The Bulletin of Symbolic Logic
Volume 14, Number 3, Sept. 2008

A NATURAL AXIOMATIZATION OF COMPUTABILITY AND PROOF OF CHURCH'S THESIS

NACHUM DERSHOWITZ AND YURI GUREVICH

Abstract. Church's Thesis asserts that the only numeric functions that can be calculated by effective means are the recursive ones, which are the same, extensionally, as the Turing-computable numeric functions. The Abstract State Machine Theorem states that every classical algorithm is behaviorally equivalent to an abstract state machine. This theorem presupposes three natural postulates about algorithmic computation. Here, we show that augmenting those postulates with an additional requirement regarding basic operations gives a natural axiomatization of computability and a proof of Church's Thesis, as Gödel and others suggested may be possible. In a similar way, but with a different set of basic operations, one can prove Turing's Thesis, characterizing the effective string functions, and—in particular—the effectively-computable functions on string representations of numbers.

Contents

Received July 10, 2007.

2000 *Mathematics Subject Classification.* 03D10.

Key words and phrases. effective computation, recursiveness, computable functions, Church's Thesis, Turing's Thesis, abstract state machines, algorithms, encodings.

First author's research supported in part by the Israel Science Foundation under grant no. 250/05.

1079-8986/08/1403-0002/$6.20

We can write down some axioms about computable functions
which most people would agree are evidently true.
It might be possible to prove Church's Thesis from such axioms.
—*Joseph Shoenfield* (1993)

§1. Introduction: Effectivity. Church formulated the thesis bearing his name to address a very fundamental issue in modern logic and mathematics.

1.1. Historical background. In the beginning of the twentieth century, Hilbert famously introduced fundamental questions of decidability to mathematics:

> **[Problem] 10. Determination of the solvability of a Diophantine equation.** Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.[1] [44]
>
> The Entscheidungsproblem [decision problem for first-order logic] is solved when we know a procedure that allows for any given logical expression to decide by finitely many operations its validity or satisfiability.... The Entscheidungsproblem must be considered the main problem of mathematical logic.... The solution of the

[1]"**10. Entscheidung der Lösbarkeit einer Diophantischen Gleichung.** Eine *Diophantische* Gleichung mit irgend welchen Unbekannten und mit ganzen rationalen Zahlencoefficienten sei vorgelegt: man soll ein Verfahren angeben, nach welchem sich mittelst einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist."

> Entscheidungsproblem is of fundamental significance for the theory of all domains whose propositions could be developed on the basis of a finite number of axioms.[2] [45, p. 73ff.]

Hilbert was looking for well-defined procedures that would solve each instance of a problem, positively or negatively, by applying a finite number of operations. He did not have a formal notion of which operations would be reasonable in this context and which not, but clearly he was looking for operations that could be carried out by a mathematician acting mechanically, sans ingenuity. "We assume that we have the capacity to name things by signs, that we can recognize them again. With these signs we can then carry out operations that are analogous to those of arithmetic and that obey analogous laws" (Hilbert, quoted in [95]).

With regard to *numeric* functions, that is, functions from the natural numbers to the natural numbers,[3] Church suggested in 1936 that the recursive functions, which had been defined by Gödel earlier that decade (based on a suggestion of Herbrand's), adequately capture the intended concept of function computable by a finite procedure.[4] He adopted this identification in the form of a definition of effectiveness, and wrote [23, pp. 346, 356]:

> The purpose of the present paper is to propose a definition of effective calculability which is thought to correspond satisfactorily to the somewhat vague intuitive notion. . . .
>
> We now define the notion . . . of an *effectively calculable* function of positive integers by identifying it with the notion of a recursive function of positive integers (or of a λ-definable function of positive integers).

Church identified "the commonly used term 'effectively calculable'" [22, p. 40] in reference to a function, that is, the existence of "an algorithm" for

[2]"Das Entscheidungsproblem ist gelöst, wenn man ein Verfahren kennt, das bei einem vorgelegten logischen Ausdruck durch endlich viele Operationen die Entscheindung über die Allgemeingültigkeit bzw. Erfüllbarkeit erlaubt. Das Entscheidungsproblem muss als das Hauptproblem der mathematischen Logik bezeichnet werden. . . . Die Lösung des Entscheidungsproblems ist für die Theorie aller Gebiete, deren Sätze überhaupt einer logischen Entwickelbarkeit aus endlich vielen Axiomen fähig sind, von grundsätzlicher Wichtigkeit."

[3]Nowadays (e.g., [82]) it is more common to work with the nonnegative integers, including zero, though in the past (e.g., [23]) it was common to deal only with the positive integers. The difference is immaterial for discussions of effectiveness of computation.

[4]Already in 1935, Church thought (in correspondence to Bernays, quoted in [94, p. 155]) that the "results of Kleene [regarding the λ-calculus] are so general and the possibilities of extending them apparently so unlimited that one is led to the conjecture that a formula can be found to represent any particular constructively defined function of positive integers whatever." Kleene is reported [2, p. 185] to have said much later, "I would like to be able to say that, at the moment of discovering how to lambda define the predecessor function, I got the idea of Church's Thesis. But I did not, Church did." In retrospect, Kleene (see [51, p. 62]) and Rosser (see [84]) felt that the lambda calculus might have been a more appropriate basis for characterizing effective computability.

computing the value of that function for any arguments [23, p. 356], with the specific requirement that there be recursion equations by means of which the evaluation of the function can be effected. Only with such a formalization of effectivity in hand could one prove "absolute" undecidability results, namely, that no algorithmic solution whatsoever exists for some particular problem—like the Entscheidungsproblem—as Church set out to show in his papers.

Church was roundly criticized by Post [75, p. 105] for hiding a debatable formalization of "effective calculability" behind a *definition*:

> The work done by Church and others carries this identification considerably beyond the working hypothesis stage. But to mask this identification under a definition hides the fact that a fundamental discovery in the limitations of mathematicizing power of Homo Sapiens has been made and blinds us to the need of its continual verification.

For Post [76, p. 418], it "is not a matter of mathematical proof but of psychological analysis of the mental processes involved in combinatory mathematical processes".

Turing, in his seminal 1936 paper [107], analyzed human computation "from the bottom up", building complex procedures from the most primitive of operations on single symbols. He asserted that computation proceeds by sequential symbol manipulation ("Computing is normally done by writing certain symbols on paper"), and argued that any such computation can be mimicked by a symbolic computation with the following characteristics:

- Deterministic behavior: "The behaviour of the [human] computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at that moment."
- Finitely many internal states: "If we admitted an infinity of states of mind, some of them will be 'arbitrarily close' and will be confused.... [T]his restriction is not one which seriously affects computation, since the use of more complicated states of mind can be avoided by writing more symbols on the tape."
- A finite symbol space: "If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrary small extent. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols."
- Finite observability and local action: "Let us imagine the operations performed by the computer to be split up into 'simple operations' which are so elementary that it is not easy to imagine them further divided.... We may suppose that there is a bound... to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations."

- Linear external memory: "The two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper."

When Church learned of Turing's work,[5] he conceded that Turing's machines have "the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately" [24, p. 43]. He responded to Post, saying: "To define effectiveness as computability by an arbitrary machine, subject to restrictions of finiteness, would seem an adequate representation of the ordinary notion" [25].

A few years later, Kleene, a student of Church, reformulated Church's contention that the recursive functions and the effective numeric functions are one and the same as a "thesis" [48, p. 60], [49, p. 300]:[6]

> **[Church's] Thesis I.** *Every effectively calculable function* (*effectively decidable predicate*) *is general recursive.*

Again, "effective" is meant in the "vague intuitive" sense of computable by humans acting in an algorithmic fashion. In order to accurately match the effect of computations, one must allow functions to be partial, as Kleene subsequently did [49, p. 332]:[7]

> **[Church's] Thesis I†.** *Every partial function which is effectively calculable* (*in the sense that there is an algorithm by which its value can be calculated for every n-tuple belonging to its range of definition*) *is . . . partial recursive.*[8]

1.2. Current status. Though Kleene spoke of this thesis as unprovable ("Since our original notion of effective calculability . . . is a somewhat vague intuitive one, the thesis cannot be proved" [49, p. 317]), he did present evidence in its favor [49, Chaps. XII–XIII]. Three main lines of argument have been adduced in support of Church's Thesis (already in [49, pp. 319–323]):

1. In years of experience, all the many effective computational models that have been investigated (starting with the lambda calculus and continuing on down to the latest programming languages) have been shown to compute only partial recursive functions.

[5]Church was Turing's official Ph.D. advisor at Princeton.

[6]The converse, namely, that all recursive functions are effectively computable, is almost universally held. (Goodstein is an exception [36, n. 29]; Kalmár is not [6, n. 10].) Kleene claims to have proved this (e.g., [50, p. 300]), though some (e.g., Folina in [31]) contend that such an argument, inasmuch as it too involves the informal notion of effectiveness, should not be designated a "proof". Cf. Gödel's assertion [38, p. 44] that primitive recursive functions "can be computed by a finite procedure".

[7]"Gödel points out that the precise notion of mechanical procedures is brought out clearly by machines producing partial rather than general recursive functions." Reported in [112, p. 84].

[8]The omitted word is "potentially", which is nowadays left unstated but understood.

2. A vast number of computational models and a multitude of variants, all yield the exact same class of functions. In particular, Turing [108] showed that his machines computed the same functions as did the lambda calculus. "The proposed characterizations of Turing and of Kleene, as well as those of Church, Post, Markov, and certain others, were all shown to be equivalent" (Rogers' *Basic Result* [82, pp. 18–19]).[9]
3. Turing's analysis of "the sorts of operations which a human computer could perform, working according to preassigned instructions" showed that these can be simulated by his machines [49, p. 321]. Turing [108] modestly wrote: "The identification of 'effectively calculable' functions with [Turing-] computable functions is possibly more convincing than an identification with the λ-definable or general recursive functions. For those who take this view the formal proof of equivalence provides a justification for Church's calculus, and allows the 'machines' which generate computable functions to be replaced by the more convenient λ-definitions."

The first, "heuristic" argument is relatively unconvincing ("heuristic" is Kleene's word). History is full of examples of delayed discoveries. Aristotelian and Newtonian mechanics lasted much longer than the seventy years that have elapsed since Church proposed identifying effectiveness with recursiveness, but still those physical theories were eventually found lacking. As Barendregt writes [2]:

> One may wonder why doubting Church's Thesis is not a completely academic question. This becomes clear by realizing that [Skolem in 1923] had introduced the class of primitive recursive functions that for some time was thought to coincide with that of the intuitively computable ones. But then [Ackermann in 1928] showed that there is a function that is intuitively computable but not primitive recursive.

The empirical second argument, from "confluence" of models, is also weak, and has been deemed so by Kreisel [58, p. 144] and Mendelson [69, p. 228, n. 4]. Clearly the notion captured by these equivalent models is a robust one, but there still could be a class of effective algorithms not captured by it. As Kreisel [58, p. 144] put it: "What excludes the case of a *systematic* error?"

The third argument is by far the strongest, so strong, in fact, that Gödel [39, p. 168] thought the idea "that this [the recursive functions] really is the correct definition of mechanical computability was established beyond any doubt by Turing".[10] For an analysis of Gödel's opinion in this matter, see [98]; see,

[9] Even the solution sets of Diophantine equations are exactly the recursively-enumerable subsets of the integers [67].

[10] So, too, Bernays in a letter to Church, quoted in [94, p. 169], "[Turing] seems to be very talented. His concept of computability is very suggestive and his proof of equivalence of this

also, Gandy [36, p. 72] in this connection. Turing's analysis was qualitatively different from those of his predecessors. Soare [104] goes so far as to write: "It was Turing *alone* who ... gave the first convincing formal definition of a computable function ... [and] proved that the informal notion coincided with this formal one" (emphasis in the original). Still, though the grand sweep of Turing's argument is overwhelming, there remain weaknesses when it comes to details. For example, the universality of Turing's supposition that "the number of states of mind which need be taken into account is finite" is debatable. Also, how certain is it that each and every elaborate data structure used during a computation can be encoded as a string, and its operations simulated by effective string manipulations? In any event, it is non-trivial to reduce Turing's analysis to a few general axioms.

Subsequent models of computation did not add much force to Turing's arguments, with the possible exception of Kolmogorov's model [56, 57], which, according to Leonid Levin [personal communication],[11] was inspired by an analysis of computation in physical space-time. This model partly addresses the issue of computations that compute with data other than mere strings of symbols. But one can only guess at the analysis of computations that was in Kolmogorov's head. Kolmogorov machines, and other variants of the pointer machine, do provide greater fidelity to algorithmic behavior than do Turing machines; see [10]. But Kolmogorov's published work does not delve into philosophical motivations for, or implications of, his model. In any event, an algorithm need not fit the constraints placed on the states of Kolmogorov's machines. (See Section 7.1.)

Hence, it remains of importance to provide a small number of convincing postulates in support of Church's Thesis. Indeed, Gödel has been reported (by Church in a letter to Kleene cited by Davis in [28]) to have believed "that it might be possible ... to state a set of axioms which would embody the generally accepted properties of [effective calculability], and to do something on that basis". As explained by Shoenfield (and partially quoted in the opening tag line) [92, p. 26]:

> It may seem that it is impossible to give a proof of Church's Thesis. However, this is not necessarily the case.... In other words, we can write down some axioms about computable functions which most people would agree are evidently true. It might be possible to prove Church's Thesis from such axioms.... However, despite strenuous efforts, no one has succeeded in doing this (although some interesting partial results have been obtained).

Kalmár [46] and Rogers [82, p. 20] (and, more recently, Folina [31]) argued against provability of the thesis, while Gandy [35] and Mendelson [69, 70]

notion with your λ-definability gives a stronger conviction of the adequacy of these concepts for expressing the popular meaning of 'effective calculability'."

[11] Levin was Kolmogorov's student.

(along with [88, 90, 60, 94, 6]) argued in favor of the possibility of axiomatizing effectivity. Kreisel described the discovery of "evident axioms about constructive functions" as "one of the really important open problems" [58] and "one of the more feasible problems at the present time" [59]. We propose just such an axiomatization of effective computation in the sections that follow.

For more on the history of Church's Thesis, see Kleene's retrospective [51], the historical remarks of Rosser in [84], and the article by Davis [28]; all three of the authors were students of Church.

1.3. Sketch of axioms. The first issue that needs to be addressed when axiomatizing effective computation is: What kind of object is a "computation"? Once we agree that it is some sort of state transition system (Postulate I in what follows), we need to formalize the appropriate notions of "state" and of "transition". To model states, we take the most generic of mathematical objects, namely, logical structures (Postulate II). To ensure that each transition step is effective, we require only that it not entail an unbounded amount of exploration of the current state (Postulate III). Finally, we need to make sure that a computation does not start out with any magical abilities (Postulate IV). We will demonstrate that under these very natural and general hypotheses regarding algorithmic activity, which certainly suffice for the computation of all recursive functions, the recursiveness of the computed function is in fact guaranteed.

More precisely, but still informally, the postulates say the following about algorithms:

I. *An algorithm determines a sequence of "computational" states for each valid input.*
II. *The states of a computational sequence are structures. And everything is invariant under isomorphism.*
III. *The transitions from state to state in computational sequences are governable by some fixed, finite description.*
IV. *Only undeniably computable operations are available in initial states.*

Postulates I–III are called the "Sequential Postulates" [42]. They axiomatize (deterministic, sequential) algorithms in general, not only those for computable functions; they apply equally to algorithms dealing with complex numbers, say, as to those for integers only. Postulate IV, which will be fleshed out later, ensures that an algorithm is not endowed from the outset with uncomputable oracles, such as infinite precision operations on real numbers, or a solvability decider for Diophantine equations. We will show in this paper that Church's Thesis provably follows from these four postulates.

In the next section, we formulate the three Sequential Postulates rigorously, motivate each of them, and adduce support for them from the classical literature. In Section 3, we recall the definition of abstract state machines [41], and the fact that they emulate any algorithm obeying those postulates [42].

(See also [80].) These machines will play a central part in our proof. Then, in Section 4, we turn Church's Thesis into a precise mathematical statement and explain why the fact that only the recursive functions can be calculated by effective means follows provably from our four postulates. In the same way, as shown in Section 5, it follows that relative effectiveness (modulo oracles) and relative recursiveness are equivalent.

