ALGORITHMS AND THE MATHEMATICAL FOUNDATIONS OF COMPUTER SCIENCE

WALTER DEAN

1. Introduction

The goal of this paper is to bring to the attention of philosophers of mathematics the concept of *algorithm* as it is studied in contemporary theoretical computer science, and at the same time address several foundational questions about the role this notion plays in mathematical practice. In the most general sense, an algorithm is simply a procedure for achieving a particular mathematical end – paradigmatically computing the values of a function, or deciding whether a given mathematical object has a particular property. Most readers will be familiar with a variety of such procedures – e.g. carry addition, long division, Euclid's greatest common divisor algorithm – and will also be able to provide a characterization of the features in virtue of which we traditionally classify these methods as practical aids to calculation.

Methods such as these form an important part of the intellectual heritage of mathematics in several respects. For instance, methods for calculating quantities such as areas, inverses, powers, and roots appeared early in the Babylonian, Sumerian, and Egyptian mathematical traditions first being recorded around 2500 BCE. In the early middle ages, Arabic mathematicians developed a variety of procedures for performing arithmetic and algebraic operations which exploit the features of positional notation systems. The greater efficiency of these algorithms – some of which are the ancestors of algorithms still taught to school children today – over earlier methods for calculating with Roman numerals is often cited as having been the determining factor in the acceptance of Hindu-Arabic numerals in medieval Europe.

In the modern era the discovery of algorithms of practical import has often gone hand in hand with significant mathematical discoveries. For instance, Gauss first noted that both long division and Euclid's algorithm can be generalized to the ring $K[X]$ of polynomials over a field K. Together with the procedure now known as Gaussian elimination for solving systems of linear equations, these algorithms figured prominently in the proofs of a variety of theorems of 19th century algebra – e.g. Gordan's demonstration that the ring of invariants of binary forms of fixed degree is finitely generated, Hilbert's Nullstensatz, and Strum's Theorem. An important contemporary example in the same tradition is Buchberger's algorithm

Forthcoming in *The limits of mathematical knowledge*, Oxford University Press, 2016. L. Horsten and P. Welch, eds.

[9] for computing Gröbner bases.[1] Many similar examples can be cited in linear algebra (e.g. the Gram-Schmidt algorithm), analysis (e.g. the Newton-Raphson method), and graph theory (e.g. Krusal's algorithm).

In the face of such examples, it seems reasonable to propose that the notion of algorithm deserves a place alongside concepts such as number, set, and function as an important substrate of contemporary mathematics. This role is partially accounted for in light of the well-known analyses of computability undertaken during the 1930s. For note that all of the methods mentioned thus far are paradigmatically *effective* – i.e. they are finitely specifiable in terms of operations (e.g. adding or subtracting natural numbers, comparing the order of leading terms of polynomials, etc.) which may be carried out using finite resources by a mechanical computing agen. Such characteristics inform the conception of an *effectively computable function* – i.e. one computable by an effective procedure – which was analyzed in distinct but extensionally equivalent ways by Church, Turing, Post, Kleene, and Gödel.

Reflection on these analyses led via a well known route to the framing of Church's Thesis:

(CT) $f : \mathbb{N}^k \to \mathbb{N}$ is effectively computable if and only if $f(\vec{x})$ is recursive.

Note, however, that an effectively computable function is standardly understood to be one whose values can be computed by an algorithm. Church's Thesis can thus also be formulated as follows:

(CT_a) There exists an algorithm for computing $f : \mathbb{N}^k \to \mathbb{N}$ if and only if $f(\vec{x})$ is recursive.

CT_a is indeed a significant contribution to our understanding of the notion of algorithm. In particular, it enables us to analyze the truth conditions of statements such as

(1) There is no algorithm for determining whether an arbitrary formula of first-order logic is valid.

in terms of a proposition which quantifies not over algorithms but rather over the members of some mathematically well defined class (e.g. that of Turing machines). CT_a thus provides a means of assigning significance to the formal results which Church and Turing demonstrated in the course of answering Hilbert's *Entscheidungsproblem* in the negative (which is what is reported by (1)). And it has a similar effect for other formal undecidability results for problems such as the word problem for semigroups or Hilbert's Tenth Problem.

Taken on its own, however, CT_a provides little direct insight into how the notion of algorithm is understood in contemporary computer science. For note that while several of the examples mentioned above suggest that mathematical advances have often been tied to the discovery of new algorithms, the statements

[1] Buchberger's algorithm can be understood as generalizing both Euclid's algorithm and Gaussian elimination. It finds a variety of applications in both pure and applied mathematics – cf., e.g., [15].

of the associated mathematical *theorems* rarely mention algorithms explicitly.[2] Thus it might appear that the task of formalizing classical mathematics (say within axiomatic set theory) does not require that we accord algorithms the status of freestanding objects or that we provide a rigorous reconstruction of the means by which we apply or analyze them in practice.

In computer science this order of inquiry is often inverted. For instance, in fields like complexity theory and algorithmic analysis, individual algorithms are treated as the presumptive objects of study and mathematical methods are then used to investigate their properties. This emphasis is reflected, for instance, in the practice of referring to individual algorithms by what appear to be grammatical proper names (e.g. Euclid's algorithm, MergeSort, etc.). Moreover, results in algorithmic analysis are often reported by predicating computational properties directly of individual procedures by using these names (e.g. "The AKS primality algorithm has polynomial running time") or by quantifying over classes of procedures (e.g. "There exists a polynomial time primality algorithm", "There is no linear time comparison sorting algorithm").

As the use of such language bears many of the hallmarks which philosophers have traditionally associated with ontological commitment, the practice of computer science thus raises a variety of questions about the status of algorithms which have been overlooked by philosophers of mathematics. Perhaps most prominent among these is the following:

(Q1) Are individual algorithms properly regarded as objects? If so, are they abstract, concrete, or somehow intermediate between these possibilities? How can we account for our apparent ability to make reference to and prove both singular and general propositions about them?

Questions of type (Q1) – which are of an overtly ontological nature – can be contrasted with epistemological concerns about the use of algorithms in mathematical practice such as the following:

(Q2) How can we justify the use of computational methods in the derivation of mathematical results? In particular, why ought we to accept a calculation carried out by using an algorithm as an adequate demonstration of a mathematical proposition?

Questions in the vicinity of (Q2) have attracted more attention within philosophy of mathematics than (Q1) largely because of the difficulties which are thought to arise in accounting for the status of proofs – most famously of the Four Color Theorem [78] – which have substantial computational components.

[2] For instance, consider Sturm's Theorem – "The number of distinct real roots of a square-free polynomial $p(x)$ located in the half-open interval $(a, b]$ is given by $\sigma(a) - \sigma(b)$ where $\sigma(x)$ is the number of sign changes in the Sturm sequences $p_0(x), p_1, (x), \ldots, p_m(x)$ for $p(x)$." Although the classical proof of this result consists largely in specifying an algorithm for computing the Sturm's sequence for $p(x)$ (and thereby the value of $\sigma(a) - \sigma(b)$) and proving it correct, the statement of the theorem itself makes no mention of this.

I will discuss such examples further in §2. Therein I will also argue that they lead naturally to the formulation of the following thesis about how questions of type (Q1) should be answered:

(A) Algorithms are mathematical objects.

(A) answers (Q1) by proposing that algorithms are no different in kind than the sorts of objects (e.g. numbers, sets, groups, matrices, graphs, etc.) studied in recognized branches of classical mathematics. Such a view also provides a convenient answer to questions of sort (Q2): if algorithms are just another type of mathematical object, then the use of an algorithm to perform a lengthy calculation which may be required in the course of a proof can be justified by a mathematical proof taking the algorithm as its subject that the algorithm is *correct* (i.e. that it computes the function that it is claimed to).

I will refer to the view expressed by (A) as *algorithmic realism*.[3] Although it is often not identified as a thesis requiring an explicit statement and defense, I will argue in §3 that (A) is presupposed throughout much of contemporary theoretical computer science.[4] Variants of (A) are also implicit in the writings of virtually all theorists who have considered the notion of algorithm in a foundational setting. This most prominently includes Yiannis Moschovakis and Yuri Gurevich, each of whom have set out theories wherein in algorithms are explicitly identified with certain classes of mathematical objects (respectively in [54], [56], [57] and [36], [37], [35]). Other formulations of algorithmic realism can also be found in the writing of many other theorists who have addressed the nature of algorithms in relation to other concepts studied in computer science – e.g. Gödel [32], Rogers [66], Kreisel [48], Milner [51], Knuth [41], Odifreddi [59], Foster [23], and Yanofsky [82] – many of whose views will be discussed (at least in passing) below.

The primary goal of this paper will be to address the question of whether algorithmic realism can be sustained as a foundational thesis about the nature of

[3] Of course one might also think that a view deserving the name "realism" about algorithms ought to be involved with providing a *non-reductive* account of our discourse about them – i.e. one which identifies algorithms not with mathematical objects as traditionally conceived, but rather with a distinctive class of entities which might (or instance) possess genuine spatio-temporal properties. Although such a view – which I will discuss further in §4 under the name *direct algorithmic realism* – does not appear to be widely held in mainstream computer science, it bears an obvious affinity to intuitionistic or constructive mathematics. For instance, the understanding of mathematical objects as idealized mental constructions within intuitionism may itself be taken to form a kind of converse to (A) – i.e. rather than seeking to identify procedural entities with mathematical ones, it seeks to identify mathematical entities with procedural ones. Unlike intuitionism, however, algorithmic realism is not intended to be a foundational thesis about mathematics as a whole, but rather a thesis about the relationship of procedural discourse – especially as it figures in theoretical computer science – to mathematics.

[4] Donald Knuth – who is generally credited with founding (as well as naming) the field now known as *algorithmic analysis* – has been one of the most outspoken proponents of algorithmic realism. One of his most explicit formulations of this view is as follows: "Algorithms are concepts which have existence apart from any programming language ... I believe algorithms were present long before Turing et al. formulated them, just as the concept of the number 'two' was in existence long before the writers of first grade textbooks and other mathematical logicians gave it a certain precise definition." [45], p. 654.

algorithms in light of the details of our mathematical and computational practices involving them. Despite the realistic character of much of our discourse about algorithms, I will ultimately argue that it *cannot be so sustained.* The central reason I will offer for this is that our practices simply do not exert sufficient pressure on the use of the expressions we employ to refer to algorithms to allow us to assign them mathematical denotations in a non-arbitrary manner. In fact, I will suggest that the fundamental problem confronting someone wishing to defend algorithmic realism is the familiar one of showing how to associate algorithms with mathematical objects such that the identified objects represent all and only the computational properties which we associate with them directly in the course of our practices.

I will refer to variants of this view on which the class of mathematical objects with which algorithms are identified are members of one of the conventional models of computation studied in computability theory – e.g. Turing machines, partial recursive function definitions, lambda terms – as *Strong Church's Thesis.*[5] When seen simultaneously through the lens of computer science and philosophy of mathematics, this view faces a strong *prima facie* challenge. For note that if we propose to identify, say, Euclid's algorithm with a *particular* mathematical object M (say a specific Turing machine), we thereby equip the former with all of the properties – e.g. having a particular number of states, an exact as opposed to asymptotic running time, etc. – which are possessed by the latter. As we will see below, however, it is out of keeping with the methodologies of algorithmic analysis and complexity theory to think that such properties are appropriately attributed to individual algorithms as opposed, e.g., to their implementations with respect to a given model of computation or their expression in a given programming language.

The adoption of Strong Church's Thesis thus faces a potential objection on the basis of what we might call the *what-numbers-could-not-be problem.* In its original form, this is the charge famously formulated by Benacerraf [5] that a successful reduction of number theory to set theory must provide not only a means of identifying numbers with sets which preserves the truth of our chosen arithmetical axioms, but also an account of what makes a particular means of making such an identification correct relative to other identifications which also preserve the truth of the axioms. For to take the familiar example, were we to propose to identify the natural numbers with the finite von Neumann ordinals – as opposed to, say, the finite Zermelo ordinals – we would be faced with the seeming unanswerable question of why 2 has the properties of $\{\varnothing, \{\varnothing\}\}$ (e.g. having two members) as opposed to those of $\{\{\varnothing\}\}$ (e.g. having a single member).

Perhaps for this reason, the contemporary consensus appears to be that individual algorithms should be identified not with particular instances of models of computation, but rather to equivalence classes of such items determined by an appropriate notion of "computational equivalence" defined over an appropriate

[5] Inchoate versions of Strong Church's Thesis can be found already in Post [64] and in Kolmogorov and Uspensky [46]. More explicit formulations are given by Knuth [41], Gandy [28], Sieg and Byrnes [74] and Dershowitz and Gurevich [18].

class $\mathfrak{C}$ of mathematical objects. I will refer to this as the *algorithms-as-abstracts* view.[6] The most popular versions of this view take $\mathfrak{C}$ to coincide with the class of machine $\mathfrak{M}$ of some (sufficiently generalized) model of computation or the class of programs $\mathfrak{P}$ over a (sufficiently generalized) programming language. These proposals respectively lead to what I will refer to as the *machine-based* and *program-based* variants of the algorithms-as-abstracts view. In order to develop this view it is also necessary to propose an equivalence relation $\sim$ by which these classes can be factored into the classes which its proponents hold to be algorithms. As I will discuss further below, standard choices for $\sim$ on the machine-based and program-based variants are respectively *mutual simulation* (which I will denote by $\approx$) and various technical definitions of *program synonymy* (which I will denote by $\simeq$).

Once appropriate definitions for these parameters have been fixed, the proponents of the algorithms-as-abstracts view can then be understood to propose that the following implicit definitions provide an analysis of statements about algorithmic identity and non-identity:

(MP) the algorithm implemented by machine M_1 = the algorithm implemented by machine M_2 if and only if $M_1 \simeq M_2$

(PP) the algorithm expressed by program Π_1 = the algorithm expressed by program Π_2 if and only if $\Pi_1 \approx \Pi_2$

Putting aside for the moment how a proponent of the algorithms-as-abstracts view might go about defining $\mathfrak{M}, \mathfrak{P}, \simeq$ and $\approx$, it is also evident that the schema (MP) and (PP) have the form of *abstraction principles* of the sort which have been widely discussed in the relation to Frege's [24] proposed analysis of natural numbers as equivalence classes of finite sets with respect to the relation of equicardinality. It might thus at first appear that it is open to proponents of the algorithms-as-abstracts view to propose that these schemas provide a means by which we can understand the status of algorithms as abstract objects in something like the manner in which neo-logicist philosophers of mathematics such as Wright and Hale [81], [38] propose that the adoption of Hume's Principle (i.e. the abstraction principle *the number of* Fs = *the number of* Gs just in case the Fs and Gs are equinumerous) can be understood to introduce the natural numbers as a new class of "logical objects".

But despite the outward affinity between the two proposals, the problems which must be confronted to develop the algorithms-as-abstract view are of a different character than those which have traditionally been taken to confront neo-logicism. In particular, the neo-logicist project is traditionally understood to be premised on the claim that Hume's Principle should be understood as a conceptual or analytic truth about our understanding of the concept *natural number*. In order to defend this view, it is usually taken to be incumbent on the neo-logicist not

[6] Inchoate forms of the algorithms-as-abstracts view can be found in Rogers [66] and Kreisel [48], whereas more refined versions have been developed by Milner [51], Moschovakis (e.g.) [56] and Yanofsky [82].

only to defend this claim but also to respond to other challenges which are often pressed against the neo-logicist program.[7]

As we will see below, however, a significant problem which the algorithms-as-abstracts view must face is that it is difficult to find non-trivial instances in which the schema (MP) or (PP) are applied in practice against which we can test our intuitions about their potential analyticity. In fact, the practice of fields like algorithmic analysis and complexity theory appear to leave not only the choice of the classes $\mathfrak{M}$ and $\mathfrak{P}$, but also the choice of the equivalence relations $\simeq$ and $\approx$ highly unconstrained. Proponents of the algorithms-as-abstracts view thus face the initial challenge not only of defining what is meant by "computational equivalence", but also of characterizing the class of objects over which it is defined. And perhaps more significantly, for such a proposal in this family to serve as a convincing *analysis* of what we mean when we speak of the notion of algorithm in the course of our informal practices, it would presumably have to be informed by the same sort of linguistic, conceptual, and technical investigation to which Frege [24], [25] famously subjected the notion of natural number. There is thus much work to be done even before analogues of the traditional challenges to neo-logicism can be raised for the algorithms-as-abstracts view.

These points notwithstanding, the algorithms-as-abstracts view is currently the best developed strategy for addressing the status of algorithmic realism and questions of type (Q1) more generally. In order to understand the challenges facing this view it will be useful to first consider in greater detail the factors which motivate algorithmic realism and also the practices of the subfields of theoretical computer science which bear on the details of our contemporary understanding of algorithms. These will be the respective purposes of §2 and §3. In §4, I will further develop several of the strategies for developing algorithmic realism just surveyed so as to provide a more systematic case that the machine based variant of the algorithms-as-abstracts view is currently the most proposing approach for defending this view. In §5, I will then consider the relation of mutual simulation which several theorists have proposed can play the part of $\simeq$. Upon concluding that it is not possible to provide a definition of this notion which serves needs of algorithmic realism, in §6 I will then discuss this conclusion in light of Moschovakis's and Gurevich's theories of algorithms, as well as offering some final thoughts on the significance of intensional identity statements in mathematical practice.

