specific plausible extension of the CTT to uncountable domains); see sec.4, for examples. Second, 'algorithm', even in its classical (and ostensibly precise) sense, has been an open-textured concept, having given rise to more than one sharpened notion used implicitly in the mathematical discourse.

The former thesis has been held by others philosophers as well.[5] However, in previous discussions, a broader notion of 'algorithm' than the classical one has been usually assumed, while here I only focus the discussion on algorithms in the sense of the 1930s. The latter thesis has first been proposed by Shapiro in (2006; 2013). But, while Shapiro has held that open texture was a feature of 'algorithms' (and 'computability') *initially*, and the notion was finally sharpened in a unique way,[6] I attempt to show that the classical —open-textured indeed— idea of the 1930s has evolved into two distinct conceptualizations that exist in parallel today in mathematics, logic, and computer science.

Specifically, I argue that the *outcome* of *one* mode of sharpening 'algorithms' in the sense of the 1930s is their common identification with the idea of 'effective procedures' (dominant in logic and subdisciplines of computer science). That is, 'algorithms' became identical to computations by an (idealized) agent following a deterministic routine and having inexhaustible space and time. Crucially, effective procedures in this sense are *symbolic*, as is explicitly captured by Turing's (1936) analysis and various later models.[7] I will hereafter refer to this sharpening of 'algorithms' as symbolic (effective) procedures as **'the Symbolic View' (SV)** (so, this is actually another name for the First Received View). But, at the same time, there exists another regimentation of 'algorithms in the sense of the 1930s' (dominant in numerical analysis and other areas of mathematics) as *abstract* —i.e., non-symbolic— processes, defined with respect to stipulated primitive operations for different contexts and structures. I will refer to this second sharpening of 'algorithms' as **'the Abstract View' (AV)**.

The paper is organized as follows. In section 2 I attempt a more precise characterization of the 1930s idea of a 'classical algorithm'. In section 3 I give a more precise account of the SV —i.e., the regimentation of the 1930s idea as effective (symbolic) procedures. In section 4

[4] In the sense that they are not computable by a TM. Thus, in this paper, I start from the assumption that the CTT correctly captures our notion of 'effective' but arrive at the conclusion that the $\text{CTT}_{(\text{alg})}$ does *not* correctly capture our notion of 'algorithmic'.

[5] See, for example, Copeland and Shagrir (2019) and Shagrir (2022, ch.3) for relevant discussions and arguments that 'algorithms' (*simpliciter*) is a broader notion than 'effective procedures' (in the sense of the term used here).

[6] In much the same manner proposed by Lakatos (1976) regarding the (historical) development of mathematical concepts.

[7] In this paper, I consider 'effective procedures' —in the sense of symbolic procedures that are in principle simulable by an ideal agent in pen and paper, given enough space and time— a sharpened, *proto-theoretic* concept (see sec.2 for explanation of this terminology). However, it should be noted that in computer science literature the term appears with variant senses, which increasingly depart more and more from the original sense of 'effectivity' of the 1930s (as a symbolic deterministic procedure); see, e.g., Boker and Dershowitz (2010) and Gurevich (2019) for different (and more recent) uses.

I examine examples of algorithms which, although in line with the pre-theoretic idea of 'classical algorithms', are not effective, and I introduce the alternative view of algorithms, i.e., the AV. In section 5 I give a more precise account of the AV, drawing upon foundational theories of algorithms that regard them as procedures directly over abstract objects as opposed to representations thereof. In section 6 I provide an account for the emergence of the different views (the SV and the AV), arguing that the classical, deterministic idea delineates an open-textured concept, thus allowing room for two different notions of a 'primitive step', an *absolute* and a *relative* one. In section 7 I discuss the dilemmas for the community in further regimenting the intuitive, pre-theoretic, concept.

2 Algorithm: the informal concept and the road to formalization

I have proposed that the Second Received View about algorithms is that we have a robust underlying informal notion of 'algorithm' (in line with the words of Gödel's, Malc'ev's and Gurevich's quoted earlier). What exactly is this notion? Virtually all practitioners might agree on something like the following features as being essential for an algorithm:[8]

1. An algorithm is a general step-by-step procedure, prescribing a sequence of operations for solving a type of problem. It must be expressed as a set of instructions of *finite size*.
2. An algorithm has a set (perhaps empty) of inputs and a (set of) output(s).
3. For any given input, the computation is carried out in a discrete stepwise fashion (that is, without use of any continuous methods or analog devices). Alternatively put, an algorithm proceeds in discrete time, so that at every given moment the state of the computation is obtained from the state at the previous moment.
4. For any given input, the computation is carried out deterministically, without resort to any random methods. The computational state at any given step/moment is *uniquely* determined by the state in the preceding step/time and the list of instructions.
5. The list of instructions that make up the algorithm are to be followed by a computing agent (human or otherwise) which carries out the computation.
6. Each step of an algorithm must precisely and unambiguously be specified with sufficient detail such that no ingenuity whatsoever is required by the computing agent.
7. An algorithm terminates after a finite number of steps.[9]

I will argue that it is this —purely intuitive— idea that has been sharpened further into two different senses of a'algorithm'. But, first, we need to quickly fix some terminology regarding the different stages of sharpening and formalizing an intuitive idea in general.

[8]The characterization I provide here is distilled from descriptions in the classics Knuth (1997), Rogers (1987), and Malc'ev (1970).

[9]While some authors (e.g., Knuth 1997) explicitly pose this requirement, others accept non-terminating procedures as well. See, e.g., Hermes (1969, 2), Gurevich (2015, 189), and Moschovakis (1998, fn.16). In any case, we do not need a definitive stance on this for the purposes of this paper.

Let us say that in moving from a purely intuitive to a purely formal concept, we start with a *pre*-theoretic idea. At this initial level, the informal idea may be precise to a greater or lesser extent, but it is typically employed and thought of on the basis of its paradigmatic cases, while it may (possibly but not necessarily) have some vagueness at the boundaries of its extension. In our case the pre-theoretic idea is something (along the lines of) the characterization of 'algorithms' just given. Then, at a second level, a process of a conceptual refinement of the pre-theoretic idea comes into play, which involves fixing the boundaries and employing appropriate idealizations and/or abstractions. The result of this "theoretic tidying" is an intermediate, more rigorous-yet-still-informal concept, with now precise boundaries.[10] Following Smith (2013), let us call this intermediate concept *proto*-theoretic. In the case that concerns us, such proto-theoretic concepts will be the distilled SV and AV, put forward in secs.3.3 and 5.4 respectively; the notion of 'effective procedure', then, is —according to this paper's proposal— one result of proto-theoretic refinement of 'algorithms'. Finally, a formal, '*fully*-theoretic' counterpart is developed to explicate the proto-theoretic concept. In our subject of concern, Turing machines and other machine-based formal models are fully-theoretic explications of 'effective procedures', while the formal theories discussed in sec.5 will be considered fully-theoretic explications of 'abstract algorithms'.

3 Building the Symbolic View: algorithms defined (formally)

The SV of algorithms is a familiar one. As said already, it is based on the well-entrenched idea that certain formal models of computation that are used to explicate effective computations (especially symbolic ones) in some sense explicate the notion of 'algorithm' as well. Such formal models are often motivated by the need to prove negative results, such as the non-existence of algorithmic methods —either *tout court* or with specific time/space costs— for the solution of given problems. In both computability and complexity theory, the dominant formal model that plays this foundational role is the Turing machine. For example, it is on the grounds of their unsolvabilty by a Turing machine (conjoined with the CTT and the co-extensionality of TMs with other formal models) that problems like the *Entscheidungsproblem* are taken to be unsolvable by effective procedures (i.e., 'algorithms' on the SV); and the same holds —at least, according to the standard foundational story— for complexity results to the effect that no algorithmic method exists for solving given problems within specific time or space limits.[11]

However, none of the early works in computability theory was concerned explicitly with defining 'algorithms' per se (but rather with extensionally delineating the class of computable

[10] This refinement process is alluded to in Carnap (1962) and discussed in more detail by Smith (2013, 344-5) (upon which I also draw here). See also Shapiro (2013; 2015) for a close (but with some differences) account.

[11] More precisely, though, negative complexity results cannot be based on *any* kind of Turing machine or equivalent model (although this is not the case with computability). The actual (equivalence) class of the appropriate models for complexity, called the *first machine class*, imposes some restrictions on the computational power of its members. For example, first machine class models must be so powerful as to be able to support representation of numbers in binary, but no so powerful as to allow parallel computations with arbitrary branching. See van Emde Boas (1990), for details.

functions). And yet later works concerned with definitions of 'algorithms' per se followed suit by remaining in the spirit of the above symbolic approaches. Two paradigmatic cases are the works of Markov (1960, 1962) and Kolmogorov (later together with his student, Uspenskii) (1963). The crucial assumption behind these approaches is that *an algorithm is always a process that exists with respect to some given alphabet and notational system*. Such is the case with Markov's definitions of 'normal algorithms', and so is the case with algorithms as defined by Kolmogorov and Uspenskii (K&U). As the latter authors succinctly put it: "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*." (Kolmogorov and Uspenskii, 1963, fn.2; emphasis added).

3.1 Adding numerical algorithms to the picture

How do the foregoing approaches to formalizing 'algorithms' relate to the informal use of the notion in mathematical practice? I have in mind here the use of algorithms in the context of establishing rule-of-thumb methods for solving mathematical problems, such as finding roots of (systems of) equations with real or complex coefficients (see, e.g., Chabert 1999).

In the last couple of centuries the search and study of such methods has formed the core of an independent mathematical area, known as *numerical analysis*. The crucial point is that in the mathematical folklore numerical algorithms are meant to be algorithms in the same, proper sense of the term used in the theory of computation, even though they prototypically operate on real numbers. That is, they are step-by-step computational procedures, always executable in principle by an idealized human agent; thus consistent with the classical idea of the 1930s.[12]

Now, if we take this assumption for granted, and if we also assume that the CTT tells us something about algorithms —i.e., if we assume the SV, on which CTT = $\text{CTT}_{(\text{alg})}$— how are numerical algorithms to be understood in relation to the CTT, given that the former concern real-valued functions while the latter imposes limitations on algorithmic computation of functions over non-negative integers? A possible answer comes from the framework of *computable analysis*, which purports to extend the classical theory of (Turing) computability (and the scope of the CTT) to real numbers and functions.

3.2 Numerical algorithms on the symbolic view: computable analysis

There are various approaches to computable analysis, but we will consider here the so-called 'Weihrauch school' (aka 'Type-2 Theory of Effectivity' (TTE)). TTE generalizes the classical Turing machine model to a *Type-2* Turing machine that computes a function $f : \mathbb{R}^k \to \mathbb{R}$, by operating over finite or infinite words from Σ^* and/or Σ^ω (where Σ a non-empty finite alphabet) on k input and one output tape. The interested reader can see for details Weihrauch (2000), Brattka and Hertling (2021), and for shorter and friendlier expositions Braverman (2005), Braverman and Cook (2006), Brattka et al. (2008), or Pégny (2016).