Church supplied examples to argue that a decision problem in a non-numerical domain could also "be interpreted as a problem in elementary number theory", since properties in other domains "can be described in number-theoretic terms" [23, p. 345]. Accordingly, in Section 6, we extend our analysis to deal with such algorithms that manipulate additional objects, besides numbers, like strings of symbols. There, we formalize what it means to "be described", without recourse to any intuitive notion of effectiveness of encodings. Furthermore, individual steps can be as large as one wishes, as long as they can be guaranteed to be effective. This analysis, turned around so that everything is viewed in terms of strings, also yields an axiomatization and proof of Turing's Thesis, namely, that every effective string-to-string (partial) function is Turing-computable.

1.4. Preliminary discussion. Turing's analysis [107] and its subsequent generalizations by Post [77] and Kolmogorov [56, 57] are on an informal level. Gandy [35] was the first to attempt an axiomatization, and was followed in this endeavor by Sieg [93, 94, 97, 96, 99, 100]; though their axioms are formal, they are expressed on the level of a specific representation of states (namely, hereditarily finite sets). In contrast, our axioms of effective computation are, at the same time, formal and generic. They are formal in the sense that they are precise mathematical statements about computation sequences. Our postulates are generic, in that they are expressed in terms of computation sequences with arbitrary states and arbitrary programmable transitions. Each transition corresponds exactly to a single step in a given algorithmic process. In these ways, our proposed axiomatization improves upon its predecessors.

All attempts to axiomatize computation inevitably formalize the notion of state as a mathematical object, and, as such, involve some measure of abstraction from, and representation of, the physical realities of human or machine computations. Indeed, science in general is impossible without modeling. Mathematics, in particular, deals with mathematical objects, not physical ones. To quote Kleene [53, p. 30]:

> Mathematicians deal with idealized systems of objects, obtained by extrapolating for the purposes of their thought from what people encounter in the real world. They imagine the infinite sequence 0, 1, 2, ... of the natural numbers, and thus get a beautiful theory with an elegant logical structure, though in actual counting of discrete objects we can never use more than finitely many of them.

Regarding the modeling of solvable and unsolvable puzzles, for instance, Turing wrote [110, p. 11]: "If one wants to treat the problem seriously and systematically one has to replace the physical puzzle by its mathematical equivalent." So the right question is whether our representation goes beyond what is absolutely necessary for formalization.

One abstraction, one that we inherit from all classical analyses, is that transitions are discrete (Postulate I). An additional abstraction we make is that states are structures. Our claim, expressed in the Abstract State Postulate (Postulate II), is that using structures is the unavoidable, bare minimum that is necessary for formalizing actual computational states.

There are strong arguments to support this claim. In the first place, the history of mathematical endeavor is on our side. Long experience supports the contention that any static mathematical reality can be viewed as a structure, without resorting to coding, translation, or the like. In this way, our Abstract State Postulate allows formal computational states to be as true to reality as is mathematically feasible. Whatever the states of some algorithmic computation "really" are (e.g., a piece of paper containing geometric drawings), their reality is faithfully modeled by logical structures, the "least common denominator" of all of mathematical modeling of static realities. No unnecessary properties of the real states (e.g., the thickness of the paper) appear in the model, nor does the model introduce unneeded and unwanted attributes (like the length of some particular textual representation of a triangle), as is invariably the case when objects of one kind are encoded as objects of another kind, be that numbers as strings, strings as graphs, graphs as matrices, or matrices as nested sets. (Compare the foundational discussion in [14].)

By virtue of the generality of structures, every other model of (sequential) computation extant in the literature is a special case of ours. As one would expect, the presentations of state are—from the mathematical point of view—structures, in every case. Here is a sampling of structures used in programming languages:

- Traditional arithmetic operates over natural numbers, having a "Platonic" existence. Numbers are endowed with elementary-school operations, like addition and multiplication, or with more complex functionality, like primality tests.
- Common Lisp's arithmetic operates over the infinite set of rational numbers.
- Counter machines, at the other extreme, make do with very simple, "neolithic" operations.
- In the classical random access (RAM) model of computation, memory cells are arbitrary integers and memory content is a dynamic function with an infinite address domain.
- The states of a pushdown automaton include a stack of symbols and simple stack operations.

- The Burroughs B5500 computer had a stack-based architecture with hardwired stack operations and no programmer-addressable registers.
- In Lisp, nested list expressions are the basic datatype, with primitive operations for adding and removing elements.
- The basic datatype in APL is arrays, including arrays of arrays. These objects are endowed with a very rich repertoire of matrix operations.
- Infinite streams are the basis for the lazy programming paradigm used in languages like Lucid and Haskell.
- Strings, with complex string operations, including matching and concatenation, are used for text processing in languages like Snobol and Perl.
- Plain lambda calculus has lambda terms as its basic objects and β-reduction as a basic operation.

All of these are straightforward instances of structures, and fit effortlessly into our framework.

No previous approach has nearly the generality of the one espoused here. For example, in Turing's formalism, states are modeled by means of strings; Kolmogorov machines are based on labeled graphs; Gandy's states are hereditarily finite sets. But strings, graphs and sets are all distinct from one another and come with very different native operations. Nested sets are not labeled graphs, nor are labeled graphs, nested sets. Neither can lay claim to genericity. Hypergraphs, for instance, are not strings, graphs, or sets, per se.

State structures can be quite intricate, involving many types of objects. For example, a programming language can itself be thought of as an algorithm (an *interpreter*) that takes a program as input and executes it on given data. Its states would comprise a variegated domain and an abundance of operations. To represent all structures used by programmers in terms only of strings, or graphs, or sets would require a daunting amount of encoding. In our setting, however, this poses no problem; witness the detailed structural-representations of C and Java given in [43] and [105], respectively.

Even algorithms computing purely numeric functions or pure string functions typically involve additional types of objects. So, to define effectiveness of numerical calculations or of string calculations, one needs to consider the effectiveness of operations over those auxiliary objects. But to show the effectiveness of a graph operation by applying standard notions of effectiveness of strings, say, requires an appeal to intuition in support of the assertion that the encoding of graphs as strings is effective. (Compare [72, pp. 430–431].) This is because the encoding does not reside in either a pure string domain or a pure graph domain, for either of which there is a standard effective model of computation (Turing machines and Kolmogorov machines, respectively). Rather, it is a function over a domain with both strings and graphs, for which there is no direct, formal definition of effectiveness. In Section 6, we provide an alternative approach to effectiveness of basic operations over non-numerical domains, requiring, instead, effectiveness of their numerical homomorphic images.[12]

In contradistinction to Turing machines or any other model of computation with a limited repertoire of basic objects, using structures for states makes it possible for an abstract state machine to provide a step-for-step emulation of any algorithm, regardless of the complexity of its data structures and primitive operations.[13]

These issues are discussed further in the concluding section.

§2. Stepwise effectivity.

Church was striving to characterize the numeric functions that are algorithmically computable. The question is how does one know that all possible algorithms have been characterized, or as Post already phrased the problem in 1921 [76]: one needs to capture "all the possible ways in which the human mind could set up finite processes".

Rogers, a student of Church's, starts his classic book [82, pp. 1–2] with the following elaboration on the informal notion of "effective procedure" (italics in original):

> Roughly speaking, an algorithm is a clerical (i.e., deterministic, bookkeeping) procedure which can be applied to any of a certain class of symbolic *inputs* and which will eventually yield, for each such input, a corresponding symbolic *output*. An example of an algorithm is the usual procedure given in elementary calculus for differentiating polynomials. ...
>
> Several features of the informal notion of algorithm appear to be essential. We describe them in approximate and intuitive terms.
> *1. *An algorithm is given as a set of instructions of finite size.* ...
> *2. *There is a computing agent, usually human, which can react to the instructions and carry out the computation.*
> *3. *There are facilities for making, storing, and retrieving steps in a computation.*

[12] Barendregt expresses the importance of transparent representations as follows [2, pp. 188–189]: "Lambda definability was introduced for functions on the set of natural numbers $\mathbb{N}$. In the resulting mathematical theory of computation (recursion theory) other domains of input or output have been treated as second class citizens by coding them as natural numbers. In more practical computer science, algorithms are also directly defined on other data types like trees or lists. Instead of coding such [inductive] data types as numbers one can treat them as first class citizens by coding them directly as lambda terms while preserving their structure. Indeed, lambda calculus is strong enough to do this. ..."

[13] Indeed, the experience of a wide range of abstract state machine applications [20] also supports this claim: In all cases, it has been possible to specify software faithfully on the precise proper level of abstraction, without introducing unnecessary details, or divulging internal details, thereby eliminating unwanted ambiguity, while—at the same time—preserving any desired ambiguities.

> *4. *Let P be a set of instructions as in* *1 *and L be a computing agent as in* *2. *Then L reacts to P in such a way that, for any given input, the computation is carried out in a discrete stepwise fashion, without use of continuous methods or analogue devices.*
> *5. *L reacts to P in such a way that a computation is carried forward deterministically, without resort to random methods or devices, e.g., dice.*
>
> Virtually all mathematicians would agree that features *1 to *5, although inexactly stated, are inherent in the idea of algorithm.

Rogers also goes on to say [82, p. 4] that there should be a "fixed finite bound on the capacity" of the agent L, which would necessitate L's using the unlimited memory resource, alluded to in *3, to "keep track of ... progress" in the computation and remember "one's place" in the program. In what follows, we formalize these considerations of finite program and computation that is stepwise-bounded and deterministic—without delineating the rôles of program P and agent L. We base ourselves on the formalization of [42], while elaborating and refining aspects that are important from the point of view of effectiveness.

2.1. Sequentiality. We begin by characterizing computations, in general. Computation, as opposed to the behavior of a physical process, is usually conceived of as a sequence of discrete computational steps.[14]

This is what Kolmogorov presumably had in mind when he presented his view of algorithms in 1953 [56]:[15]

> We start with the following obvious ideas concerning algorithms:
> (1) An algorithm Γ being applied to any "input" (= "initial state") A which belongs to some set (the "domain" of the algorithm) gives a "solution" (= "final state") B.

[14]Turing was explicitly interested in formalizing "discrete" machines, not continuous processes [109]: "The nervous system is certainly not a discrete-state machine. A small error in the information about the size of a nervous impulse impinging on a neuron, may make a large difference to the size of the outgoing impulse. It may be argued that, this being so, one cannot expect to be able to mimic the behaviour of the nervous system with a discrete-state system. It is true that a discrete-state machine must be different from a continuous machine. But if we adhere to the conditions of the imitation game, the interrogator will not be able to take any advantage of this difference." The first attempt, as far as we know, at characterizing analogue computation is [72].

[15]Cf. Knuth in 1966 [54]: "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.... A computational method comprises a set Q (finite or infinite) of 'states', containing a subset X of 'inputs' and a subset Y of 'outputs'; and a function F from Q into itself. (These quantities are usually also restricted to be finitely definable, in some sense that corresponds to what human beings can comprehend.) ... In this way we can divorce abstract algorithms from particular programs that represent them."

(2) An algorithmic process splits into separate steps of limited complexity; each step consists of an "immediate transformation" of the state S obtained up to this moment into the state $S^* = \Omega_\Gamma(S)$.
(3) The process transforming $A^0 = A$ into $A^1 = \Omega_\Gamma(A^0)$, then A^1 into $A^2 = \Omega_\Gamma(A^1)$, then A^2 into $A^3 = \Omega_\Gamma(A^2)$, etc. is continued until the next step is impossible (i.e., the operator Ω_Γ is undefined on the current state) or a signal indicating the appearance of the "solution" is received. It is possible, however, that this process of transformations would never stop (if we get no signal at all).
(4) The immediate transformation of S into $S^* = \Omega_\Gamma(S)$ is based only on information about the limited "active part" of S and affects this part only.

Much earlier, in 1922, Behmann [4, p. 166] expressed the stepwise nature of algorithmic activity by saying (cited in [113]):[16]

> A completely determined general [set of] instructions shall be exhibited, according to which the correctness or falsity of an arbitrary given claim, which can be formulated with purely logical means, can be decided after a finite number of steps.

This is also what Kleene envisioned when he wrote [53, pp. 16–17] (emphasis in the original):

> Such a method is given by a set of rules or instructions, describing a [decision] procedure that works as follows. *After* the procedure has been described, if we select *any* question from the class, the procedure will then tell us how to perform successive steps, so that after a finite number of them we will have the answer to the question selected.... After our performing any step to which the procedure has led us, the rules or instructions will *either* enable us to recognize that now we have the answer before us and read it off, *or else* that we do not yet have the answer before us, in which case they will tell us what steps to perform next.

Furthermore, we view computation as proceeding *deterministically*, "leaving no place to arbitrariness" [66, p. 1]. As Rosser, also a student of Church, puts it [83]:

> "Effective method" is used here in the rather special sense of a method each step of which is precisely determined and which is certain to produce the answer in a finite number of steps.... An effective method of solving certain sets of problems exists if one

[16]"Es soll eine ganz bestimmte allgemeine Vorschrift angegeben werden, die über die Richtigkeit oder Falschheit einer beliebig vorgelegten mit rein logischen Mitteln darstellbaren Behauptung nach einer endlichen Anzahl von Schritten zu entscheiden gestattet."

> can build a machine which will then solve any problem of the set with no human intervention beyond inserting the question and (later) reading the answer.

Shoenfield adds [91, p. 107], "A method must be *mechanical*.... Methods which involve chance procedures are excluded; ... methods which involve magic are excluded; ... methods which require insight are excluded."

Computations may, therefore, be formalized as a (*deterministic*) *state-transition system*, comprising a set of *states* S, a subset I of which are *initial*, and a (typically partial) *transition function* $\rightsquigarrow$ on states, which determines the next-state relation. States with no "next" state, namely, $\{\beta \in S \mid \neg \exists \gamma. \beta \rightsquigarrow \gamma\}$, will be, for us, *terminal states*.

Remark 2.1. For Kolmogorov [56], terminal states somehow "signal" their appearance. In the original definition of abstract state machines [41], the transition function is always total; intuitively terminal states are their own next state. Since we are interested in the output of algorithms, we distinguish between a terminal state that marks the end of a computational sequence, and a non-terminating state that may be its own next state.

This understanding of computation as proceeding in discrete steps is encapsulated as follows:

POSTULATE I (Sequential time). *An algorithm is a state-transition system. Its transitions are partial functions.*

Continuous (analogue) processes, transfinite computation sequences (involving limits) [78, 40], nondeterministic transitions, and nonprocedural input-output specifications are thereby excluded from consideration.

Classical algorithms, of the sort Church was considering, never leave room for choices. For example, though segments of the evaluation of recursive functions could, in principle, proceed in a nondeterministic fashion, or even in parallel, when it came down to specifying their computation, a particular order was always fixed in advance. Thus, Rogers [82, p. 7] writes, "We obtain the computation uniquely by working from the inside out and from left to right" (and, similarly, in [48, p. 45] and [91, p. 109]). Classical algorithms also do not involve any sort of interaction with the environment to determine the next step. For this reason, we are justified in restricting our attention to fully deterministic algorithms.

Remark 2.2. Moschovakis [73, p. 919] claims that "algorithms are recursive definitions while machines model implementations, a special kind of algorithms". We beg to differ. Algorithms, for us, are deterministic transition systems. This traditional viewpoint is in accord with that of most students of computability—including those quoted above—and that of virtually all computer scientists and engineers. Besides, recursive definitions by themselves are open to more than one interpretation. Most programming languages in fact use an eager evaluation strategy and compute a function that is, in general, less defined than the least fixed point. Recursive definition have, besides their least fixed-point solution, a unique "optimal" (maximally consistent) fixed point [65], which (though not necessarily computable) is, in general, more defined than the least fixed point and could also be taken as the intended semantics of the definition. Even after specifying that the least fixed point is what is meant, as Moschovakis does, there is much room for algorithmic variation. There can be significant algorithmic distinctions and performance differences between reasonable methods of computing that least fixed point, such as "call by name" and "call by need". So, for us (see [8]), recursion equations are a partial specification of desired properties of the algorithm in question, not the algorithm itself. Nevertheless, if one does accept Moschovakis's point of view, then it is the various "implementations" of such recursive definitions that we have set out to characterize here. In [8], it is argued that such a machine-independent implementation does not reduce the level of abstraction; in [7], it is argued that there is no "one size fits all" notion of equivalence between algorithms—or between faithful implementations of recursive definitions.

Remark 2.3. Though Turing's (human) computers operate deterministically when computing functions, he did envision *choice-machines*, which wait for an "arbitrary choice . . . by an external operator" before continuing (in an exploration of proofs, say). The Sequential Postulates have been adapted to admit the essential use of nondeterminism, as in modern distributed computations; see [11, 12, 13] for an in-depth treatment.