2. Motivating algorithmic realism

The purpose of this section will be to set out a series of considerations which serve to motivate the consensus view that algorithmic realism is a correct means of responding to questions of type (Q1). As mentioned above, questions of this sort have received considerably less attention in philosophy of mathematics than those of type (Q2). For although there has been relatively little philosophical engagement with theoretical computer science to date, what interaction there

[7] E.g. the Julius Caesar or "bad company" objects as discussed in [38].

has been has focused around evaluating the epistemic questions raised by mathematical arguments which involve lengthy algorithmic calculations. The best known example of this sort is, of course, the so-called "computer proof" which was employed by Appel and Hanken [2], [3] in their original demonstration of the Four Color Theorem.[8] However, in order to illustrate how attempts to answer questions of type (Q2) lead inevitably to consideration of questions of type (Q1), it will be useful to examine not this example but rather the use of algorithmic methods to solve a more commonplace problem: given a natural number n, is n prime?[9]

Until the late 19th century, all known methods for solving this problem appear to have been based on variants of the "naive" method of trial division – i.e. in order to determine if n is prime, show $p_1 \not| n, p_2 \not| n, \ldots$ for all primes $p_i \leqslant \sqrt{n}$. But it was also recognized that this method is both laborious and unreliable due to the large number of divisions it required. The first significant advance in primality testing was the development of a procedure by the French mathematician Édouard Lucas who showed how it was possible to determine whether a Mersenne number (i.e a number of the form $m_p = 2^p - 1$ for p prime) is prime without checking it for divisibility by a large number of smaller primes.

The basis of Lucas's method is a number theoretic lemma whose general form is now known as the *Lucas-Lehmer primality test*:

(2) For p an odd prime, $2^p - 1$ is prime if and only if $r_{p-2} \equiv 0 \mod 2^p - 1$ where $r_0 = 4$ and $r_{i+1} = r_i^2 - 2$.

This statement implies that in order to determine whether $2^p - 1$ is prime it suffices to carry out the following procedure (which I will refer to here as *Lucas's algorithm*): compute the sequence of numbers $r_0, r_1, \ldots, r_{p-2}$ and then check if $2^p - 1$ divides r_{p-2}.

In 1876 Lucas employed this procedure to show that the 39 digit number m_{127} was prime. This required that he compute the values of 125 products on numbers up to 39 decimal digits in length. While this is still an arduous task to perform

[8] The version of the proof which was originally published included an exhaustive case analysis conducted by computer to check whether a collection of almost 2000 graphs possessed a property known as *reducibility*. Although this number has been reduced in subsequent proofs, all known demonstrations of the Four Color Theorem rely on verifying a number of decidable graph theoretic statements which is greater than the number which can be feasibly tested (or even surveyed) by a human mathematician. On this basis Tymoczko[78] famously argued that the Appel and Hanken demonstration is not a mathematical proof in the traditional sense as its correctness relies on the empirical (and hence *a posteriori*) assumption that the computing device which is employed to carry out the relevant calculations has operated correctly.

[9] The of primality testing was described by Gauss [30] (p. 329) as "the most important and useful in arithmetic." Anticipating later developments in complexity theory, he went on to say that "nevertheless we must confess that all methods that have been proposed thus far are either restricted to very special cases or are so laborious and difficult that even for numbers that do not exceed the limits of tables constructed by estimable men, they try the patience of even the practiced calculator." The significance of primality testing – and in particular of the Lucas-Lehmer test described below – to the status of the debate about "computer proofs" was first observed by Detlfesen and Luker [19].

without mechanical computing equipment, it should be compared with the fact that more than 1.9×10^{36} divisions would be required to test the primality of m_{127} using the trial division algorithm. The contrast between trial division and Lucas's algorithm thus illustrates the reason why we are often concerned with the existence of an algorithm for solving a given mathematical problem – i.e. by exploiting a "clever" or indirect method, algorithms often allow us to solve instances of problems of antecedent interest which would be infeasible for us to solve by direct or "naive" methods alone.[10]

The question arises, however, why we ought to accept Lucas's calculation as an admissible mathematical demonstration of the proposition expressed by "m_{127} is prime". For even if we put aside the possibility that Lucas made a mistake during his calculation, we must presumably also that demonstrate Lucas's algorithm accurately decides the primality of $2^p - 1$ before we are justified in accepting that a computation carried out by this algorithm is sufficient to demonstrate that m_p is prime. In order to understand the significance of such a requirement, note that the specification of the algorithm does not bear a transparent relationship to the traditional definition of primality (i.e. n is prime just in case n is divisible by only 1 and itself). A computation carried out by the algorithm will thus bear little resemblance to an execution of the trial division algorithm described above and even less resemblance to a "canonical" deductive proof that m_{127} is prime – i.e. one whose structure mirrors the logical form of the proposition $\forall x[x|m_{127} \rightarrow (x = 1 \vee x = m_{127})])$. When viewed in isolation, we would thus have little reason to accept such a calculation as a demonstration that m_{127} is prime.

The desired connection between this calculation and the definition of primality is, of course, provided by the Lucas-Lehmer Test. This result allows us to see informally why repeatedly carrying out the operation $s \mapsto (s^2 - 2)$ by which the value associated with the variable s is replaced by that of $s^2 - 2$ results in a final value of s such that $s \mod 2^p - 1 = 0$ just in case $2^p - 1$ is prime. Note, however, that in order to turn this observation into what we normally regard as a mathematical proof, some explanation must presumably be given of the meaning of expressions such as *repeatedly carrying out* or *replacing* one value of a variable with another.

A related question is how we are to understand such language if we wish to give a purely mathematical proof that a given algorithm computes a function or decides a predicate which has an antecedently given mathematical definition. For note that while it is standard to think of mathematical objects as static, the use of operational terms in the specification of algorithms reflects our understanding

[10] Note this is often true regardless of whether the algorithm is carried out by hand or with a computer. For even using a computer, our ability to practically apply an algorithm to solve particular instances of a problem will still be determined by its running time complexity. As such, Lucas's algorithm remains a significant improvement over the naive primality algorithm for testing Mersenne numbers even when they are both carried out mechanically rather than with paper and pencil calculation.

that to execute an algorithm is to carry out a sequence of operations which are *ordered in time*.[11]

In the current case, however, a resolution to this apparent incongruity still appears near at hand: in order to prove that Lucas's algorithm is correct, it suffices to construct a formal representation of the procedure – call it ϕ_{LUCAS} – which relates the value associated with the variable s at the $i+1$st stage in its operation to the value of the term r_i occurring in (2). This may be accomplished by specifying a mathematical structure ϕ_{LUCAS} sometimes known as an *iterator*.[12]. ϕ_{LUCAS} may be taken to consist of a set of *computational states* $St \subseteq \mathbb{N} \times \mathbb{N}$ whose first member keeps track of the index i and whose second member keeps track of the value of s_i together with a *transition function* $\sigma : St \rightarrow St$ such that $\sigma(\langle x, y\rangle) = \langle x-1, y^2-2\rangle$. We may then define $App(\phi_{\text{LUCAS}}, p)$ – i.e. the result of applying LUCAS to p – to be $\mathtt{1}$ if upon iterating σ we obtain a sequence of states of the form $s_0 \equiv \langle p, 4\rangle, s_1 = \sigma(s_0), \ldots, s_{i+1} = \sigma(s_i), \ldots, \sigma_{p-1} = \sigma(s_{p-1}) \equiv \langle 2, y\rangle$ and $y = 0$ and $\mathtt{0}$ otherwise.[13]

Using (2) it is now straightforward to prove a statement expressing the correctness of LUCAS in the following form:

Proposition 2.1. For all odd primes p, $App(\phi_{\text{LUCAS}}, p) = \mathtt{1}$ if and only if $2^p - 1$ is prime.

Unlike the observations about the operation of Lucas's algorithm recorded above, Proposition 2.1 is a result involving only mathematical definitions and structures (i.e. natural numbers, finite sequences of ordered pairs, etc.). As such, it admits to a standard proof by mathematical induction which does not require us to reason informally about notions like "repeatedly carrying out" an operation.

Proposition 2.1 reports what computer scientists call the *correctness* of Lucas's algorithm – i.e. that for all odd primes p the algorithm outputs $\mathtt{1}$ or $\mathtt{0}$ according to whether $2^p - 1$ is prime. The derivation of Proposition 2.1 from the Lucas-Lehmer test is sufficiently straightforward that this result might at first appear trivial. In the general case, however, correctness proofs for more complex

[11] Observations about the tension between the abstract subject matter of mathematics and the use of dynamic language in the formulation of mathematical proofs are at least as old as Plato's critique of the geometers in the *Republic* VII.1 [62]. The resolution typically favored by proponents of classical mathematics is to interpret talk of constructions (e.g. of a line bisecting an angle, the minimal closure of a set) as a heuristic way of expressing atemporal existence assertions which may in some cases lack constructive proofs – c.f., e.g., [72]. But as the methods which will be at issue in this paper are all effective, the issue at stake will not be whether dynamic discourse involving algorithms has genuine constructive content. Rather it is whether classical mathematics possesses the resources to interpret the operational discourse itself in a manner which is faithful to our understanding of the mode of operation of the individual algorithms in question and, more broadly, of our means of classifying them according to their computational properties.

[12] The term "iterator" to describe such a structure owes to Moschovakis [56]. However, similar models have been independently proposed many other theorists – e.g. Kolmogorov and Uspensky [47], Knuth [41], and Gandy [28].

[13] For instance the computation induced by ϕ_{LUCAS} on the input 5 is the sequence of states $\langle 5, 4\rangle, \langle 4, 17\rangle, \langle 3, 8\rangle, \langle 2, 0\rangle$ and $App(\phi_{\text{LUCAS}}, 5) = \mathtt{1}$.

algorithms may require sophisticated mathematical arguments themselves. But what is most significant in the current case is not the difficulty of demonstrating correctness, but rather the proposal that algorithms admit to mathematical proof of correctness in the first place.

For as we have just seen, the availability of such a proof for an algorithm A appears to presuppose the existence of a mathematical structure such as ϕ_A which provides a precise representation of what we might informally describe as A's "mode of operation". Once we have specified A informally, it is also often a routine exercise in formalization to construct ϕ_A. But even after we have provided such a representation, the question remains what relationship such a structure bears to the algorithm A in the sense we originally understood it relative to an informal specification similar to that by which Lucas's algorithm was introduced above.

One obvious proposal is that the relationship between an algorithm and its formal representation as a structure such as an iterator is sufficiently direct and systematic that the former may simply be *identified* with the latter. A consequence of this is that the procedure we started out calling Lucas's algorithm simply *is* the structure φ_{LUCAS} and similarly that all of the other algorithms we have previously identified in our mathematical practices may be identified with such structures. This is not to deny that our initial apprehension of how such procedures operate is often grounded in the sort of informal description provided for Lucas's algorithm above or that such descriptions are sufficiently precise to guide us in their execution. Rather the import of both the current example as well as several others considered below is to suggest that nothing of mathematical import is lost when we move from such informal descriptions to the sort of formal representation of a procedure which is required to support a correctness proof.

Our apparent ability to make such identifications in a manner which preserves the details of how we apply and reason about algorithms in practice appears to provide the basis of the consensus that algorithmic realism is correct. In particular, the need to provide correctness proofs before we can assimilate algorithmic calculations to traditional mathematical proof coupled with the apparent consensus that we are indeed justified in believing the outcomes of algorithmic calculations on the basis of something akin to proof appears to provide a strong *prima facie* case in favor of this view. The goal of §3-§5 will be to show that the practice of contemporary computer science exerts sufficiently many other pressures on the form which an adequate foundational theory of algorithms must take so as to render algorithmic realism untenable. The goal of §6 will then be to provide an alternative reconstruction of the justification of the use of computational methods in mathematics.

3. Algorithms in theoretical computer science

The goal of this section will be to provide a concise account of how algorithms are treated within theoretical computer science. This will inform the detailed considerations which are required to evaluate algorithmic realism relative to our contemporary practices. In so doing we face the challenge that despite its relative youth as a subject, theoretical computer science has spawned subfields in which algorithms are treated, each with its own motivating questions and methodology. Of these, considerations originating in complexity theory and algorithmic analysis (which will be considered in this section), and programming language semantics (which will be considered in §4) will be most relevant below.[14]

An elementary (but deceptively simple) class of examples is provided by the study of procedures for sorting a finite array B of numbers or other items linearly ordered by a relation $\leqslant$.[15] Consider, for instance, the following passage from a popular algorithmic analysis textbook:

> In this chapter, we introduce another sorting algorithm: HeapSort. Like MergeSort, but unlike insertion sort, HeapSort's running time is $O(n \log_2(n))$. Like InsertionSort, but unlike MergeSort, HeapSort sorts in place: only a constant number of array elements are stored outside the input array at any time. Thus, HeapSort combines the better attributes of the two sorting algorithms we have already discussed. [14], p. 151

This passage exemplifies how expressions like HeapSort, MergeSort, and InsertionSort are used to denote individual methods for computing the sorting function – i.e. the the function $\mathrm{sort}(x)$ which on input array B returns a permutation B' of B such that $B'[0] \leqslant B'[1] \leqslant \ldots \leqslant B'[n]$. A notable feature of the discourse of algorithmic analysis and related subjects is that terms play the apparent grammatical role of *proper names* – i.e. linguistically primitive expressions which function as singular terms for purpose of predication and formulation of statements of identity and non-identity. I will refer to such as expressions as *algorithmic names*.

A question close to the heart of algorithmic realism is how we should understand the reference of expressions likeHeapSort or MergeSort. As a first step toward addressing this, note that algorithmic names are often introduced along with informal mathematical descriptions as exemplified by the following specification of InsertionSort:

> [C]onsider the elements one at a time, inserting each in its proper place among those already considered (keeping them sorted). The element being considered is inserted merely by moving larger elements one position to the right, then inserting the element into the vacated position. [71], p. 95-96

Although such informal descriptions are sometimes taken to be sufficient for purposes of proving that an algorithm is correct or analyzing its computational complexity (a task I will discuss further in a moment), it is also conventional to supplement them with more regimented formulations in an idiom known as *pseudocode*. A specification of InsertionSort in this manner might take the following form:[16]

[14] Standard references for these subjects are as follows: complexity theory [29], [60], [20]; algorithmic analysis [41], [42], [43], [71], [14]); semantics of programming languages [75], [58], [52].

[15] For a history of such methods and their role in the development of algorithmic analysis, see [41].

```
InsertionSort($B$)
for $j \leftarrow 2$ to $|B|$
    do $m \leftarrow B[j]$
       $i \leftarrow j - 1$
       while $i > 0$ and $m \leqslant B[j]$
           do $B[i+1] \leftarrow B[i]$
              $i \leftarrow i - 1$
       $B[i+1] \leftarrow m$
```

Expressions like Lucas, InsertionSort, or Euclid (i.e. Euclid's algorithm) are introduced in much the same way that names for theorems and definitions are used in conventional mathematical texts (e.g. Desargue's Throrem, the Snake Lemma, the Riemann Integral, etc.) – i.e. as means of facilitating reference to prior complex definitions or results. Within computer science, however, algorithmic names also commonly appear as the grammatical subjects of sentences reporting that individual algorithms have particular computational properties – e.g.

(3) a) Euclid computes the function $\gcd(x, y)$.
b) InsertionSort has running time complexity $O(n^2)$.
c) MergeSort has faster running time complexity than InsertSort.

The frequency with which such statements appear in computer science textbooks and other sources underscores the fact that what is often of interest in fields like algorithmic analysis and complexity theory is not merely the discovery of algorithms, but rather the use of mathematical methods to show that certain properties and relations hold of procedures which have already been introduced.

In addition to the property of correctness with respect to a given mathematically defined function (e.g. as reported by (3a)), of paramount concern in computer science are the sorts of complexity theoretic properties reported by (3b,c). For it is the attribution of such properties by which the efficiency of algorithms for solving particular mathematical problems are measured and compared. The manner in such attributions are treated in algorithmic analysis and complexity theory will play a significant role in the evaluation of algorithmic realism developed below. It will thus be useful to offer the following brief account of how such statements are demonstrated.

Informally speaking, the *running time complexity* of an algorithm A on an input x is the number of basic computational steps which are required for A to halt and return a output when applied to the input x measured as a function of some standard measure $|x| = n$ of the size of x (e.g. the length of its binary

[16] As is evident from this example, psuedocode employs a combination of natural language and formal programming constructions. Many of the latter express instructions which are awkward to describe in natural language – e.g. that several operations should be iterated until a specified condition is met, or that a subprocedure should be called recursively. The conventions for interpreting pseudocode thus involve many conventions about how scope and variable binding are to be interpreted which are akin to those employed in formal logic. See [14] p. 19-20 for a more detailed discussion of such conventions.

representation in the case x is a natural number). This property of A is typically reported as a function $time_A(n)$ of type $\mathbb{N} \rightarrow \mathbb{N}$ which returns the maximum number of steps consumed by A for all inputs x of length equal to n.[17] As much of theoretical computer science is aimed at finding and analyzing efficient methods for solving mathematical problems such as primality testing or sorting a list of numbers, this property is often taken as the single most significant feature of an algorithm. But before $time_A(n)$ can be precisely defined, some prior stipulation must be made as to what should be counted as a "basic computational step" of A.