[12] I consider it uncontentious that numerical algorithms —more precisely, their deterministic subset that concerns us here— are algorithms in the classical sense of the 1930s. Just consider how many of them bear the names of mathematicians who lived long before the computer era and made use of them in real practice.

Now, accepting that the Type-2 TM formalism naturally explicates the informal idea of 'effective computation of real-valued functions',[13] we can call the following statement '*the Uncountable Church-Turing Thesis*':

> **Uncountable-CTT**: *The effectively computable real-valued functions are exactly the functions that are TTE-computable.*

Analogously to the countable case, and assuming the SV, we can also assert an algorithmic version, since the SV presumably identifies also algorithms over the reals with effective procedures:

> **Uncountable-CTT$_{\textbf{(alg)}}$**: *The real-valued functions that can be computed by following a (numerical) algorithm are exactly the functions that are TTE-computable.*

Crucially, computable analysis introduces some special mathematical results that we will need for our later discussion. Although these results may initially strike one as counter-intuitive, they are in fact straightforward consequences of the requirement of effectivity for processes that deal with infinite objects.

(A) *Only continuous functions are computable*. That is, no discontinuous function can be effectively computed; at least, near the discontinuity points. (See, e.g., Weihrauch 2000, p.6,p.30 or any other of the above references). Intuitively, this holds because if we are computing an approximation to a value $f(x)$, we want this approximation to be a good one for all points near x, since the latter is also given by an approximation.

(B) A consequence of (A) is that the following relations are not TTE-computable:

$$\{(x,y) \in \mathbb{R}^2 \mid x = y\},\ \{(x,y) \in \mathbb{R}^2 \mid x < y\},\ \{(x,y) \in \mathbb{R}^2 \mid x \leqslant y\}$$

The equality relation, for example, is not computable, intuitively, because a TM with two equal real numbers on its input tape(s) would not be able to decide within any finite number of steps —i.e., after having read only a finite prefix of the inputs— that the two numbers are equal and do not differ at some later digit, not yet reached by the machine's head. However, if the two numbers *are* different, a decision is possible. Similarly, the total order relation is also semi-decidable.

(C) Computability of particular operations and functions in $\mathbb{R}$ does not hold only by virtue of their intrinsic properties but also depends on how the input/output data are represented on the tapes. This is in contrast with classical computability theory, where there is no dependence of whether a function $f : \mathbb{N} \to \mathbb{N}$ is computable on how the natural numbers are represented. As a quick example, consider the function: $g(x) = 3x$ ($g : \mathbb{R} \to \mathbb{R}$). Its simplicity notwithstanding, this function is not computable on the decimal representation (see, e.g., Weihrauch 2000, 18).

[13] See Pégny (2016) for a thorough discussion and arguments for that view.

But, it is computable on the *Cauchy representation*, by which a real x is represented as an infinite sequence of rational numbers converging rapidly to x.[14]

3.3 Making the SV more precise

Assuming the SV, let us try to identify central features of 'algorithms' upon which the use of the concept in logic and some subdisciplines of computer science (e.g., complexity) implicitly rests. Drawing upon the above-mentioned formalizations (TMs, K&U machines, TTE), I will pin down essential central themes that I suggest are constitutive of our *proto*-theoretic understanding of 'algorithms' as *effective* procedures. Such features may not be explicitly posited in the theoretical discourse of the relevant areas (computability theory, computable analysis, etc.), but they are still constitutive of their theoretical structure (or so I claim).

We start by reiterating that, on the SV, algorithms are *symbolic* procedures. Thus, they are tightly interlocked with the representations of the data they operate upon. Given a symbolic representation of some input, on the most basic level of analysis an algorithm is a stepwise procedure for effectively manipulating simple or composite parts of the arrangements of symbols that constitute the representation on order to get a different arrangement of symbols that represents the output.

On this account, algorithmic computations are *linguistic/concrete*, in the sense that they are procedures for pushing symbols around.[15] As we saw in the previous section (result (C)), their close connection to representations comes clearly to the fore in the case of uncountable domains; but this is not the only case where sensitivity to representations becomes apparent. A Turing- or K&U-machine (or a Markov algorithm) computing a number-theoretic function would follow a different program if the numbers were represented in unary or binary notation. And, even from a pre-formal point of view, a schoolchild in ancient China would learn different sequences of steps for multiplying (say) 538 by 127 from a schoolchild in contemporary New York. In a clear sense then, we would say that in the foregoing examples, every different notation system requires a *different algorithm* for multiplication.[16] This is a crucial point and we will return to it

[14]This very brief discussion on representations should not leave the reader with the impression that real computability is merely a matter of conventional or arbitrary choice, and that we get to make a given function computable at will, by opting for those representations that "support" the desired result. There are, in fact, inherent mathematical reasons for singling out natural equivalence classes of *admissible representations* for given mathematical structures or spaces. The interested reader can see, e.g., Hertling (1999) and Weihrauch (2000).

[15]Shapiro (1982, 14) echoes exactly this view:

> Mechanical devices engaged in computation and humans following algorithms[1] 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. ... It follows that, strictly speaking, computability applies only to string-theoretic functions and not to number-theoretic functions.

The fn. clarifies that "[t]he term 'algorithm' is understood here in its intuitive or preformal sense as an effective procedure for computation."

[16]Shapiro (2017, 269) expresses this view, in the context of a different debate with Rescorla (2007) on computability:

later.[17]

A further essential feature of algorithms, on the SV, is that any permitted primitive operations are finite. This suggests that both the existence of algorithms for given problems, as well as the notion of 'algorithm' itself, are understood in an *absolute* sense. To see what I mean here, recall that in ordinary computability theory there is an absolute answer whether any function $f : \mathbb{N}^n \to \mathbb{N}$ is algorithmically computable or not. The justification for the absolute character of this answer is implicitly premised upon employing a family of *natural* notations (i.e., notations that do not "cheat" by encoding already crucial parts of the sought answers—they are not "deviant", so to speak[18]) and upon taking for granted that any operations over such notations are finite.[19] The same holds for computable analysis and functions $f : \mathbb{R}^k \to \mathbb{R}$, based again on an analogous assumption of "admissible" representations (fn.14) and of finite primitive operations. This is in contrast with the view of algorithms that we will discuss in sec.5, where the notion (and existence) of algorithms is understood primarily as *relative* to some stipulated *model* (of computation) or *level of abstraction*, which become shaped by virtue of what operations are *stipulated* as the primitive ones. The relative character of this view of 'algorithms' stems from the fact that in this case the stipulated primitive operations can be either finite or infinite.

I contend that, despite being widely accepted, the features of algorithms I have articulated in this section are not essential to or constitutive of the intuitive *pre*-theoretic idea of 'algorithms' that was presented in sec.2. That is, notwithstanding being a well-established view, the SV about algorithms expresses only a *proto*-theoretic sharpening of our primary original idea of an 'algorithm'. As I am about to suggest, we can discern a second conceptualization in the mathematical discourse that has been in parallel use for a long time.

4 Algorithms which do not square with the SV

I suggest that there is a latent tension between how algorithms are conceptualized on the SV and how they actually function in the quotidian mathematical discourse (but also in some subareas of computer science, such as algorithmics). To see this, recall that numerical analysis is concerned with a huge range of algorithms, from simpler to more sophisticated ones. Are such algorithms always effective procedures? Since these examples typically involve computations of operations and comparisons between real numbers, and calculations of real-valued functions, we need to look at the corresponding SV explication of real effectivity; i.e., in this case, TTE. But, recall, order comparisons and equality checking are not always computable by Type-2 TMs (results

> Notice, however, that the standard algorithm for multiplication ... works on numbers written in decimal notation. The same algorithm would either give the wrong results, or senseless results, for numbers written in unary, binary, hexadecimal or Roman notation.

[17] Put differently, one could say that, on the SV, the idea of 'algorithm' is closer to the notion of 'program'.

[18] It is not a trivial problem to single out exactly this family of natural notations. For discussions on deviant notational systems, see Shapiro (1982, 2017), Rescorla (2007), Copeland and Proudfoot (2010), Quinon (2018), Brauer (2021), and Shapiro et al. (2022).

[19] This finitistic requirement applies only to the elementary operations themselves, not to the whole computation.

(B), sec.3.2). As a result, it is certainly possible that various numerical algorithms, which very often include comparisons as necessary intermediate operations, involve non-effective (i.e., not TTE-computable) steps. Let us see some examples, to make this concern more specific.

Example 1: The floor function, $f(x) = \lfloor x \rfloor$, $(x \in \mathbb{R}^+)$.
This is a function that takes as input a real number x, and gives as output the greatest integer less than or equal to x (e.g., $f(2.71) = 2$, $f(83) = 83$). Let us consider it for non-negative reals only. As can be seen in fig. 1, this function is discontinuous. Therefore, it is not Turing computable (see result (A) or Weihrauch 2000 p.7, p.108), and by the Uncountable-CTT it is not effectively computable either. Is there an algorithm for its computation? Consider the following procedure, on any input $x = x_0$ and initial value zero in some register r.[20]

FLOORALG (x, r)
1. If $x < r$, go to 3, else go to 2;
2. Set $r = r + 1$, and go to 1;
3. If $x < r$, go to 4, else go to 5;
4. Set $r = r - 1$, and go to 3;
5. Stop and return r;

Here is also a recursive formulation $(x \in \mathbb{R}^+)$:

$$f(x) = \begin{cases} 0 & \text{if } x < 1, \\ f(x-1) + 1 & \text{otherwise.} \end{cases}$$

The (TTE/Turing) uncomputability of these processes stems from the existence of comparisons and subsequent branching steps. In essence, each such branching introduces a point of discontinuity. Does this mean that FLOORALG is not a "genuine" algorithm? On the face of it, it looks like a perfectly legitimate algorithm in the intuitive sense, for it seems to conform with the informal characterization in sec.2. Yet it clearly is not an effective procedure as it stands, and it violates the assumptions behind the SV (it neither depends on the particular choice of symbolic representations nor it assumes exclusively finite operations).

Example 2: Consider the well-known bisection algorithm. It is a method for finding a root $x_0 \in [a, b]$ of a continuous function $f(x)$, such that $f(x_0) = 0$, when the values of the endpoints of the interval are of opposite signs; that is, $f(a)f(b) < 0$ (existence in that case is guaranteed by the intermediate value theorem). The idea behind the algorithm is to start with an interval that is known to contain a root, x_0, and try to approach the root by iterated bisections of the interval. The algorithm receives as inputs the function f and the endpoints a_1, b_1 of the interval, such that $f(a_1)$ and $f(b_1)$ are of opposite signs. One formulation is as follows:

BISECT (f, a_1, b_1)

[20] This algorithm is an adaptation of one given in Weihrauch (2000, 261).

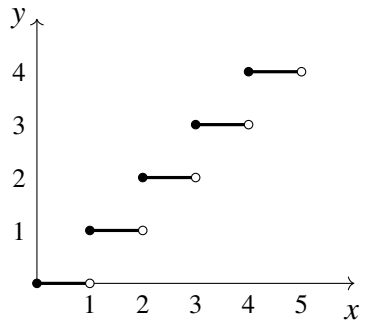

Figure 1: The floor function, $f(x) = \lfloor x \rfloor$, for non-negative reals.