2.2. Abstractness. Specific models of computation work with specific data structures. For example, Turing designed a model that works with tapes, though he explained why such a one-dimensional medium suffices for what is normally carried out by people in two dimensions [107]: "I think that it will be agreed that the two-dimensional character of paper is no essential of computation." Kolmogorov–Uspensky's model [57], and later variants of the "pointer machine" (e.g., [85]), use graphs as a more free-form representation of state. Gandy [35] suggests hereditarily finite sets as a generic data structure for the same purpose.

Since we are interested here in characterizing arbitrary algorithms, we ought not limit the form of states a priori. Accordingly, we let the states of state-transition systems be first-order structures with equality [106], first-order structures being the most general thing mathematicians have in their arsenal for representing discrete states. It will simplify matters if the relations (predicates) of a first-order structure are viewed as their characteristic, truth-valued functions, but this is merely a matter of convenience.

Structures, like our states, whose vocabulary does not include relation symbols are called *algebras* (in the universal algebra sense). We shall refer to state-transition systems with algebraic states as *abstract transition systems* (*ATSs*).

Remark 2.4. If one chooses to forego the convenience of dealing with algebras and work with states that are classical structures, with relations as well as functions, then Boolean truth values and Boolean connectives need not be an integral part of states. Instead, the semantics of Boolean operations could be imparted by the "outside world", whereas here Boolean operations are an integral part of states. Cf. Remark 3.3. Both the logical structure-based approach and the programming-oriented algebra-based approach yield precisely the same results on effectivity. The differences are purely æsthetic.

As already indicated in Section 1.4, the justification for postulating that structures or algebras are appropriate for capturing algorithmic states is the enormous experience of mathematicians who have faithfully and transparently presented every kind of static mathematical reality as a first-order structure.

The reader should not be misled by the modifier "first-order": Should an algorithm make use of sets and/or higher-order functions, then sets and functions should be incorporated in the domain (a.k.a. base set, universe, underlying set, or carrier) of the structure, and the appropriate "higher" operations can be included in the "first-order" structure.

We assume, furthermore, that the classes of states and initial states are closed under isomorphism, and that transitions commute with isomorphisms, so that isomorphic states transition to isomorphic states. Closure under isomorphism is justified by the intention that all essential information about a state be given by the basic functions, as is usual in logic and algebra. The individual nature of elements is unimportant; therefore, structures are mere representations of isomorphism types. There should be no more reality to a domain element than what is observable from the structure.

This isomorphism constraint is what makes states "abstract". It reflects the fact that an ATS works at a fixed level of abstraction, and "lower-level" representations do not matter. The relevance of insisting on closure under isomorphism was underscored by Gandy [35, p. 128]. For more discussion, see [42, Sect. 4.6].

All states of an algorithm should share the same fixed vocabulary. Furthermore, transitions do not change the underlying domain. Whatever vocabulary or domain elements may possibly be needed at some point in a computation are included from the outset. Since, as we will see in the next subsection, algorithms are finitely describable, we may assume that the vocabulary is finite. See [42, Sect. 4.4–4.5] for more detailed considerations.

Usually, the states of an ATS include some static functions that are present in initial states and are never changed by transitions. In particular, equality and the Boolean operations are always static and inviolate. The values (interpretation) of any other ("defined") functions, however, are dynamic and may change from state to state. But in any case, all the functions of a state are *total*, formally speaking.

Since a state is a structure, it "holds" a value for each function in its vocabulary, applied to every possible combination of domain values. A specific *location* in a state α is given by a function symbol f from the vocabulary of α and by a tuple $(a_1, \ldots, a_n)$ of elements of the domain $\mathrm{Dom}\,\alpha$ of α, where n is the arity of f. Let f_α signify the interpretation of f in state α. The value stored at location $f(a_1, \ldots, a_n)$ is just $f_\alpha(a_1, \ldots, a_n)$. In this way, every state α assigns a value $[\![t]\!]_\alpha \in \mathrm{Dom}\,\alpha$ to each (ground, that is, variable-free) term t over its vocabulary.

A domain might use a particular element to signify a singularity, or that the arguments in question are for all practical purposes invalid, or that its value has not yet been ascertained. For example, a state might contain the datum, $3/0 = \bot$, where $\bot$ is some particular domain element, to indicate that the result of division by zero is undefined, and has no numerical value.[17]

Remark 2.5. Following [42], we will henceforth assume that the domain of states includes an undefined value, $\bot$, for this purpose. This is a mere convenience; it is not an essential ingredient of abstract states.

Of course, each state must contain all the data required by the algorithm for making the next step. As Turing [107, pp. 232, 253–254] explains:

> At any stage of the motion of the machine, the number of the scanned square, the complete sequence of all symbols on the tape, and the m-configuration will be said to describe the *complete configuration* at that stage. . . .
>
> It is always possible for the computer to break off from his work, to go away and forget all about it, and later to come back and go on with it. If he does this he must leave a note of instructions (written in some standard form) explaining how the work is to be continued. This note is the counterpart of the "state of mind". We will suppose that the computer works by such a desultory manner that he never does more than one step at a sitting. The note of instructions must enable him to carry out one step and write the next note.

So, to completely characterize the intermediate status of a Turing machine computation, while the computer is "out to lunch", the abstract state should include the condition of the control (Turing's "state of mind" or "m-configuration"), the complete contents of the tape, and the position of the read/write head (what is called, nowadays, the "instantaneous description" of an intermediate state of the machine).

[17] A non-terminating computation (denoted $\bot$ in some other contexts) is something altogether different, and corresponds to an infinite sequence of transitions.

To sum up, we have:

POSTULATE II (Abstract state). *States are structures, sharing the same fixed, finite vocabulary. States and initial states are closed under isomorphism. Transitions preserve the domain, and transitions and isomorphisms commute.*

Since transitions do not change the vocabulary or the domain, it is only the interpretation that a state gives to the symbols in its vocabulary that changes from state to state.

Commutation here means that whenever there is a transition $\alpha \rightsquigarrow \alpha'$ and an isomorphism π from α to another state $\beta = \pi(\alpha)$, then there must also be a transition $\beta \rightsquigarrow \beta'$ from β to the corresponding isomorphic image $\beta' = \pi(\alpha')$ of α'. It follows that terminal states are also closed under isomorphisms.

Infinitary operations, like taking limits, are excluded, since the vocabulary is first-order, but—then again—operations that are not finitary cannot be expected to be evaluatable in a single algorithmic step. As already mentioned, algorithms working with higher-order structures are not precluded, since the domain may include sets and higher-order functions. Similarly, one can have a limit operator operating on a whole sequence as a unity, along with operations for building sequences provided as part of the state.

Note that we are not saying that the states of an algorithm can be somehow "encoded" as (isomorphism-closed) structures. Rather, we are postulating that, once formalized, states are essentially structures—whether or not the author of the algorithm conceived of them that way. In other words, there is always a set of relevant objects (which is the domain), and the salient characteristics of the state are in fact relations or functions over those objects.

Post, according to his own testimony in [76, pp. 420, 426–429], had the following intuitions about abstract representations of computational states in 1922:

> We are ... to regard our symbols as without properties except that of permanence, distinguishability and that of being part of certain symbol-complexes.
>
> We ... give what is at least a first approximation to a definitive solution of the difficulty of finding a natural normal form for symbolic representation....
>
> We ... assume [symbolic representations] to be finite and we might say discrete.... Each symbolization can be considered to consist of a finite number of unanalysable parts (unanalysable from the standpoint of the symbolization) these parts having certain properties and certain relations with each other.... The ways in which these parts can be related will be assumed to be specified for the whole system of symbolizations.... The number of these

> elementary properties and relations used is finite and ... there is a certain specific finite number of elements in each relation....
>
> The symbol-complexes are completely determined by specifying all the properties and relations of [their] parts.... Each complex of the system can be completely described [by a conjunction of relations]....
>
> Due to discreteness and finiteness we would thus have a finite sequence of symbol-complexes representing the various stages in the method.

Post is asserting here that computational states are completely determined by the relations of a first-order structure.[18]

2.3. Boundedness. So far, nothing we have said guarantees that the behavior of a transition system is effective. For effectivity, it must be possible to express the rules for going from state to state in some *finite* fashion. Kleene stresses this point repeatedly:

> An algorithm in our sense must be fully and finitely described before any particular question to which it is applied is selected. When the question has been selected, all steps must then be predetermined and performable without any exercise of ingenuity or mathematical invention by the person doing the computing. [50, pp. 240–241n.]

> The notion of an "effective calculation procedure" or "algorithm" (for which I believe Church's thesis) involves its being possible to convey a complete description of the effective procedure or algorithm by a finite communication, in advance of performing computations in accordance with it. [52, p. 493]

> An algorithm is a *finitely* described procedure.... In performing the steps, we simply follow the instructions like robots; no ingenuity or mathematical invention is required of us. Such methods as I have described ... have been called "algorithms". [53, p. 17]

Turing analyzed the need for transitions to depend on only a finite segment of the state, for his model, as follows [107, Sect. 9]:

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

[18] The idea of providing the basic operations of recursively-defined functions over arbitrary domains by means of a logical structure also appears in [62].

> this are of the same character as those which restrict the number of symbols.

This finiteness requirement is expressed in more general terms by Kolmogorov and Uspensky [57, pp. 6, 16]:

> The mathematical notion of Algorithm has to preserve two properties. . . .
>
> 1. The computational operations are carried out in discrete steps, where every step only uses a bounded part of the results of all preceding operations.
>
> 2. The unboundedness of memory is only quantitative: i.e., we allow an unbounded number of elements to be accumulated, but they are drawn from a finite set of types, and the relations that connect them have limited complexity. . . .
>
> It seems plausible to us that an arbitrary algorithmic process satisfies our definition of algorithms. We would like to emphasize that we are talking not about a reduction of an arbitrary algorithm to an algorithm in the sense of our definition, but that every algorithm essentially satisfies the proposed definition.

To achieve this for arbitrary abstract transition systems, we demand the following:

POSTULATE III (Bounded exploration). *Transitions are determined by a fixed finite "glossary" of "critical" terms. That is, there exists some finite set of* (*variable-free*) *terms over the vocabulary of the states, such that states that agree on the values of these glossary terms, also agree on all next-step state changes.*

Algorithms, by their nature, explain how to update states by manipulating values stored at locations in the current state. For an algorithm to refer to a particular location $f(a_1, \ldots, a_n)$, it needs to specify the function f, and also to identify each of the arguments a_i. But how can the algorithm specify domain elements a_i in an abstract state? It can indirectly, by means of locations. So, in the final analysis, terms provide a perfectly general means of specifying locations and elements of states. Each critical term $f(t_1, \ldots, t_n)$ "points" to the location $f(a_1, \ldots, a_n)$, containing $f_\alpha(a_1, \ldots, a_n)$, where $a_i = [\![t_i]\!]_\alpha$ is the value of the term t_i in state α. Thus, this postulate means that only a bounded number of locations need to be explored for the algorithm to make a transition. Needing only a bounded number of critical terms corresponds exactly to the ability to describe *finitely* how transitions are effected, whatever the language or format of description. Additional arguments in support of this postulate may be found in [42].

By an *update*, we will mean a triple, written as an "assignment" $f(\bar{a}) := b$, indicating that the value b is to be assigned to location $f(\bar{a})$, changing the graph of function f. Let

$$\Delta(\alpha) = \{f(\bar{a}) := b \mid \alpha \rightsquigarrow \alpha', f \in \mathcal{F}, a_i \in \mathrm{Dom}\,\alpha, f_\alpha(\bar{a}) \neq f_{\alpha'}(\bar{a}) = b\},$$

be the set of updates that transpire in a transition out of α. Bounded exploration demands that whenever $[\![t]\!]_\alpha = [\![t]\!]_\beta$ holds for all critical terms t in the glossary, then either α and β are both terminal states, or else $\Delta(\alpha) = \Delta(\beta)$, meaning that they both change in the same way.[19] Together with isomorphism preservation of transitions (Postulate II), the equality of the Δ's implies that any updated value b in $\Delta(\alpha)$ is the interpretation of one of the critical terms in the glossary [42, Lemma 6.2]. Bounded exploration is what ensures that the step-by-step behavior of the procedure is effective, since it implies that the algorithm can be described by a finite text [42], as we will see in the next section.

Infinite programs, as well as individual steps that require examination of unboundedly many locations within states, or which update unboundedly many values in one fell swoop, are precluded by this postulate. To the extent that an unbounded operation is effective on account of its having a bounded schematic description, that schema should be incorporated in the state itself. For example, a transition cannot be governed by an "instruction" like

$$n := 2^{3^{\cdot^{\cdot^{\cdot^{n}}}}},$$

as it refers to unboundedly many terms (depending on the value of n). One would need, instead, a new operation for such an exponential tower. Nor can the transition be something like

$$n := 3 \underbrace{\uparrow\uparrow \cdots \uparrow}_{n \text{ times}} n,$$

where the arrow is Knuth's "uparrow" notation for iterated operations.[20] This instruction is also unbounded, as it is really a schema for infinitely many conditional assignments, each involving a different operation:

$$\begin{array}{l} \vdots \\ \textbf{if } n = 2 \textbf{ then } n := 3 \uparrow\uparrow 2 \\ \textbf{if } n = 3 \textbf{ then } n := 3 \uparrow\uparrow\uparrow 3 \\ \textbf{if } n = 4 \textbf{ then } n := 3 \uparrow\uparrow\uparrow\uparrow 4 \\ \vdots \end{array}$$

To obtain the equivalent effect would require some form of iteration involving a long sequence of state transitions, or a new ternary operation $a \uparrow^n b$. Similarly, a bounded quantifier, like $\exists\, i{<}n.f(i){=}f(n)$, which refers to an

[19] This definition of Bounded Exploration takes into account the possibility of the transition function being partial, along the lines of [11].

[20] A single $\uparrow$ is ordinary exponentiation, $\uparrow\uparrow$ is tetration (repeated exponentiation), $\uparrow\uparrow\uparrow$ is "pentration" (repeated tetration), and so on.

unbounded number of values of f, is actually an operation in its own right, taking the predicate $\lambda\, i,n.f(i)=f(n)$ as an argument.

As explained in the previous subsection, each abstract state is meant to explicitly incorporate *all* information needed for the continuation of the computation from that point on. To determine the next state only a bounded amount of that information needs to be explored. With Bounded Exploration, an algorithm computes in "steps of limited complexity", as demanded by Kolmogorov [56] (quoted above). This postulate, thereby, goes straight to the heart of Kolmogorov's implicit question: What does it mean to bound the complexity of each individual step?

§**3. Abstract state machines.** An *abstract state machine*, or *ASM*, is a state-transition system in which algebraic states store the values of functions, in other words, their "graphs", and each transition updates a finite number of locations of the current state. Transitions are programmed using a convenient language based on guarded commands for updating individual locations in states. ASMs capture the notion that each step of an algorithm performs a bounded amount of work, whatever domain it operates over, so are central to our development.

DEFINITION 3.1 (ASM [42]). An *abstract state machine* (*ASM*) is given by:[21]

- a set (or proper class)[22] S of algebraic states, closed under isomorphism, sharing a vocabulary $\mathcal{F}$,
- a set (or proper class) $I \subseteq S$ of initial states, closed under isomorphism, and
- a *program* P, consisting of finitely many *commands*, each taking the form of a *guarded assignment*

$$\textbf{if } p \textbf{ then } t := u,$$

for terms t and u over $\mathcal{F}$ and conjunction p of equalities and disequalities between terms.

At each step of a computation, all assignments in P whose guards hold true in the current state are executed in parallel so as to give the next state. More precisely, given a state $\alpha \in S$, program P defines the following set $\Delta^+(\alpha)$ of updates:

$$\big\{f([\![\bar{s}]\!]_\alpha) := [\![u]\!]_\alpha \bigm| (\textbf{if } p \textbf{ then } f(\bar{s}) := u) \in P,\ [\![p]\!]_\alpha = \mathit{True}\big\},$$

[21] What is defined here is what are called "small-step" (or "sequential") ASMs, but since that is the only kind of ASM used in this paper, the modifier "small-step" will be omitted. A richer and more liberal language for small-step ASMs is provided in [42], but the simplistic form given here suffices for our purposes.