Complexity theory and algorithmic analysis can be understood to offer distinct but complementary answers to this question. On the one hand, complexity theory is typically developed so that measures of *time* (and also *space* or *memory*) *complexity* are reported relative to a particular fixed model of computation $\mathfrak{M}$.[18] Although most familiar models are sufficient for this purpose, complexity theory does impose additional constraints beyond those imposed by computability theory on the choice of $\mathfrak{M}$ (wherein in $\mathfrak{M}$ is typically only required to be Turing complete and provide a suitable means of indexing machines so that results like the *s-n-m* Theorem hold). In particular, $\mathfrak{M}$ must be powerful enough to allow for the representation of numbers in binary form, but not so powerful as to allow for arbitrarily branching parallel computations.[19] Such models – which include the familiar single- or k-tape Turing machine model $\mathfrak{T}_k$ with a binary tape alphabet or the standard Random Access [RAM] model $\mathfrak{R}$ with unit time addition and subtraction – are often referred to as *reasonable* and are taken to comprise a natural category known as the *first machine class* [79].

Although the reliance of complexity theory on the existence of a well defined class of models of this sort is often not stressed explicitly, it is difficult to understate its overall significance for theoretical computer science. For complexity theory is standardly taken to provide our most complete account of which mathematical problems are *feasibly decidable* – i.e. decidable by a method which can

[17] More precisely, this is the so-called *worst case running time* of A. Although it is possible to also study the best and average case running time (relative to some probability distribution on inputs) of A, the worst case metric is often taken to be the most useful in practice – see, e.g., [71] for discussion.

[18] For present purposes I will assume that it is clear what it means for a particular mathematical formalism – e.g. the lambda calculus or the class of Turing machines – to serve as such a model. Matters of family resemblance aside, however, an important foundational question which is shared by computability theory, complexity theory, and algorithmic analysis is that of making the notion of a model of computation precise. I will return to this issue in §4 (see also [28], [73], [79]).

[19] For instance, the former restriction excludes the variant of the Turing machine model with only a single non-blank symbol in its tape alphabet, while the latter excludes the so-called Parallel RAM [PRAM] model, or the so-called MBRAM variant of the traditional sequential model which allows for unit time multiplication. Although these models are Turing complete – and hence sufficient for the development of computability theory – they are members of the so-called *second machine class* for which the complexity classes **P** and **PSpace** defined below collapse. See [79] for details.

be carried out in practice for inputs of a reasonable size, as opposed to only in the "in principle" sense of computability theory. Thus if there were no precise means of distinguishing those models of computation which provide abstract characterizations of the sort of basic operations which can be carried out in a single by the sort concretely embodied computing devices we employ in practice, there would be little hope of giving a *mathematical* analysis of feasible computability in the manner which complexity theory aims to provide.[20]

Once a model $\mathfrak{M}$ in the first machine class has been fixed as a benchmark, complexity theory suggests that the running time $time_A(x)$ of an algorithm A should be defined by first constructing a machine $M \in \mathfrak{M}$ which is said to *implement* A. In order to state in precise terms what it means for M to implement A requires solving what I will refer to as the *implementation problem* – i.e. that of providing necessary and sufficient conditions for when a given $M \in \mathfrak{M}$ is an adequate mathematical representation of A. As should already been evident from the example of §2, providing a non-stipulative answer to this question is of central importance to assessing the status of algorithmic realism itself. In broad terms, however, the manner in which M is often constructed from the specification of A will be familiar to anyone who has attempted (e.g.) to construct a Turing machine for performing addition or multiplication on natural numbers represented in binary notation – i.e. it must be shown how the basic operations and data structures of $\mathfrak{M}$ may be used to mimic the step-by-step operation of A in terms of the (possibly "higher-level") primitive operations in terms of which A is informally specified. Upon so doing, the function $time_A(n)$ can then be defined as $time_M(n)$ – i.e. the mathematically defined running time complexity of the machine M constructed in this manner.

After the choice of $\mathfrak{M}$ has been fixed, complexity theory then goes on to study the properties of $time_M(n)$ where M varies over the machines which solve mathematical problems – e.g. primality testing, determining whether a given propositional formula is satisfiable or is a tautology, determining whether a given graph has a clique of a certain size or whether two graphs are isomorphic. This gives rise to the notion of a *complexity class* $\mathbf{Time}_{\mathfrak{M}}(f(n))$ which is defined to consist of those computational problems X which can be solved by a machine $M \in \mathfrak{M}$ such that $time_M(n) \leqslant O(f(n))$.[21] Such classes form the familiar hierarchy $\mathbf{LogTime} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSpace}$, respectively defined as the classes of

[20] For instance, in distinguishing the first machine class from the second, van Emde Boas states "The first machine class represents the class of reasonable sequential models. However, for the machine models in the second machine class it is not clear at all that they can be considered to be reasonable. It seems that the marvelous speed-ups obtained by the parallel models of the second machine class require severe violations of basic laws of nature. Stated differently: if physical constraints are taken into account, all gains of parallelism seem to be lost." [79], p. 14.

[21] See below for discussion on the use of the order of growth notation $O(f(n))$. Also note that many of the problems mentioned above pertain to the classification of mathematical (e.g. logical formulas or graphs) objects which are not natural numbers. As such, suitable encodings of these items will often need to be found in order that they may be presented as inputs to an appropriate $M \in \mathfrak{M}$. In the case where $\mathfrak{M}$ is taken to be the Turing machine model $\mathfrak{T}_k$, it is

computational problems which can be solved in time $k \log_2(n)$ and $k_1 \cdot n^{k_2}$ using (e.g.) the deterministic Turing machine model $\mathfrak{T}_k$ (respectively described as *logarithmic time* and *polynomial time*), the class of problems which can be solved in time $k_1 \cdot n^{k_2}$ using the non-deterministic Turing machine model $\mathfrak{NT}_k$ (*non-deterministic polynomial time*), and the class of problems which can be solved in space $k_1 \cdot n^{k_2}$ using $\mathfrak{T}_k$ (*polynomial space*) – see, e.g., [60] for details.

As should be evident from the foregoing discussion, complexity theory is primarily concerned with the classification of computational problems understood extensionally as classes of their instances. The most fundamental distinction in this regard is between those problems which can be shown to be members of **P** – which is commonly thought to provide an upper bound on the class of problems whose instances can be uniformly solved in practice – and those which can be shown to be *hard* for a class such as **NP** which is strongly believed (but not known) to properly contain **P**. A problem X is shown to be hard in this sense by showing that there is a polynomial time reduction which efficiently transforms any instance of X into an instance of some problem which is known to be *complete* for **NP**. Such a problem can thus be understood to embody the underlying computational difficulty of solving all problems in this class.[22]

One well known example of such a problem is the set SAT consisting of the set of (binary strings encoding) formulas of propositional logic for which there exists a satisfying assignment – e.g. $P_0 \vee P_1 \in \text{SAT}$, but $P_0 \wedge \neg P_0 \notin \text{SAT}$. Since SAT is known to be **NP**-complete, it is very unlikely that SAT is feasibly decidable. Such a fact would typically be reported in complexity theory as follows:

(4) If $\mathbf{P} \neq \mathbf{NP}$, then there does not exist an algorithm with polynomial running time complexity which solves the problem SAT of determining whether a formula of propositional logic is satisfiable.

Note that like many other limitative results in computability theory such as (1), statements like (4) are often reported informally using quantifiers over algorithms. But it should be clear on the basis of the foregoing that just as CT_a allows us to paraphrase away the quantification over algorithms in "There is no algorithm for deciding first-order validity" in favor of quantification over the members of any Turing complete model of computation, it is possible to paraphrase away the quantifier in (4) in favor of quantification over the members of an appropriate model of computation in the first machine class.[23]

Matters are somewhat different in algorithmic analysis wherein the focus of study is not on computational problems, but rather on individual algorithms themselves. Note, however, that in order to justify a statement such as (3b)

typical to assume problem instances are encoded as binary strings. In this case a computational problem X will be a subset of $\{0,1\}^*$.

[22] See, e.g., chapter 8 of [60] for the relevant definitions and examples.

[23] Note, however, that unlike the classical limitative results of computability theory, the statement of (4) is premised on the non-coincidence of the complexity classes **P** and **NP**. Such a qualification can be removed by considering a computational problem which is known to be complete for a complexity class such as **NExp** (i.e. non-deterministic exponential time, see chapter 20 of [60] for examples) which can be proven to properly contain **P**.

requires that we are able to somehow count the number of steps which are required by an algorithm like InsertionSort or MergeSort to sort a list of length n. This is typically accomplished by performing a so-called *running time analysis* of the algorithm – i.e. a combinatorial argument which counts the maximum number of distinct operations which must be performed when the algorithm is carried out for an input of length n. For instance, a routine calculation shows that as by the described by the pseudocode specification above, InsertionsSort has *exact* running time complexity $(3/2)n^2 + 5/2(n) - 4$.

It should be borne in mind, however, that such calculations are conducted under the assumption that it is admissible to determine the running time complexity of an algorithm by counting each instruction in its pseudocode specification as a single primitive step. This convention raises two questions: 1) should the exact running time complexity derived in this manner be understood as an intrinsic property of an algorithm itself?; 2) given that a pseudocode specification can in principle be stated in terms of arbitrary mathematical operations (inclusive, e.g., of non-effective ones), does it make sense to ascribe a running time complexity to an algorithm directly, or should such ascriptions be understood modulo a further specification of what operations are to be taken as primitive?

A comparison of different textbooks is often sufficient to confirm that conventional usage of common algorithmic names allows for slight variations in the way in which an algorithm is specified – e.g. InsertionSort might be specified by using a "for" or "repeat-until" loop instead of a "while" loop, or it might increment its counter variables in a different order. Experience bears out, however, that such small variations in how an algorithm is specified typically affect the computation of its exact running time by at most a scalar factor. In virtue of this, the running time complexity of individual algorithms are conventionally reported not as exact functions, but rather using so-called *asymptotic* notation.

Recall in particular that the notation $O(f(n))$ is used to denote the class of functions whose rate of growth is dominated by a scalar multiple of $f(n)$ for sufficiently large n[24] – i.e.

(5) $$O(f(n)) =_{df} \{g : \mathbb{N} \to \mathbb{N} \mid \exists c, n_0 \forall n \geqslant n_0 [g(n) \leqslant cf(n)]\}$$

Based on the calculation reported above, InsertionSort would thus typically be reported as having running time $O(n^2)$. And based on similar (but more complex) calculations, the algorithms MergeSort and HeapSort would both be reported as having running time $O(n \log_2(n))$. It is easy to see that any function in the latter class is eventually dominated by one in the former. And it is such facts which are typically understood to provide the mathematical content of comparisons of the efficiency of algorithms as reported by (3c).

The use of asymptotic notation goes some distance towards justifying the claim (which appears implicit in the methodology of algorithmic analysis) that a running time complexity can be associated directly with an algorithm. However, a yet more serious challenge to the basis of this practice arises in regard to the

[24] The use of asymptotic notation in computer science was originally introduced and popularized by Knuth – see., e.g., [44].

second question flagged above – i.e. is it licit to use arbitrary effective operations in a pseudocode specification of an algorithm?

In order to address this question, it is useful to keep in mind that the development of algorithmic analysis has been largely guided by the practical goal of developing methods for comparing the efficiency of procedures for solving problems which are of some practical interest. But at the same time, it is also easy to see that by selecting an appropriate choice of primitive operations, it is often possible to trivialize the comparison of the relative efficiency of algorithms A_1 and A_2, by simply defining another algorithm A_3 which performs the same task as A_1 and A_2 in fewer (effective, but intuitively complex) steps.

For instance, although one might think that (3c) reports that a genuine gain in practical efficiency can be had by sorting a list using MergeSort instead of InsertionSort, even the $O(n \log_2(n))$ running time complexity of the latter is expensive in comparison with that of the $O(1)$ (i.e. constant time) algorithm

```
TrivialSort($B$)
1  **return** sort($B$)
```

TrivialSort sorts lists of arbitrary length in a single step by simply calling the function *sort*(B) – an operation whose effectiveness is in turn attested to by the existence of algorithms such as MergeSort instead of InsertionSort.

The conventional manner in which this problem is addressed in algorithmic analysis is to assume that while it is legitimate to specify and reason about algorithms informally via their pseudocode specifications, such specifications should be understood relative to a choice of a fixed model of computation $\mathfrak{M}$ for which it is assumed that they can be implementable. Although this model is often chosen for convenience, it is still important that it be among those in the first machine class. One hallmark of such models is that they allow the sorts of algorithms A which arise most often in mathematical practice to be implemented in a manner such that it is possible to construct $M \in \mathfrak{M}$ such that A and M not only compute the same function (possibly up to an efficiently computable encoding of inputs and outputs), but also such that $time_M(n) \in O(time_A(n))$ – i.e. although M may have a larger *exact* running time than A (e.g. in virtue of needing to explicitly compute the value of a function which is assumed as a primitive in the informal specification of A), it will have the same *asymptotic* running time.[25]

The foregoing observations provide further justification for the use of asymptotic notation to report the running time of individual algorithms. However, they also further highlight the significance of the implementation problem – i.e. in order for $M \in \mathfrak{M}$ to count as an implementation of A, not only must it mimic its operation in a step-by-step sense, but it must do so in a manner which preserves its asymptotic running time complexity. The fact that we are able to construct such implementations using the models $\mathfrak{R}$ or $\mathfrak{T}_k$ for the sorts of procedures which have traditionally been valued as effective methods in mathematical practice is

[25] For further discussion of the status of these assumptions as well as the choice of specific models for the purpose of implementing algorithms arising in mathematical practice, see, e.g. [14] pp. 23-28 and [41], §1.3.

suggestive of the fact that there may be a much closure connection between the specific properties of these models and our detailed intuitions about effectivity than is often acknowledged.[26]

With these details in place, it may finally be observed that algorithmic analysis is concerned not only with the sorts of attributions of running time complexity as reported by (3b,c) but also with proving so-called *lower-bound* results which report that a given algorithm is optimal in terms of time or space complexity. Because they require combinatorial analysis not only of a single algorithm, but of all algorithms of a certain sort which compute a given function such results are often much harder to obtain than individual running time calculations. Recall, for instance that InsertionSort is known as a *comparison sorting algorithm* in virtue of the fact that it sorts the array B using comparisons performed with $\leqslant$ but without making use, e.g., of the size or multiplicity of the elements which comprise this array. For such methods it is possible to prove the following:

(6) There is no comparison sorting algorithm with running time complexity asymptotically less than $\Omega(n \log_2(n))$.[27]

In terms of its logical structure (6) should be compared with other limitative propositions about computability like (1) and (4). Suppose, however, our ultimate goal is to show – in conformity with the thesis (A) – that the quantifier over algorithms in (6) can be replaced with a quantifier over some other well defined class of mathematical objects in a manner which preserves the meaning of the proposition expressed. In this case, we are faced with a refinement of the problems which make it more challenging to find a suitable paraphrase of (4) than of (1). For recall finding such a paraphrase for (1) requires only that we have at our disposal a Turing complete model of computation – i.e. one that (per CT_a contains a machine computing every function which is computable by an algorithm), whereas in the case of (4) we must also at least ensure that this model is in the first machine class. But now note that providing an adequate paraphrase of (6) requires that we consider a model of computation $\mathfrak{M}_2$ which satisfies at least the following properties: i) $\mathfrak{M}_2$ is in the first machine class; ii) the implementation problem for sorting algorithms can be solved for $\mathfrak{M}_2$ in a manner which preserves asymptotic running time complexity; iii) it is still meaningful to talk about what it means for a machine $M \in \mathfrak{M}_2$ to be an intuitively correct implementation of a comparison sorting algorithm such as InsertionSort or MergeSort.

[26] It is, for instance, by no means a trivial observation that the sort of primitive algebraic operations in terms of which procedures like Gaussian elimination or the Gröbner basis algorithm are stated can be implemented using the limited set of operations and data structures made available by the standard RAM model in a manner which preserves informal estimates of their running time complexity. In particular, such implementations often exist only in virtue of sophisticated techniques such as hashing which allows rapid look up of complex data objects in a manner which is independent of their size.

[27] Here $\Omega(f(n))$ denotes the *asymptotic lower bound* for $f(x)$ – i.e. the class of functions which $c \cdot f(n)$ eventually dominates for some constant c.

4. In search of a foundational framework

Given that complexity theory and algorithmic analysis represent our most refined methods for reasoning about the sorts of effective procedures which arise in mathematical practice, the sorts of considerations surveyed in §3 are typical of the primary linguistic and technical data to which a foundational theory of algorithms ought to be responsive. On this basis, it seems reasonable to extract the following general observations about the status we accord algorithms in our mathematical and computational practices:

I) Algorithms are mathematical procedures which may be described either informally using standard mathematical prose or through the use of a pseudocode specification. Such procedures can be *executed* or *carried out* for a given *input*, resulting in a sequence of intermediate states which leads to the calculation of an *output*.

II) Algorithms can be *implemented* by members of models of computation $\mathfrak{M}$. To implement an algorithm A is to specify a machine $M \in \mathfrak{M}$ which not only computes the same function as A (possibly up to an efficient encoding of inputs and outputs), but also operates in the same step-by-step manner.

III) Algorithms possess their *asymptotic running time complexity* intrinsically. This imposes additional constraints on the implementation relation – e.g. i) $\mathfrak{M}$ must be a model in the first machine class; ii) if $M \in \mathfrak{M}$ implements A, then M and A must have the same asymptotic running time complexity.

The question which we must now confront is how a proponent of algorithmic realism might develop a general theory which simultaneously accords algorithms the status of mathematics objects and also provides a satisfactory account of constraints like I)-III). In order to address this question it will be useful to consider three views about the nature of algorithms, two of which were briefly discussed in §1:

Direct algorithmic realism: Algorithms comprise a class $\mathcal{A}$ of intrinsically intensional objects which are distinct from those traditionally recognized in classical mathematics.