1. Compute $c_i = \frac{a_i + b_i}{2}$ and go to 2;
2. If $f(c_i) = 0$, go to 5, else go to 3;
3. If $f(a_i)f(c_i) < 0$, set $b_{i+1} = c_i$ and $a_{i+1} = a_i$. Else, set $a_{i+1} = c_i$ and $b_{i+1} = b_i$. Go to 4;
4. Set $i = i + 1$ and go to 1.
5. Stop and return c_i

Similarly to the previous example (and for the same reasons), this procedure cannot be implemented in a Type-2 Turing machine (even assuming admissible representations), on account of the repeated comparisons and equality tests it requires (in other words, the functional mapping it induces is not Turing computable). It is, however, one of the most typical examples of algorithms found in numerical analysis (together, perhaps, with Newton's algorithm, which could also be another relevant example).

Example 3: Consider a very simple decision problem of determining whether a quadratic equation $ax^2 + bx + c = 0$ has real roots. A simple algorithm to decide this can be read off from the formula that gives the real roots of this equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

The method consists in determining whether $b^2 - 4ac \geqslant 0$, and high school students learn to perform this algorithm routinely.[21] Now this simple decision problem is again based on evaluating an order relation, so it is definitely possible to have instances, for certain parameters a, b, c that render it (TTE/Turing-)undecidable.

These considerations suggest that the way algorithms are perceived in the broader mathematical community goes beyond a strict understanding of them as effective, symbolic procedures. In actual practice, we know that all such algorithms can be implemented with finite operations

[21] In algorithmic fashion, the process could look something like the following: Step 1: Compute $D = b^2 - 4ac$. Step 2: If $D \leqslant 0$, return 'no real roots'; else return "real roots exist".

and executed approximately (or even exactly, on convenient data) by means of several devices, such as floating-point or symbolic computations,[22] rounded representations, extra terminating conditions, etc. We also assume (probably for good reason) that whenever a computer program successfully executes one of the above algorithms, we have not actually witnessed a case of hypercomputation or the completion of a supertask. What this means, then, is that non-effectivity does not imply non-implementability —despite that the inverse does hold; i.e., effectivity implies implementability— and so these notions should be separated. We will not dwell on this distinction, however, but restrict our attention to how the above bear on the SV. The point is that the discrepancy between what is an effective method and what counts as an algorithm in mathematics undermines the SV, which is not merely the claim that algorithms are abstract procedures defined by their ability to be effectively carried out by some —sufficiently close for one's purposes— *approximate* implementation, but the *stronger* claim that algorithms (in the sense of the '30s) *are* effective procedures themselves.

Now, what is the relation between algorithms of the kind we discuss in this section and symbolic finitistic manipulations (i.e., the SV)? It seems clear that in developing and studying numerical methods, considerations about concrete symbolic representations of their input/output data are not involved in some essential way; even though the choice of representations may potentially bear on their effectivity (result (C), sec.3.2). Attending to the long-standing study of numerical algorithms and their purposes[23] shows that they are largely understood as having a natural structure which is independent of the employed symbolic systems, and so it would be stretching a point to see them as procedures whose main function is the description of mechanical symbol manipulation. While one could say that in carrying out a numerical algorithm one is ultimately caught up in such manipulations, nevertheless, the algorithm itself possesses some properties that are difficult to make sense of if interpreted as mainly concerning moving symbols around. Some such properties are, for example, *convergence/accuracy*, and *stability*, and they all have to do in some way with the proximity of the output value of the algorithm to the target mathematical value. Convergence/accuracy tell us something about whether and how well the algorithm's output approaches the correct result after a number of iterations, while stability expresses how accurate the algorithm's output remains with respect to small perturbations in its input (due to rounding, data uncertainty or truncation errors).[24] But such properties are meaningful and

[22] In this sentence the term 'symbolic' was used in a different sense than in the rest of this paper. 'Symbolic computations' here is a technical term referring to the area of scientific computing that studies computations with symbols that represent mathematical concepts, such as polynomials, algebraic numbers, trigonometric functions, integrals, etc. (for example, computations with expressions of the form '$\cos \pi$', '$e^2 + \frac{\pi}{4}$', etc.). This is a rather unfortunate parallel use of the term 'symbolic computation', but it is already well-established in the relevant literature. To avoid confusion, in this paper this was the unique occurrence of 'symbolic computation' in this sense.

[23] For a large pool of examples in a historical context, see, e.g., Goldstine (1977) and Chabert (1999).

[24] These terms take different precise meanings in different contexts. For example, an algorithm can be 'stable' with respect to *backward* or to *forward* errors, or can be called 'stable' in a *mixed backward-forward* sense, and the exact definitions of these notions vary in the contexts of linear algebra or differential equations. One can see, e.g., Higham (2002) and Corless and Fillion (2013) for details (here we generally use 'stability' in the mixed

useful for the practice of numerical analysis (which substantially involves comparing different algorithms for the same problem, in a manner that does not depend on any particular choice of notational systems) only insofar as by 'input' and 'output' here we refer to the actual real values themselves and not to any symbolic representations thereof. For what makes a "small" perturbation, as well as what it means to be "near" a target value, take a very different meaning when we talk about distances between real values (i.e., points in the real line) and between symbolic representations (e.g., sequences in a symbolic space). And it is with respect to the former notion of distance that algorithms are evaluated in the tradition of numerical analysis.

5 Building the Abstract View: algorithms defined (again)

To make the case for the existence of an AV on 'algorithms' stronger, we first examine some foundational approaches to the theory of algorithms, and then we propose a more detailed characterization, addressing various conceptual points.

We first discuss the BSS (standing for 'Blum-Shub-Smale') framework. This is an algebraic approach to real algorithms and computations, aimed primarily at formalizing the notion of 'algorithm' as used in numerical analysis. By formulating a formal explicatum of 'numerical algorithm', based on a machine model that expands the classical TM, it becomes possible to formally investigate the complexity of numerical methods and develop a real complexity theory based on this model, along similar lines with ordinary complexity theory and the foundational reliance of the latter on TMs. Here we are mainly interested in the conceptualization of 'algorithms' that underlines this active area of research. After BSS, we very briefly look at the foundational approaches of Gurevich and Moschovakis.

5.1 The BSS approach

A main motivation behind Blum et al.'s work is that although numerical analysis is all about algorithms, "there is not even a formal definition of algorithm in the subject" (Blum et al., 1997, 23), despite the field's origins going centuries back:

> [T]he Turing model ... with its dependence on 0s and 1s is fundamentally inadequate for giving ... a foundation to ... scientific computation, where most of the algorithms ... are *real number algorithms*. (Blum et al., 1997, 3; emphasis in original)

Thus:

> 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. (Blum, 2004, 1028)

backward-forward sense).

The BSS model is a model of computation not exclusively over continuous spaces, but over any arbitrary field or ring R. If R is $\mathbb{Z}_2 =< \{0,1\}, +, \times >$, then the model becomes reduced to classical computability theory. The standard reference is Blum et al. (1997), but briefer expositions can also be found in Blum (2004), Smale (1990), and Cucker (1999).

A numerical algorithm is generally formalized as a machine over $\mathbb{R}$. But the theory provides a more general model that is based on the notion of a *machine $\mathcal{M}$ over R*, where R is any commutative, possibly ordered, ring or field. A (finite-dimensional) machine $\mathcal{M}$ has an *input* and *output* space associated with it, a two-way *infinite* tape with cells, and, similarly to Turing machines, a *read-write* head. The machine's *program* is a finite directed graph with five types of nodes, linked by operations or *next node* mappings. When both the input and output spaces are $\mathbb{R}$, the machine can be considered a formal model for numerical algorithms.

Within this framework, the authors prove several results about the decidability and complexity of sets and problems over $\mathbb{R}$ and $\mathbb{C}$. Importantly, a crucial idealization in the approach is that the machine $\mathcal{M}$ is able to manipulate the *exact value* of any real number it is operating on. Real numbers are viewed as "unanalyzed" (atomic, so to speak) entities in the algebra R, and algebraic operations and comparisons are each counted as one unit of work; for instance, if $R = \mathbb{R}$, it takes one step to add, subtract, multiply, divide, and compare any two real numbers, even irrationals.

The crucial take-away lesson from this model —for our purposes— is that in its algebraic (and highly idealized) nature lies the goal of formalizing the quotidian conceptualization of 'numerical algorithms' as direct manipulations of *numbers* (unmediated by any particular representations).[25] This facilitates a more practice-oriented study of algorithms, concerned with properties such as running time complexity, which in the numerical tradition are measured in terms of numbers of operations; whence the treatment of operations and comparisons as unit steps.

5.2 Gurevich's axiomatization and the Abstract State Machine

The view of algorithms as abstract (non-symbolic) entities underlies Gurevich's work as well, in a series of articles trying to define the notion. Gurevich follows the axiomatic route to founding the theory of (sequential) algorithms and then poses a machine model such that "*any* sequential algorithm, however abstract, could be simulated step-for-step by a machine of that model" (Gurevich, 2000, 5). A crucial assumption, which he takes to be fundamental of the notion, is that *every algorithm has its native level of abstraction*. Crucially for our purposes, an additional motivation for Gurevich's specifying his corresponding machine model, the *Abstract State Machine* (ASM), is to sidestep the problem of dependence on data representation, which other machine models (like Type-2 TMs) face. By employing ASMs, algorithms can be modeled in

[25] Compare: "there is a long-standing tradition of decidability results in algebra and analysis that we refer to as the numerical tradition. This theory led to algorithms ... such as Newton's method ... and Gaussian elimination ... It is important to notice that these algorithms manipulate real numbers in much the way proposed [by the model]." (Blum et al., 1997, 31). To see the relevance of this quote, recall that their model stipulates operations over the *exact* real numbers.

a *representation-independent* way (2000, 78). The aim, then, becomes to prove that for every algorithm, on any level of abstraction, there is an equivalent ASM. The relevant level of abstraction of a particular informal algorithm is determined by *what operations are to be executed in one step*, as well as by the abstraction levels of the algorithm's states (2015, 204), and dictates the choice of the vocabulary (2000, 87).

5.3 Moschovakis's set theoretic definition and recursors

The last foundational approach we look into is by Moschovakis; a work also spanning several decades, as Gurevich's. Similarly to Gurevich, Moschovakis too adopts an abstract view of algorithms; that is, algorithms are viewed as processes that are not representation-dependent and not sufficiently modeled by Turing machines.

In Moschovakis's framework, algorithms are viewed as abstract mathematical entities, formalized in set-theoretic terms. He regards algorithms as concepts with intrinsic mathematical properties, which are different and must be distinguished from their *implementations*. Algorithms can be mathematically defined and studied in and of themselves, based solely on their intrinsic properties and not on properties that may be inherited by any particular machine implementation. The proposal, in a nutshell, is that he explicates 'implementations' by means of a specific abstract machine model, called an *iterator* (which is very similar to an ASM) and 'algorithms' themselves as abstract objects by means of a set-theoretic concept that he calls a *recursor* on a partially ordered set. Then, having shown that iterators can too be represented by recursors, the relation of implementability is formalized by means of the existence of a *reduction* relation between recursors. For more details, one can see, e.g., Moschovakis (1998, 2001).