[22] See [42, fn. 2]

where *True* is Boolean truth and the valuation given by state α is extended to tuples $\bar{s}$ in the obvious way. A set of updates is *inconsistent* if it contains two updates, $f(\bar{a}) := b$ and $f(\bar{a}) := b'$, with $b \neq b'$. If $\Delta^+(\alpha)$ is empty or if it is inconsistent, then α is a terminal state. Otherwise, α has a next state α', with the same vocabulary and domain as α, and with its valuations updated as follows:

$$f_{\alpha'}(\bar{a}) = \begin{cases} b & \text{if } (f(\bar{a}) := b) \in \Delta^+(\alpha), \\ f_\alpha(\bar{a}) & \text{otherwise.} \end{cases}$$

Remark 3.2. In [42], there was no need to deal with the output of algorithms, so there was no need to single out terminal states. Since we need to do just that here, we have slightly modified the behavior of ASMs, so that a state α has no next state when an ASM performs no updates (rather than have α also be the next state, as in [42]). On the other hand, when there are only trivial updates, assigning the same value to each location as it already has, the result is an infinite, nonterminating computation. Despite this minor difference (which is but a small part of the ASM enhancements made in [11, 12, 13]), the proof in [42] of Theorem 3.4 below goes through.

Remark 3.3. Note that guards are essentially propositional formulæ. If one prefers to view states as traditional structures, having both functions and relations, then one needs to extend the valuation provided by states to also give propositional formulæ their usual meaning. From this more traditional viewpoint, truth values and propositional connectives need not be an actual part of states, but get their meaning from the "outside".

Obviously, every ASM satisfies the three postulates of the previous section, collectively referred to as the *Sequential Postulates*. Moreover, any process that satisfies the Sequential Postulates provably behaves just like some ASM:

THEOREM 3.4 (ASM Theorem [42]). *For every process satisfying the Sequential Postulates, there is an abstract state machine in the same vocabulary (and with the same sets of states and initial states) that emulates it.*

We use the term "emulate" for *step-by-step* simulation.[23] The proof is based in large part on the ability to use critical terms to express all transitions of abstract states.

This emulation is effective, in that the abstract state machine provides an effective means of computing each state of a computation sequence from its predecessor, provided the latter is finitely representable. So, whenever initial states can be represented effectively—a notion that will be axiomatized in Sections 4 and 6—the whole computation becomes effective.

[23]"**Emulate.** To duplicate the functions of one system with a different system, so that the second system appears to behave like the first system." American National Standards Institute [1].

Methods satisfying the Sequential Postulates include: (1) the classical algorithm for greatest common divisor—which Euclid applied to both rational and irrational values, and which can be applied more generally to Euclidean rings; (2) the ancient "rectangular array" method for solving linear equations (from the two-millennia old Chinese classic, *Jiuzhang suanshu* [47]); and (3) the very similar method of Gaussian elimination, even when the field (or division ring) over which it is applied is unspecified. On the other hand, the postulates exclude underspecified methods (like, "Guess a solution to a system of linear equalities"), and non-algorithms ("Try all numbers to see whether or not there is a solution to a system of linear equalities" or "... to a Diophantine equation"). They are also meant to exclude nondeterministic methods (like "pivot on any non-zero element"), randomized algorithms (like multiplying by a random matrix prior to performing Gaussian elimination), probabilistic methods (like Rabin's algorithm for testing primality), modern distributed processes (like Internet routing), or massively parallel DNA computations (for the traveling salesman problem, say).[24]

Up to this point, we have established conditions under which an algorithm is effective to the extent that the operations in the initial state are. In addition, it is universally required that states have constructive representation. "We are", Kleene writes [52, p. 493], "dealing with discrete objects (the arguments and the result included) – it is digital, not analog, computing." Uspensky and Semenov [111, p. 9] write that, "We insist ... that algorithms can deal directly only with constructive objects but not with finite objects not being constructive ones."

As Knuth writes [55, p. 6]:

> An algorithm is also generally expected to be *effective*. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using pencil and paper. [Euclid's Algorithm] uses only the operations of dividing one positive integer by another, testing if an integer is zero, and setting the value of one variable equal to the value of another. These operations are effective, because integers can be represented on paper in a finite manner, and because there is at least one method (the "division algorithm") for dividing one by another. But the same operations would *not* be effective if the values involved were arbitrary real numbers specified by an infinite decimal expansion, nor if the values were lengths of physical line segments (which cannot be specified exactly).

Indeed, if all initial states of an ATS are finitely-representable objects, then Bounded Exploration ensures that all subsequent states also are, and

[24]Large classes of such non-classical algorithms are covered by the generalizations of the ASM Theorem in [9, 13, 16, 37].

the ASM formalism effectively computes the ensuing sequence of states, until a terminal state is obtained—if ever. But, the Sequential Postulates do not, in and of themselves, guarantee that an algorithm computes a *computable* function, since they allow initial states to be pre-endowed with non-computable functionalities (something that can be necessary for general algorithms, which may operate over domains like the real line or a Hilbert space).

This issue is taken up next.

§**4. Arithmetical effectivity.** Since we are interested in the computation of functions, we may suppose that the vocabulary of an ATS includes (nullary) symbols *Out* for the output and $In_1, \ldots, In_n$, for $n \geq 0$ input values. (Constants are treated as nullary functions.)[25] Furthermore, we should insist that there is exactly one initial state, up to isomorphism, for each n-tuple of input values. An ATS operating over a domain D may be said to *compute* the following function:

$$\{(\llbracket In_1 \rrbracket_\alpha, \ldots, \llbracket In_n \rrbracket_\alpha) \mapsto \llbracket Out \rrbracket_\beta \mid \alpha \in I, \beta \in O, \alpha \leadsto^* \beta\},$$

where I and O are the sets of initial and terminal states with domain D, respectively, and $\leadsto^*$ is the reflexive-transitive closure of its transition function $\leadsto$. This input-output relation is a partial function over D, since $\alpha \leadsto^* \beta \in O$ is a partial function, by virtue of $\leadsto$ being undefined for terminal states.

Remark 4.1. It is often convenient to distinguish between successful and unsuccessful termination of algorithms. So, it is natural to declare that a terminal state in which the value of *Out* is $\perp$ constitutes failure, while obtaining a defined value for the output is deemed a success. This distinction is of no consequence in the development that follows.

Church was interested in formalizing numerical algorithms, that is, algorithms that apply arithmetic operations to the natural numbers. So we should endow our states with basic arithmetic abilities.

4.1. Arithmetical states. The choice of basic functions is somewhat flexible, as we will see. But the standard "grade school" operations are what one typically has in mind. Menabrea,[26] in his 1842 description of Babbage's Analytical Engine, wrote [68]:

> We must limit ourselves to admitting that the first four operations of arithmetic, that is addition, subtraction, multiplication and division, can be performed in a direct manner through the intervention of the machine. The machine is thence capable of

[25]Neither constants, nor functions, are guaranteed to maintain their values, because their value may vary from state to state, which is what would make it awkward were we to talk about "variable constants".

[26]Luigi Federico Menabrea, an engineer, later became prime minister of Italy.

> performing every species of numerical calculation, for all such calculations ultimately resolve themselves into the four operations we have just named.

Babbage's design also included a conditional branch on zero; see, for example, [36].

DEFINITION 4.2 (Arithmetical state). Up to isomorphism, an *arithmetical state* is as follows: Its domain includes the natural numbers **N**, as well as the two (distinct) Boolean truth values, *True* and *False*, and some (other) distinguished value $\perp$ signifying "undefined". Its operations include some or all of the "grade school" operations of arithmetic, namely, zero (0), successor $(+1)$, addition $(+)$, subtraction $(-)$, multiplication $(\cdot)$, integer quotient ($\div$, which ignores any remainder), equality $(=)$, and inequality $(>)$, as well as logical constants and standard operations for the Booleans. Besides symbols for all these operations, the vocabulary of an arithmetical state may also have various symbols for dynamic functions.

So that these arithmetic functions are all total, we let $m - n = 0$ when $n > m$ (this is "proper" or "natural" subtraction for the natural numbers) and let $n \div 0 = \perp$ for all n. Arithmetic and Boolean operations are "typed" and "strict", so applying an arithmetic operation to a non-number or a Boolean operation to anything but truth values results in $\perp$.

Dynamic operations act as "variables" in the programming sense; their values may be updated by the algorithm in the course of a computation. When all dynamic operations in a state, *other than the inputs* In_i, are completely undefined—that is, are assigned the value $\perp$ for all arguments—we say that the state is *blank*. This will be the case initially.

To capture the fact that Church's Thesis is dealing specifically with numerical calculations, we need the following additional postulate:

POSTULATE IV (Arithmetical state). *Initial states are arithmetical and blank. Up to isomorphism, all initial states share the same static operations, and there is exactly one initial state for any given input values.*

This postulate will be considerably weakened in Section 6 to allow richer domains than the purely numeric.

4.2. Arithmetical machines. Our goal is to characterize everything that is effectively computable, starting with arithmetical states. Accordingly, we are interested in the following class of machines:

DEFINITION 4.3 (Arithmetical ASM). An *arithmetical ASM* is an ASM satisfying the Arithmetical State Postulate (IV).

To begin with, since transitions do not change the domain, vocabulary, or static operations, we obviously have the following:

PROPOSITION 4.4. *All states of an arithmetical ASM are arithmetical.*

The following characterization is the main point of this section:

THEOREM 4.5. *A numeric function is partial recursive if and only if it is computable by an arithmetical ASM.*

PROOF. It is well-known that counter (Minsky) machines can compute any partial recursive function [71]. And any counter machine (which only uses $0, = 0, +1, -1$) can be directly emulated by an arithmetical ASM, with nullary symbols for each counter (which include the inputs In_i and output Out), plus another "program counter" to keep track of the current counter-machine instruction. The ASM commands take the following forms:

- Initialization instructions:
  if $p = \bot$ **then** $p := 0$ (initialize program counter p)
  if $c = \bot$ **then** $c := 0$ (initialize non-input counter c).
- Increment instructions:
  if $p = i$ **then** $c := c + 1$ (increment counter c)
  if $p = i$ **then** $p := i + 1$ (move to next instruction).
- Decrement instructions:
  if $p = i$ **then** $c := c - 1$ (decrement counter c)
  if $p = i$ **then** $p := i + 1$ (move to next instruction).
- Branch instructions:
  if $p = i \wedge c = 0$ **then** $p := j$ (branch on zero)
  if $p = i \wedge c \neq 0$ **then** $p := k$ (branch on non-zero).

The program counter p and non-input counters c are initially $\bot$; 0 and $\bot$ in the above commands are the nullary symbols for those values; i, j and k are (instruction) numbers; numbers appearing in commands are written in successor notation $(0 + 1 + \cdots + 1)$.[27]

On the other hand, it is also clear that any arithmetical ASM can be programmed in a standard programming language,[28] by storing the current non-$\bot$ values of all dynamic functions (and possibly some $\bot$ values, as well) and interpreting the ASM's conditional updates step-by-step. Such programs, of course, can compute only partial recursive functions.[29]

---

[27] For details of counter-machine ASMs, consult [17].

[28] By "standard programming language", we mean the "idealized" version of existing languages, in the sense that programs are allowed to be arbitrarily large and complex, the namespace is unlimited, and arbitrarily large numbers can be manipulated. The textbook, [29], describes one such idealized language in detail.

[29] This implicit appeal to the formal effectiveness of standard programming techniques (viz. what can be programmed in any formalism can be expressed as general recursion) is sometimes also referred to as an invocation of Church's Thesis (cf. [82, pp. 20–21]), but the omitted details could be fleshed out in what amounts to no more than a programming assignment for an undergraduate course. Shoenfield [91, p. 121] refers to such uses of the thesis as "very convenient", but "not really essential". Kripke [60, p. 13] explains: "If a recursion theorist is given an informal effective procedure for computing a function, he or she will regard it as proved that that function is recursive. However, an experienced recursion theorist will easily be able to convert this proof into a rigorous proof which makes no appeal

In Lisp, for example, one can simply maintain an associative list (an "environment") that records the values of all dynamic functions as location-value pairs $\langle (f, a_1, \dots, a_n), b\rangle$. Initially, only the input values are placed on this list. At that stage, all other dynamic functions have undefined ($\perp$) values for all arguments, so their location-value pairs are not listed. Simulating a step of the ASM involves using equality tests to search for values in the list, in order to evaluate all the expressions in the ASM program's commands. Anything not in the list is presumed to be undefined. If an attempt is made to perform a basic arithmetic or Boolean operation on $\perp$, the result is also $\perp$; otherwise, native Lisp operations are applied to the arguments. The updates for assignments whose guards evaluate to true are computed using the old values stored in the list, before prepending all the new values *en masse* to the list, so that the effect corresponds to the parallel execution of the ASM's commands. (Values closer to the head of the list take precedence, so there is no need to erase old, stored values.) If no update is performed, or if a clash is detected between the values assigned by different updates, the program halts, and outputs the value of *Out*. ⊣

4.3. Church's Thesis. It follows from Theorem 4.5 that:

COROLLARY 4.6. *Every numeric function computed by a state-transition system satisfying the Sequential Postulates, and provided initially with only basic arithmetic, is partial recursive.*

PROOF. By the ASM Theorem (Theorem 3.4), every such algorithm can be emulated by an ASM whose initial states are provided only with the basic arithmetic operations. By Theorem 4.5, such an ASM computes a partial recursive function. ⊣

DEFINITION 4.7 (Arithmetical algorithm). A state-transition system satisfying the Sequential and Arithmetical Postulates is called an *arithmetical algorithm*.

The above corollary, rephrased, is precisely what we have set out to establish, namely:

THEOREM 4.8 (Church's Thesis). *Every numeric (partial) function computed by an arithmetical algorithm is (partial) recursive.*

The converse also follows from Theorem 4.5, since arithmetical ASMs—by their very nature—satisfy Postulates I–IV:

PROPOSITION 4.9. *Every partial recursive function can be computed by an arithmetical algorithm.*

4.4. Recursive oracles. There is nothing very special about the particular set of arithmetic operations allowed in arithmetical states. It can be seen from the proof of Theorem 4.5, and the fact that three counters suffice to compute

whatsoever to Church's Thesis. So working recursion theorists should not be regarded as appealing to Church's Thesis in the sense of assuming an unproved conjecture."

all recursive functions,[30] that it would be enough if states had just zero, successor, equality, and a handful of dynamic nullary operations—serving as counters—for all recursive functions to be computable. (Predecessor of a positive integer can be computed from zero, successor, and equality.) In fact, the (partial) functions computed by all arithmetical ASMs with $n \geq 1$ inputs plus three additional dynamic nullary symbols (including *Out*) are precisely the (partial) recursive functions of arity n.

At the other extreme, effectiveness is maintained no matter how many additional recursive functions are permitted in arithmetical states. Since our algebraic states always include the Boolean truth values, *True* and *False*, and the undefined value, $\perp$, we extend the notion of recursiveness to cover functions that may also involve these values.

In what follows, we apply any unary function, say σ, to n-tuples $\bar{x}$, so $\sigma(\bar{x})$ will serve as shorthand for $(\sigma(x_1), \ldots, \sigma(x_n))$.

Let $\mathbf{B}^{\perp}$ stand for $\{\mathit{True}, \mathit{False}, \perp\}$ and let σ be the following bijection from $\mathbf{N}$ to $\mathbf{N} \uplus \mathbf{B}^{\perp}$:

$$\sigma(x) = \begin{cases} \perp, \mathit{False}, \mathit{True} & \text{if } x = 0, 1, 2, \text{ respectively,} \\ x - 3 & \text{if } x > 2. \end{cases}$$

We say that a function f of arity n over $\mathbf{N} \uplus \mathbf{B}^{\perp}$ is (*partial*) *recursive* if its numeric conjugate $\sigma \circ f \circ \sigma^{-1}$ is (partial) recursive in the ordinary, purely numeric sense. Our terminology is sensible, since a numeric function can be recursive in both senses, or partial recursive in both senses, or else it is not partial recursive in either sense.

The quotient operator ($\div$) in arithmetical states, for instance, which is only partially defined, is recursive in this wider sense:

$$m \div n = \begin{cases} \perp & \text{if } n = 0, \\ \lfloor m/n \rfloor & \text{if } n > 0. \end{cases}$$

A divisibility predicate defined as

$$n \mid m = \begin{cases} \perp & \text{if } n = 0, \\ \mathit{True} & \text{if } n > 0 \text{ and } n \times (m \div n) = m, \\ \mathit{False} & \text{otherwise} \end{cases}$$

is also recursive in this sense.

The proof of Theorem 4.5 holds fast even when initial states incorporate many such recursive functions (because they are all programmable like basic arithmetic is). Later, we will make use of the following variant of Corollary 4.6, which uses this slightly more general notion of recursive function and which allows for arbitrarily many recursive oracles:

[30] See [3] (also [86]) for the fundamental weakness of a two-counter machine, as compared to a machine with three counters or more.