Strong Church's Thesis: Algorithms may be identified with (or *reduced to*) the members of a class of mathematical objects $\mathcal{M}$.

Algorithms-as-abstracts: Algorithms correspond to equivalence classes defined over some class of mathematical objects $\mathcal{M}$ factored by an appropriate equivalence relation.

In §1, the second (and also implicitly the first) of these options were dismissed summarily. It will now behoove us to backtrack slightly and examine them again in light of the considerations adduced in §2 and §3. This will allow for a better appreciation of why the algorithms-as-abstracts view appears to be the most plausible of the enumerated options.

As noted above, direct algorithmic realism might seem like the most straightforward means of making sense of the realistic tone of much of our informal discourse about algorithms (as typified by the passage cited at the beginning of

§3). For it seems that this discourse possesses many of the characteristics which philosophers often associate with ontological commitment to a category of objects – e.g. use the of singular terms such as Lucas, InsertionSort, etc. to make (apparent) singular reference, the use of quantification over algorithms to express limitative results, etc. And thus when taken in conjunction with the apparent incongruity flagged in §2 between the abstractness of mathematical objects and the use of temporal language to describe algorithms and their executions, one might reasonably conclude that there is at least *prima facie* justification for considering algorithms to comprise a class of freestanding "procedural" entities which unlike those traditionally countenanced as mathematical objects,

Another potential motivation for exploring this view is the observation that algorithms have traditionally conceive of as *intensional* entities. Some motivation for this classification derives from the observation that algorithms evidently cannot be assimilated to functions understood in the extensional sense. For whatever we take algorithms to be, it must be acknowledged can exist distinct algorithms for computing the same extensional function – e.g. although InsertionSort and MergeSort both compute the function *sort*(B), they cannot both be equal to this function in virtue of the fact that we regard them as having distinct computational properties (e.g. running time complexity). For reasons in this vicinity, it might seem that there is some hope of developing direct algorithmic realism long the same lines which philosophers have occasionally attempted to develop theories of other classes of intensionally individuated objects – e.g. Fregean senses, propositions, constructive proofs, etc.

While it seems that there is nothing which blocks proceeding in this manner, the prospects for developing and defending such a theory of algorithms along these lines begins to seem less plausible when we start to look at the methodologies of complexity theory and algorithmic analysis in detail. For on the one hand, while there do exist formal theories of notions like senses (e.g. [12], [76], [40]), propositions (e.g. [4], [16], [83]), and constructive proofs (e.g. [50], [31]), many of these make use of modal apparatus which appears out of place in a mathematical context, or are parasitic on technical notions which are already employed in logic or theoretical computer science (e.g. the typed lambda calculus or operational semantics for programming languages of the sort discussed below). Such observations serve to deflate the hope that extant theories of intensional entities can be used to explain the sense in which there is something *distintive* about algorithms which distinguishes them from, e.g., mathematically defined models of computation.[28] And on the other, none of these philosophically motivated accounts

[28] These considerations notwithstanding, it has been repeatedly suggested that the relationship between functions (understood extensionally) and algorithms can be taken to mirror that of the relationship between the reference of an expression and its Fregean sense – e.g [80], [21], [55], [40]. However, this represents at best a partial reconstruction of Frege's own understanding of the relation between sense and reference for functional expression. For on the one hand while an algorithm for computing a function $f(x)$ prototypically provides an *effective* means of determining the denotation of $f(a)$, a mathematical function can be introduced by a non-constructive definition (e.g. $f(x) = 1$ if the Riemann Hypothesis is true and 0 otherwise) which still might be taken to give its sense. And on the other hand, while we have seen that we

offers a principled explanation of the complex relationship which exist between algorithms, antecedently defined mathematical problems, and specific kinds of models of computation as described in §3.

Turning now to Strong Church's Thesis [SCT], another *prima facie* plausible view which one might attempt to develop is that it is possible to construct a mathematical object which could be taken to correspond to an algorithm starting from its pseudocode representation. An obvious point of comparison for such a view is the so-called *sententialist* theory of propositions according to which the proposition expressed by a natural language sentence just is that sentence itself. By analogy, a straightforward version of SCT might propose that we might similarly take an algorithm as nothing other than the text comprising its pseudocode specification.

Note, however, that sententialism is generally put forth as a form of *eliminativism* about propositions (a sentiment which is presumably counter to the spirit of algorithmic realism). And it is also typically held to suffer from a variety of problems – e.g. if unamended it leaves no room for two sentences in different natural languages to express the same proposition – versions of which can readily be seen to beset the analogous view about algorithms. A more plausible view is thus that algorithms should not be identified with linguistic descriptions such as pseudocode specifications themselves, but rather with some other sort of object which is derived by interpreting such specifications according to a form of compositional semantics. Such a view has obvious affinities to the conventional view that the proposition expressed by a sentence of a natural language is determined by a compositional semantic theory – e.g. of the sort originally propose by Montague [53] and now widely studied in formal linguistics.

Note, however, that in developing such a proposal we face the initial problem that pseudocode cannot be understood to be a part of any extant natural language. For instance, it employs constructions – e.g. those used to describe iteration, recursion, and flow control – which appear to have no natural language counterparts. But at the same time, pseudocode is also not typically taken to be a fully fledged formal language with a precisely defined syntax and semantics. It would thus appear that there is little hope that current semantic theories for natural languages can be directly applied to pseudocode specifications without substantial modifications and other precifications which would most likely have to be supplied by the practice of computer science itself.

A related view is that algorithms are the sorts of entities which are expressed not by pseudocode specifications themselves but rather their regimentations expressed using a formal programming language. It is often said that such languages were originally developed in part to provide a medium for expressing algorithms in a manner which is similar to pseudocode but which has a sufficiently precise syntax to allow direct translation into the primitive instruction sets which can

typically speak of the application of an algorithm to an argument as something which happens *in time*, Frege explicitly rejects the view that the composition of functions and arguments in the domain of sense should be understood as a temporal process (cf. [27]).

be carried out by conventional digital computers. But although it can be reasonably maintained that this is true of familiar "high level" languages like `Algol`, `C`, or `LISP`, it should also be kept in mind that unlike pseudocode specifications – which are at least intended to be continuous with informal mathematical language – programs constructed over such languages are originally *uninterpreted*.

It is now standardly acknowledged that formal semantics for programming languages are needed to enable rigorous correctness proofs similar to that considered in §2 – i.e. that when a programs is carried out in accordance with the intended interpretation of the constructs in which it is specified, it computes the values of an independently defined mathematical function. The recognition of this fact (which began in the 1960s) gave rise to the subject now known as *programming language semantics*. For present purposes, we may take a formal semantics for a programming language `L` to be a function $[\![\cdot]\!]^{\mathtt{L}}$ which maps $\mathtt{Prog}_{\mathtt{L}}$ – the class of programs over `L` – onto some domain $\mathcal{D}$ of mathematical objects which in some (yet to be specified sense) are taken to be the interpretations of `L`-programs.

Suppose we now also assume that given an informally specified algorithm A, it is unproblematic to construct a program $\Pi \in \mathtt{Prog}_{\mathtt{L}}$ which we take to be an adequate expression of A. We might then attempt to understand expressions of the form

(7) A is the algorithm expressed by Π

as a sort of canonical means of referring to algorithms. And on this basis, we might also propose that the reference of terms of the form "the algorithm expressed by Π" is given by applying the function $[\![\cdot]\!]^{\mathtt{L}}$ to the program Π.

It is, however, by no means a trivial question what sort of objects should be taken to comprise the class $\mathcal{D}$ which forms the range of $[\![\cdot]\!]^{\mathtt{L}}$, or how the value of $[\![\Pi]\!]^{\mathtt{L}}$ ought to be determined according to the structure of Π. Abstracting away from many details, it is possible to identify two broad approaches to these problems – respectively known as *denotational semantics* and *operational semantics*. In the first case, $\mathcal{D}$ is taken to be an appropriately defined function space X^Y such that the value of $[\![\Pi]\!]^{\mathtt{L}}_{den}$ will be a (possibly partial) function of type $X \to Y$ which is intended to correspond to the function induced by executing Π for all inputs in X. From this it follows that a denotation semantics will associate any two programs Π_1 and Π_2 which compute the same function with the same extensional object. But this presumably is an unsatisfactory result in the current context given that, e.g., we can easily construct programs which naturally express intuitively distinct algorithms (e.g. InsertionSort and MergeSort) but which compute the same function.

The specification of an operational semantics for `L` can be understood to address this issue by assigning to a program Π not a function but rather an abstract mathematical representation of its mode of operation, as determined compositionally according to the interpretation assigned by basic constructs made available by `L`. Such a representation can take a number of forms.[29] What is significant

[29] For instance, in the *structural operational semantics* of Plotkin [63], the class $\mathcal{D}$ is comprised of labeled trees whose structure represents the compositional structure of Π and whose

for our current purposes, however, is that in the case of operational semantics, the members of $\mathcal{D}$ are recognizable as instances of *models of computation* in the sense introduced in §3. Such a model is a class of mathematical structures $\mathfrak{M}$ together with a definition of a (possibly partial) function $App : \mathfrak{M} \times X \to Y$ which determines the output of applying $M \in \mathfrak{M}$ to input $x \in X$. In familiar cases like $\mathfrak{T}_k$ and $\mathfrak{R}$ the members of $\mathfrak{M}$ can be reasonably described as *machines* – i.e. they may be described as classes of *computational states*, over which various operations are defined and which induce a *transition* between states – while the $App(M, x)$ can be seen as providing a mathematical analysis of what it means to *carry out* or *execute* the computation which is induced by applying M to x.

In order to make these notions somewhat more precise it will be useful to introduce Moschovakis's [56] notion of an iterator as was described informally in §1:

Definition 4.1. An iterator from a set X to a set W $\phi : X \to Y$ is a quintuple $\phi = \langle in, S, \sigma, T, out\rangle$ such that

- St is an arbitrary non-empty set, the *computational states* of ϕ;
- $in : X \to St$ is the *input function* of ϕ;
- $\sigma : St \to St$ is the *transition function* of ϕ;
- $T \subseteq St$ is the set of *terminal states* of ϕ and $s \in T$ implies $\sigma(s) \in T$;
- $out : St \to Y$ is the *output function* of ϕ.

A *computation* induced by the iterator ϕ on an input $x \in X$ is a sequence of states $s_0(x), s_1(x), s_2(x), \ldots$ such that $s_0(x) = in(x)$ and $s_{n+1}(x) = \sigma(s_n(x))$ if $s_n(x) \notin T$ and is undefined otherwise. The *length* of such a computation is given by $len_\phi(x) = n + 1$ where n is the least such that $s_n(x) \in T + 1$. The result of applying ϕ to x induces a (possibly partial) function $App(\phi, x) =_{df} out(s_{len_\phi}(x))$.

It is a straightforward but useful exercise to see how familiar models of computation like $\mathfrak{T}_k$ and $\mathfrak{R}$ can be modified so that their instances satisfy the definition of an iterator. It is evident, however, that this definition cannot itself be taken to serve as an adequate analysis of either the general notion of an *effective* model of computation (as is needed for the development of computability theory), or for the definition of the first machine class as discussed in §3 (as is needed for the development of complexity theory and algorithmic analysis).[30] Nonetheless,

nodes are labeled with representations of state transitions of the form $\langle \alpha, s\rangle \to s'$ with the intended interpretation "if statement α is executed in state s, then the resulting computation will terminate in state s'". On the other hand, in the so-called *abstract machine semantics* of Nielson and Nielson [58], $\mathcal{D}$ is comprised of instances of a simple RAM model.

[30] For on the one hand, it is evident that since Definition 4.1 places no constrains itself on the mapping τ, there will exist iterators ϕ which compute (e.g.) all functions of type $\mathbb{N} \to \mathbb{N}$ (inclusive of non-recursive ones). And on the other, even if various so-called *locality* and *boundedness* constraints are placed on this definition to rule out such examples (e.g. in the manner of [28] or [73]), it will still be possible to canonically represent various models from the second machine class as iterators.

Definition 4.1 is still useful in the sense that it allows us to speak uniformly about notions like computational states and transition functions across different models of computation.

Let us return now to assess the plausibility of the version of SCT considered above. Relative to this proposal the reference of an expression of the form "the algorithm expressed by Π" is asserted to correspond to the value of $[\![\Pi]\!]^{\mathtt{L}}_{op}$ where $[\![\cdot]\!]^{\mathtt{L}}_{op}$ is the denotation function of some form of operational semantics for a programming language L for which $\Pi \in \mathtt{Prog}_{\mathtt{L}}$. Based on the foregoing discussion, we can now see that (at least in prototypical cases), we will have $[\![\Pi]\!]^{\mathtt{L}}_{op} \in \mathfrak{M}$ where $\mathfrak{M}$ is some fixed model of computation. The question which we must now ask is whether this view gives rise to a version of algorithmic realism which satisfies the constraints described at the beginning of this section.

To see that it does not, it is sufficient to observe that although the proposal in question succeeds in providing an account of how an informally specified algorithm A can be associated with a mathematical object – i.e. by first regimenting a pseudocode specification of A as a program Π over a language L, and then identifying A with $[\![\Pi]\!]^{\mathtt{L}}_{op}$ – it still leaves the choice of this object highly unconstrained. For note that even if we fix the schema (7) as a canonical means of making reference to algorithms, the choice of both the programming language L as well as the precise form of the denotational semantics $[\![\cdot]\!]^{\mathtt{L}}_{op}$ are still left undetermined. But as will be evident to anyone who has ever attempted to express an algorithm A of even moderate complexity in a formal programming language, not only will different languages lend themselves in different ways to expressing the operation of A, but there will typically be many programs over the same language which we will accept as equally "apt" expressions of A.

When compounded with the further variation which may be introduced by different forms of operational semantics we might adopt for the various choices of L, it would seem that the current proposal provides little insight into why we are justified (e.g.) in attributing a particular running time complexity (either exact or asymptotic) to the algorithm A itself. As such, it seems that the current proposal is unable to account for the constraints imposed on how we interpret our informal discourse about algorithms in a manner which is compatible with constraints imposed (per requirement III) above) by complexity theory and algorithmic analysis.

The difficultly just described evidently represents a variant of the familiar *what-numbers-could-not-be-problem* described in §1 – i.e. when we attempt to identify algorithms directly with either their linguistic specifications or the mathematical objects which we might take to be the semantic interpretations of these specifications, we are faced with an abundance of seemingly arbitrary choices about which objects should be taken as their "canonical" representations. A well known reply to this problem comes in the form of *structuralist* propsosal (cf., e.g., [72]) which holds that "algebraic" objects like groups or graphs should not be indentified with individual sets, but rather with *structures* (conceived roughly as in first-order model theory), which themselves are identified only up to isomorphism. An analogous refinement of the foregoing proposal is as follows: i) rather than seeking to identify algorithms with particular programs or their interpretations, we seek to define an appropriate equivalence relation $\approx$ over the class $\mathfrak{P}$ of programs drawn from the different programming languages $\mathtt{L}_1, \mathtt{L}_2, \ldots$ which we take to be adequate media for expressing the sorts of algorithms we encounter in practice; ii) we then identify an algorithm with the $\approx$-equivalence class of some program we take to express it.

This proposal leads us away from SCT toward the second form of the algorithms-as-abstracts view I discussed at the end of §1 – i.e. the view that algorithms should be understood as the "logical objects" obtained via the abstraction principle (PP) – i.e.

(PP) the algorithm expressed by Π_1 = the algorithm expressed by Π_2
if and only if $\Pi_1 \approx \Pi_2$

Recall, however, that I suggested there that this view also admits to a machine-based variant according to which the equivalence relation in question should be defined not over programs, but over an appropriate class $\mathfrak{M}$ of machine models. This gives rise to the rival abstraction principle

(MP) the algorithm implemented by M_1 = the algorithm implemented by M_2
if and only if $M_1 \simeq M_2$

where $\simeq$ is an appropriate equivalence relation defined over the class $\mathfrak{M}$.

Given that the algorithms-as-abstracts view appears to have significant advantages over direct algorithmic realism or SCT, it seems that the best prospects for an algorithmic realist lie with developing the form of this view based on (PP) or (MP) so as to provide a foundational account of algorithms which is in conformity with I)-III). But before entering into the details of how this might be accomplished in §5, it will also be useful to observe that there are a number of reasons to think that the the machine-based variant based on (MP) will ultimately fare better than the program-based variant based on (PP).[31]

In order to see why this is so, it will also be useful to briefly return to the comparison between the algorithms-as-abstracts view and the neo-logicist view of number theory mentioned in §1. Recall in particular that part of the standard defense of neo-logicism is that Hume's Principle – i.e.