5.4 Making the AV more precise

By grouping the above three approaches —BSS, Gurevich, Moschovakis— together, I do not mean to suggest that they are similar. Rather, I mean to stress that there is some common conceptualization of 'algorithm' underlying all three of them which is the same (I suggest) that also underlies the informal practice of numerical analysis. A crucial common feature is that algorithms are procedures whose *identity and natural structure* do not depend on particular symbolic representations (and thus on a specification of their exact sequence of steps). This claim may sound obscure however, so here is a way in which I think we can understand it. Let us accept that in any discourse about algorithms, a tacit assumption is always at play: only acceptable, non-deviant, notations (for number-theoretic algorithms) and only admissible representations (for numerical algorithms) are considered. If this natural (and essential) presupposition is granted, then we can say that in all the above cases a given algorithm retains its identity as long as it operates over any member of some (equivalence) class of acceptable/admissible notations/representations. This is the sense of 'abstractness' in which I call this understanding of algorithms "abstract", and in which I claim that algorithms "operate over abstract (and not symbolic) entities".

But here lurks an important tension. As said already, not every member of the class of admissible notations/representations entails the same sequence of concrete actions during the

implementation of the algorithm (and we have assumed that algorithms in the sense of the 1930s are procedures to be implemented by agents). In fact, as discussed earlier, most changes in any notational or representational system would result in a different actual procedure. But if the identity of these algorithms is stipulated to remain the same even when the exact sequence of steps changes, this seems to bring about a contrast with a basic intuition that an algorithm is made up of steps whose exact sequence is determined in complete detail. So either abstract algorithms are not algorithms *proper* or, if we want to remain faithful to actual mathematical practice and discourse, we need some way out of this apparent discrepancy. I can see two possibilities here.

One is to introduce some distinction between *higher-* and *lower-level algorithms*. Following this route, one could say that abstract algorithms (like the earlier examples) are higher-level descriptions, which do not specify exact computational procedures but some sort of patterns of actions or "algorithmic schemas" (cf. what Sipser 2013, p.185 calls 'high-level descriptions of algorithms'). Then, the Bisection method tells us something like "given any admissible representations of the endpoints of the interval, employ the (lower-level) algorithm of addition on them, then the (lower-level) algorithm of division by two and then..." This seems a fair (and common) interpretation, but it is crucial to stress that the invoked lower-level algorithms cannot be thought of as effective procedures (algorithms in the sense of the SV) but they too have to be algorithms in the AV sense. This is because algorithms like the Bisection method have steps (e.g., the comparisons) for which there exist no effective concrete algorithms, since they are not TTE-computable as we have seen. The aforementioned description of the Bisection method would include at some point "... use the (lower-level) algorithm to compare (the representation of) the result of the previous step with (the representation of) zero. If it is less, go to step...". But there is no effective procedure for comparing reals, so lower-level algorithms must be abstract algorithms (in the AV sense) too. We are back where we started, having only managed to give a different name to the underlying dilemma that if we recognize the existence of the AV, we recognize the existence of algorithms that are not effective.

Another possibility is to accept that abstract algorithms describe *de re* manipulations of abstract objects (e.g., real numbers) *themselves* (an assumption that I have implicitly made in various places until now). The merit of this approach is that it remains faithful to an understanding of algorithms as problem-solving stratagems. For example, the bisection method and the algorithm described by the quadratic formula (1) tell us how to pin down some specific real *number* (or, at least, one that lives arbitrarily close to it) beginning from the knowledge of other real numbers. Talking about *de re* operations avoids the above problem of accounting for the existence of non-effective steps within a procedure that, overall, is supposed to be always reducible to an effective specification, and it dovetails naturally with our attribution of certain mathematical properties to algorithms (e.g., stability and convergence/accuracy) which, as mentioned already, get distorted if interpreted as concerning symbol manipulation. But talk of *de re* processes has also some disadvantages, such as the problem of our epistemic access to numbers and other platonic identities. Such algorithms in fact lose touch with computations as physical procedures carried out by human agents. I will defer discussion of these issues until the last

section (7).[26]

Back to identifying the main features of the AV:[27] Besides abstract algorithms being invariant under different notations/representations, a second feature of them is that there does not seem to be any *absolutist* standard of whether a procedure is algorithmic or not. While algorithms are *ex hypothesi* assumed to be systematic, mechanical procedures —thus, comprised of "small" steps, blindly executable—, abstract algorithms in particular do not seem to imply any independently-determined "measure" of step-complexity, as is the case with, e.g., computing with Turing Machines (or the machines considered by Markov 1962 or by Kolmogorov and Uspenskii 1963). Rather, what kind of actions comprise a single step is now a *relative* issue, shaped by the level of abstraction, or the model of computation, that reifies the algorithm. This, in fact, goes hand in hand with abstracting away algorithms from symbolic representations, since if what is a step in an algorithm is shaped by what operations are *stipulated* as given *within some structure*, then even single steps of infinite work can be sanctioned, such as some comparison between reals.

Relatedly, all the three formal approaches from this camp that aim to formalize common-or-garden algorithms are underpinned by a model- (or level- or structure-) relative view of admissible steps. BSS assumes that carrying out any of the arithmetic operations $(\pm, \times, \div, \leqslant,)$ makes a single step, and a numerical algorithm operates on a universe that comprises the closure of the base set (typically $\mathbb{R}$ or $\mathbb{C}$) under these operations and compositions. Gurevich's approach poses a native level of abstraction for any algorithm and his ASM model purposely permits steps of any generality.[28] And Moschovakis explicitly accepts that algorithms make sense only relative to operations we accept as primitive on the relevant sets of data.[29]

To summarize, on the AV, algorithms are understood as abstract procedures for *de re* manipulations of mathematical entities. They have an identity and natural structure that do not depend on the employed representations and, hence, on any particular precise sequence of steps implied by these representations. They can also admit primitive operations that are not finitistic, and so

[26]The use of '*de re*' in this context serves strictly the purpose of expressing the fact that numerical algorithms instruct *direct* manipulations of real numbers (rather than via some description of them). Hence, I do not mean to suggest a parallel between the classical *de re*/*de dicto* distinction and the AV/SV one.

[27]This paragraph is meant to be read in contrast with paragraph 3.3.

[28]"In applications, an algorithm may use powerful operations —matrix multiplication, discrete Fourier transform, etc.— as givens. On the abstraction level of the algorithm, such an operation is performed within one step and the trouble of actual execution of an operation is left to an implementation. ...

I sought a machine model ... such that any sequential algorithm, however abstract, could be simulated step-for-step by a machine of that model." (Gurevich, 2000, 81)

[29]"It is tempting to assume that the successor operation $S(n) = n + 1$ on the natural numbers is "immediately computable," an absolute "given," presumably because of the trivial nature of the algorithm for constructing the unary (tally) representation of $S(n)$ from that of n—just *add one tally*; if we use binary notation, however, then the computation of $S(n)$ is not so trivial, ... while multiplication by 2 becomes trivial now—*just add one* 0. The point [.. is] that ... there is no corresponding absolute notion of "algorithm" on the natural numbers—much less on arbitrary sets. Algorithms make sense only *relative* to operations which we wish to admit as *immediately given* on the relevant sets of *data*." (1998, sec.8; emphasis in original).

there exist algorithms that are not effective procedures. Finally, algorithms live within models of computation (or native levels of abstraction or recursor structures), so that their existence is in reference to a given collection of primitive operations and a set of abstract entities to be operated upon.

6 Explaining the emergence of the different views

In sec.2, we suggested an informal characterization of the pre-theoretic idea of 'algorithms'. In the 1930s, this idea —which was from the outset concerned with procedures carried out by humans— *became regimented* as 'effective procedures'; that is, finitistic manipulations of (parts of) any particular symbolic names that are used to stand as "proxies", so to speak, for the actual objects of concern (the SV). Yet, owing to the parallel development of scientific computing, the same idea was also heavily used in a different sense, according to which algorithms have an identity and a natural structure that are both independent of any symbolic names that may have been chosen to represent their data; such algorithms exist rather relative to specific domains and stipulated primitive operations over them (the AV). Let us, then, refer to the proto-theoretic concept that derives from the former regimentation (SV) as '**algorithms**$_{\mathcal{S}}$' and to the one that stems from the latter (AV) as '**algorithms**$_{\mathcal{A}}$'.[30]

In this section, I suggest a conceptual etiology for the two different regimentations, and in the next one I discuss implications for the mathematical practice that different conceptual choices have.

6.1 The Open Texture of Algorithms

How are we to account for the appearance of incompatible formalizations of 'algorithm', despite the alleged consensus at the informal level? The explanation I propose is that we are dealing with an *open textured* concept. That is, the so far established use of 'classical algorithms' in the language and practice of mathematics is not adequate to delimit it in all possible directions.[31]

As said already in the introduction, Shapiro (2006, 2013) has argued that both 'computability' and 'algorithm' had initially some open texture, which was later removed by virtue of results like the CTT. Nevertheless, while Shapiro holds that the notion "of (idealized) human computability ... [is] now about as sharp as anything gets in mathematics ... [without] much room

[30] Although I have been speaking all along about 'algorithms$_{\mathcal{A}}$' in a way that implies that this is also a well-sharpened and precise concept, I believe that if one looks further into it, one will find an idea that is less refined and precise (i.e., still having some vagueness at the borders) than its counterpart proto-theoretic idea of 'algorithms$_{\mathcal{S}}$'. This is partly because the respective *fully*-theoretic regimentations of 'algorithms$_{\mathcal{A}}$' (BSS-machines, ASMs, recursors, etc.) are not entirely co-extensional —and none of them is universally accepted either—, thereby allowing room for disagreement at the borders of the proto-theoretic idea itself (assuming a Lakatos-Shapiro picture, according to which formal results contribute to a backward refinement of the informal concepts too). This is in contrast with 'algorithms$_{\mathcal{S}}$' where the formal concepts are all co-extensional.

[31] The notion of an 'open textured term' is due to Waismann (1945). It is further explained in Shapiro (2006, 2013). See also Makovec and Shapiro (2019).

for open texture" (2013, 178), the view that I propose is that 'human computability' and 'algorithm' are sharp only insofar as they are taken from the outset as symbolic procedures,[32] which *presupposes already a stage of proto-theoretic refinement*. But 'algorithms' and 'human computability', in the sense of the 1930s, allow also for non-symbolic refinements (or so I claim), indicating that these concepts still have some remaining open texture, despite the unanimous acceptance of the CTT. In other words, I contend that the informal characterization of classical algorithms (sec.2) does not reveal on its face whether algorithms are meant to just mean effective linguistic (symbolic) procedures over names of mathematical objects, or whether they can also refer to procedures that go above and beyond manipulation of symbolic names. I argued, however, that the standard take —the (first) received view— has been the former option. Why has it been so?