COROLLARY 4.10. *Every partial function computed by a state-transition system satisfying the Sequential Postulates, whose states are arithmetic and whose initial states only have recursive (possibly partially defined) operations, is partial recursive.*

Recall that the operations of algebraic states are always total, formally speaking, though they need not be numeric. So, when this corollary speaks of a partially-defined function f as being recursive, it means that the total function f, whose range includes $\perp$, is recursive (not just partial recursive) in the expanded sense given above.

The partial function computed by such an algorithm, on the other hand, may be only partial recursive, since the algorithm might not terminate for some inputs. For example, there is an algorithm that satisfies the conditions of the above corollary and which interprets Turing machines that act on string-representations of the natural numbers. The interpreter computes the following *partial* recursive function:

$$\mathrm{TM}(m, n) = \begin{cases} z & \text{if machine number } m \text{ on input } n \\ & \quad \text{terminates with numeric output } z, \\ \perp & \text{if machine number } m \text{ on input } n \\ & \quad \text{aborts or terminates with non-numeric output.} \end{cases}$$

When machine number m does not terminate at all on input n, the algorithm also does not terminate, and $\mathrm{TM}(m, n)$ returns nothing, not even $\perp$.

In Section 6, we will see how to incorporate richer domains than mere arithmetic.

§**5. Relative effectivity.** Church provided some justifications for his claim that "every function, an algorithm for the calculation of the values of which exists, is effectively calculable" by means of recursion [23, p. 357]. But, as Sieg [94, p. 78] clarifies, Church's argument lacked a formal analysis of the recursiveness of the individual steps in the performance of an algorithm, a point regarding which Church was quite aware. (See also [103, p. 291].) So, as Shoenfield [91, pp. 120–121] also makes clear, it all boils down to the effectivity of the operations that are applied at each step of a computation.

We have seen in the previous section that effectiveness is guaranteed by the Sequential Postulates and the Arithmetical State Postulate. We also saw that additional recursive functions in initial states do not increase computational power: Bounded Exploration guarantees that each single step is in fact effective, because it allows only a bounded number of applications of those initial functions to values derived in the same fashion during the preceding finitely many steps. Adding non-recursive functions to the initial state, on the other hand, is a different story, taken up next.

A (partial) function f is said to be (*partial*) *recursive relative to* a set of functions $\mathcal{B}$ if its values can be inferred by equational reasoning from a set of (true) equations involving $\mathcal{B}$. This is equivalent to stating that f can be obtained by composition, primitive recursion, and/or minimization from the *oracles* $\mathcal{B}$. See [49, §63]. We assume that $\mathcal{B}$ always includes zero, successor, and equality. The ordinary recursive functions are just those that are defined in this way from only zero, successor, and equality.

THEOREM 5.1. *A numeric function is partial recursive relative to "oracular" functions $\mathcal{B}$ if and only if it is computable by an ASM operating over domain* $\mathbf{N} \uplus \mathbf{B}^{\perp}$ *and initial functions $\mathcal{B}$ (containing at least zero, successor, and equality), but no other functions defined in its initial state.*

PROOF. It is an ordinary programming exercise to show how to obtain an ASM for the composition of the functions computed by two given ASMs, or for primitive recursion, given ASMs for the zero and non-zero cases, or for iterating to look for the minimal input value (if there is one) such that a given ASM returns zero. So, by induction on the construction of a function that is partial recursive in $\mathcal{B}$, we know that there is an ASM that computes any such function, given oracles for $\mathcal{B}$.

On the other hand, one can write an interpreter for such ASMs, which can be programmed in any standard programming language,[28] except for calls to the oracles. Such an interpreter can in turn be implemented in terms of the functions in $\mathcal{B}$, using composition, primitive recursion, and minimization.[29] ⊣

Clearly, ASMs necessarily satisfy the Sequential Postulates, so any relatively recursive function can be computed by a process satisfying those postulates. By Theorems 3.4 and 5.1, the converse is also true:

COROLLARY 5.2. *The only numeric functions that are algorithmically computable by a process satisfying the Sequential Postulates are those that are partial recursive relative to the initial functions.*

We have, then, what Kleene [49, p. 332] (paraphrased) refers to as:

> **Thesis $\mathbf{I}^{*\dagger}$.** *Every (partial) function which is effectively calculable relative to some initial functions is (partial) recursive relative to those functions.*

This version of Church's Thesis follows from the Sequential Postulates alone, without Arithmetical State. The clause "relative to some initial functions" means that the algorithm is allowed to apply "black box" primitive operations $\mathcal{B}$. The results of the previous section are obtained whenever $\mathcal{B}$ is a subset of the recursive functions.

In the development so far, all functions operate over numbers or truth values. But one can add as many non-numerical values to the domain as one wishes to more naturally mimic human "notations" to keep track of things while in the midst of arithmetic calculations. Moreover, one may incorporate algorithmic operations on strings of symbols, or on other kinds of objects, as shown in the next section.

§6. Arithmetized effectivity. We consider now algorithms that operate over larger domains than just natural numbers, domains that may include rationals, vectors, matrices, strings, lists, graphs, etc. To work with such objects, an algorithm would be provided with operations like division of rationals, vector addition, matrix multiplication, string concatenation, list sorting, or graph complementation.

6.1. Arithmetizable states. Indeed, one might naturally employ non-numerical capabilities in the process of computing what is a strictly numeric function. The object domain of such an algorithm would include elements besides numbers and truth values, and its initial states would include operations over those auxiliary domains and operations connecting those domains with the natural numbers, in addition to purely arithmetic operations.

As we are only interested in effective computations, it is necessary to limit the initial repertoire of operations to what is undeniably effective. Anything more complicated should be programmed, just as sophisticated arithmetic operations are.

Let the enriched object domain of an algorithm be $D \supseteq \mathbf{N} \uplus \mathbf{B}^{\perp}$ and let $\rho \colon D \to \mathbf{N} \uplus \mathbf{B}^{\perp}$ be an injection of that domain into the natural numbers that preserves the special elements $\mathbf{B}^{\perp} = \{\mathit{True}, \mathit{False}, \perp\}$. We will use ρ_{N} to denote the restriction $\rho \restriction \mathbf{N}$ of ρ to the natural numbers in D.

We will say that a function f of arity n over D is *ρ-recursive* if there exists a recursive function $\widehat{f}$ over $\mathbf{N} \uplus \mathbf{B}^{\perp}$ of the same arity such that $\widehat{f}(\rho(\bar{x})) = \rho(f(\bar{x}))$ for all $\bar{x} \in D^n$, that is, if $\rho \circ \widehat{f} = f \circ \rho$. In other words, f is ρ-recursive if there is some recursive extension $\widehat{f}$ of the function $\widetilde{f}$ over the image $\rho(D)$ whose graph is $\{(\rho(\bar{x}), \rho(y)) \mid (\bar{x}, y) \in f\}$. Recursiveness of the witness $\widehat{f}$ is meant in the expanded sense of Section 4.4. Note that the witness is not necessarily unique, since ρ need not be onto; what exactly $\widehat{f}$ produces for arguments not all in $\rho(D)$ is immaterial.

We adopt a generous version of the presumption that initial operations are effective:

POSTULATE IVB (Arithmetizability). *Initial states are blank. Up to isomorphism, all initial states share the same domain and static operations, and there is exactly one initial state for any given numerical input values. There is an encoding ρ whose restriction ρ_{N} to the natural numbers is recursive* (*in the ordinary sense*), *and via which all static operations of the initial states are ρ-recursive.*

The completely undefined dynamic functions of blank states are trivially ρ-recursive for any encoding ρ (they are witnessed by a completely undefined function), as are the nullary input operations (witnessed by their images under ρ). Note that this postulate places no demands on the dynamic functions that evolve over the course of a computation, only on the static operations of initial states.

Remark 6.1. It is quite natural to demand that an encoding and its inverse be "effective" in some informal sense (cf. [82, p. 27]). We demand less: only that the encoding ρ_{N} of the natural numbers $\mathbf{N}$ within the domain D be effective. Indeed, as long as the successor function s on the natural numbers is ρ-recursive, then ρ_{N} is in fact recursive, since $\rho(0)$ is some constant and $\rho(s(n)) = \widehat{s}(\rho(n))$, where $\widehat{s}$ is the recursive witness for successor (cf. [81, 18]). A similar approach to effectiveness in non-numerical domains D is taken in the field of computable algebra, namely that operations on D are tracked by homomorphic images in $\mathbf{N}$ (e.g., [34, 64, 79]), but our conditions on the encoding are noticeably weaker.

Remark 6.2. When, in addition to ρ_{N} being recursive, the image $\rho_{\mathrm{N}}(\mathbf{N})$ it gives of the natural numbers is recursive (as in [64, 79]), then the inverse of ρ_{N} is also (formally total and) recursive:

$$\rho_{\mathrm{N}}^{-1}(x) = \begin{cases} \min n.\, \rho(n) = x & \text{if } x \in \rho(\mathbf{N}), \\ \bot & \text{otherwise.} \end{cases}$$

In that case, every recursive function f over $\mathbf{N} \uplus \mathbf{B}^{\bot}$ is automatically ρ-recursive, since its conjugate $\rho_{\mathrm{N}}^{-1} \circ f \circ \rho_{\mathrm{N}}$ by ρ_{N} is its recursive witness. However, Lemma 6.8 and Theorem 6.4 below hold even when ρ_{N} has a perversely non-recursive image. So, we do not impose this otherwise perfectly reasonable requirement on encodings. See the discussion in Section 6.4.

6.2. Arithmetized algorithms. We will say that an algorithm (and its set of static operations) operating over a domain D is *arithmetizable* if there is an encoding ρ of D such that the conditions of the Arithmetizability Postulate are fulfilled.

We generalize now the notion of arithmetical algorithm (Definition 4.2) to allow for basic operations over other domains:

DEFINITION 6.3 (Arithmetized algorithm). A state-transition system satisfying the Sequential and Arithmetizability Postulates is called an *arithmetized algorithm*.

This gives the following variation on Church's Thesis (Theorem 4.8) for enriched domains:

THEOREM 6.4. *Every numeric* (*partial*) *function computed by an arithmetized algorithm is* (*partial*) *recursive.*

The special case when the object domain is not enriched ($D = \mathbf{N} \uplus \mathbf{B}^{\bot}$) was already dealt with in Section 4.

We will make use of several easy lemmata.

LEMMA 6.5. *Suppose* $\alpha \rightsquigarrow^* \alpha^*$ *by a computation of an ATS* A, *and let* $\beta = \pi(\alpha)$ *for some isomorphism* π. *Then one also has* $\beta \rightsquigarrow^* \beta^*$ *via* A, *where* $\beta^* = \pi(\alpha^*)$. *Furthermore,*

$$[\![t]\!]_{\beta^*} = \pi([\![t]\!]_{\alpha^*}),$$

for all (*ground*) *terms* t *over the vocabulary of* A.

PROOF. To begin with, by the nature of isomorphisms, $[\![t]\!]_\beta = \pi([\![t]\!]_\alpha)$ for all t. By virtue of the Abstract State Postulate (II), the isomorphism π is preserved by every transition of A, from which it follows that $[\![t]\!]_{\beta'} = \pi([\![t]\!]_{\alpha'})$ whenever $\alpha \rightsquigarrow \alpha'$ and $\beta \rightsquigarrow \beta' = \pi(\beta)$. The result follows by induction on the computation $\alpha \rightsquigarrow^* \alpha^*$. ⊣

Let $D \subseteq \widehat{D}$ for domains D and $\widehat{D}$, and suppose that algebras α and β have domains D and $\widehat{D}$, respectively, and share the same vocabulary. Then α is called a *subalgebra* of β, denoted $\alpha \subseteq \beta$, if the operations of β are closed on the subdomain D and every operation f_α of α is the restriction $f_\beta \restriction D$ of the corresponding operation f_β of β. This implies, of course, that $[\![t]\!]_\alpha = [\![t]\!]_\beta \in D$ for all (ground) terms t over their joint vocabulary.

LEMMA 6.6. *Suppose an ASM M with states S has a transition $\alpha \rightsquigarrow \alpha'$ for states α and α' with domain $D \subseteq \widehat{D}$. Suppose further that $\alpha \subseteq \beta$, where* Dom $\beta = \widehat{D}$. *Then an ASM $\widehat{M}$ with exactly the same vocabulary and the same program as M, but with extended states $\{\psi \mid \exists \varphi \in S. \varphi \subseteq \psi\}$ (closed under isomorphisms), engenders the corresponding transition $\beta \rightsquigarrow \beta'$ (for extended states β, β'), where $\alpha' \subseteq \beta'$.*

PROOF. All the extra elements introduced by the enlarging of the domain D of α are inaccessible in β (i.e., there is no ground term t over the vocabulary for which $[\![t]\!]_\beta \in \widehat{D} \setminus D$), and, therefore, those extra domain elements have no influence on transitions. Specifically, the update set $\Delta(\alpha)$ for M is identical to $\Delta(\beta)$ for $\widehat{M}$ whenever $\alpha \subseteq \beta$. ⊣

Let $\rho\colon D \to \widehat{D}$ be an injection from domain D to domain $\widehat{D}$, and suppose that algebras α and β have domains D and $\widehat{D}$, respectively, and share the same vocabulary. Then one says that α is *embedded* in β via ρ, written as $\alpha \hookrightarrow \beta$, if $\widetilde{\rho}(\alpha) \subseteq \beta$, where $\widetilde{\rho}$ is the bijection obtained by restricting ρ to its image $\rho(D) \subseteq \widehat{D}$. This implies (using Lemma 6.5) that

$$[\![t]\!]_\beta = [\![t]\!]_{\widetilde{\rho}(\alpha)} = \widetilde{\rho}([\![t]\!]_\alpha) = \rho([\![t]\!]_\alpha) \in \rho(D),$$

for all (ground) terms t over their joint vocabulary. For any collection S of abstract states, let the corresponding collection $\widehat{S}$ (for the embedding ρ) be $\{\psi \mid \exists \varphi \in S. \varphi \hookrightarrow \psi\}$, closed under isomorphisms.

Combining the above lemmata, we get the following:

LEMMA 6.7. *Suppose an ASM M with states S has a computation $\alpha \rightsquigarrow^* \alpha^*$ for states α and α^* with domain D. Suppose further that $\alpha \hookrightarrow \beta$ via some injection $\rho\colon D \to \widehat{D}$. Then an ASM $\widehat{M}$ with exactly the same vocabulary and the same program as M, but with states $\widehat{S}$, engenders the corresponding computation $\beta \rightsquigarrow^* \beta^*$ $(\beta, \beta^* \in \widehat{S})$, where $\alpha^* \hookrightarrow \beta^*$. Furthermore,*

$$[\![t]\!]_{\beta^*} = \rho([\![t]\!]_{\alpha^*}),$$

for all (ground) terms t over the vocabulary of M.

PROOF. By Lemma 6.5, M also has a computation $\widetilde{\rho}(\alpha) \rightsquigarrow^* \widetilde{\rho}(\alpha^*)$, and, by repeated application of Lemma 6.6, $\widehat{M}$ has a computation $\beta \rightsquigarrow^* \beta^*$, with $\widetilde{\rho}(\alpha^*) \subseteq \beta^*$, for any β such that $\widetilde{\rho}(\alpha) \subseteq \beta$. In other words, $\beta \rightsquigarrow^* \beta^* \hookleftarrow \alpha^*$, when $\beta \hookleftarrow \alpha \rightsquigarrow^* \alpha^*$. Accordingly, we also have $[\![t]\!]_{\beta^*} = [\![t]\!]_{\widetilde{\rho}(\alpha^*)} = \rho([\![t]\!]_{\alpha^*})$, for all terms t. ⊣

LEMMA 6.8. *If* $\widehat{g}\colon \mathbf{N} \to \mathbf{N}$ *is a partial recursive function and* $\rho_{\mathrm{N}}\colon \mathbf{N} \to \mathbf{N}$ *is a recursive injection, then* $\rho_{\mathrm{N}} \circ \widehat{g} \circ \rho_{\mathrm{N}}^{-1}$ *is also partial recursive.*

PROOF. Since the injection ρ_{N} is recursive, its inverse $\rho_{\mathrm{N}}^{-1}(x) = \min n.\, \rho(n) = x$ is partial recursive. Its composition with other partial recursive functions is partial recursive. ⊣

PROOF OF THEOREM 6.4. Suppose A is the arithmetized algorithm operating over the rich domain D and $\rho\colon D \to \mathbf{N} \uplus \mathbf{B}^{\perp}$ is the postulated injective encoding. By the ASM Theorem (Theorem 3.4), algorithm A (which satisfies Postulates I–III) is emulated by an ASM M that computes the same numeric function g as does A over D. Up to isomorphism, M has one initial state for every possible combination of numeric inputs.