(HP) the number of F = the number of G if and only if $F \equiv G$

where $\equiv$ denotes the second-order definable relation of *equinumerosity* – represents an *analytic* feature of our concept of natural number. Although this claim has often been challenged (e.g. [8]), neo-logicists at least start out with a precise mathematical definition of the equivalence relation by which they proposes to factor the class of finite sets into equivalence classes. And additionally, HP itself at least seems to have the following in its favor: 1) it is *extensionally adequate*

[31] I will discuss the machine-based variant in greater depth in §5. For general discussion of the program-based variant see, e.g., [65], [39], [22], [77]. For a well-worked out technical proposal along these lines see Yanofsky [82] who writes "'For us, an algorithm is the sum total of all the programs that express it. In other words, we look at all computer programs and partition them into different subsets. Two programs in the same subset will be two implementations of the same algorithm. These two programs are 'essentially' the same." [82], p. 3

in the sense that it explains both the truth of "the number of days in a fortnight = the number of moons of Neptune" and the falsity of "the number of planets = the number of US supreme court justices"; 2) even if it is not an *analytic* feature our of our basic understanding of number, HP can plausibly be maintained to follow from such an understanding plus suitable definitions (cf., e.g., [38]).[32]

I will ultimately argue that no foreseeable version of the algorithms-as-abstracts views fares particularly well with respect to either of these criteria. But in order to see that the machine-based variant may at least fare better, note that a proponent of either form of the view faces the initial challenge of providing a precise definition of either $\mathfrak{M}$ and $\simeq$ or of $\mathfrak{P}$ and $\approx$. As I suggested above, although there are many different models of computation have been introduced, their members can typically be described in a transparent manner as iterators. This provides the proponent of the machine-based variant with a natural suggestion as to the appropriate choice of $\mathfrak{M}$ – e.g. for maximal generality, he can just take $\mathfrak{M}$ to coincide with the (proper) class of sets satisfying a canonical translation of Definition 4.1 into the language of set theory. And as we will see in §5, over such a class it is at least possible to provide a general definition of equivalence – i.e. that of *mutual simulation* – which provides a *prima facie* plausible means of responding to concerns of type 1) (i.e. extensional adequacy) and type 2) (i.e. fidelity to an antecedently recognized notion of "procedural equivalence").

But matters would appear to stand somewhat differently with respect to the program-based variant of the algorithms-as-abstracts view. To get an impression for why this is so, note first that as with the notion of machine model, the number of programming languages which have been introduced in computer science numbers at least into the hundreds. However, the characteristics which qualify a formalism as a programming language seem to be less well defined than in the case of machine models. This is witnessed, for instance, by the existence of different *programming paradigms* – e.g. declarative, functional, object-oriented, etc. – each of which can be understood to be based on a fundamentally different conception of what is involved with providing a linguistic description of a mathematical procedure (cf., e.g., [70]). Given that these languages employ primitive constructs drawn from a wide range of developments in logic and mathematics – e.g. the typed and untyped-lambda calculus (`LISP` and `Haskell`), first- and higher-order logic (`ProLog`, `HiLog`), graph rewriting (`GP`), linear algebra (`Fortran`) – it seems unlikely that we can find an overarching mathematical definition analogous to

[32] Another dimension along which the neo-logicist project compares favorably with the algorithms-as-abstracts view pertains to the fact that we possess a prior axiomatic theory of the natural numbers whose interpretability in a putatively analytic extension of second-order logic can be used a criterion of success. In particular, the adjunction of HP to pure second-order logic yields a system in which it is possible to derive the second-order Peano axioms PA^2 once zero and successor are analyzed in the manner which Frege suggests (i.e. "Frege's theroem" – cf., e.g., [10]). Note on the other hand that not only do we not at present possess anything like an axiomatic theory of algorithms (over what language would such a theory be formulated? what would its axioms be?) relative to which the success of principles like PP and MP in accounting for our procedural discourse can be judged.

that of an iterator relative to which all of the programming languages studied in computer science may be naturally circumscribed.

Such diversity considerations aside, however, a proponent of the program-based variant might still attempt to nominate some specific language `L` relative to which it might be argued that all algorithms may be naturally expressed.[33] Putting aside the question of what could justify the choice of `L` over other languages which might be put forth for this purpose, one might then attempt to explicitly define the relation $\approx$ over $\mathtt{Prog}_{\mathtt{L}}$ by attempting to identify pairs of programs (or potentially even individual programming language constructs) which satisfy some antecedent intuitions about when two programs (or constructions) expressed the same algorithm, or as we might put it, are "procedurally equivalent".

Such an approach bears an evident affinity to the notion of *synonymy isomorphism* explored by Carnap [11] and Church [13] in the case of sentence meanings. Various approaches to defining procedural equivalence have also been studied in the literature on programming language semantics. But such proposals often have an *ad hoc* flavor in the sense that they concentrate on only particular species of "procedure preserving" transformations such as renaming of bound variables, substitution of evaluable expressions, or transformations of one form of flow control construction into another.[34] They thus provide little confidence that a notion adequate to meeting the needs of algorithmic realism – e.g. with respect to extensional adequacy or the justification of attributing asymptotic running times directly to algorithms – can be defined in a manner which does not ultimately rely on the same intuitions which underlie the machine-based variant of the algorithms-as-abstracts view which will be considered in §5.[35]

[33] Understood in the current context, such a claim is considerably stronger than Church's Thesis. For in order to show that `L` is "expressively complete" in the relevant sense, it must be shown that not only does it allow for the formulation of programs which compute all recursive functions, but also that these formulations are adequately representative of the relevant class of all algorithms. And it would appear that it would be hard to make a systematic case for this (even in the case of widely used languages like `C`) without having at our disposal a detailed account of which sorts of computational properties apply in the first instance to algorithms as opposed (e.g.) to their implementations.

[34] The technical goal of such accounts is typically not that of providing a conceptual analysis of "procedural equivalence" but the much more modest one of determining (e.g.) when two program fragments can be substituted for one another in a manner which does not affect the execution of a program into which they are substituted (see, e.g., [61]).

[35] To take just one example (derive from [82]), consider the following two program fragments: 1) $\mathsf{for}\ i = 1\ \mathsf{to}\ n{+}1\ \{x = f(x)\}$ and 2) $\mathsf{for}\ i = 1\ \mathsf{to}\ n\ \{x = f(x)\};\ x = f(x)$. One might claim that 1) and 2) express the same "procedural meaning" in the sense that they each describe a method for iterating the application of function $f(x)$ to itself $n + 1$ times (where in the latter case, the iteration by which this is achieved is simply "unraveled" by one step). In speaking in this manner, however, we seem to rely on *extra-linguistic* intuitions about the effect of executing the program in question – e.g. based on how data is stored in the registers of an abstract machine relative to which it is carried out. And as I will suggest in §5, such intuitions seem to be most naturally accounted for in terms of the notion of simulation equivalence which is defined on machines rather than programs.

5. Procedural equivalence

The view which I will discuss in this section is a further refinement of the proposal which I just argued represents the most promising means of defending algorithmic realism – i.e. the machine-based variant of the algorithms-as-abstracts view. In order to develop such a proposal in greater detail, recall that one needs to put forth both a class of machine models $\mathfrak{M}$ and an equivalence relation $\simeq$ over $\mathfrak{M}$ such that if one then identifies algorithms with $\simeq$-equivalence classes over $\mathfrak{M}$ determined by their implementations, we may then provide a plausible account of requirements such as I)-III) from above.

Such a proposal has historically attracted the most attention among proponents of algorithmic realism. For unlike the situation we face with respect to the program-based variant of the algorithms-as-abstracts view, it seems possible to nominate definitions of $\mathfrak{M}$ and $\simeq$ which are both technically and conceptually plausible. With respect to the former, for instance, we can see that both complexity theory and algorithmic analysis provide a principled basis for requiring that an implementation of an algorithm be a member of a model of computation from the first-machine class. Thus although the general definition of an iterator introduced above is it itself too broad to serve as a definition of $\mathfrak{M}$, we might plausibly try to develop the machine based view by considering the class $\mathfrak{M}^1$ comprised of the iterator representations of the union of all models in the first machine class.[36]

Since machines in $\mathfrak{M}^1$ can be naturally represented as iterators, they share a common mathematical structure which appears to be lacked by programs formulated over different languages. Not only might we hope to take advantage of this common structure so as to provide a definition of $\simeq$ which somehow engages with the specification of machines in virtue of their properties as iterators, but there might even we some hope that this can be accomplished in a manner which can be understood to analyze a salient *pretheoretical* notion of computational equivalence. This might in turn be thought to improve the chances that the principle (MP) can be regarded as akin to a logical or conceptual truth which grounds our understanding of algorithms in something like the way the neo-logistics claim that Hume's Principle grounds our understanding of the notion of number. It has

[36] In [79] the following precise definition of $\mathfrak{M}^1$ is given: a model $\mathfrak{M}$ is in the first machine class just in case there exists a polynomial time, linear space overhead simulation of $\mathfrak{M}$ relative to the single-tape Turing machine model $\mathfrak{T}_1$. (It may be seen that this definition is sufficiently broad to encompass $\mathfrak{T}_k$ (for $k > 1$) and the standard RAM model $\mathfrak{R}$, as well as many more refined models based on computational architectures which may be concretely embodied as practical microprocessor designs.) On the order hand, the second machine class $\mathfrak{M}^2$ is defined to contain those models $\mathfrak{M}$ such that $\mathbf{P}^{\mathfrak{M}} = \mathbf{PSpace}^{\mathfrak{M}}$ – i.e. for which the classes of polynomial time and polynomial space coincide when these notions are analyzed relative to $\mathfrak{M}$. Since it is currently unknown even if $\mathbf{P}^{\mathfrak{T}_1} \neq \mathbf{PSpace}^{\mathfrak{T}_1}$, it is in principle possible that $\mathfrak{M}^1$ and $\mathfrak{M}^2$ coincide. Although of course this is taken to be exceedingly unlikely, any indeterminacy in the definition of $\mathfrak{M}^1$ lies with the status of open separation questions in complexity theory and not with the imprecision of notions like "reasonableness" by which this class might be initially characterized.

in fact been repeatedly claimed that $\simeq$ can be taken to be a form of simulation equivalence of the sort which has been extensiely studied in a variety of contexts in theoretical computer science. I will introduce and motivate this notion in §5.1, before arguing in §5.2 that it is also unlikely that a version of (MP) based on simulation equivalence will be able to meet the needs of algorithmic realism.

5.1. **Simulation Equivalence.** The origins of the notion of simulation which will be at issue in this section can be traced back to the results demonstrating that the models of computation originally defined in the 1930s all determine the same class of functions – i.e. the partial recursive ones. Such results are *extensional* in the sense that they pertain to the class of functions which are determined by the machines comprising a model rather than to how they are computed. However, they are typically demonstrated in a paradigmatically *intensional* manner as described in the following passage:

> The proofs for the results ... have the following common structure. In every instance, the fact that one formally characterized class of partial functions is contained in another is demonstrated by supplying and justifying a uniform procedure according to which, given any [machine M_1] from the first characterization, we can find a [machine M_2] from the second characterization for the same partial function. ... [Results of this type show] that there is a sense in which each standard characterization appears to include all possible algorithms ... For, given a formal characterization ... there is a uniform effective way to "translate" any set of instructions (i.e. algorithm) of that characterization into a set of instructions of one of the standard formal characterizations. [66], p. 19

Consideration of the equivalence proofs in question also makes clear that the notion of a "translation" between models is what would later come to be known as a *simulation* – i.e. that of a uniform transition-preserving maping between the states comprising the computations induced by two machines M_1 and M_2 from different models.[37] The proposal that it is possible to *define* the notion of algorithm in terms of an appropriate refinement of this notion appears to have originated in early development of the field which would come to be known as *process algebra* (cf. [67]) wherein the notion of simulation was first rigorously defined.

In one of the founding papers in this subject Milner writes as follows:

> One aim ... is to make precise a sense in which two programs may be said to be realizations of the same algorithm. We can say loosely that for this to be true it is sufficient though perhaps not necessary that the programs do the same "important" computations in the same sequence, even though they differ in other ways: for example 1) we may disregard other computations perhaps different in the two programs, which are "unimportant" in the sense that they are only concerned with controlling the "important" ones, (2) the data may flow differently through the variables or registers, (3) the data may be differently represented in the two programs. ... [W]e give a relation of simulation between programs which may fairly be said to match [this description] ... Mutual simulation is an equivalence relation, and it is the equivalence

[37] See Kreisel [48], p. 177-178 for similar comments about the significance of such "translation-based" proofs in regard to Strong Church's Thesis.

> classes under this relation which may be regarded as algorithms - at least this is an approximation to a definition of algorithm. [51] p. 2

What Milner refers to here as "programs" are in fact structures very similar to iterators. If we let $\phi_1 = \langle in_1, St_1, \Sigma_1, T_1, out_1 \rangle$ and $\phi_2 = \langle in_2, St_2, \Sigma_2, T_2, out_2 \rangle$ be iterators of type $X \rightarrow Y$, Milner's proposal may thus be understood as the claim that such structures should be taken to implement the same algorithm just in case there exists a relation which correlates the steps in the computations induced by the input x $\vec{s}(x) = s_0(x), s_1(x), s_2(x), \ldots$ of ϕ_1 and $\vec{t}(x) = t_0(x), t_1(x), t_2(x), \ldots$ and of ϕ_2 which satisfies certain properties. I will refer to these properties respectively as the *transitional* and *representational* conditions on a definition of simulation.

The transitional requirement is intended to analyze the intuition that iterators ϕ_1 and ϕ_2 corresponding to machines which implement the same algorithm A ought to each perform the same sequences of operations which must be performed to carry out A in the same order. In the most straightforward situation, this could be formalized by requiring that there exists a *simulation relation* $R \subseteq St_1 \times St_2$ such that every transition between states $s, s' \in S_1$ mediated by σ_1 in a computation of ϕ_1 is matched by a similar transition between R-related states mediated by σ_2 in the corresponding computation of ϕ_2.[38] Writing $s \overset{i}{\mapsto} s'$ for $\sigma_i(s) = s'$ $(i \in \{1, 2\})$, such a condition can be schematized as follows:

$$(8) \qquad \forall s \in S_1\ \forall s' \in S_2\ \forall t \in S_2[(s \overset{1}{\mapsto} s' \wedge R(s,t)) \rightarrow \exists t' \in S_2(t \overset{2}{\mapsto} t' \wedge R(s',t'))]$$

It is a condition of this type which is typically demonstrated to hold between machines from different models in the course of demonstrating both the sort of extensional equivalence and complexity overhead results mentioned above.

But it is also not hard to see that as stated (8) is almost certainly too restrictive to apply to many cases in which we wish to regard ϕ_1 and ϕ_2 as implementations of the same algorithm. Milner flags this problem by noting that a simulation should only be required to relate sequences of states corresponding to "important" subcomputations but (presumably) not the "unimportant" sequences of states which comprise them. We can, for instance, imagine that the former correspond to implementations of the steps of the algorithm A which we take ϕ_1 and ϕ_2 to mutually implement, while the latter correspond to the sequences of "finer grained" steps by which the iterators carry out these molar steps.

If we write $s \overset{i\,*}{\mapsto} s'$ to denote that the state s' is derivable by some finite number of iterations of σ_i from s, then one way of revising (8) to take this observation into account is as follows:

$$(9) \qquad \forall s \in S_1\ \forall s' \in S_2\ \forall t \in S_2[(s \overset{1\,*}{\mapsto} s' \wedge R(s,t)) \rightarrow \exists t' \in S_2(t \overset{2\,*}{\mapsto} t' \wedge R(s',t'))]$$

[38] Since the models we are considering here are all deterministic, the relations R in question will all be single-valued – a point which will be taken into account when we adopt the definition of iterator isomorphism below. I have introduced the notion of simulation in terms of relations so as to conform to the tradition of process algebra wherein non-deterministic and concurrent models are also typically considered.

As van Emde Boas [79] observes, however, in moving from (8) to (9) there is a risk that we will make the transitional condition too weak by allowing, e.g., a simulation which links only the initial and final states in each computation of ϕ_1 and ϕ_2. It is hence generally acknowledged that an adequate formalization of the transitional condition must find a compromise between these constraints which is compatible with the extensional adequacy condition that $\simeq$ holds between ϕ_1 and ϕ_2 just in case they derive from machines M_1 and M_2 which we are prepared to accept as implementations of the same algorithm.

The representational requirement on the analysis of simulation is intended to formalize the fact that if we regard the iterators ϕ_1 and ϕ_2 as implementations of the same algorithm, then a simulation between them ought to relate states which represent the same mathematical structures (or "data") on which we understand them to operate. Suppose, for instance, that we are willing to regard ϕ_1 and ϕ_2 as implementations of Euclid's algorithm but that neither is based on a machine which is able to operate directly on natural numbers. (For instance, ϕ_1 might be a Turing machine and ϕ_2 might be a graph rewriting machine similar to that introduced by Schönage [68].) In this case, both ϕ_1 and ϕ_2 will need to use some form of alternative encoding to represent both the numerical inputs n, m to Euclid's algorithm, as well as the sequence of intermediate values (i.e. $r_0 = n \mod m, r_1 = m \mod r_0, r_2 = r_1 \mod r_0, \ldots$) which are computed during the course of its operation. (We might, for instance, imagine that ϕ_1 uses binary stings written on its tape to store these values, whereas ϕ_2 might uses a sequence of nodes and pointers.) In such a case the representational requirement imposes the condition that a simulation R between ϕ_1 and ϕ_2 must relate states which represent the same numerical values for each computation of Euclid's algorithm as implemented by these iterators.

5.2. **The exigencies of simulation.** While the foregoing conditions characterize the conceptual role which the notion of simulation is expected to play, they fall short of providing a precise definition of an equivalence relation $\simeq$ defined on $\mathfrak{M}^1$. A variety of different technical definitions of simulation have been proposed in computer science. Many of these are designed specifically to treat non-deterministic or concurrent computation. But since all of the models we have been considering are deterministic and non-concurrent, it will be useful to take as our paradigm Moschovakis's [56] definition of *iterator isomorphism*.