As it is well-known various early formalizations of mechanical procedures restricted the analysis to symbolic computations (e.g., Turing machines, Post's systems, Markov's normal algorithms, etc.). This was a well-motivated choice at the time, since such formalizations were intended to regiment mechanical processes (within the logic tradition) as a means of demarcating the computable number-theoretic functions. But it seems to me that there is no *conceptual* —i.e., separated from practical concerns— reason to assume that a mathematical stratagem which is intended for identifying the specific *number* that is the solution to a problem (e.g., how to identify the particular *number* which is the greatest common divisor of two other *numbers*) is from the outset conceived as pertaining to combinatorially manipulating the components of any particular linguistic description that has been employed to represent the involved numbers, rather than prescribing operations on the numbers themselves. The distinction between these two intentions might not come to the fore during physical executions of algorithms —i.e., computations—, since any infinite abstract object is represented in practice by some composite linguistic object in order for a physical agent to operate on it. But it is an existing distinction and should be borne in mind, for it does have potential consequences. For one, operating over composite linguistic descriptions that are assumed of infinite length (e.g., strings on Type-2 TMs) distorts the natural structure of an algorithm that is designed to feasibly produce the solution to some problem. For another, translating a process which is constructed with numbers as unanalyzed entities in mind into a process of piece-by-piece manipulation of the descriptions of these objects (e.g., large strings, even finite) distorts formal properties of the original algorithm, such as its identity conditions, estimated convergence and stability, and even estimated computational costs.[33] The reason, however, that such discrepancies do not typically become apparent in everyday computational practice is because we purposely restrict our attention and efforts to developing

[32] Something that Shapiro implicitly does, as can be seen in the quotes in fn.15 and 16.

[33] For example, a major factor that affects how well the algorithm from eq.(1) behaves is whether $b^2 \approx 4ac$ (a case in which the algorithm becomes prone to unreliable results due to catastrophic cancellation). Now, it would be impractical (and would likely lead us to the wrong conclusions) to try to decide whether such properties hold for any problem instances by examining properties of the symbolic strings that represent these numbers, such as (say) their lengths or their distance within some symbolic space with a relevant metric. It is well conceivable that b^2 might indeed be approximately equal to $4ac$, and yet the relevant properties of the corresponding strings to indicate the opposite (and vice-versa).

algorithms that are *stable* ones (i.e., that "behave well" with respect to any perturbed data and error accumulation arising from the "translation" into symbolic descriptions).

The foregoing comments notwithstanding, the important fact for the subject of this section is that a distinction between symbolic and *de re* manipulations does not bring about a noticeable difference in the *extensions* of the classes of 'algorithms' and 'algorithmically computable functions', as long as we remain in the countable realm. We could think of (say) Euclid's algorithm as being either a(n equivalence) class of effective procedures (i.e., algorithms$_{\mathcal{S}}$; but different ones for each different notation, e.g., decimal or Roman, etc.) or a (unique) algorithm$_{\mathcal{A}}$ (a model/level-relative sequence of direct operations over numbers).[34] Both choices lead us to the same extension for the class of algorithmically computable *number-theoretic* functions, and so any reasons to opt for either one would be rather philosophical than purely mathematical. This explains why the long-standing identification of the CTT (which concerns effective procedures) with its algorithmic variant, CTT$_{(\text{alg})}$, has not led us astray in the practice and still remains so widely assumed.[35]

Still, though, would there be any compelling *philosophical* reasons to choose between *number-theoretic* algorithms$_{\mathcal{S}}$ and algorithms$_{\mathcal{A}}$? I believe no. I submit that there is nothing in the intuitive notion of 'algorithm' to suggest a definitive choice between algorithms$_{\mathcal{S}}$ and algorithms$_{\mathcal{A}}$ as the (supposedly) correct regimentation. The informal characterization of sec.2 just does not have enough shape to definitively weigh against one or the other; any direction taken would amount to further sharpening the informal idea. This is the open texture of the pre-theoretical idea, the origin of which we are now turning to discuss.

6.2 Suggesting an etiology: Two notions of an 'immediate' step

I propose that the source of the problem —the exact place where the open texture is to be found in the pre-theoretic concept of sec.2, so to speak— comes down to this: how exactly is the informal requirement that steps be specified in the "smallest detail" (so that no ingenuity whatsoever may be required to perform the algorithm) to be understood and sharpened? Let us try to make this requirement more precise by rephrasing it as something like this: at any given state the application of the next step must be immediately clear and recognizable, in the sense that no actual thought is needed to perform it. This is better, but still vague. How can we embed this idea in a proto-theoretic regimentation of 'algorithms'? For what kind of actions exactly can we say that they require no thought whatsoever? I suggest that there are two possibilities of going about this, which respectively underlie the two regimentations, SV and AV.

The first possibility is to try to implement this requirement in the most absolute sense possible. That is, to ensure that algorithmic steps somehow be immediate and "intuitively obvious"

[34] For problems faced by both these views, see Dean (2016a).

[35] Recall that both the CTT and the CTT$_{(\text{alg})}$ are claims concerning *extensions* of concepts. See also Smith (2013, 345,350) and Dean (2016a, §2.1) in this regard. Nevertheless, my claim about the co-extensionality of 'algorithms$_{\mathcal{S}}$' and 'algorithms$_{\mathcal{A}}$' in the countable realm is in fact premised on a condition of always finitistic primitive operations, which I will discuss and put under more scrutiny in the following subsections.

tout court.[36] A way to implement this ideal suggests itself when looking at preceding philosophical attempts to found mathematics in self-evident operations of that sort. Specifically, aspects of Hilbert's program can be seen as hinting at a suitable solution: express the entities under consideration in ways that are "intuitively present as immediate experience prior to all thought" (Hilbert, 1926). For Hilbert, this is achieved by the use of *concrete signs* (p.376) "whose shape is ... immediately clear and recognizable" (such as a unary notational system). Then, various aspects of these objects (concatenation, order of occurrence, etc.) are "given intuitively ... as something that neither can be reduced to anything else nor requires reduction". So, if these claims of Hilbert's are accepted, then the symbolic regimentation of algorithms —e.g., as a TM operating on stroke numerals— responds satisfactorily to the requirement of actions being immediately clear and recognizable in the strictest possible sense. Essentially, the same requirements are found in Markov's and K&U definitions of algorithms, expressed in terms of allowing only *local* computational steps (see Markov 1962, Kolmogorov and Uspenskii 1963, and Uspensky and Semenov 1993).

The second possibility is that instead of requiring that the computing agent deal with actions that are immediately recognizable in some absolute —or strictest possible— sense, we require that the agent deal with actions that are immediately recognizable in the sense that the agent has achieved such a degree of familiarity with them as to immediately recognize them as obvious, requiring no thought. Think, for example, how in our early school years we struggle to memorize multiplication tables, but later on we become so familiar with them as to treat them as immediate one-step operations in further calculations. Under this understanding, then, what specifically the immediately recognizable steps are turns on the particular domain of discourse under consideration. As some examples, formal models in logic, such as μ-recursive functions or λ-calculus, take certain operations, such as composition or substitution, as immediately recognizable; truth-table validity tests (propositional logic) take the assignment of truth values to the basic logical connectives as primitive; the algorithms we learn at elementary school for long multiplication and long division take the multiplication tables we have memorized in first grades as primitive; and more advanced number-theoretic algorithms, such as Euclid's algorithm, take all four arithmetical operations, as well as comparisons between integers, as primitive. By the same token, then, it would make some sense to assume that in the centuries-old practice of constructing algorithms for solving problems (numerical analysis), the notion of an 'immediate step' that has tacitly been at play is also that of the familiar-immediate primitives in the domains of interest (e.g., arithmetical operations, comparisons, and even n^{th} roots).

Let us call the above two senses of an 'immediately recognizable step', respectively, the 'absolute-immediate' and the 'relative-immediate' sense. Crucially, both senses lead us to extensionally equivalent outcomes in the countable realm (i.e., when we restrict attention to number-theoretic algorithms). Why is this so? I suggest two reasons: an underlying fact and a condition. The underlying fact is that in countable domains we can have a *finite* name for every entity in the domain. This allows for a one-to-one correspondence between symbolic and *de re* operations.

[36] Compare: "Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided." (Turing, 1936, 249).

That is, fixing an appropriate notational system, we can assign to any primitive *de re* operation described by our algorithm a unique symbolic operation (and vice-versa).[37] But, the primitive operations assumed by the algorithm may not be of the 'absolute-immediate' type; for example, the algorithm might involve using the product of two integers within a step. Then the following condition becomes relevant, ensuring the co-extensionality as long as it is met: *no primitive operation can be of infinite work*. Crucially, as long as operations of infinitary work are not taken as primitive, any "relative-immediate" primitive operation (e.g., '$24 \leftarrow 4 \times 6$') can always be ultimately reduced to a finite number of "absolute-immediate" operations (e.g., as repeated unary addition), by a reduction process which is itself finitistic and unique. This means a one-to-one correspondence between absolute- and relative-immediate operations as well (assuming that the notational systems have been pre-determined); ultimately ensuring the existence of one-to-one correspondences between any two of *de re*, absolute-symbolic and relative-symbolic operations. Thus, owing to this finitistic-all-the-way-down structure, we have equivalent up-to-isomorphism notions of 'step' in countable domains, accordingly leading to co-extensional notions of 'algorithms$_{\mathcal{S}}$' and 'algorithms$_{\mathcal{A}}$'.

6.3 The open texture shows its face in uncountable domains

Despite that the open texture exhibited by the 'small step' requirement is benign for the countable case, trying to sharpen this requirement in uncountable domains opens up Pandora's conceptual box. Are we to understand an "immediately recognizable" step as an absolutely minimal operation on symbols, or are operations such as "divide π by 2", "check whether $b^2 - 4ac \geqslant 0$" (with a, b, c reals), immediately recognizable steps in the sense of intimate familiarity with them in quotidian mathematical practice? While such operations may involve infinitary work to be completed, the use of rounded numbers (or other workarounds) we typically employ in *implementing* these operations underpins a familiarity that makes us assume operations of this kind as primitive (I think this familiarity is ultimately the product of our extensive use of operations over buck-stopper[38] representations of reals in our mathematical education). But, now, such 'relative-immediate' operations, if accepted, do not lead anymore to outcomes that are co-extensional with those based only on 'absolute-immediate' operations, as was the case in the

[37] Such a correspondence ceases to be bijective in uncountable domains, since then we can have multiple names to denote the same entity. This holds either if we use only finite representations (so different rounded numerals may be used to denote the same real) or if we even allow the use of infinite strings to ensure equal cardinalities between the set of names and the set of denoted entities (think e.g. the infinite names '0.999...' and '1.000...', or of different Cauchy sequences that converge to the same number).

[38] A term t is called a 'buck-stopper' for an agent, in a given context, if the term is such that when encountered by the agent it does not leave much room for wondering "how much is t?" In other words, by seeing t an agent automatically grasps the quantity denoted by it. To give some examples: when we see 'III' or '34501' we recognize instantly the actual *numbers* (quantities) denoted by these terms. It would not make much sense to ask "but how much really is 'III'?" On the other hand, when we look at 'IIIIIIIIIIIIIII' or '3450516999154' we do not get an instant grasp of the denoted quantities and might be justified to ask "what numbers are these?" This idea (and the term 'buck-stopper' itself) comes from Kripke's unpublished Whitehead Lectures (1992). An explanation of it can be found in Shapiro et al. (2022).

countable realm. This is because the two reasons we gave above (the underlying fact of the existence of finite representations for every entity and the condition of allowing only finitistic operations) are not anymore relevant together. We get confronted with the following dilemma.