Consider an ASM $\widehat{M}$ with the same vocabulary and program as M, but with states $\widehat{S}$, and with initial states in which every static symbol f is interpreted as its witness $\widehat{f}$, each dynamic symbol other than the input symbols is completely undefined, and input symbols take all possible values in $\mathbf{N}$. Modulo isomorphism, all states of $\widehat{M}$ have domain $\mathbf{N} \uplus \mathbf{B}^{\perp}$. Restricting an initial state $\widehat{\alpha}$ of $\widehat{M}$ to $\rho(D)$ gives a state $\widetilde{\alpha}$ that is isomorphic by ρ to an initial state α of M, since the restriction $\widehat{f} \upharpoonright \rho(D)$ of each witness $\widehat{f}$ is isomorphic to the original static operation f of M.

Static arithmetic operations of M are reinterpreted as their witnesses in $\widehat{M}$, but Boolean operations and equality remain intact. By the Arithmetizability Postulate, $\widehat{M}$'s initial states provide recursive functions only. Since Definition 4.2 does not require arithmetical states to have any specific operations, other than equality and Boolean connectives, $\widehat{M}$ does in fact satisfy the requirements of Corollary 4.10, and must compute a recursive function $\widehat{g}$.

Consider any computation $\alpha \rightsquigarrow^* \alpha^*$, where α is an initial state of M and α^* is a terminal state. The ASM $\widehat{M}$ has an initial state $\widehat{\alpha}$ such that $\alpha \hookrightarrow \widehat{\alpha}$, and wherein $[\![In_i]\!]_{\widehat{\alpha}} = \rho([\![In_i]\!]_{\alpha})$ for each input In_i. By Lemma 6.7, there is a corresponding computation $\widehat{\alpha} \rightsquigarrow^* \widehat{\alpha}^*$ for $\widehat{M}$, such that $\alpha^* \hookrightarrow \widehat{\alpha}^*$. Furthermore, by that lemma, $\widehat{z} = \rho(z)$, where $\widehat{z} = [\![Out]\!]_{\alpha^*}$ is the output of M and $z = [\![Out]\!]_{\widehat{\alpha}^*}$ is the output of $\widehat{M}$.

By construction, $\widehat{x} = \rho(\bar{x})$, where $\bar{x} = [\![\overline{In}]\!]_{\alpha}$ are the inputs to M and $\widehat{x} = [\![\overline{In}]\!]_{\widehat{\alpha}}$ are the inputs to $\widehat{M}$. By convention, $z = g(\bar{x})$ and $\widehat{z} = \widehat{g}(\widehat{x})$ are the output values computed by M and $\widehat{M}$, respectively. It follows that

$$\rho(g(\bar{x})) = \rho(z) = \widehat{z} = \widehat{g}(\widehat{x}) = \widehat{g}(\rho(\bar{x})),$$

for all $\bar{x}$, or, put another way, $g \circ \rho = \rho \circ \widehat{g}$.

Recall that ρ_{N} is postulated to be recursive and that the partial function g computed by A is numeric. Hence, it follows from Lemma 6.8 that $g = \rho_{\mathrm{N}} \circ \widehat{g} \circ \rho_{\mathrm{N}}^{-1}$ must also be partial recursive, as claimed. And if g is total, it is recursive. ⊣

Remark 6.9. Suppose we wish to arithmetize a many-sorted domain $D = D_0 \uplus D_1 \uplus \cdots \uplus D_m \uplus \{\bot\}$, $m \geq 1$. Specifically, we are assuming that the domain is partitioned into $m+1$ *disjoint* sorts: the natural numbers $D_0 = \mathbf{N}$, zero or more auxiliary domains $D_1, \ldots, D_{m-1}$, and the Booleans, D_m. (The undefined value, $\bot$, is part of the domain but does not belong to any of the sorts.) It may be convenient to arithmetize such a D, sort by sort: Let $\mathbf{N}$ be stratified into m disjoint parts $\mathbf{N}_j$, $0 \leq j < m$, via m *stratification* injections $\sigma_j \colon \mathbf{N} \to \mathbf{N}$, such that $\uplus_j \mathbf{N}_j \subseteq \mathbf{N}$ and $\sigma_j(\mathbf{N}) \subseteq \mathbf{N}_j$ for each j. Then, given individual embeddings $\tau_j \colon D_j \to \mathbf{N}$, and letting $\tau_0 \colon \mathbf{N} \to \mathbf{N}$ and $\tau_m, \sigma_m \colon \mathbf{B}^\bot \to \mathbf{B}^\bot$ all be identity maps (so that the natural numbers and the Boolean truth values are preserved), we can define the encoding $\rho(x) = \sigma_j(\tau_j(x))$ for $x \in D_j$, and $\rho(\bot) = \bot$. Let $\tau(x) = \tau_j(x)$ for $x \in D_j$ and $\tau(\bot) = \bot$. The restriction ρ_{N} of such a ρ to $\mathbf{N}$ is recursive as long as the given stratification function σ_0 for encoded natural numbers is recursive. If σ_0^{-1} is also recursive, then all purely numeric functions over D are in fact ρ-recursive, as explained in Remark 6.2. Compare [5].

Example 6.10. One convenient stratification mapping is obtained by partitioning the natural numbers into m residue classes $\mathbf{N}_j$ modulo m, corresponding to the m sorts of D. The mappings and their inverses are easily computable, as follows:

$$\sigma_j(x) = mx + j,$$
$$\sigma^{-1}(x) = x \div m,$$

for each j, $0 \leq j < m$, where $\div$ gives the integer quotient. Since this σ^{-1} does not depend on the stratum, we have omitted the subscript j. The classification function

$$\kappa(x) = x \bmod m$$

is also effective. This regimen is tantamount to the standard type-tagging scheme used in programming languages, and is clearly recursive.

Example 6.11. As an example of such a stratified encoding, consider an algorithm that uses finite strings over some alphabet Σ, in addition to the natural numbers. It is not hard to conjure up a mapping that encodes finite strings from the auxiliary domain $D_1 = \Sigma^*$ as natural numbers. For example, if Σ is a binary alphabet, then the injection $\tau_1(uv)$ could be $2^i(2j+1)$, where u is the prefix of all leading zeroes, i is the length of u, and j is the binary value of the suffix v. Composing this with the stratification mappings σ

of the previous example gives the following family ρ of domain encodings: $\rho_0(n) = 2n$ for numbers n; $\rho_1(w) = 2\tau_1(w) + 1$ for strings w.

Remark 6.12. When all the operations (save equality) of an ATS are simply typed, the condition needed for arithmetizability can be simplified by ignoring the stratification. A *simple type* takes the shape $S_1 \times \cdots \times S_n \to S_0$ $(n \geq 0)$, where every S_i is one of the sorts D_k of the domain D, meaning that $f(x_1, \ldots, x_n)$ yields an element of S_0 or $\perp$ whenever $x_i \in S_i$ for all the arguments x_i, and that $f(x_1, \ldots, x_n)$ yields $\perp$, otherwise. Let τ_j and σ_j be as described in Remark 6.9; suppose the σ_j and their inverses are recursive as in Example 6.10; and let ρ be the composite encoding defined in that remark. Consider a function f with result sort $S_0 = D_j$ and input sorts $\bar{S}$ having indices $\bar{\imath}$. Define the tuple-mapping $\sigma_{\bar{\imath}}$ applied to a list of arguments $\bar{x}$ to be the result of applying the corresponding functions $\sigma_{\imath_i}$ to each component x_i of $\bar{x}$. With this machinery, a function f is ρ-recursive if it is τ-recursive, since if $\widetilde{f}$ is the witness of τ-recursiveness, then $\widehat{f} = \sigma_{\bar{\imath}}^{-1} \circ \widetilde{f} \circ \sigma_j$ is a recursive witness of ρ-recursiveness. To wit,

$$\rho \circ \widehat{f} = \tau \circ \sigma_{\bar{\imath}} \circ \sigma_{\bar{\imath}}^{-1} \circ \widetilde{f} \circ \sigma_j = \tau \circ \widetilde{f} \circ \sigma_j = f \circ \tau_j \circ \sigma_j = f \circ \rho.$$

Note that we are not insisting that the encodings τ_j are effective in any sense; it is enough that the stratification σ is effective for the requirements of the postulate to be met.

Example 6.13. To manipulate strings, Post's tag machines [77] use the following all-powerful set of basic string operations:

- Read: $\Sigma^* \to \Sigma$ (read first letter);
- Delete: $\Sigma^* \to \Sigma^*$ (delete first letter);
- Add: $\Sigma^* \times \Sigma \to \Sigma^*$ (add letter to end).

The encoding τ_1 of Σ^* given in Example 6.11 turns these basic string operations into combinations of addition, multiplication, exponentiation, and taking remainders. For example, $\widetilde{\mathrm{Read}}(n)$ would return $2 = \tau_1(0)$ if n is even, and $3 = \tau_1(1)$ if it's odd.

Remark 6.14. When the types of operations are the union of simple types (like the "overloaded" $>$ operator in many programming languages, which compares strings as well as numbers and returns a Boolean value in both cases), it is enough if there are recursive witnesses for every combination of input types, since operations can be "dispatched" to the appropriate witness based on the sorts of the arguments. (By "union" we mean that the graph of the function on correctly-typed arguments is the union of the graphs of simply-typed functions on their correct types.) For example, suppose a domain D includes three sorts D_0, D_1, and D_2, and some function $f: D \to D$ is the union of two simply-typed unary functions $f_1: D_1 \to D_0$ and $f_2: D_2 \to D_0$. Suppose further that there are witnesses $\widetilde{f_1}, \widetilde{f_2}: \mathbf{N} \to \mathbf{N}$

for some τ_i as in the previous remark. Then we can use the stratification scheme σ of Example 6.10 and note that the recursive function

$$\widehat{f}(n) = \begin{cases} \sigma_0(\widetilde{f}_1(\sigma^{-1}(n))) & \text{if } \kappa(n) = 1, \\ \sigma_0(\widetilde{f}_2(\sigma^{-1}(n))) & \text{if } \kappa(n) = 2, \\ \perp & \text{otherwise} \end{cases}$$

is a witness that f is ρ-recursive for $\rho(x) = \sigma_j(\tau_j(x))$, as in Remarks 6.9 and 6.12.

Remark 6.15. When operations have generalized types and return results of more than one sort (as can list operations in Lisp), ρ-recursiveness with respect to a more complicated encoding ρ of all the sorts of elements of D, as in Remark 6.9, is called for.

6.3. Turing's Thesis. As a byproduct of the ASM Theorem (Theorem 3.4) and the above result (Theorem 6.4), one obtains a straightforward method of showing that other deterministic models of computation cannot compute more than the partial recursive functions.

Consider, by way of example, Markov's *normal algorithms* [66], which repeatedly apply a series of substring-replacement rules of the form $u \to v$ to a given input string. To connect effectiveness of normal algorithms with that of the recursive functions, one needs to show that the operations of testing for the occurrence of a substring u in a string w and of replacing the first such occurrence (of u in w) with another substring (v) are both arithmetizable. This is quite easy with an encoding ρ in the style of that given in Example 6.11 above. By the ASM Theorem, issues of control—like that of determining the first applicable rule, if any, in a Markov program, and of terminating as soon as one of the rules that are marked "final" is applied—can always be handled effectively, using numbers to represent control states. As a consequence of the previous theorem, one may conclude that regardless of how numerical inputs are given as strings to normal algorithms (whether a number n is represented by a string of n marks in "tally" notation, or—more compactly—in decimal notation, or using any other convention), as long as the homomorphic image under ρ of that number-representation function is recursive, one can be sure that only partial-recursive functions can be computed.

Just as we have shown above that the recursive functions are the only numeric functions that can be computed by an effective algorithm, we could likewise have shown that Turing machines compute all effective *string* functions, by adopting some basic set of primitive string operations, like Post's (see Example 6.13), and postulating that initial states of string algorithms are endowed with nothing additional. In other words, the exact same approach as that taken above gives an axiomatization and proof of Turing's Thesis regarding string-based effective computation. We have seen how to enrich an arithmetic model with strings. By mapping numbers to strings, instead of strings to numbers, one gets the analogous enrichment of string-based machines, empowering them to compute all recursive number-theoretic functions.

6.4. Significance. Theorem 6.4 and its proof demonstrate the full strength of Church's Thesis: No matter what additional data structures an algorithm has at its disposal, it cannot compute any non-recursive numeric functions, since essentially the same computations can be performed over the natural numbers. The ASM Theorem was crucial in this argument, enabling us to cover all possible algorithms, regardless of the data structures employed.

To axiomatize effectiveness of algorithms operating over such domains, we have formalized requirements for domain encodings. Church [23, p. 345] asserted that non-numerical domains "can be described in number-theoretic terms". Particular encodings (like Gödel numberings) are used all the time; some are more "natural" than others. Rogers [82, p. 27] demands the following of such encodings:

> The coding is chosen so that (a) it is itself given by an informal algorithm in the unrestricted sense; and (b) it is *reversible*; i.e., there exists an informal algorithm (in the unrestricted sense) for recognizing code numbers and carrying out the reverse "decoding" mappings from code numbers to nonnumerical entities. Furthermore, it is stipulated that a coding shall be used only when (c) an informal algorithm exists for recognizing the expressions that constitute the uncoded, nonnumerical class.

Demanding the existence of "informal algorithms" for the encoding and decoding of the additional data structures, however, is problematic. And there is no accepted formal sense of effectiveness that covers operations over arbitrary domains. (See [72, 88, 19] for discussions of this problem.) Instead, we have insisted only that—under *some* representation—the homomorphic images of the basic native operations, which track the native operations on the natural numbers, be effective in the technical, recursive sense, and that the mapping ρ_{N} of the natural numbers alone be recursive. We propose that such encodings be deemed "reasonable"; they provide just the right amount of effectiveness. With this approach, recourse to informal considerations is not needed.

As Ada Lovelace asserted in 1843 [68]:

> Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they

> were letters or any other general symbols; and in fact it might bring out its results in algebraical notation were provisions made accordingly.

§7. Conclusion.

Our goal in this work has been to remedy the situation described thus by Montague [72, p. 432]: "Discussion of Church's thesis has suffered for lack of a precise general framework within which it could be conducted." We have shown how the Sequential ASM Postulates provide just such a framework, and how Church's Thesis follows from them, plus the postulate that nothing uncomputable is given *ab initio*.

We saw in the introduction that Gödel surmised that Church's Thesis may follow from appropriate axioms of computability. But, as far as we can ascertain, no complete axiomatization has previously been presented in the literature. In fact, the challenge of proving Church's Thesis is first in Shore's list of "pie-in-the-sky problems" for the twenty-first century [21]. Whereas Kripke [60, p. 14] feels that it is "a very difficult task", Friedman [33] predicted that sometime in this century, "There will be an unexpected striking discovery that any model of computation satisfying certain remarkably weak conditions must stay within the recursive sets and functions, thus providing a dramatic 'proof' of Church's Thesis."

Our axiomatization provides a small number of principles that imply Church's Thesis, and focuses attention on the axioms. Thus, to the extent that one might entertain the notion that there exist non-recursive effective functions, one must reject one or more of these postulates.[31] The need for "continual verification" of the legitimacy of Church's identification of recursiveness with effectivity, to which Post [75, p. 105] referred, can now center around the universality of the individual axioms.

7.1. Previous analyses. Turing long ago dissected the essentials of computation. (See Section 1.2.) As Gödel [38, p. 72] commented: "Turing's work gives an analysis of the concept of 'mechanical procedure' (alias 'algorithm' or 'computation procedure' or 'finite combinatorial procedure'). This concept is shown to be equivalent with that of a Turing machine." See also Kleene in [53, p. 30]:

> Computation, theoretically considered (to be performable for all possible values of the independent variables), is idealized. Turing's analysis takes this idealized aspect of it into account. A Turing machine is like an actual digital computing machine, except that

[31] For one well-known example of the claim that there is an "effective" mode of reasoning that computes a non-recursive function, see Lucas [63]: "One can feel confident without having an effective method within the meaning given to effective—i.e., programmable into a Turing machine. . . . It is not necessary that all reasoning must in this sense be effective. And in the sense in which reasoning might be necessarily effective, effectiveness does not imply computability."