Definition 5.1. An *isomorphism* between two iterators ϕ_1 and ϕ_2 is a bijection $\rho : St_1 \to St_2$ such that $\rho(in_1(x)) = in_2(x)$ for all $x \in X$, $\rho[T_1] = T_2$, $\rho(\sigma_1(s)) = \sigma_2(\rho(s))$ for all $s \in S_1$ and $out_1(s) = out_2(\rho(s))$, for every $s \in T_1$ which is *input-accessible* (i.e., such that for some $x \in X$ and some $n \in \mathbb{N}$, $s = \sigma_1^n(in_1(x))$). We say ϕ_1 and ϕ_2 are *isomorphic* just in case there exists a mapping with these properties.

Requiring simulation relations to be bijections has the advantage of ensuring that simulation is an equivalence relation.[39] Clearly this definition can be modified in a number of ways – e.g. by relaxing the requirement that $\rho(x)$ be a function, or requiring only that it satisfy a requirement analogous to (8). But as I will now argue, however, there appear to be several reasons why any definition in this vicinity is still likely to fail to satisfy one or more of the requirements which would be necessary to vindicate the version of the algorithms-as-abstracts view under consideration.

5.2.1. *Formalizing the transitional condition.* The need to provide a definition of simulation which strikes a balance between (8) and (9) in order to satisfy Milner's transitional requirement obviously raises the concern that no definition in the spirit of iterator isomorphism can be found which induces the correct identity conditions for all of the algorithms we speak of in the course of our mathematical practices. For on the one hand it is easy to construct pairs of such structures which intuitively implement the same algorithm but for which this condition does not hold. And on the other hand, note that once we have decided to uniformly represent machines as iterators, it follows that the running time complexity $time_M(n)$ of a machine M will be given by $\max\{len_\phi(x) : |x| = n\}$. But also note that if ϕ_1 and ϕ_2 are isomorphic in the sense of Definition 5.1, then it will follow that $len_{\phi_1}(x) = len_{\phi_2}(x)$ for all $x \in X$. Thus if ϕ_1 and ϕ_2 are isomorphic in this sense, it will follow that they have the same exact running time complexity.[40] And this would appear to imply – counter to the argument given in §3 – that *exact* as opposed to *asymptotic* running time is an intrinsic property of individual algorithms.

These observations can be taken to illustrate another way in which the classical problem of "grain" – which is often pressed against theories of intensional entities like Fregean senses and propositions – also arises when we attempt to provide a definition of $\simeq$ which matches our intuitions about algorithmic identity and non-identity.[41] But such concerns can also be understood to point towards a yet more general worry for any view which attempts to analyze the notion of algorithm in terms of a principle akin to (MP) wherein $\simeq$ is taken to be a form of simulation equivalence similar to iterator isomorphism.

The underlying difficulty may be illustrated by first recalling that for a given model $\mathfrak{M}$ in the first machine class and algorithm A, there will often be multiple equally "apt" or "faithful" ways of implementing A as instances of $\mathfrak{M}$. To better

[39] Milner originally achieved this by requiring that there exist a pair of relations R and R' such that R satisfies (8) between ϕ_1 and ϕ_2 and conversely R' satisfies (8) between ϕ_2 and ϕ_1. These conditions respectively correspond to back and forth clauses of the familiar definition of *bisimulation* which is studied widely in process algebra and modal logic – cf., e.g., [6]. However, we are only interested in deterministic algorithms whose computations can be represented as linear sequences of states rather than trees. In such cases, it is more straightforward to adopt a definition like 40 in which simulations are functions rather than relations.

[40] In other words, iterator isomorphism in the sense of Definition is a *congruence* with respect to the property of the exact running time complexity $time_M(f(n))$.

[41] For a number of concrete examples of this type see [17] and [7].

appreciate the scope of this concern, note that many mathematical algorithms operate on structured mathematical entities like polynomials, matrices, of graphs and may also involve auxiliary data structures like trees, stacks, or heaps. Practical experience again bears out the fact that efficient implementations of such structures can be founded using the RAM model $\mathfrak{R}$.[42] But predictably, these representations are almost never unique – e.g. a matrix can be represented as a sequence of appropriately indexed registers or (if sufficiently sparse) as a hash table, a graph may be represented either as a list of adjacent vertices or as an adjacency matrix, a tree may be represented either as a prefix-closed list of sequences or as a graph, etc. And in addition, operations on these structures which we often treat as unmediated operations when specifying algorithms informally – e.g. looking up an entry in a matrix, adding or deleting a node from a graph – will generally be associated with operations which require multiple steps when implemented using a RAM machine.

Consider, for instance, a specific algorithm A for operating on graphs – e.g. Kruskal's minimum spanning tree algorithm.[43] A basic operation employed in this algorithm is that of adjoining an edge (u, v) to a set of edges E' of the graph $G' = \langle V, E' \rangle$ which will ultimately form a minimal spanning tree for the input graph $G = \langle V, E \rangle$. But now suppose ϕ_1 and ϕ_2 are iterators representing RAM machines which implement A in different ways – say ϕ_1 represents G and G' as adjacency matrices and ϕ_2 represents them as lists. And finally consider the sequence of steps $\vec{s}(G) = s_0(G), \ldots, s_{c_1(n)}(G)$ of $\vec{t}(G) = t_0(G), \ldots, t_{c_2(n)}(G)$ of ϕ_1 and ϕ_2 which will be required for A to adjoin the edge (u, v) to a particular spanning tree U which it is constructing.

Although it is reasonable to assume that the lengths of $\vec{s}(G)$ and $\vec{t}(G)$ will vary as linear functions $c_1(n)$ and $c_2(n)$ for $n = |V|$, there is no *a priori* reason to suspect that these functions will be identical. Nonetheless, the transitional condition on the definition of simulation requires that any simulation between ϕ_1 and ϕ_2 ought to correlate not the individual steps of $\vec{s}(G)$ and $\vec{t}(G)$ (which will be impossible if $c_1(n)$ and $c_2(n)$ ever differ in value), but rather these sequences themselves. This suggests that in order to satisfy the spirit of Milner's transitional requirement, we must find an appropriate way of partitioning the sequences of states corresponding to the computations of ϕ_1 and ϕ_2 into subsequences to take into account how these iterators (or more accurately, the machines from they are derived) implement the basic steps in terms of which A is defined.

In specific cases, it will generally be clear how this can be accomplished by modifying Definition 5.1 so that a simulation $\rho(x)$ is now understood as correlating not individual states, but rather sequences of states determined by the

[42] Techniques for constructing RAM implementations of such structures are considered systematically in the context of Knuth's meticulous implementations of informally specified algorithms as so-called MIX machines (a variant of the RAM model). See in particular Chapter II of [42].

[43] See, e.g., [14] p. 631-633.

relevant partitioning.[44] But such a modification will obviously depend on the particular partitioning of states required to ensure that the definition holds between the pair of iterators ϕ_1 and ϕ_2 in question. And it thus seems that the relevant modification to the definition of iterator can only be formulated given *after* we have decided that ϕ_1 and ϕ_2 should both be understood as implementing the same algorithm.

Recall, however, that according to the algorithms-as-abstracts view it is only through a principle such as (MP) that we understand what algorithms are in the first place. But if this is so, then the definition of $\simeq$ must presumably be fixed in a manner which is independent of the details of specific algorithms and the class of machines which we are willing to regard as their implementations. The proposed modification to the definition of iterator isomorphism using partition functions thus highlights that in moving away from a purely structural account of Milner's transitional requirement as exemplified by (8) there appears to be no natural stopping point short of (9) which is fixed independently of our prior understanding of *specific* algorithms and implementations. But it would seem that if a proponent of the algorithms-as-abstract view has any hope of maintaing that (MP) reflects a conceptual or analytic feature of our informal notion of algorithm, he cannot allow that the definition of $\simeq$ is fixed on a case-by-case basis in this manner.

5.2.2. *Formalizing the representational requirement.* In order to formalize Milner's representational requirement, we must find some relation between the computational states of iterators which formalizes the fact that they encode the same mathematical "data" on which a single algorithm A operates. As a first step in this direction, suppose that $\phi = \langle in, St, \Sigma, T, out\rangle$ and that $s \in St$ is one of the states of ϕ. Then we will typically be able to think of s as a mathematical structure akin to a Turing or RAM machine configuration. Such a structure can in turn be viewed in the conventional manner of first-order model theory – i.e. as a model of a first-order language $\mathcal{L}_s$ of the form $\mathcal{A}_s = \langle A, R_1, \ldots, R_n, f_1, \ldots, f_n, \{c_i : i \in I\}\rangle$ consisting of a non-empty domain A, a set of relations R_i, functions f_j, together with a designated collection $\{c_i : i \in I\} \subseteq A$.[45]

[44] For instance, in the case under consideration, there would presumably exist monotone functions $\tau_1, \tau_2 : X \times \mathbb{N} \rightarrow \mathbb{N}$ which uniformly partition the computation $\vec{s}(x)$ and $\vec{t}(x)$ of ϕ_1 and ϕ_2 into subsequences $\vec{s}(x) = s_{\tau_1(x,0)}(x), \ldots, s_{\tau_1(x,1)}(x), s_{\tau_1(x,1)+1}(x), \ldots, s_{\tau_1(x,2)}(x), \ldots, s_{\tau_1(x,k-1)+1}(x), \ldots, s_{\tau_1(x,k)}(x)$ and $\vec{t}(x) = t_{\tau_2(x,0)}(x), \ldots, t_{\tau_2(x,1)}(x), t_{\tau_2(x,1)+1}(x), \ldots, t_{\tau_2(x,2)}(x), \ldots, t_{\tau_2(x,k-1)+1}(x), \ldots, t_{\tau_2(x,k)}(x)$ such that the subsequences $s_{\tau_1(x,i)}(x), \ldots, s_{\tau_1(x,i)}(x)$ and $t_{\tau_2(x,i)}(x), \ldots, t_{\tau_2(x,i)}(x)$ correspond to the "unimportant" computations which implement the same step in the operation of A. On this basis, we could imagine redefining an isomorphism between ϕ_1 and ϕ_2 to be a function $\rho^* : St_1^* \rightarrow St_2^*$ mapping all and only sequences of the form $s_{\tau_1(x,i)}(x), \ldots, s_{\tau_1(x,i)}(x)$ to $t_{\tau_2(x,i)}(x), \ldots, t_{\tau_2(x,i)}(x)$ for $i \leqslant time_A(x)$

[45] For instance, in the case of the Turing machine model, we could represent states as structures of the form $\langle \mathbb{N}, f, c\rangle$ where $f : \mathbb{N} \rightarrow \{0, \ldots, k-1\}$ and $f(n)$ represents the contents of the nth tape cell, and c records the location of the machine's head.

In this setting, formalizing the representational requirement thus amounts to saying when the structures $\mathcal{A}_s$ and $\mathcal{A}_t$ representing the same configuration of data (and presumably also control parameters such as the values of loop variables) on which A operates. Since these structures are conventional first-order models, it is natural to first ask if any of the familiar notions of structural equivalence from model theory – e.g. isomorphism, elementary equivalence, back-and-forth equivalence – can be of use in this regard. Note, however, that if ϕ_1 and ϕ_2 are based on machines from different models of computation (e.g. one is derived from a Turing machine, and one from a RAM machine), then $\mathcal{A}_s$ and $\mathcal{A}_t$ will generally interpret distinct languages $\mathcal{L}_s$ and $\mathcal{L}_t$. But in this case, these structures cannot be isomorphic to one another in the traditional sense.

In the case that ϕ_1 and ϕ_2 are based on machines from a single model of computation $\mathfrak{M}$, the situation is likely to be more complex. To continue with the example from above, for instance, suppose that ϕ_1 and ϕ_2 are both based on RAM machines which we wish to interpret as implementing an algorithm which operates on a graph $G = \langle V, E \rangle$. A concrete illustration of the problem we face in formalizing the representational condition is to nominate some notion relation $U \subseteq St_1 \times St_2$ which holds just in case they contain sequences of registers encoding graphs which are isomorphic. Note, however, that by proceeding in this way, we will almost certainly incur the same problem highlighted above with respect to formalizing the representational condition. For if ϕ_1 and ϕ_2 represent G differently in their computational states – e.g. one uses a matrix representation, and the other a list representation – then we will be forced to define U in a manner that will be unlikely to generalize to arbitrary algorithms operating on arbitrary data structures.

We might attempt to circumvent both of the foregoing problems by trying to identify a more general definition which analyzes what it means for one structure $\mathcal{A}_t$ to be representable in another $\mathcal{A}_s$ even if these structures are for different signatures $\mathcal{L}_1$ and $\mathcal{L}_2$. One possibility is the notion of *definable interpretability* from model theory which holds between structures $\mathcal{A}_s$ and $\mathcal{A}_t$ just in case it is possible to define an $\mathcal{L}_1$-structure $\mathcal{N}$ whose domain is an $\mathcal{L}_1$-definable subset X of the domain of $\mathcal{M}_1$ consisting of $\mathcal{L}_1$-definable elements, subsets, and functions of X so that $\mathcal{N}$ is isomorphic to $\mathcal{A}_t$.[46] We might then attempt to analyze the representational condition by requiring that computational states correlated by a simulation relation are mutually interpretable in one another in this sense.

But if we attempt to apply this definition to the sorts of cases which are likely to be encountered in practice, we quickly run into what appears to be a complication arises in virtue of the way first-order model theory interacts with the mathematical definitions of common models of computation. For note that in order to describe the relation U which a state s of the iterator ϕ_1 described above bears to a state t of ϕ_2 in virtue of representing the same graph G, it is not sufficient to work in a language which describes s and t simply as vectors of natural numbers representing the contents of the registers of ϕ_1 and ϕ_2. Rather, we

[46] See [49] for definitions and examples of this relation.

must work in a language containing additional arithmetical vocabulary sufficient for describing how these iterators encode and operate on finite graphs as sequences of natural numbers stored in their registers.

The most natural way to do this is to view s and t as structures of the form $\mathcal{A}_s = \langle \mathbb{N}, R_1, 0, s, +, \times \rangle$ and $\mathcal{A}_t = \langle \mathbb{N}, R_2, 0, s, +, \times \rangle$ where R_1, R_2 are binary relations such that $R_i(n, m)$ holds just in case m is stored in the nth register of ϕ_i as represented by s (if $i = 1$) or t (if $i = 2$). It is now easy to see that $\mathcal{A}_s$ and $\mathcal{A}_t$ are mutually interpretable in one another since we can find formulas in either language which allow us to represent either structure in terms of the other using standard techniques for coding finite sequences to construct an $\mathcal{L}_1$-formula $\psi_{R_2}(x, y)$ which defines R_2 up to isomorphism as a subset of $\mathbb{N}^2$ and an $\mathcal{L}_2$-formula $\psi_{R_1}(x, y)$ which defines R_1 up isomorphism also as a subset of $\mathbb{N}^2$. But since R_1 and R_2 will only encode a finite number of register-value pairs which will be employed in the computations of ϕ_1 and ϕ_2 in their computations on a fixed graph, we can see that $\psi_{R_1}(x, y)$ and $\psi_{R_2}(x, y)$ can be taken to be purely *arithmetical* formulas – i.e. to simply show that $\mathcal{A}_t$ is *definable* over $\mathcal{A}_s$ does not require that we make use of the relation R_2 itself (and conversely for $\mathcal{A}_s$ and R_2).

In fact, since $\mathcal{A}_s$ and $\mathcal{A}_t$ are structures over a language which extend that of first-order arithmetic, it is also easy to see (again using standard coding techniques) that there will be a vast array of other structures of the form $\langle \mathbb{N}, R, 0, s, +, \times \rangle$ with which they will be mutually interpretable but for which R will not bear any discernible relation of structural similarity to the graph G which we are assuming the states s and s' encode. This would seem to illustrate another fundamental limitation on our ability to use traditional notions of sameness of structure taken from model theory to analyze Milner's representational condition.

5.2.3. *Implementing recursion.* The foregoing observations raise the general concern that despite the intuitive appeal of Milner's original analysis of simulation in terms of transitional and representational requirements, his characterization does not succeed in implicitly defining a relation (or even a family of relations) which holds between iterators in virtue of what we might broadly describe as their general structural properties which also induces an extensionally adequate criterion of identity for algorithms. But as I will now also attempt to illustrate, there are also instances in which these conditions can themselves be seen as pulling in opposite directions on the form which a definition of simulation ought to take.

Most of the procedures mentioned above would traditionally be classified as *iterative* (or *sequential*) algorithms in the sense that the computations induced by their operation can be understood as sequences of states induced by the repeated application of a transition function in the sense formalized by the notion of an iterator. There is, however, another class of procedures studied in algorithmic analysis known as *recursive* (or *divide and conquer*) algorithms which operate by successively decomposing their input into two or more *subproblems* (i.e. structural components) on which they then repeatedly call themselves until a base case is reached.

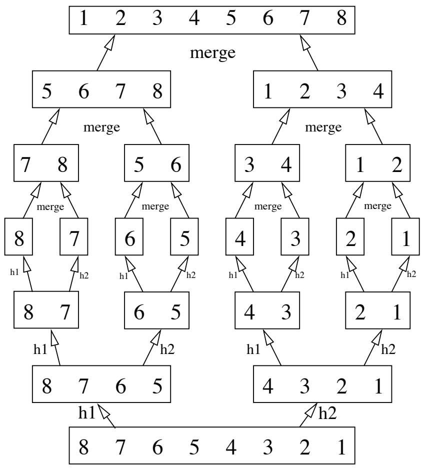

FIGURE 1. The operation of MERGESORT on the array $[8, 7, 6, 5, 4, 3, 2, 1]$. h_1 denotes the operation of taking the first half of an array, h_2 the second half, and *merge* the merging operation described in the text.