On the one hand, accepting only the 'absolute-immediate' understanding of 'step' (e.g., by representing real numbers by strings of '0's and '1's, so that steps make only immediately recognizable changes to them) adheres to the condition of finitistic primitives but means that we have to represent the reals by infinite strings (i.e., we meet the condition but lose the underlying fact; this approach is exemplified, for example, by TTE). The consequence is that we rule out operations like the aforementioned examples as permissible; accordingly, we also rule out the procedures from sec.4 as proper algorithms, and remain in line with the SV.

On the other hand, admitting *also*[39] the 'relative-immediate' conception of 'step' allows us to permit operations like the above as primitive but at the same time opens the door to cases which involve infinitary work within a primitive operation. This option is to some extent reconcilable with mathematical practice through the employment of suitable workarounds (rounded representations, stopping rules, etc) that assure the reliability of the approximated output. But it means that we can only preserve the finitistic condition by replacing the original algorithm by an approximate substitute for it. And, such substitutions notwithstanding, it also means that we admit inside the class of algorithms processes that are not effective *by themselves* (i.e., prior to being approximated), putting us in line with the AV. For example, we recognize the existence of a FLOOR-algorithm$_{\mathcal{A}}$, even though no *exact* FLOOR-algorithm$_{\mathcal{S}}$ can exist.

The outcome of all this is that the one-to-one correspondences that existed before between the *de re*, absolute- and relative-(familiar-)immediate operations do not hold anymore; thereby breaking the up-to-isomorphism equivalence between these notions. We are now forced to make conceptual choices whose consequences matter. But, again, the open texture of the pre-theoretic concept does not tell us which way to go. The informal idea just does not have enough shape to tell us which notion of 'immediate step' to give conceptual priority to, when it comes to uncountable domains: the 'absolute-immediate' one or the 'relative-(familiar-)immediate' one. Any answer would just be a community's additional *choice* toward more conceptual sharpening.

6.4 Why the condition of finitistic operations?[40]

There is a remaining wrinkle that needs to be ironed out. In the previous subsections, I argued that the co-extensionality between 'algorithms$_{\mathcal{S}}$' and 'algorithms$_{\mathcal{A}}$' holds in the countable case on the condition that all primitive operations are of only finitistic work. But, why accept this condition, in the first place? Seen from the perspective of trying to capture the ideal of an absolutely immediate and intuitively present step (as in the logic and foundations of mathematics tradition) the answer seems clear: only at most finitistic processes can be of that sort (as, e.g., Kronecker, Skolem, Hilbert and the constructivists had it), and so the condition is not only

[39]The use of 'also' here is because this second scenario is not mutually exclusive with the first but actually expands it, since familiar-immediate steps include absolute-immediate ones.

[40]The issues discussed in this subsection arose from extended comments by an anonymous reviewer of *Philosophia Mathematica*, to whom I am indeed grateful.

justified but necessary. However, this is not the case from the point of view of algorithms as primarily problem-solving stratagems. As long as an algorithm is seen as a method leading us to discover a specific mathematical entity (which is either the exact or a close enough solution to our problem), then there is no conceptual requirement for all the intermediate steps to be formulated finitistically. For the concern in this case is not to secure the process foundationally, or to formalize some intuitive idea of 'effectivity', but to come up with stratagems that just "do the job" for the particular problem at hand and show us a reliable way of locating the sought entity. The only requirement, then, is that any such stratagem is susceptible of being approximated by finite means in a way that does not affect its reliability. That is, the unavoidable errors that occur from the approximating process are tolerable for any particular application and context at hand. These conditions —whenever satisfied— compensate for the appearance of primitive operations that involve infinitary work in the case of numerical algorithms (e.g., arithmetical operations and comparisons between any two reals). But they also raise the question: why do we not see similar primitive steps of an infinitary flavor in number-theoretic algorithms as well? If such operations had indeed been seen in practice, then an extensional divergence between algorithms on the SV and on the AV could have been noticeable *also* in countable domains.

Before attempting an answer, let us see what such operations could look like. The recursive (and thus already finitistic) nature of the arithmetic operations in $\mathbb{N}$ makes it so that any alleged non-effective primitive operations would have to introduce any infinite work in some other, indirect way. As an example, then, consider stipulating a primitive operation such that on some input $n \in \mathbb{N}$ the result of the operation is "1" if n encodes a Turing machine that halts on its own code, otherwise the operation outputs "0":

HALT(n): given $n \in \mathbb{N}$, return 1 if the n^{th} TM halts on its own code; else return 0.

Now, any primitive infinitary steps in numerical algorithms are allowed on account of being in principle implementable; i.e., they are susceptible to finite approximations by means of rounded representations and stopping rules. In a similar vein, the HALT(n) operation could be considered implementable by employing some stopping rule like the following:[41]

HALT(n,m): given $n, m \in \mathbb{N}$, return 1 if the n^{th} TM halts on its own code after m steps; else return 0.

Nevertheless, although necessary, plain implementability is not *the* crucial requirement. In order for a prescribed sequence of steps to find its way into the algorithmic practice, it rather needs to be *reliably* implementable. That is, if the sequence of steps is implemented approximately (i.e., in case that the process is not effective/Turing computable), then the result needs to be adequately satisfactory, i.e. the induced errors remain small for the particular problems at hand. This, of course, is a context-dependent and essentially pragmatic criterion, which indicates that the motivations lying behind the study of algorithms as problem-solving processes are very

[41] I thank the same anonymous reviewer for suggesting these examples of the HALT operation.

different from the motivations that lay behind the development of the SV.[42] In sharpening the former idea, the concern was from the outset with useful and reliable (i.e., accurate/convergent and error-robust) implementability,[43] while in sharpening the latter idea, the concern was with a rigorous delineation of the concepts themselves. Now, given these motivations for the AV, it is difficult to imagine some practical number-theoretic problem for which a sequence of steps involving a HALT operation would make a useful algorithm —let alone a stable and reliable one. Given that the only sense in which infinitary operations can penetrate number-theoretic algorithms is in the form of operations like the HALT one (because natural numbers have only finite representations and arithmetical operations are recursive, hence finitistic), this partially explains why we have not seen number-theoretic algorithms that violate the finitistic condition so far.

There is a second reason that I also think responsible for the lack of infinitary operations in number-theoretic algorithms$_{\mathcal{A}}$; though, this one is more of a sociological nature. The SV of algorithms provided an extremely fruitful ground for both foundational and practical breakthroughs in the theory of computation. Owing to its huge success, the syntactic approach to algorithmics and the theory of computation became the orthodoxy, and the foundational framework underpinned by the CTT became the yardstick against which many later aspiring foundational frameworks would be evaluated.[44] For reasons that are not yet clear to me (and which I think worthy of further investigation) the conceptual advances in the theory computation were considered as refining the informal notion of 'algorithm' in almost its entirety (i.e., the resulted refinements of the notion of a'well-defined effective method' were brought to bear on the problem-solving conception of 'algorithms' as well), although no conceptual reasons for the necessity of such a link really existed (except, perhaps, reasons of conceptual unification). The result is that the widespread acceptance of the CTT$_{(\text{alg})}$ became so deeply entrenched in the mathematical folklore that it would really require a major breakthrough (or a number of them) in algorithmics to accept as a bona fide algorithm any problem-solving stratagem over *countable* domains which

[42] To see the contrast, recall that in the process of sharpening 'algorithms$_{\mathcal{S}}$', and in order to comply with the then new formal results, a number of limiting cases had to be included in the extension of the concept as well. Such cases were, for example, procedures for which it is impossible to tell in advance whether they will terminate on a given input (owing to the crystallization of the class of effectively computable functions as the *partial* recursive ones), or algorithms that do nothing in particular and yet never terminate (owing to accepting the *empty* function as a limiting case of a partial recursive function. I think that such theoretical and limiting cases would hardly be considered as falling under the concept of 'algorithm' within the problem-solving tradition that underpins the AV.

[43] Such concern with selecting only error-robust methods is already found in Gauss (1857, 31) (see also Goldstine 1977, 258 in this regard); and also later in von Neumann and Goldstine (1947) and Turing (1948).

[44] To wit, although Kolmogorov and Uspenskii (1963) aimed to give the broadest account of 'algorithm', they still considered as the measure of success for any such account that only partial recursive functions are algorithmically computable (see their introduction and §3). Similarly, the BSS model is meant to be exactly reducible to the classical TM model when the examined computations and algorithms are over discrete domains. And even Chabert (1999), after having examined a long list of (non-effective) numerical methods, presents in a final chapter (titled "Towards the Concept of Algorithm") the developments of the 1930s as the culmination of the long-standing study of algorithms as problem solving methods. The only exceptions that I am aware of to the "unwritten rule" that any foundational theory of algorithms should comply with the CTT are the frameworks of Gurevich and Moschovakis (cf. also Dean 2016a and Kapantaïs 2016, 2018).

would defy the CTT. In other words, it seems to me that even if a useful method with a HALT operation were to be discovered tomorrow —one that would indeed solve some problem from the practice when implemented approximately (via a HALT(n,m) stopping rule)— there would still be strong reluctance from the community to accept it as a proper algorithm (even though *numerical* algorithms *are* essentially of the same nature).

7 The road ahead

I have argued that in examining the mathematical practice one comes to see more than one informal proto-theoretic variant of the concept of 'algorithm'. And this holds even when the concept is considered only in its classical, sequential, deterministic conception.[45] I have also suggested that a reason for this phenomenon is the open texture of the informal, pre-theoretic idea. Accordingly, it is a matter of deliberate, conceptual choice by the community, as regards the key idea of a small step, which variant(s) to retain, sharpen further, and use their formalized theories as frameworks for novel mathematical work. In this section, I discuss some conceptual implications that each possible choice has.

The two existing variants are algorithms$_{\mathcal{S}}$ and algorithms$_{\mathcal{A}}$ (based on the SV and the AV). Let us assume first that the community has to keep only *one* of them. Let us also assume that a reasonable goal is to end up with one, unified (proto-theoretic) concept of algorithm for the countable and the uncountable cases alike. Indeed, it would be unnatural to deliberately aim for different definitions of algorithms over $\mathbb{N}$ and $\mathbb{R}$, knowing that they will be unrelated or even incompatible. But this last assumption, natural though it may be, has crucial implications. For it brings about an inescapable need for trade-offs in any attempt to define algorithms in a unified way. These trade-offs are between generality and inclusiveness on the one hand and domain-specific fecundity on the other. The point has been made aptly by P. Smith for mathematical definitions in general. Let us see it, and then examine how it fittingly shows up in our case.