> (1) it is error free..., and (2) by its access to an unlimited tape it is unhampered by any bound on the quantity of its storage of information or "memory".

Turing [107] stressed the finite limitations on exploration of state by an "idealized" human computer, but his analysis was on an informal level, and related only to two-dimensional symbolic manipulations, though his interests extended into higher dimensions [110]. Turing analyzed human reckoning, but there are many things that modern computers (let alone future computers) can do that are very hard or even impossible for a human computer to imitate.

Kolmogorov [56] generalized Turing's analysis, insisting on the "limited complexity" of operations and on the locality of information needed to determine the next state, but he too gave no precise characterizations. Kolmogorov and Uspensky [57] and Sieg and Byrnes [101] proposed sufficient conditions on labeled graphs to ensure boundedness of complexity and locality of action for Kolmogorov-like machines, but their conditions are overly restrictive, and cannot characterize effective computation, in its generality.

Gandy [35] proposed postulates for human and machine effectivity. He defined a model, "Gandy machines", whose states are described by hereditarily finite sets. Effectivity of Gandy machines is achieved by bounding the rank (depth) of states, insisting that they be unambiguously assemblable from individual "parts" of bounded size, and requiring that transitions have local causes. Gandy's ideas have been expounded and simplified by Sieg and Byrnes [102].[32] Gandy admitted that his model cannot emulate all computations [35, p. 146]: "Despite the liberality advertised ... there is a limit to what a machine can do in a single step."[33]

An alternative approach to proving Church's Thesis has been suggested by Kripke [61], based on "Hilbert's Thesis" that "any mathematical argument ... can be formalized in some first-order language", and—in particular—arguments about the effects of applying the instructions of an algorithm can be so formalized.

7.2. This work. Unlike all previous formalizations of effectiveness, the postulates proposed here apply to transition systems with *arbitrary* structures as states. Our abstract states can hold one-dimensional tapes (as in Turing's original work [107]), two-dimensional arrays (as extended in [93]), bounded-degree graphs (as in Kolmogorov machines [56, 57]), or bounded-rank hereditarily finite sets (as in Gandy machines [35, 102]). But, in fact, states can also be multi-dimensional grids, or unbounded-degree graphs, or sets with unbounded rank, so long as the program satisfies the Bounded Exploration Postulate. In this sense, we are truer to the claim of Kolmogorov

[32]The explicit bound on rank is removed in Sieg's more recent work [96, 97, 99, 100].

[33]Indeed, the algorithm in [30] (determining the "parity" of certain graphs) cannot be naturally encoded as hereditarily finite sets of bounded rank, as shown there.

and Uspensky [57, 16]: "We . . . are talking not about a reduction of an arbitrary algorithm to an algorithm in the sense of our definition, but [contend] that every algorithm essentially satisfies the proposed definition."

Our postulates of effectivity relate to these arbitrary structures—without encoding complete states as numbers, strings, or graphs. Moreover, we place no restrictions on state transitions, other than the respecting of isomorphism and Bounded Exploration. We do not straitjacket transitions to follow any particular format—as, for example, in Turing's formalism. The critical restriction, the one that makes individual steps effective, is Bounded Exploration.

Abstract states are what allow this crucial Bounded Exploration Postulate to be phrased at the abstract level of algorithms, rather than on some particular representation level. Our analysis culminated in Theorem 6.4, which is Church's Thesis expressed so as to allow algorithms to employ arbitrary non-numerical auxiliary domains, while placing no restrictions on the effectiveness of the encoding of those domains as numbers.

On account of their abstractness, our postulates apply equally well to all sequential machine models in the literature. The three Sequential Postulates apply as is; the fourth axiom, that initial states include only undeniably effective operations, would need to be expressed in terms of the domain and primitive operations of the specific computational model.

Some might aver that convoluted structures can be *encoded* linearly or graphically, and that complex transitions can be *decomposed* into smaller steps, and then effectiveness of the encoded operations can be established. But it is far from evident that computational power is not increased or decreased by particular representations. It can be the case that the implementation f' of what is an intuitively effective function f over some rich domain D does not satisfy axioms of effectivity expressed on the level of the domain D' in which it is implemented—simply because the choice of encoding is not ideal. In fact, there is no a priori reason to believe that there always is a reasonable encoding such that everything that can be done effectively (in the intuitive sense) in one domain can be done effectively in another. On the other hand, it may be that f is uncomputable in one model of computation, but—having been encoded—its implementation f' becomes computable over D'. Given a sufficiently malevolent encoding, one can "effectively" compute blatantly *uncomputable* functions. (See [89, 18, 81] for discussions of this point.) It is, therefore, imperative to deal with effectiveness in the same terms as those in which the algorithm operates—as undertaken here.

7.3. Related issues. Some of Gandy's considerations were motivated by physical limitations of *machines*, like "the finite velocity of propagation of effects and signals" [35, p. 135]. We, on the other hand, are not concerned at all with what Gandy calls "Thesis M" (also called the "Physical Church–Turing Thesis"), namely, that whatever can be calculated by a *physical machine* can be computed by a Turing machine, a claim regarding limitations that physics may, or may not, place on *physical* computing devices.[34] As already pointed out, our analysis is not specific to a computational model operating over hereditarily finite sets, but applies to arbitrary state-transition systems with arbitrary structures for states. For a recent critique of Gandy, see [87]. For another set of (informal) physical postulates, see [32].

We should point out that, nowadays, one deals daily with more flexible notions of algorithm, such as interactive and distributed computations. To capture such non-sequential processes and non-classical algorithms, additional postulates are required. For these developments, see [9, 10, 11, 12, 13, 15, 16, 37].

We also do not address the question of the computational capabilities of the human mind, what Shagrir [87, p. 223] refers to as "The Human version of the Church–Turing Thesis" (more generally called the "AI [Artificial Intelligence] Thesis"), that (idealized) humans cannot compute any uncomputable function. Nevertheless, it does follow from our axiomatization that any state-transition mechanism that computes a non-recursive function, whether physical or biological, must violate (at least) one of the Sequential Postulates, and/or must include at least one non-recursive function in its initial states.

See [74, pp. 101–123] and [26] for discussions of these and other variants of Church's Thesis.

Finally, the question of what effectiveness means for computations over arbitrary, non-numerical domains is taken up in [64, 62, 17, 19], and elsewhere.

Acknowledgments. We thank Nikolaj Bjørner, Andreas Blass, Maria Paola Bonacina, Martin Davis, Andreas Glausch, Pierre Lescanne, Wilfried Sieg, Robert Soare, and Moshe Vardi for their valuable comments.

[***The Sliced Bread Thesis*:**]
"Turing machine = coolest idea since sliced bread"

Even a rabid fan of the Turing machine concept, who firmly believes the Sliced Bread Thesis, would not claim that the Sliced Bread Thesis is formalizable in ZFC (or whatever).
Possibly one could come up with an axiomatic definition of "effective algorithm" that is not trivially equivalent to the definition of a Turing machine, and then one could formalize Church's Thesis and ask for a proof of it.
Shoenfield worked on this for a while, I am told, but didn't get very far.

—Tim Chow, *Foundations of Mathematics (FOM) Forum* (January 12, 2004)

[34] Copeland [26] argues against the all too common misconstrual of Turing as having himself asserted such a physical claim.

REFERENCES

[1] ALLIANCE FOR TELECOMMUNICATIONS INDUSTRY SOLUTIONS, COMMITTEE T1A1 PERFORMANCE AND SIGNAL PROCESSING, *ATIS Telecom Glossary* 2000, American National Standards Institute Document ANS T1.523-2001, approved 28 February 2001. Latest version available at `http://www.atis.org/glossary/definition.aspx?id=7687` (viewed Apr. 23, 2008).

[2] HENK BARENDREGT, *The impact of the lambda calculus in logic and computer science*, this BULLETIN, vol. 3 (1997), no. 2, pp. 181–215, available at `http://www.math.ucla.edu/~asl/bsl/0302/0302-003.ps` (viewed Apr. 21, 2008).

[3] JANIS M. BARZDIN, *Ob odnom klasse mashin Tjuringa* (*mashiny Minskogo*) [*On a class of Turing machines* (*Minsky machines*)], ***Algebra i Logika*** [***Algebra and Logic***], vol. 1 (1963), no. 6, pp. 42–51, in Russian.

[4] HEINRICH BEHMANN, *Beitrage zur Algebra der Logik, insbesondere zum Entscheidungsproblem*, ***Mathematische Annalen***, vol. 86 (1922), pp. 163–229, in German.

[5] JAN A. BERGSTRA and JOHN V. TUCKER, *Algebraic specifications of computable and semi-computable datastructures*, ***Theoretical Computer Science***, vol. 50 (1987), no. 2, pp. 137–181.

[6] ROBERT BLACK, *Proving Church's thesis*, ***Philosophia Mathematica***, vol. 8 (2000), pp. 244–258.

[7] ANDREAS BLASS, NACHUM DERSHOWITZ, and YURI GUREVICH, *When are two algorithms the same*?, Technical Report MSR-TR-2008-20, Microsoft Research, Redmond, WA, February 2008, available at `http://research.microsoft.com/~gurevich/Opera/192.pdf` (viewed Apr. 30, 2008).

[8] ANDREAS BLASS and YURI GUREVICH, *Algorithms vs. machines*, ***Bulletin of the European Association for Theoretical Computer Science***, (2002), no. 77, pp. 96–118, reprinted in ***Current Trends in Theoretical Computer Science: The Challenge of the New Century***, vol. 2, ***Formal Models and Semantics*** (G. Paun, G. Rozenberg, and A. Salomaa, editors), World Scientific Publishing Company, 2004, pp. 215–236. Available at `http://research.microsoft.com/~gurevich/Opera/158.pdf` (viewed Nov. 28, 2007).

[9] ———, *Abstract state machines capture parallel algorithms*, ***ACM Transactions on Computational Logic***, vol. 4 (2003), no. 4, pp. 578–651, available at `http://research.microsoft.com/~gurevich/Opera/157-1.pdf` (viewed Nov. 28, 2007). See also: *Abstract state machines capture parallel algorithms: Correction and extension*, ***ACM Transactions on Computational Logic***, vol. 9, no. 3, in press, available at `http://tocl.acm.org/accepted/314gurevich.pdf` (viewed Nov. 28, 2007).

[10] ———, *Algorithms*: *A quest for absolute definitions*, ***Church's Thesis after 70 years*** (A. Olszewski, J. Wolenski, and R. Janusz, editors), Ontos Verlag, 2006, pp. 24–57. Also in ***Bulletin of the European Association for Theoretical Computer Science***, (2003), no. 81, pp. 195–225, and in ***Current trends in theoretical computer science***, World Scientific, 2004, pp. 283–311, available at `http://research.microsoft.com/~gurevich/Opera/164.pdf` (viewed Nov. 28, 2007).

[11] ———, *Ordinary interactive small-step algorithms, I*, ***ACM Transactions on Computational Logic***, vol. 7 (2006), no. 2, pp. 363–419, available at `http://tocl.acm.org/accepted/blass04.ps` (viewed Apr. 23, 2008).

[12] ———, *Ordinary interactive small-step algorithms, II*, ***ACM Transactions on Computational Logic***, vol. 8 (2007), no. 3, available at `http://tocl.acm.org/accepted/blass2.ps` (viewed Apr. 23, 2008).

[13] ———, *Ordinary interactive small-step algorithms, III*, ***ACM Transactions on Computational Logic***, vol. 8 (2007), no. 3, available at `http://tocl.acm.org/accepted/250blass.pdf` (viewed Apr. 23, 2008).

[14] ———, *Why sets*?, ***Pillars of computer science: Essays dedicated to Boris*** (***Boaz***) ***Trakhtenbrot on the occasion of his 85th birthday*** (A. Avron, N. Dershowitz, and A. Rabi-

novich, editors), Lecture Notes in Computer Science, vol. 4800, Springer-Verlag, Berlin, 2008, pp. 179–198, available at `http://research.microsoft.com/~gurevich/Opera/172.pdf` (viewed Mar. 8, 2008).

[15] ANDREAS BLASS, YURI GUREVICH, DEAN ROSENZWEIG, and BENJAMIN ROSSMAN, *Interactive small-step algorithms I: Axiomatization*, ***Logical Methods in Computer Science***, vol. 3 (2007), no. 4, available at `http://research.microsoft.com/~gurevich/Opera/176.pdf` (viewed Dec. 16, 2007).

[16] ———, *Interactive small-step algorithms II: Abstract state machines and the characterization theorem*, ***Logical Methods in Computer Science***, vol. 3 (2007), no. 4, available at `http://research.microsoft.com/~gurevich/Opera/182.pdf` (viewed Dec. 16, 2007).

[17] UDI BOKER and NACHUM DERSHOWITZ, *Abstract effective models*, ***New developments in computational models: Proceedings of the first international workshop on Developments in Computational Models*** (***DCM*** 2005) (***Lisbon, Portugal, July*** 2005) (M. Fernández and I. Mackie, editors), vol. 135, Electronic Notes in Theoretical Computer Science, no. 3, February 2006, pp. 15–23, available at `http://www.cs.tau.ac.il/~nachum/papers/AbstractEffectiveModels.pdf` (viewed Nov. 28, 2007).

[18] ———, *Comparing computational power*, ***Logic Journal of the IGPL***, vol. 14 (2006), no. 5, pp. 633–648, available at `http://www.cs.tau.ac.il/~nachum/papers/ComparingComputationalPower.pdf` (viewed Nov. 28, 2007).

[19] ———, *The Church–Turing Thesis over arbitrary domains*, ***Pillars of computer science: Essays dedicated to Boris*** (***Boaz***) ***Trakhtenbrot on the occasion of his 85th birthday*** (A. Avron, N. Dershowitz, and A. Rabinovich, editors), Lecture Notes in Computer Science, vol. 4800, Springer-Verlag, Berlin, 2008, pp. 199–229, available at `http://www.cs.tau.ac.il/~nachum/papers/ArbitraryDomains.pdf` (viewed Nov. 28, 2007).

[20] EGON BÖRGER, *The origins and the development of the ASM method for high level system design and analysis*, ***Journal of Universal Computer Science***, vol. 8 (2002), no. 1, pp. 2–74.

[21] SAMUEL R. BUSS, ALEXANDER A. KECHRIS, ANAND PILLAY, and RICHARD A. SHORE, *The prospects for mathematical logic in the twenty-first century*, this BULLETIN, vol. 7 (2001), no. 2, pp. 169–196, available at `http://www.math.ucla.edu/~asl/bsl/0702/0702-001.ps` (viewed Apr. 21, 2008).

[22] ALONZO CHURCH, *A note on the Entscheidungsproblem*, ***The Journal of Symbolic Logic***, vol. 1 (1936), no. 1, pp. 40–41; Corrigendum, ***The Journal of Symbolic Logic***, vol. 1 (1936), no. 3, pp. 101–102. Reprinted in [27], pp. 110–115.

[23] ———, *An unsolvable problem of elementary number theory*, ***American Journal of Mathematics***, vol. 58 (1936), pp. 345–363, reprinted in [27], pp. 89–107. Available at `doi:10.2307/2371045`.

[24] ———, *Review of Alan M. Turing, On computable numbers, with an application to the Entscheidungsproblem* (***Proceedings of the London Mathematical Society***, vol. 2 (1936), no. 42, pp. 230–265), ***The Journal of Symbolic Logic***, vol. 2 (1937), pp. 42–43.

[25] ———, *Review of Emil Post, Finite combinatory processes, Formulation I* (***The Journal of Symbolic Logic***, vol. 1 (1936), pp. 103–105), ***The Journal of Symbolic Logic***, vol. 2 (1937), p. 43.

[26] B. JACK COPELAND, *The Church–Turing Thesis*, ***Stanford encyclopedia of philosophy*** (E. Zalta, editor), 1996, available at `http://plato.stanford.edu/entries/church-turing` (viewed Nov. 28, 2007).

[27] Martin Davis (editor), ***The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions***, Raven Press, Hewlett, NY, 1965, reprinted by Dover Publications, Mineola, NY, 2004.

[28] ———, *Why Gödel didn't have Church's thesis*, ***Information and Control***, vol. 54 (1982), pp. 3–24.

[29] MARTIN D. DAVIS, RON SIGAL, and ELAINE J. WEYUKER, ***Computability, complexity, and languages: Fundamentals of theoretical computer science***, 2nd ed., Academic Press, San Diego, CA, 1994.