A commonly cited example is the sorting algorithm known as MERGESORT. This procedure can be described informally as follows: 1) divide the input array B into two subarrays B_1 and B_2 comprising the first and second halves of B (rounding indices up or down as needed); 2) if either of these arrays are of length greater than 1, recursively call MERGESORT (i.e. the procedure here described) on B_1 and B_2; 3) combine the sorted arrays B_1' and B_2' which result from this into a single sorted array by calling the auxiliary procedure MERGE.[47] In pseudocode such a procedure may be expressed as follows:

```
MERGESORT($B, p, r$)
if $p < r$
    then $q \leftarrow \lfloor (p + r)/2 \rfloor$
         MERGESORT($B, p, q$)
         MERGESORT($B, q + 1, r$)
         MERGE($B, p, q, r$)
```

It will be useful to illustrate the operation of MERGESORT by considering an example. To this end consider the initial array $B = [8, 7, 6, 5, 4, 3, 2, 1]$ as depicted at the bottom of Figure 1. The bottom four levels depict the decomposition of B into halves as effected by the recursive calls to MERGESORT. This

[47] MERGE itself can be understood as an iterative procedure which operates by repeatedly comparing the first elements of its inputs B_1 and B_2 (which are assumed to be sorted), selecting the one which is smaller with respect to $<$, removing it from the relevant list, and then adding it to the end of a new sorted list which it returns as output. See, e.g., [14], p. 31.

process terminates after calls of the form MergeSort$(B, i, i + 1)$ (as the next calls would be of the form MergeSort(B, i, i) and MergeSort$(B, i + 1, i + 1)$ for which the condition $p < r$ will fail). At this point control passes back to the point of the recursive call and the operations Merge$(B, i, i, i + 1)$ are performed. This results in the fifth level from the bottom. After these values have been computed, control passes to the prior call to MergeSort by which the resulting values and the subarrays of length two are merged by performing operations of the form Merge$(B, i, i + 1, i + 3)$, yielding the second row from the top. Finally, control passes to the initial calls of the form MergeSort$(B, 1, 4)$ and MergeSort$(B, 5, 8)$ which leads to the computation of Merge$(B, 1, 4, 8)$ yielding the sorted array at the top.

As is evident from the foregoing description, MergeSort is most naturally described as operating in a manner which assumes that certain of its steps can be carried out in parallel – e.g the calls to the Merge operation for the inputs $[7, 8]$, $[5, 6]$ and $[3, 4]$, $[1, 2]$ corresponding to the third row from the top are understood to be carried out simultaneously. A question which often arises in computational practice, however, is whether such a procedure can be implemented using a machine drawn from a model in the first machine class. For as we have observed above, such machines do not allow for unbounded parallelism of the sort which would be required to construct a direct implementation of the procedure just described.

In order to demonstrate that such implementations are available, it is typical to employ a sequential model of computation equipped with an auxiliary data structure known as a *stack*. Such a device can be used to keep track of both the subarrays which are being operated on and the location of a particular splitting or merging operation in the structure depicted in Figure 1. A method for transforming recursive specifications of procedures into machines in a class such as $\mathfrak{R}$ (to which a stack may either be added or implemented in terms of its other primitives) is known as an *implementation of recursion*. The details of how this is accomplished in practice are often quite complex (cf., e.g., [1]) and need not concern us here. What is of current concern, however, is that there will often not be a *unique* way of implementing a recursive algorithm specified using an sequential model such as $\mathfrak{R}$. For instance, there are distinct stack-based implementations of MergeSort R_{left} and R_{right} such that the former always performs the left-most possible merge operation first (relative to the sort of computation tree depicted in Figure 1) and the latter always performs the right-most possible merge operation first (as well as a variety of other intermediate possibilities).

These complications notwithstanding, the relation borne by both R_{left} and R_{right} to MergeSort would conventionally be regarded as paradigmatic of the implementation relation in computational practice. There is thus a strong *prima facie* reason to think that any adequate definition of $\simeq$ ought to hold between R_{left} and R_{right}. As should now be evident, however, no relation satisfying these properties can satisfy both the transitional and representational requirements on a definition of simulation simultaneously. For suppose that we let $\vec{l_1}, \ldots, \vec{l_7}$ and

$\vec{r}_1, \ldots, \vec{r}_7$ respectively represent the sequence of transitions of R_{left} and R_{right} responsible for carrying out the merges depicted in Figure 1. According to the transitional condition, a simulation relation of type $R^* \subseteq St^*_{left} \times St^*_{right}$ ought to relate sequences of states which occur in the same order – i.e. $R^*(\vec{l}_1, \vec{r}_1), \ldots, R^*(\vec{l}_7, \vec{r}_7)$. But on the other hand, according to the representational condition R^* ought to relate sequences corresponding to the operations which MergeSort performs on the same data – e.g. $R^*(\vec{l}_1, \vec{r}_5), R^*(\vec{l}_2, \vec{r}_4), R^*(\vec{l}_3, \vec{r}_7), \ldots$ The transitional and representational requirements thus cannot be simultaneously satisfied by any single-valued relation on $St^*_1 \times St^*_2$. As such, it would seem that there are instances in which the transitional and representational conditions on the definition of simulation are in genuine conflict with respect to certain of our pretheoretical intuitions about computational equivalence.

6. Taking stock

6.1. **Moschovakis, Gurevich, and the level-relativity of algorithms.** As I mentioned in §1, the task of providing a foundational account of the nature of algorithms of the sort we have been considering here has been largely overlooked by both mathematicians and computer scientists. Two notable exceptions are the theories of algorithms which been developed by Yiannis Moschovakis and Yuri Gurevich, each of whom can be understood as offering an extended defense of algorithmic realism. There are, however, several respects in which their views differ not only from one another, but also from the general framework I have argued in §4 and §5 represents the most promising means of developing this view. It will thus be useful to investigate their proposals in more detail before attempting to

In a series of papers spanning over twenty years (including [36], [37], [35]) Gurevich develops the proposal that individual algorithms should be identified with instances of a class of formalisms known as Abstract State Machines [ASMs] – a proposal he refers to as the *ASM Thesis*:

> The ASM thesis is that every sequential algorithm, on any level of abstraction, can be viewed as a sequential abstract state machine. [37], p. 1

The definition of an ASM is very similar to that of an iterator – e.g. in [35] (p. 7) such a model M is defined to consist of class of *states* $\mathcal{S}(M)$ (a subset $\mathcal{I}(M) \subseteq \mathcal{S}(M)$ of which are referred to as *initial*), and a transition function $\tau_M : \mathcal{S}(M) \to \mathcal{S}(M)$. A similar definition of the computation induced by M on input $x_0 \in \mathcal{I}(M)$ as the sequence derived by iterating $\tau(x)$ on x_0 is also provided.

Moschovakis's proposal is many respects similar, but is developed in terms of a formalism known as a *recursor*. Unlike any of the models we have considered thus far, this model takes recursion (rather than iteration) as its basic computational paradigm. A recursor formally is defined ([56], p. 85) to be a triple $\alpha = \langle \mathcal{D}, \tau, value \rangle$ such that $\mathcal{D} = \langle D, < \rangle$ is an inductive partially ordered set, $\tau : X \times D \to D$, and the result $\alpha(x)$ of applying α to $x \in X$ is defined to be value(d) where d is the $<$-least element of D satisfying the fixed-point equation

$d = \tau(x, d)$.[48] Two recursors $\alpha_1 = \langle D_1, \tau_1, \text{value}_1\rangle$ and $\alpha_2 = \langle D_2, \tau_2, \text{value}_2\rangle$ are defined to be isomorphic just in case there exists an order-preserving bijection $\pi : D_1 \to D_2$ for which $\pi(\tau_1(x, d)) = \tau_2(x, \pi(d))$ and $\text{value}_1(x, d) = \text{value}_2(x, \pi(d))$ ([56], pp. 86-87). On the basis of these definitions, Moschovakis proposes a view which might aptly be called the *Recursor Thesis* – i.e.

> *Proposal: Algorithms are recursors.* The mathematical structure of every algorithm on a poset X to a set W is modeled faithfully by some recursor $\alpha : X \to W$; and two recursors model the same algorithm if they are isomorphic. [56], p. 86

Relative to the terminology adopted in §4, such remarks make clear that Gurevich is a proponent of Strong Church's Thesis,[49] while Moschovakis can be understood to be a proponent of the machine-based variant of the algorithms-as-abstracts view where $\mathfrak{M}$ is taken to be the class of recursors and $\simeq$ is the relation of recursor isomorphism. Thus although Moschovakis and Gurevich can thus both be reasonably classified as algorithmic realists, it would at first appear to disagree with respect to how they would answer the following questions: 1) should algorithms be identified directly with the instances of a model of computation or rather with equivalence classes thereof? 2) should the mathematical class to which algorithms are reduced (either by direct identification or by factoring by a notion of procedural equivalence) be comprised of machines which take iteration or recursion as a basic computational paradigm?

Further reflection on their views would seem to suggest that their disagreement over 1) is not as substantial as it might initially appear. For instance, Moschovakis himself issues the following caveat about the significance of recursor isomorphism:

> [R]ecursor isomorphism is a very fine equivalence relationship which is not preserved by many useful algorithm transformations (optimizations, refinements, etc.), and we must often introduce 'coarser' equivalences . . . to express important facts of informal algorithm analysis. [56], p. 87

Some experimentation with formalizing algorithms as recursors bears out this point – i.e. in the course of constructing recursor representations of informally specified algorithms, we are often confronted with arbitrary choices about how to represent the data on which an algorithm operates together with its computational state as a partially ordered set. And since different choices will lead to different recursors among which the defined isomorphism relation need not

[48] Less formally, a recursor can be understood as the interpretation of a set of mutually recursive equation definitions $f_1(\vec{x}), \ldots, f_n(\vec{x})$ (or more generally as a *recursion scheme* in the sense of Greibach [34]) relative to a fixed-point semantics of the sort originally proposed by Scott [69] in the form of *domain theory*.

[49] Or at least this appears to be true of Gurevich's views up to at least his joint 2008 paper with Dershowitz [18]. In his joint subsequent 2009 paper with Blass and Dershowitz [7], it is argued that the program-based variant of the algorithms-as-abstracts view is unsustainable in virtue of our inability to define an appropriate equivalence relation $\approx$ on programs for reasons resembling those discussed in §4. For instance it repeatedly suggested in this paper that the relation between programs of "expressing the same algorithm" is subjective or interest relative. But since no account of why the same problem does not also beset the machine-based variant of this view is given in this paper, it is unclear at present whether Gurevich should be understood as continuing to support the ASM Thesis in its original form.

hold, it would seem that factoring representations of algorithms by recursor isomorphism does not lead to a satisfactory resolution to the problems of "grain" discussed in §5. It is, moreover, easy to see that isomorphic recursors will have the same running time complexity relative to the analysis Moschovakis provides. Hence both his view and Gurevich's predict that exact (as opposed to asymptotic) running time complexities are directly attributable to individual algorithms.

Moschovakis and Gurevich's proposals do, however, make different predictions about the identity conditions of certain recursive algorithms. Recall for instance, that we saw in §4.2.3 that in order to implement MergeSort as a RAM machine we needed to make a seemingly arbitrary decision about how the merge operations it performs in parallel are put into sequential order, leading to distinct implementations R_{left} and R_{right} which cannot stand in a simulation relation which simultaneously satisfies Milner's transitional and representational conditions. On the other hand, it is possible to canonically define a recursor α_{merge} whose operation can be understood to formalize the sort of parallel computation tree for MergeSort depicted in Figure 1. On this basis Moscovakis ([56], p. 90) suggests that α_{merge} is a more apt mathematical representation of MergeSort than iterators such as R_{left} and R_{right} deriving from sequential models of computation like $\mathfrak{R}$.

This example illustrates that the fundamental distinction between the proposals of Gurevich and Moschovakis is not so much that one prefers a "reductionist" ontology for algorithms while the other prefers an "abstractionist" one, but rather whether the sort of objects with which they seek to identify algorithms derive from a model which is based on iteration or recursion. In virtue of the argument I made against taking simulation to be an adequate analysis of computational equivalence in §4.3, the foregoing observations may reasonably be taken to suggest that Moschovakis's proposal fares better than Gurevich's in accounting for the identity conditions of recursive algorithms like MergeSort.[50]

There is, however, another sense in which the proposals of Moschovakis and Gurevich are alike in the sense that they depart in the same way from the strategy which I argued at the beginning of §5 constitutes the most promising way of vindicating algorithmic realism. In particular, rather than attempting to define an algorithm in terms of its implementations from the class of machines $\mathfrak{M}^1$ from the first machine class, they take individual algorithms to be defined only relative to the primitive operations in terms of which they are specified (it is such a specification of primitives which is typically taken to correspond to a "level of

[50] There is, however, some question of how general this advantage can be taken to be. For on the one hand, there are standard techniques which allow for the uniform translation between members of iterative and recursive models of computation (cf, e.g., the *flowchart* and *recursion scheme* models developed in [34]). And on the other, an open but widely believed conjecture in complexity theory (i.e. $\mathbf{NC} \neq \mathbf{P}$, cf. [33]) predicts that there is a wide class of naturally occurring computational problems (i.e. those which are complete for the class $\mathbf{P}$) which can only be solved by sequential algorithms. If this is the case, then problems like sorting for which their exist parallel algorithms which are more efficient than the most efficient sequential algorithm by which they can be solved by more than a scalar factor may represent only an exceptional case of the sort of problems we confront in mathematical practice.

abstraction" in the sense of Gurevich's formulation of the ASM thesis). Thus like the general definition of iterator introduced in §3, the ASM and recursor models are very different from models like $\mathfrak{T}_k$ or $\mathfrak{R}$. For whereas arbitrary mathematical operations may be used to define the states and transition functions of the former, the latter make available only very restrictive classes of functions and primitive data structures.

One potential advantage of the level-relative approach is borne out by the elegant reconstruction which Moscovakis ([56], p. 83-84) provides of the sort of informal running time analysis of MergeSort as would typically be given in algorithmic analysis. As his analysis is developed directly by reasoning about the properties of the primitive mathematical notions in terms of which this algorithm is specified, it hence avoids the necessity of reasoning about the details of how machines like R_{left} and R_{right} discussed in §4.2.3 operate on registers or stacks. But in the case of many sequential algorithms which operate (e.g.) on graphs, matrices, or polynomials, the same will also be true of the ASM model in virtue of the considerations described in §4.2.1 and 4.2.2 – i.e. since an ASM can be specified at an arbitrary "level of abstraction" (i.e. such that its steps are defined in terms of arbitrary mathematical operations, regardless of their underlying feasibility) it is also possible to simplify running time analyses by working with ASMs instead of (e.g.) RAM machines.

These considerations aside, however, the question also arises as to the utility of applying such analyses directly to "level-relative" models such as ASMs or recursors. For although it may be that an ASM M or recursor α which we employ to model an informally specified algorithm A will introduce *fewer* arbitrary properties into its mathematical representation than, say, an implementation R of A as a RAM machine, neither Moschovakis nor Gurevich provides an argument that M or α must be *unique*. As such, their proposals must still presumably face variants of the what-numbers-could-not-be problem introduced in §1. But much more seriously than this, the generality of the ASM and recursor theses is purchased precisely at the price of severing the link between our practice of assigning running time complexities to informally specified algorithms and the underlying complexity costs of models like $\mathfrak{R}$ and $\mathfrak{T}_k$.

As I argued in §3, it is our ability to implement algorithms with respect to these models which ultimately explains how it is that complexity theory and algorithmic analysis provide a means by which we can meaningfully measure and compare the utility of different computational procedures in practice. As such, it seems reasonable to conclude that the level-relative view of algorithms advocated by Moschovakis and Gurevich does not provide an account which makes good on all of the foundational aspirations of algorithmic realism described above.

6.2. **Algorithms, identify, and mathematical practice.** After initially stressing the importance of foundational questions about algorithms to our mathematical practices in §1, §4 and §5 of this paper have been largely devoted to arguing that despite its evident allure, there is no apparent means of substantiating the algorithmic realist's claim that algorithms are freestanding mathematical objects which is compatible with the practice of complexity theory and algorithmic analysis. In order to better orient ourselves with respect to how the diminished prospects of algorithmic realism bear on how one might ultimately go about answering (Q1) and (Q2) from §1, it will be useful to first recall that much of the evidence that was initially marshaled in favor of the plausibility of this view derives from our willingness to *speak* of algorithms as if they were abstract objects on a par with those studied in mathematics – e.g. natural numbers, graphs, groups, etc. This is evident not only from our practice of introducing singular terms (e.g. in the form of algorithmic names) to refer to individual procedures, but also in our willingness to quantify over certain classes of algorithms as exemplified by statements like (1), (4), and (6).

Note, however, that another linguistic phenomenon which is often taken to be diagnostic of whether a class of expressions $t_1, t_2, \ldots$ is properly regarded as denoting objects is our willingness to use them in identity statements of the forms $t_i =$ *the* φ, $t_i = t_j$, or $t_i \neq t_j$. We have seen some evidence that algorithmic names can be used in the first of these schema in the course of considering statements such as

(10) a) InsertionSort = the algorithm expressed by program Π
b) MergeSort = the algorithm implemented by machine M

Proponents of the algorithms-as-abstracts view will presumably take such statements as paradigmatic of the way in which algorithmic names are introduced into our mathematical language.