> Definitions in mathematics get shaped by a number of pressures. We may start with a cluster of informal basic results which we want our formal definitions broadly to sustain ... There is then, on the one hand, the desire for increasing generality, inclusiveness, abstractness. But on the other hand, we also want the defined concepts to feature in powerful theorems.
>
> ...
>
> The desire for generality and the desire for a rich network of theorems evidently push in different directions, for the more general and all-embracing a concept, the fewer the interesting truths about all its instances. And hence there may be a number of acceptable ways of trading off the virtue of generality against the virtue of theorem-generation, and a number of concepts encapsulating these different trade-offs. (Smith, 1998, 174-5)

[45] Arguing that the *general* idea of algorithm, with all its extensions (parallel, non-deterministic, interactive, quantum, geometric, deep algorithms etc.) has not a unique, precise meaning would not be any news really.

Trying to define algorithms certainly exemplifies this predicament. Regimenting algorithms as algorithms$_{\mathcal{A}}$ provides generality, inclusiveness and abstractness but loses touch with the necessary concreteness for basing classical complexity theory on them. And regimenting them as algorithms$_{\mathcal{S}}$ (i.e., always effective) loses touch with algorithms in numerical analysis, geometry, etc. Here is how the first problem comes about.

In a discussion on different foundational frameworks for 'algorithms', Dean (2016a) puts forward three aspects of the status we grant to algorithms in our mathematical and computational practice. Dean considers algorithms mainly in the context of complexity theory and algorithmic analysis, and, accordingly proposes that any foundational theory of 'algorithms' should be responsive to the linguistic and technical practices of *these* fields and so capture all the three aspects he proposes. These are roughly as follows (Dean, 2016a, 34): (a) Algorithms are mathematical procedures (described informally or in pseudocode) that can be carried out for given inputs and lead to obtaining an output through a sequence of intermediate states. (b) Algorithms can be implemented by members of models of computation $\mathfrak{M}$; that is, by some machine $M \in \mathfrak{M}$, in a way that M computes the same function as the algorithm does and operates in the same step-by-step manner. (c) Each algorithm has an intrinsic asymptotic running time complexity, which imposes certain constraints on the implementation relation between it and the machines from $\mathfrak{M}$. Specifically, $\mathfrak{M}$ must belong in the first machine class (see fn.11), and an $M \in \mathfrak{M}$ that implements an algorithm $\mathcal{A}$ must have the same asymptotic running time complexity as $\mathcal{A}$.

The above conditions are crucial for delimiting the basic operations which can reasonably be regarded as basic computational steps in any analysis of problems and algorithms that purports to be usefully applicable to implementations on real-world computers. Let us accept them as they are for our discussion.[46] The first two aspects raise no issue with what concerns us here. The third aspect (c) —which admittedly does the main work of providing the necessary restrictions for bearing on physical computations— poses interesting challenges. More specifically, it is not easy to see how algorithms$_{\mathcal{A}}$ could conform with it. As Dean points outs, in computer science, typical algorithms (e.g., a Mergesort) possess their running time complexity (up to asymptotic bounds) by virtue of counting their number of operations in some appropriate pseudocode specification. A goal of this practice is to offer grounds for comparing different algorithms for the same task in terms of their efficiency. But in order to ensure natural results — by avoiding, for example, steps that are effective yet so complex as to trivialize the process[47]— it is assumed that any pseudocode specification underpinning an algorithm's analysis is such that it can be implemented in some *reasonable* machine model, and in a manner that preserves the same step-by-step evolution. And, crucially, the requirement of "reasonableness", which provides the safeguard against triviality, is captured by the restriction to models from the first machine class only (see also van Emde Boas 1990 and Dean 2016b).

[46] Strictly, the second (b) aspect of Dean's list seems to assume that algorithms exist independently from their implementations, which we have not taken for granted in this discussion. But it would suffice to consider it true for our purposes in this section.

[47] See, e,g., Dean (2016a, 33) for an example.

Nevertheless, these conditions are too restrictive for numerical analysis. In this tradition, algorithmic$_{\mathcal{A}}$ analysis is not grounded in relevant fixed models of computation, but based on informal (yet rigorous) estimations of the number of operations in their mathematical presentation. Very often, algorithmic$_{\mathcal{A}}$ costs are estimated in terms of floating-point operations (FLOPs; see, e.g., Higham 2002, 3); yet these are not part of some formal model of computation either.[48] And even when formal models have been employed, in a manner that remains faithful to the needs of this tradition, such models do not satisfy condition (c). Recall the BSS model for formalizing algorithms$_{\mathcal{A}}$ (sec.5.1). On the basis of it a whole theory of complexity of numerical problems has been developed (Blum et al., 1997). A BSS machine, however, is much too powerful (and idealized) to be considered a reasonable model from the first class.[49] And yet, the whole BSS framework does provide a useful complexity theory, predicated upon the notion of 'algorithm$_{\mathcal{A}}$'. Similarly, Moschovakis's framework also enables more mathematically elegant analyses of algorithmic costs, as he shows in a focused example of how the analysis of Mergesort can be mathematically founded on the theory of recursors, with no implementation "distractions" in the process (1998). And it seems *prima facie* possible that the ASM framework of Gurevich can too simplify such analyses in an implementation-independent manner.

Hence, it occurs that while Dean's conditions accurately reflect the aspects in which a foundational theory of algorithms should be responsible to the practice of computer science (and especially classical complexity theory), they are too restrictive for a unified framework that would subsume algorithms on countable and uncountable data, and would be responsible to complexity theory, algorithmic analysis *and* numerical analysis together. But, on the other hand, formalizations of 'algorithm$_{\mathcal{A}}$' —such as the BSS model— are too inclusive to feature in a rich network of theorems in classical complexity theory, since algorithms$_{\mathcal{A}}$ are not even always effective, as opposed to algorithms$_{\mathcal{S}}$, and since they even ignore subtle yet fundamental differences in the difficulties of addition and multiplication. And, as Dean (2016a, 54) also points out, the generality of the recursor and ASM frameworks severs the foundational link between the practice of informal algorithmic analysis and the complexity costs of reasonable (i.e., first class) machine models to which the informal algorithms are tacitly assumed to be always reducible. Once again, the different pressures on developing mathematical definitions become increasingly apparent in the quest for definitions of 'algorithms'.

The road ahead for the community, then, might be as follows. One possibility could be to argue for just identifying 'algorithms' with 'algorithms$_{\mathcal{S}}$' in all cases, on the basis that algorithms that

[48] Typically (but not universally), one FLOP signifies one addition, subtraction, multiplication, division or comparison. However, textbooks in numerical analysis display a big variety in the exact conventions. The widespread practice of analyzing numerical algorithms in terms of numbers of FLOPs goes further distance toward justifying our claim that in NA algorithms are better understood as model-relative, where what counts as an elementary step is determined by the stipulated primitive operations.

[49] Despite that BSS machines are too powerful and clearly unrealistic with respect to physically implementable computations, the BSS model is successful in providing a complexity framework for taxonomizing numerical problems and algorithms that are developed with the goal of been implemented on digital computers. An account of how this is possible is provided in Papayannopoulos (2021).

are not unequivocally effective procedures are in fact terminological abuse and should strictly be called otherwise ('methods', 'rules', 'schemes', 'general procedures', etc.). Only when a straightforward way of fixing the details and turning the routine to a symbolic one that proceeds at each step in an unambiguous manner —so that it is also translatable to a reasonable model of computation— exists, we can legitimately talk about 'algorithms' in the proper sense of the term.

Such a choice, however, would be oddly at variance with mathematical practice in (at least) two ways: First, it would go against standard practice in numerical analysis, where there is virtually no concern with specific representations of the data.[50] Second, we may face additional conceptual implications for computability and complexity, if we also presuppose that we aim at a a unified regimentation of 'algorithms' that would subsume both the countable and the uncountable case (a natural presupposition, nonetheless). For one, the well-entrenched idea that computability in $\mathbb{N}$ is absolute would lose some of its force (unless we are willing to accept cumbersome representations as canonical ones). This is because real computability has strong sensitivity to how we represent the real numbers (sec.3.2), since even standard arithmetical operations like addition or multiplication can become uncomputable —a no-go for any theory of computability— if decimal (or other base-b) representations are employed. For another, since we know that decimal and binary notations are privileged for complexity purposes,[51] if we were to abandon them as canonical representations, this might add an extra obstacle to the possible goal of finding a concept of 'algorithm' that satisfies Dean's (2016a) requirements and is responsive to the practice of complexity theory.

In the opposite direction, the choice could be to regiment 'algorithms' as 'algorithms$_{\mathcal{A}}$'; that is, to argue that the correct view of algorithms is to be regarded as model-relative procedures, which possess an identity and natural structure that are both representation-independent. The crucial implication in my opinion is a conceptual split between 'algorithms' and 'computations'. Understood as the actual physical processes carried out by an agent (human, machine or otherwise), computations depend for their exact sequence of steps on the representations of the entities the agent operates upon. Thus, procedures that do not rely in any kind of way on the form of the representational entities operated upon are, strictly speaking, *not* computations.[52] I would then suggest that algorithms$_{\mathcal{A}}$ be seen as collections of singled out operations that can

[50] And even when representational concerns do appear in numerical analysis, they are are very different from those in TTE, for they relate to particular floating-point systems with specific precision, unit round-offs, etc.

[51] We know, for example, that TMs operating on decimal or binary notations give rise to reasonable (first-machine class) models of computation, while TMs operating on more cumbersome notations, such as stroke numerals, are too "weak" for this purpose; see van Emde Boas (1990). While this, by itself, is not necessarily in contrast with the approach of computable analysis, since natural and dyadic rationals can still be represented in binary without problems (for they have finite representations), it remains an interesting and unexplored issue how complexity is affected when we consider numbers whose binary representations require infinite strings. My conjecture is that this might force us to prefer an oracle-based formalization of Type-2 TMs (which involves only finite inputs and outputs) to the infinite-string approach discussed here and in Weihrauch (2000) (the two formalizations are equivalent from a computability point of view).

[52] Moschovakis has also expressed the view that we should not identify 'algorithms' with computational procedures (e.g., 1998, 4.3). My view, then, comes to parallel his own on this issue, although via a different route.

potentially be turned into actual computations. This approach raises a philosophical question and a further mathematical challenge.

The philosophical question has to do with an epistemological problem, arising from my proposal that, prior to and independently of their implementation, algorithms$_{\mathcal{A}}$ had better be seen as *de re* procedures.[53] Does this analysis entail the need for a view according to which our epistemological access to numbers is not mediated by their symbolic representations? I think that this is not so. As long as an algorithm$_{\mathcal{A}}$ does not describe a computation per se (but only a possibility thereof), it says nothing about our relation to the manipulated abstract entities, when we implement the algorithm. It only tells us that either an approximate or an exact computation is possible, *once* some appropriate mediating representation of the relevant domains has been adopted.

The mathematical challenge is that regimenting 'algorithms' as 'algorithms$_{\mathcal{A}}$' shifts the conceptual burden to the development of a precise definition of the 'implementation relation'. And I conjecture that such a definition would too face the challenges of trading off between on the one hand inclusiveness —i.e., articulating a notion of 'implementation' that subsumes both implementations on floating-point arithmetic (as in the practice of numerical analysis) *and* on any given formal model of computation (as in the practice of computer science)— and, on the other hand, relatability to well-entrenched results.[54]

These theoretical possibilities notwithstanding, the likeliest actual scenario, in my opinion, is that the community will keep working with both regimentations. This is not an uncommon situation in mathematical practice, where we have parallel formalizations of intuitive notions that differently satisfy trade-offs between generality, inclusiveness, and relatability to other domain-specific concepts and results (cf., e.g., the intuitive idea of 'measure' and different accepted formalizations of it as 'Lebesque measure' 'Borel measure', etc.). Accordingly, in complexity theory we need a notion of 'algorithm' that is specific enough to underpin a robust classification of problems into classes of computational difficulty, on the (constitutive) assumption of effective/symbolic computation. This need —which would exclude for example algorithms operating on infinite objects or formal models that cannot distinguish between the difficulties of multiplication and addition— would be satisfied by algorithms$_{\mathcal{S}}$ and respective explicata, such as TMs from the first machine class, and Type-2 Turing machines. But, at the same time, we want to study algorithms as freestanding mathematical entities (but cf. Dean 2016a, §6), and to that end we need increasingly generalized conceptions of them, as, e.g., operating over continuous spaces or over arbitrary structures as data sets. This need would be met by certain explicata of 'algorithms$_{\mathcal{A}}$', such as recursors, ASMs or BSS algorithms (and the latter have already been proved fruitful in supporting a rich area of complexity in numerical analysis).

[53] I thank an anonymous reviewer for bringing up this problem.

[54] For a discussion of such challenges for the *countable* case, see Dean (2016a). Yet, the uncountable case would presumably involve far more challenges, given the variation in how 'implementation' is understood in computer science and in numerical analysis (see fn.50).

Acknowledgments

I am deeply grateful to Alberto Naibo, Oron Shagrir and Walter Dean for many detailed comments and suggestions to improve this and earlier versions of this article. I am also thankful to the members of the Jerusalem Working Group in Logic (Rea Golan, Balthasar Grabmayr, Amit Karmon, David Kashtan, Aviv Keren, Naomi Korem, Ran Lanzet, Talia Leven, Amit Pinsker, Carl Posy, and Gil Sagi) for their comments when I presented an earlier draft. I have benefited greatly from continuous support from several colleagues in Jerusalem and in Paris, and from stimulating discussions, during postdoctoral fellowships in both places. I would also like to thank Wayne Myrvold, Michael Cuffaro, and Matthew Parker for helping me sharpen and clarify ideas when this work was still in a premature stage. Finally, I am indebted to various anonymous referees for some very useful comments and suggestions. This work has been partially supported by the ANR project *The Geometry of Algorithms* (ANR-20-CE27-0004).

References

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

Blum, L., F. Cucker, M. Shub, and S. Smale (1997). *Complexity and Real Computation*. Springer Science.

Boker, U. and N. Dershowitz (2010). Three paths to effectiveness. In A. Blass, N. Dershowitz, and W. Reisig (Eds.), *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, pp. 135–146. Berlin, Heidelberg: Springer Berlin Heidelberg.

Brattka, V. and P. Hertling (2021). *Handbook of Computability and Complexity in Analysis*. Springer.

Brattka, V., P. Hertling, and K. Weihrauch (2008). A tutorial on computable analysis. In S. B. Cooper, B. Löwe, and A. Sorbi (Eds.), *New Computational Paradigms: Changing Conceptions of What is Computable*, pp. 425–491. New York, NY: Springer New York.

Brauer, E. (2021). The dependence of computability on numerical notations. *Synthese 198*(11), 10485–10511.

Braverman, M. (2005). On the complexity of real functions. In *Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pp. 155–164.

Braverman, M. and S. Cook (2006). Computing over the reals: Foundations for scientific computing. *Notices of the AMS 53*(3), 318–329.

Carnap, R. (1962). *Logical Foundations of Probability* (2nd ed.). The University of Chicago Press.

Chabert, J.-L. (Ed.) (1999). *A History of Algorithms: From the Pebble to the Microchip*. Springer-Verlag.

Copeland, B. J. and D. Proudfoot (2010). Deviant encodings and Turing's analysis of computability. *Studies in History and Philosophy of Science Part A 41*(3), 247–252. Computation and cognitive science.

Copeland, B. J. and O. Shagrir (2019). The Church-Turing Thesis: logical limit or breachable barrier? *Communications of the ACM 62*(1), 66–74.

Corless, R. M. and N. Fillion (2013). *A Graduate Introduction to Numerical Methods: From the Viewpoint of Backward Error Analysis*. Springer-Verlag New York.

Cucker, F. (1999). Real computations with fake numbers. In J. Wiedermann, P. van Emde Boas, and M. Nielsen (Eds.), *Automata, Languages and Programming*, pp. 55–73. Springer Berlin Heidelberg.

Dean, W. (2016a). Algorithms and the mathematical foundations of computer science. In L. Horsten and P. Welch (Eds.), *Gödel's Disjunction: The scope and Limits of Mathematical Knowledge*, pp. 19–66. Oxford University Press.

Dean, W. (2016b). Squeezing feasibility. In A. Beckmann, L. Bienvenu, and N. Jonoska (Eds.), *Pursuit of the Universal: 12th Conference on Computability in Europe, CiE 2016, Paris, France, June 27 - July 1, 2016, Proceedings*, pp. 78–88.

Gauss, C. F. (1857). *Theory of the Motion of the Heavenly Bodies Moving about the Sun in Conic Sections: A Translation of Gauss's "Theoria Motus." With an Appendix*. Little, Brown.

Goldreich, O. (2008). *Computational Complexity: A Conceptual Perspective*. Cambridge University Press.

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

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

Gurevich, Y. (2015). Semantics-to-syntax analyses of algorithms. In G. Sommaruga and T. Strahm (Eds.), *Turing's Revolution: The Impact of His Ideas about Computability*, pp. 187–206. Cham: Springer International Publishing.

Gurevich, Y. (2019). Unconstrained Church-Turing thesis cannot possibly be true. *The Bulletin of European Association for Theoretical Computer Science 1*(127).

Hermes, H. (1969). *Enumerability, Decidability, Computability: An Introduction to the Theory of Recursive Functions* (2nd. ed.). Springer-Verlag.

Hertling, P. (1999). A real number structure that is effectively categorical. *Mathematical Logic Quarterly 45*(2), 147–182.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms* (2nd ed.). USA: Society for Industrial and Applied Mathematics.

Hilbert, D. (1926). Über das unendliche. *Mathematische Annalen 95*(1), 161–190. English translation "On the Infinite" in J. van Heijenoort (1967) *From Frege to Gödel* (transl. by S. Bauer-Mengelberg).

Kapantaïs, D. (2016). A refutation of the Church-Turing Thesis according to some interpretation of what the thesis says. In V. C. Müller (Ed.), *Computing and Philosophy: Selected Papers from IACAP 2014*, pp. 45–62. Springer (Synthese Library).

Kapantaïs, D. (2018). A counterexample to the Church-Turing Thesis as standardly interpreted. *Newsletter of the American Philosophical Association (Philosophy and Computers) 18*(1), 24–27.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Kolmogorov, A. N. and V. A. Uspenskii (1963). On the definition of an algorithm. *American Mathematical Society Translations 29*, 217–245. Translated from the Russian by Elliott Mendelson. Original publication 1958.

Lakatos, I. (1976). *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press.

Lewis, H. R. and C. H. Papadimitriou (1998). *Elements of the Theory of Computation* (2nd ed.). Prentice-Hall.

Makovec, D. and S. Shapiro (2019). *Friedrich Waismann: The Open Texture of Analytic Philosophy*. Palgrave-Macmillan.

Malc'ev, A. I. (1970). *Algorithms and Recursive Functions*. Wolters-Noordhoff Pub. Co Groningen. Translated from the Russian ed. by Leo F. Boron, with the collaboration of Luis E. Sanchis, John Stillwell and Kiyoshi Iseki.

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

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

Moschovakis, Y. N. (1998). On founding the theory of algorithms. In H. G. Dales and G. Oliveri (Eds.), *Truth in Mathematics*, pp. 71–104. Clarendon Press, Oxford.

Moschovakis, Y. N. (2001). What is an algorithm? In B. Engquist and W. Schmid (Eds.), *Mathematics Unlimited — 2001 and Beyond*, pp. 929–936. Springer.

Papayannopoulos, P. (2021). Unrealistic models for realistic computations: How idealisations help represent mathematical structures and found scientific computing. *Synthese 199*, 249–283.

Pégny, M. (2016). How to make a meaningful comparison of models: The Church–Turing thesis over the reals. *Minds and Machines 26*(4), 359–388.

Quinon, P. (2018). A taxonomy of deviant encodings. In F. Manea, R. G. Miller, and D. Nowotka (Eds.), *Sailing Routes in the World of Computation*, pp. 338–348. Cham: Springer International Publishing.

Rescorla, M. (2007). Church's thesis and the conceptual analysis of computability. *Notre Dame Journal of Formal Logic 48*(2), 253 – 280.

Rogers, Jr., H. (1987). *Theory of Recursive Functions and Effective Computability*. Cambridge, MA, USA: MIT Press.

Shagrir, O. (2022). *The Nature of Physical Computation*. Oxford University Press.

Shapiro, S. (1982). Acceptable notation. *Notre Dame Journal of Formal Logic 23*(1), 14 – 20.

Shapiro, S. (2006). Computability, proof, and open-texture. In A. Olszewski, J. Wolenski, and R. Janusz (Eds.), *Church's Thesis After 70 Years*, pp. 420–455. Ontos Verlag.

Shapiro, S. (2013). The open texture of computability. In B. J. Copeland, C. J. Posy, and O. Shagrir (Eds.), *Computability: Turing, Gödel, Church, and Beyond*, pp. 153–181. The MIT Press.

Shapiro, S. (2015). Proving things about the informal. In G. Sommaruga and T. Strahm (Eds.), *Turing's Revolution*, pp. 283–296. Springer.

Shapiro, S. (2017). Computing with numbers and other non-syntactic things: De re knowledge of abstract objects. *Philosophia Mathematica 25*(2), 268–281.

Shapiro, S., E. Snyder, and R. Samuels (2022). Computability, notation, and *de re* knowledge of numbers. *Philosophies 7*(1).

Sipser, M. (2013). *Introduction to the Theory of Computation* (3rd. ed.). Cengage Learning.

Smale, S. (1990). Some remarks on the foundations of numerical analysis. *SIAM Rev. 32*(2), 211–220.

Smith, P. (1998). *Explaining Chaos*. Cambridge University Press.

Smith, P. (2013). *An Introduction to Gödel's Theorems* (2nd. ed.). Cambridge University Press.

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

Turing, A. M. (1948). Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics 1*(1), 287–308.

Uspensky, V. and A. Semenov (1993). *Algorithms: Main Ideas and Applications*, Volume 251. Kluwer Academic Publishers. Translated from the Russian by A. Shen.

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

von Neumann, J. and H. H. Goldstine (1947). Numerical inverting of matrices of high order. *Bull. Amer. Math. Soc. 53*(11), 1021–1099.

Waismann, F. (1945). Verifiability. In *Proceedings of the Aristotelian Society*, Volume 19.

Wang, H. (1997). *A Logical Journey: From Gödel to Philosophy*. The MIT Press.

Weihrauch, K. (2000). *Computable Analysis: An Introduction*. Springer Science.