Based on their grammatical form, it might seem reasonable to take these statements to be of a piece with the following:

(11) a) 3 = the number of roots of $x^3 - 6x^2 + 11x - 6$
b) D_4 = the group represented by $\langle x, y \mid x^4 = 1, y^2 = 1, yxy = x^{-1} \rangle$

But in the case of *bona fide* mathematical items like natural numbers and groups it is also typically easy to find within our discourse statements instance of another form of identify statement – i.e. *the* $\varphi_1 =$ *the* φ_2 – as exemplified by

(12) a) the number of primes less than 6 = the number of roots of $x^3 - 6x^2 + 11x - 6$
b) the group represented by $\langle x, y \mid x^4 = 1, y^4 = 1, xyxy = 1 \rangle$ = the group represented by $\langle x, y \mid x^4 = 1, y^2 = 1, yxy = x^{-1} \rangle$

It is the analogs of such statements involving the expressions "the algorithm implemented by M" and "the algorithm expressed by Π" which the two variants of the algorithms-as-abstracts views suggest ought to be regarded as fundamental to our general understanding of what it is to be an algorithm. In particular, it precisely this kind of statement which figures on the lefthand side of the principles (MP) or (PP) which proponents of this view regard as an implicit definition of algorithmic identity. It is hence a significant observation that our everyday computational-cum-mathematical discourse seems to provide few examples of such statements about which we appear to have firm convictions. As the case described in §4.3.2 attests, we do occasionally attempt to adjudicate the status of statements like "the algorithm implemented R_{left} = the algorithm implemented by R_{right}". But it would seem that our judgements about the truth or falsity of such statements is conventional at least to the extent that we do not expect that their truth values are fixed relative to other pre-established mathematical facts. On the other hand, it seems that not only do we regard statements like (12a,b) as potentially informative in the sense that they admit to non-trivial proof, but also that their truth values are fixed in advance by our prior understanding of the meanings of the expressions in terms of which they are formulated.

Reflection on this point goes a fair distance towards explaining why we do not need to be algorithmic realists in order to explain the manifest utility of subjects like complexity theory and algorithmic analysis. For, recall the example of the algorithm LUCAS discussed in §1. As we saw, it was by applying this algorithm not only was Lucas able to establish the primality of m_{127}. Nonetheless, there is no evident sense in which the algorithm LUCAS itself figures in the proposition expressed by "m_{127} is prime". Rather, the algorithm functions as an auxiliary device which aids in our demonstration of the theorem.

Note, however, that it is via a similar route by which algebraic entities like groups also began their life in mathematics – i.e. as auxiliary devices figuring in the proof of purely number theoretic propositions.[51] Although the contemporary definition of a group had not been formulated at the time which what we would now identify as algebraic methods started to be used, such structures are now universally recognized as *bona fide* mathematical objects. There thus appears to be little reason to regard an algebraic proof of a result in number theory to be in any sense "extra-mathematical". As we have seen, algorithmic realism seeks to secure the same status for results derived by computational methods by assimilating algorithms into the framework of classical mathematics.

If algorithmic realism were true, it thus seems reasonable to think that we should regard the status of algorithms in mathematics as something like that of groups. As we have seen, the general notion of algorithm has a considerably older pedigree than that of group. And although it is somewhat newer in origin than abstract algebra, we also do possess a theory which takes algorithms as its ostensible subject matter – i.e. algorithmic analysis. It is notable, however, that no consensus has arisen *within* this field as to the appropriate criterion of identity for algorithms or even as to the form such a condition might take. In comparing algorithmic analysis and group theory, it is thus reasonable to wonder why the development of the latter prompted the formulation of a family of notions of structural equivalence (e.g. homomorphism, isomorphism, embedding, etc.) for groups while no family of similar notions has developed in the context of algorithmic analysis to serve the role of canonically relating one algorithm (or representation thereof) to another.

[51] A prototypical example is Gauss's [30] proof of Fermat's Little Theorem. We would now describe this proof as relying on the fact that the set of integers $\{1, \ldots, p\}$ form a group under multiplication and that this group is isomorphic to the cyclic group $\mathbb{Z}_p$. However, no mention of either group appears in the statement of theorem (i.e. "if p is prime, then for any integer a, $a^p - a$ is divisible by p") nor had the notion of a group been defined at the time of Gauss's proof.

The answer to this question ultimately seems straightforward: the primary significance we place in the discovery of an algoriothm A is likely to be in applying it to solve instances of an antecedently identified mathematical problem – e.g. computing the values of a function $f(x)$. For this reason, our interest in A is also likely to be exhausted by proving its correctness with respect to $f(x)$ and possibly of comparing its efficiency with that of other means of computing the values of this function.

If we are to perform the latter task in a manner which is responsive to the analysis of feasibility provided by complexity theory and algorithmic analysis, it seems that we have no choice but to work with a representation M of A drawn from a model $\mathfrak{M}$ in the first machine class. When working with such models, we have also seen that the implementation problem can become genuinely non-trivial – i.e. based on the informal specification of A, it may not be initially obvious how to employ the basic operations and data structures made available by $\mathfrak{M}$ to construct M so as to mimic A's operation in a step-by-step manner.

Given that we are able to construct such a machine, however, it is also very likely that we will also be able to construct other formally distinct $M' \in \mathfrak{M}$ which implement A in a different manner (e.g. by using a different representation of the data structures on which M operates as discussed in §5). However, once we have constructed our initial representation M (thereby making it possible to give rigorous proofs both of A's correctness and its running time complexity), our interest will typically shift from the properties of M to that of employing it to compute the values of $f(x)$. Our willingness to acknowledge that M and M' are both implementations of A notwithstanding, it would seem that mathematical considerations alone never dictate that we return to consider what (if anything) about the structural properties of M and M' accounts for the fact that we are willing to regard them as implementations of the same algorithm.

Thus although there will be many instances in which our practices may dispose us to regard two programs or machines as representing the same algorithm, the question of whether such judgements can be made precise does not seem to be of any independent mathematical significance. And there thus seems to be no abiding reason why ontological questions of type (Q1) need to be answered before we can provide detailed answers to epistemological questions of type (Q2). At least to date, the story of algorithmic realism might thus be taken to represent a cautionary tale for the study of ontological commitment. For it is indeed difficult to deny that our discourse is highly suggestive of the fact that parts of theoretical computer science are engaged in the study of a class of procedural entities which are closely related to but yet somehow distinct from those studied in classical mathematics. But at the same time, a more detailed appraisal of the methodologies of the relevant subjects suggests that there may be no way to take this language at face value without contravening some of the assumptions about the nature of algorithms in which these fields themselves seems to be grounded.

This conclusion may at first sound like a negative one with respect to the proposal formulated in §1 that the general notion of algorithm deserves a place alongside concepts such as number, set, and function as a substrates of contemporary mathematics. However, it should now also be evident that many of the considerations which have been touched on in this paper point to the import of questions refining (Q2) in regard to the practice of algorithmic analysis and complexity theory. Some of these are as follows:

(Q3) What does it mean for a mathematical model to provide an accurate representation of the constraints we face in concretely embodied computation? What does it mean to say that a function is *intrinsically difficult* to compute given that it is recursive or for one recursive function to be more difficult to compute than another? How can the practical utility of different effective methods for solving the same mathematical problem be objectively measured and compared?

It would seem that such questions must be subjected to closer scrutiny before it can reasonably be said that we have a philosophically satisfactory account of the growing role which computational techniques play in mathematical practice. But beyond what has been said in §3, a more thorough consideration of these issues must await another occasion.

References

[1] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs.* MIT Press, Cambridge, MA, 1985.

[2] K. Appel, W. Haken, et al. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

[3] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. Part II: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 1977.

[4] G. Bealer. A solution to Frege's puzzle. In *Philosophical Perspectives, Volume 7: Language and Logic*, pages 17–60. Blackwell Publishers, 1993.

[5] P. Benacerraf. What numbers could not be. *The Philosophical Review*, 74(1):47–73, 1965.

[6] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic.* Cambridge University Press, Cambridge, England, 2001.

[7] A. Blass, N. Dershowitz, and Y. Gurevich. When are two algorithms the same? *The Bulletin of Symbolic Logic*, 15(2):145–168, 2009.

[8] G. Boolos. Is Hume's Principle Analytic? In *Language, Thought, and Logic: Essays in Honour of Michael Dummett*, pages 245–261. Oxford University Press, Oxford, 1997. Edited by Richard G. Heck Jr.

[9] B. Buchberger. A theoretical basis for the reduction of polynomials to canonical forms. *ACM SIGSAM Bulletin*, 10(3):19–29, 1976.

[10] J. P. Burgess. *Fixing Frege.* Princeton Monographs in Philosophy. Princeton University Press, Princeton, 2005.

[11] R. Carnap. *Meaning and Necessity.* The University of Chicago Press, Chicago, 1947.

[12] A. Church. A formulation of the logic of sense and denotation. *Structure, Method, and Meaning: Essays in Honor of Henry M. Sheffer, Liberal Arts Press, New York*, pages 3–24, 1951.

[13] A. Church. Intensional isomorphism and identity of belief. *Philosophical Studies*, 5(5):65–73, 1954.

[14] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms.* MIT Press, second edition, 2005.

[15] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms.* Springer-Verlag, New York, 1992.

[16] M. J. Cresswell. *Structured meanings: The semantics of propositional attitudes.* The MIT Press, 1985.

[17] W. Dean. *What Algorithms could not be.* PhD thesis, Rutgers University, 2007.

[18] N. Dershowitz and Y. Gurevich. A natural axiomatization of computability and proof of Church's Thesis. *The Bulletin of Symbolic Logic*, 14(3):299–350, 2008.

[19] M. Detlefsen and M. Luker. The four-color theorem and mathematical proof. *The Journal of Philosophy*, 77, 1980.

[20] D. Du and K. Ko. *Theory of computational complexity.* John Wiley, 2000.

[21] M. Dummett. Frege's distinction between sense and reference. In *Truth and other enigmas*, pages 116–144. Harvard University Press, 1978.

[22] L. Floridi. Philosophy and computing. an introduction. *Ethics and Information Technology*, 2(2):137–138, 2000.

[23] C. Foster. *Algorithms, abstraction and implementation: Levels of detail in cognitive science.* Academic press, 1992.

[24] G. Frege. *Die Grundlagen der Arithmetik.* Koebner, Breslau, 1884.

[25] G. Frege. *Grundgesetze der Arithmetik: begriffsschriftlich abgeleitet.* Pohle, Jena, 1893, 1903. Two volumes. Reprinted in [26].

[26] G. Frege. *Grundgesetze der Arithmetik.* Olms, Hildesheim, 1962.

[27] G. Frege. Compound thoughts. *Mind*, 72(285):1–17, 1963.

[28] R. Gandy. Church's Thesis and principles for mechanisms. In H. K. J. Barwise and K. Kunen, editors, *The Kleene Symposium*, volume 101, pages 123–148. North Holland, 1980.

[29] M. Garey and D. Johnson. *Computers and intractability. A guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[30] C. F. Gauss. *Disquisitiones Arithmeticae.* Springer, New York, 1986. Translated by Arthur A. Clarke.

[31] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[32] K. Gödel. On undecidable propositions of formal mathematical systems (with postcript dated 1964). In S. Feferman, editor, *Kurt Gödel Collected Works. Vol. I. Publications 1929-1936*, pages 346–371. Oxford University Press, 1986.

[33] R. Greenlaw, H. Hoover, and W. Ruzzo. *Limits to parallel computation:* ***P****-completeness theory.* Oxford University Press, USA, 1995.

[34] S. A. Greibach. *Theory of program structures: schemes, semantics, verification*, volume 36 of *Lecture Notes in Computer Science.* Springer, 1975.

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

[36] Y. Gurevich. Evolving algebras 1993: Lipari guide. *Specification and validation methods*, pages 9–36, 1995.

[37] Y. Gurevich. The Sequential ASM Thesis. *Bulletin of the EATCS*, 67:93–124, 1999.

[38] B. Hale and C. Wright. *The reason's proper study.* Oxford University Press, Oxford, 2001.

[39] D. Harel. *Algorithmics: the spirit of computing.* Addison-Wesley, Reading, Massachusetts, 2006.

[40] J. F. Horty. *Frege on definitions: a sase study of semantic content.* Oxford, 2007.

[41] D. Knuth. *The art of computer programming*, volume I: Fundamental algorithms. Addison Wesley, 1973.

[42] D. Knuth. *The art of computer programming*, volume II: Seminumerical algorithms. Addison Wesley, 1973.

[43] D. Knuth. *The art of computer programming*, volume III: Searching and sorting. Addison Wesley, 1973.

[44] D. Knuth. Big omicron and big omega and big theta. *ACM Sigact News*, 8(2):18–24, 1976.

[45] D. E. Knuth. Algorithm and program; information and data. *Communications of the ACM*, 9(9):654, 1966.

[46] A. Kolmogorov and V. Uspensky. On the notion of algorithm. *Uspekhi Mat. Nauk*, 8(4/56):175–176, 1953.

[47] A. Kolmogorov and V. Uspensky. To the definition of algorithms. *Uspekhi Mat. Nauk*, 13(4):3–28, 1958.

[48] G. Kreisel. Some reasons for generalizing recursion theory. In R. Gandy and C. Yates, editors, *Logic Colloquium 69*. North-Holland, Amsterdam, 1971.

[49] D. Marker. *Model Theory*, volume 217 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, 2002.

[50] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[51] R. Milner. An algebraic definition of simulation between programs. Technical Report CS-205, Department of computer science Stanford University, 1971.

[52] J. Mitchell. *Foundations for programming languages*, volume 1. MIT press Cambridge, MA, 2002.

[53] R. Montague. The proper treatment of quantification in ordinary english. In J. M. J. Hintikka and P. Suppes, editors, *Approaches to Natural Language: Proceedings of the 1970 Stanford Workshop on Grammar and Semantics*, pages 221–242. Reidel, 1970.

[54] Y. N. Moschovakis. Abstract recursion as a foundation for the theory of algorithms. *Computation and proof theory*, pages 289–364, 1984.

[55] Y. N. Moschovakis. Sense and denotation as algorithm and value. In *Logic Colloquium '90*, pages 210–249. North Holland, 1990.

[56] Y. N. Moschovakis. On the founding of a theory of algorithms. In H. Dales and G. Oliveri, editors, *Truth in mathematics*, pages 71–104. Clarendon Press, 1998.

[57] Y. N. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited—2001 and Beyond*, pages 919–936. Springer-Verlag, Berlin, 2001.

[58] F. Nielson and H. R. Neilson. *Semantics with applications - a formal introduction.* John Wiley and Sons, New York, NY, 1992.

[59] P. Odifreddi. *Classical recursion theory. Vol. I*, volume 125 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 1989.

[60] C. Papadimitriou. *Computational complexity.* Addison-Wesley, New York, 1994.

[61] A. M. Pitts. Operationally-based theories of program equivalence. *Semantics and Logics of Computation*, 14:241, 1997.

[62] Plato. *Republic.* Hackett Publishing Company, Indianapolis, 2004. Translated by C.D.C. Reeve.

[63] G. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19. *Computer Science Department, Aarhus University*, 1981.

[64] E. Post. Finite combinatory processes-formulation 1. *Journal of Symbolic Logic*, 1(3):103–105, 1936.

[65] Z. Pylyshyn. *Computation and cognition: toward a foundation for cognitive science.* The MIT Press, Cambridge, Massachusetts, 1984.

[66] H. Rogers. *Theory of recursive functions and effective computability.* MIT Press, Cambridge, MA, second edition, 1987.

[67] D. Sangiorgi and J. Rutten. Origins of bisimulation and coinduction. In *Advanced Topics in Bisimulation and Coinduction.* Cambridge University Press, 2011.

[68] A. Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, 1980.

[69] D. Scott. Outline of a mathematical theory of computation. Technical report, Oxford University Computing Laboratory, 1970.

[70] R. W. Sebesta. *Concepts of programming languages*, volume 4. Addison Wesley, 2002.

[71] R. Sedgewick. *Algorithms.* Addison-Wesley, 1983.

[72] S. Shapiro. *Philosophy of mathematics: structure and ontology.* Oxford University Press, Oxford, 1997.

[73] W. Sieg. On computability. In A. Irvine, editor, *Philosophy of mathematics*, volume 4 of *Handbook of the philosophy of science*, pages 549–630. North Holland, 2009.

[74] W. Sieg and J. Byrnes. K-graph machines: generalizing Turing's machines and arguments. *Gödel'96: logical foundations of mathematics, computer science, and physics*, pages 98–119, 2001.

[75] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, USA, 1981.

[76] P. Tichý. *The foundations of Frege's logic.* Walter de Gruyter, 1988.

[77] R. Turner and A. Eden. The philosophy of computer science. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Summer 2009 edition, 2009.

[78] T. Tymoczko. The four-color problem and its philosophical significance. *Journal of Philosophy*, 76(57-83), 1979.

[79] P. van Emde Boas. Machine models and simulations. In J. Van Leeuwen, editor, *Handbook of theoretical computer science (vol. A): algorithms and complexity.* MIT Press, Cambridge, MA, 1990.

[80] J. Van Heijenoort. Frege on sense identity. *Journal of Philosophical Logic*, 6(1):103–108, 1977.

[81] C. Wright. *Frege's conception of numbers as objects*, volume 2 of *Scots Philosophical Monographs.* Aberdeen University Press, Aberdeen, 1983.

[82] N. S. Yanofsky. Towards a definition of an algorithm. *Journal of Logic and Computation*, 21(2):253–286, 2011.

[83] E. N. Zalta. *Intensional logic and the metaphysics of intentionality*, volume 92. MIT Press Cambridge, 1988.

E-mail address: W.H.Dean@warwick.ac.uk