[30] ANUJ DAWAR, DAVID RICHERBY, and BENJAMIN ROSSMAN, *Choiceless polynomial time, counting and the Cai–Fürer–Immerman graphs*, ***Annals of Pure and Applied Logic***, vol. 152 (2008), pp. 31–50, available at `http://www.mit.edu/~brossman/CPT-CFI.pdf` (viewed Apr. 30, 2008).

[31] JANET FOLINA, *Church's thesis: Prelude to a proof*, ***Philosophia Mathematica***, vol. 6 (1998), no. 3, pp. 302–323.

[32] EDWARD F. FREDKIN and TOMMASO TOFFOLI, *Conservative logic*, ***International Journal of Theoretical Physics***, vol. 21 (1982), no. 3/4, pp. 219–253.

[33] HARVEY M. FRIEDMAN, *Mathematical logic in the* 20*th and* 21*st centuries*, ***FOM – Foundations of Mathematics mailing list***, April 27 2000, available at `http://cs.nyu.edu/pipermail/fom/2000-April/003913.html` (viewed Mar. 27, 2008).

[34] ALBRECHT FRÖHLICH and JOHN C. SHEPHERDSON, *Effective procedures in field theory*, ***Philosophical transactions of the Royal Society of London, Series A***, vol. 248 (1956), pp. 407–432.

[35] ROBIN O. GANDY, *Church's Thesis and principles for mechanisms*, ***The Kleene symposium*** (J. Barwise, D. Kaplan, H. J. Keisler, P. Suppes, and A. S. Troelstra, editors), Studies in Logic and the Foundations of Mathematics, vol. 101, North-Holland, Amsterdam, 1980, pp. 123–148.

[36] ———, *The confluence of ideas in* 1936, ***The universal Turing machine: A half-century survey*** (R. Herken, editor), Oxford University Press, New York, 1988, pp. 55–111.

[37] ANDREAS GLAUSCH and WOLFGANG REISIG, *An ASM-characterization of a class of distributed algorithms*, ***Proceedings of the Dagstuhl seminar on rigorous methods for software construction and analysis*** (J.-R. Abrial and U. Glässer, editors), May 2006, available at `http://www2.informatik.hu-berlin.de/top/download/publications/GlauschR2007_dagstuhl.pdf` (viewed Mar. 26, 2008). Longer version: *Distributed abstract state machines and their expressive power*, ***Informatik-Berichte*** 196, Humboldt-Universität zu Berlin, Jan. 2006, available at `http://www2.informatik.hu-berlin.de/top/download/publications/GlauschR2006_hub_tr196.pdf` (viewed Mar. 26, 2008).

[38] KURT GÖDEL, *On undecidable propositions of formal mathematical systems*, in [27], pp. 41–73, based on lecture notes from 1934.

[39] ———, *Gödel* *193?: [*Undecidable diophantine propositions*], *Unpublished essays and lectures*, ***Kurt Gödel: Collected works*** (S. Feferman, J. W. Dawson, Jr., W. Goldfarb, C. Parsons, and R. N. Solovay, editors), vol. III, Oxford University Press, Oxford, 1995, pp. 164–174.

[40] E. MARK GOLD, *Limiting recursion*, ***The Journal of Symbolic Logic***, vol. 30 (1965), no. 1, pp. 28–48.

[41] YURI GUREVICH, *Evolving algebras* 1993: *Lipari guide*, ***Specification and validation methods*** (E. Börger, editor), Oxford University Press, Oxford, 1995, available at `http://research.microsoft.com/~gurevich/Opera/103.pdf` (viewed Nov. 28, 2007), pp. 9–36.

[42] ———, *Sequential abstract state machines capture sequential algorithms*, ***ACM Transactions on Computational Logic***, vol. 1 (2000), no. 1, pp. 77–111, available at `http://tocl.acm.org/accepted/gurevich.ps` (viewed Apr. 23, 2008).

[43] YURI GUREVICH and JAMES K. HUGGINS, *The semantics of the C programming language*, ***Proceedings of the sixth workshop on Computer Science Logic (CSL '92)*** (Berlin), Lecture Notes in Computer Science, vol. 702, Springer-Verlag, 1993, pp. 274–308, 334–336, available at `http://research.microsoft.com/~gurevich/Opera/98.pdf` (viewed Mar. 13, 2008).

[44] DAVID HILBERT, *Mathematische Probleme*: *Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris* 1900, in German, available at `http://www.mathematik.uni-bielefeld.de/~kersten/hilbert/rede.html` (viewed Nov. 28, 2007).

[45] DAVID HILBERT and WILHELM ACKERMANN, ***Grundzüge der theoretischen Logik***, Springer-Verlag, Berlin, 1928, in German. English version of the second (1938) edition: ***Principles of theoretical logic*** (R. E. Luce, translator and editor), AMS Chelsea Publishing, New York, 1950.

[46] LÁSZLÓ KALMÁR, *An argument against the plausibility of Church's thesis*, ***Constructivity in mathematics*** (A. Heyting, editor), North-Holland, Amsterdam, 1959, pp. 201–205.

[47] SHEN KANGSHEN, JOHN N. CROSSLEY, and ANTHONY W. C. LUN, ***The nine chapters on the mathematical art: Companion and commentary***, Oxford University Press, Oxford, 1999.

[48] STEPHEN C. KLEENE, *Recursive predicates and quantifiers*, ***Transactions of the American Mathematical Society***, vol. 53 (1943), no. 1, pp. 41–73, reprinted in [27], pp. 255–287.

[49] ———, ***Introduction to metamathematics***, D. Van Nostrand, New York, 1952.

[50] ———, ***Mathematical logic***, John Wiley, New York, 1967, reprinted by Dover, Mineola, NY, 2002.

[51] ———, *Origins of recursive function theory*, ***IEEE Annals of the History of Computing***, vol. 3 (1981), no. 1, pp. 52–67, preliminary version in ***Proceedings of the 20th annual IEEE symposium on Foundations of Computer Science*** (***FOCS***), San Juan, PR, Oct. 1979, pp. 371–382.

[52] ———, *Reflections on Church's thesis*, ***Notre Dame Journal of Formal Logic***, vol. 28 (1987), no. 4, pp. 490–498.

[53] ———, *Turing's analysis of computability, and major applications of it*, ***The universal Turing machine: A half-century survey*** (R. Herken, editor), Oxford University Press, New York, 1988, pp. 17–54.

[54] DONALD E. KNUTH, *Algorithm and program*; *information and data*, ***Communications of the ACM***, vol. 9 (1966), no. 9, p. 654.

[55] ———, *Fundamental algorithms*, ***The art of computer programming***, vol. 1, Addison-Wesley, Reading, MA, 3rd ed., 1997.

[56] ANDREĬ N. KOLMOGOROV, *O ponyatii algoritma* [*On the concept of algorithm*], ***Uspekhi Matematicheskikh Nauk*** [***Russian Mathematical Surveys***], vol. 8 (1953), no. 4, pp. 175–176, in Russian. English version in [111, pp. 18–19].

[57] ANDREĬ N. KOLMOGOROV and VLADIMIR A. USPENSKY, *K opredeleniu algoritma*, ***Uspekhi Matematicheskikh Nauk*** [***Russian Mathematical Surveys***], vol. 13 (1958), no. 4, pp. 3–28, in Russian. English version: *On the definition of an algorithm*, American Mathematical Society Translations, Series II, vol. 29, 1963, pp. 217–245.

[58] GEORG KREISEL, *Mathematical logic*, ***Lectures in modern mathematics III*** (T. L. Saaty, editor), Wiley and Sons, New York, 1965, pp. 95–195.

[59] ———, *Mathematical logic*: *What has it done for the philosophy of mathematics*, ***Bertrand Russell: Philosopher of the century*** (R. Schoenman, editor), Allen & Unwin, London, 1967, pp. 201–272.

[60] SAUL KRIPKE, *Elementary recursion theory and its applications to formal systems*, unpublished preliminary draft, 1996.

[61] ———, *From the Church–Turing thesis to the first-order algorithm theorem*, ***Proceedings of the 15th annual IEEE symposium on Logic in Computer Science, Santa Barbara, CA***, June 2000, p. 177. Recorded lecture at `http://www.vanleer.org.il/eng/videoShow.asp?id=317` (viewed Mar. 28, 2008).

[62] WILLIAM M. LAMBERT, JR., *A notion of effectiveness in arbitrary structures*, ***The Journal of Symbolic Logic***, vol. 33 (1968), no. 4, pp. 577–602.

[63] JOHN R. LUCAS, *Review of Judson C. Webb*, ***Mechanism, mentalism and metamathematics: An essay on finitism*** (*D. Reidel, Dordrecht*, 1980), ***The British Journal for the Philosophy of Science***, vol. 33 (1982), no. 4, pp. 441–444.

[64] ANATOLIĬ I. MAL'CEV, *Konstruktivnyye algyebry.* 1, ***Uspekhi Matematicheskikh Nauk***, vol. 16 (1961), no. 3, pp. 3–60. English version: *Constructive algebras, I, The metamathematics of algebraic systems* (K. A. Hirsch, translator), ***Russian Mathematical Surveys***, vol. 16 (1961), no. 3, pp. 77–129. Also in: ***The metamathematics of algebraic systems. Collected papers 1936–1967***, B. F. Wells, III, editor, North-Holland, Amsterdam, 1971, pp. 148–212.

[65] ZOHAR MANNA and ADI SHAMIR, *The optimal approach to recursive programs*, ***Communications of the ACM***, vol. 20 (1977), no. 11, pp. 824–831.

[66] ANDREĬ A. MARKOV, *Teorii͡a algorifmov*, ***Works of the Mathematics Institute Im. V. A. Steklov***, vol. 42, 1954, in Russian. English version: ***Theory of algorithms*** (J. J. Schorr-Kon, translator), American Mathematical Society Translations, Series 2, vol. 15, pp. 1–14, and Israel Program for Scientific Translations, Jerusalem, 1961.

[67] YURI V. MATIJASEVIČ, *Enumerable sets are Diophantine*, ***Doklady Akademiia Nauk SSSR***, vol. 191 (1970), no. 2, pp. 279–282, in Russian. English version: ***Soviet Mathematics***, vol. 11 (1970), no. 2, pp. 354–357.

[68] LUIGI FEDERICO MENABREA, *Notions sur la machine analytique de M Charles Babbage*, ***Bibliothèque Universelle de Genève n.s.***, vol. 41 (1842), pp. 352–376, in French. Also in: CHARLES BABBAGE, ***The works of Charles Babbage*** (M. Campbell-Kelly, editor), vol. 3, ***The difference engine and table making***, NYU Press, New York, 1989, pp. 62–82. English version: *Sketch of the Analytical Engine invented by Charles Babbage* (*with notes upon the memoir by the translator A. A. L.* [*Ada Augusta, Countess of Lovelace*]), ***Scientific Memoirs*** (R. Taylor, editor), vol. 3, 1843, pp. 666–731, available at `http://www.fourmilab.ch/babbage/sketch.html` (viewed Nov. 28, 2007).

[69] ELLIOTT MENDELSON, *Second thoughts about Church's thesis and mathematical proofs*, ***The Journal of Philosophy***, vol. 87 (1990), no. 5, pp. 225–233.

[70] ———, *On the impossibility of proving the 'hard-half' of Church's thesis*, ***Church's thesis after 70 years*** (A. Olszewski, J. Wolenski, and R. Janusz, editors), Ontos Verlag, Frankfurt, 2006, pp. 304–309.

[71] MARVIN MINSKY, ***Computation: Finite and infinite machines***, 1st ed., Prentice-Hall, Englewood Cliffs, NJ, 1967.

[72] RICHARD MONTAGUE, *Towards a general theory of computability*, ***Synthese***, vol. 12 (1960), no. 4, pp. 429–438.

[73] YIANNIS N. MOSCHOVAKIS, *What is an algorithm*?, ***Mathematics unlimited—2001 and beyond*** (B. Engquist and W. Schmid, editors), Springer-Verlag, Berlin, 2001, pp. 919–936, available at `http://www.math.ucla.edu/~ynm/papers/eng.ps` (viewed Nov. 28, 2007).

[74] PIERGIORGIO ODIFREDDI, ***Classical recursion theory: The theory of functions and sets of natural numbers***, Studies in Logic and the Foundations of Mathematics, vol. 125, North-Holland, Amsterdam, 1992.

[75] EMIL L. POST, *Finite combinatory processes, Formulation I*, ***The Journal of Symbolic Logic***, vol. 1 (1936), pp. 103–105, reprinted in [27], pp. 289–303.

[76] ———, *Absolutely unsolvable problems and relatively undecidable propositions*: *Account of an anticipation*, unpublished paper, 1941, in [27], pp. 340–433. Also in ***Solvability, provability, definability: The collected works of Emil L. Post*** (M. Davis, editor), Birkhäuser, Boston, MA, 1994, pp. 375–441.

[77] ———, *Formal reductions of the general combinatorial decision problem*, ***American Journal of Mathematics***, vol. 65 (1943), no. 2, pp. 197–215, available at `doi:10.2307/2371809`.

[78] HILARY PUTNAM, *Trial and error predicates and the solution to a problem of Mostowski*, ***The Journal of Symbolic Logic***, vol. 30 (1965), no. 1, pp. 49–57.

[79] MICHAEL O. RABIN, *Computable algebra, general theory and the theory of computable fields*, ***Transactions of the American Mathematical Society***, vol. 95 (1960), no. 2, pp. 341–360.

[80] WOLFGANG REISIG, *On Gurevich's theorem on sequential algorithms*, ***Acta Informatica***, vol. 39 (2003), no. 5, pp. 273–305, available at `http://www.informatik.hu-berlin.de/top/download/publications/Reisig2003_ai395.pdf` (viewed Nov. 28, 2007).

[81] MICHAEL RESCORLA, *Church's Thesis and the conceptual analysis of computability*, ***Notre Dame Journal of Formal Logic***, vol. 48 (2007), no. 2, pp. 253–280.

[82] HARTLEY ROGERS, JR., ***Theory of recursive functions and effective computability***, McGraw-Hill, New York, 1967, reprinted by MIT Press, Cambridge, MA, 1987.

[83] J. BARKLEY ROSSER, *An informal exposition of proofs of Gödel's Theorem and Church's Theorem*, ***The Journal of Symbolic Logic***, vol. 4 (1939), pp. 53–60, reprinted in [27], pp. 223–230.

[84] ———, *Highlights of the history of lambda calculus*, ***IEEE Annals of the History of Computing***, vol. 6 (1984), no. 4, pp. 337–349.

[85] ARNOLD SCHÖNHAGE, *Storage modification machines*, ***SIAM Journal on Computing***, vol. 9 (1980), no. 3, pp. 490–508.

[86] RICH SCHROEPPEL, *A two counter machine cannot calculate* 2^N, Technical Report AIM-257, A. I. Laboratory, Massachusetts Institute of Technology, Cambridge, MA, May 1972, available at `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-257.pdf` (viewed Nov. 28, 2007).

[87] ORON SHAGRIR, *Effective computation by humans and machines*, ***Minds and Machines***, vol. 12 (2002), no. 2, pp. 221–240, available at `http://edelstein.huji.ac.il/staff/shagrir/papers/Effective_computation_by_human_and_machine.pdf` (viewed Nov. 28, 2007).

[88] STEWART SHAPIRO, *Understanding Church's thesis*, ***Journal of Philosophical Logic***, vol. 10 (1981), pp. 353–365.

[89] ———, *Acceptable notation*, ***Notre Dame Journal of Formal Logic***, vol. 23 (1982), pp. 14–20.

[90] ———, *Understanding Church's thesis, again*, ***Acta Analytica***, vol. 11 (1995), pp. 59–77.

[91] JOSEPH R. SHOENFIELD, ***Mathematical logic***, Addison-Wesley, Reading, MA, 1967.

[92] ———, ***Recursion theory***, Lecture Notes in Logic, vol. 1, Springer-Verlag, Berlin, 1993, reprinted by A. K. Peters, Natick, MA, 2001.

[93] WILFRIED SIEG, *Mechanical procedures and mathematical experiences*, ***Mathematics and mind*** (A. George, editor), Oxford University Press, Oxford, 1994, pp. 71–117.

[94] ———, *Step by recursive step*: *Church's analysis of effective computability*, this BULLETIN, vol. 3 (1997), no. 2, pp. 154–180, available at `http://www.math.ucla.edu/~asl/bsl/0302/0302-002.ps` (viewed Nov. 28, 2007).

[95] ———, *Hilbert's programs*: 1917–1922, this BULLETIN, vol. 5 (1999), no. 1, pp. 1–44, available at `http://www.math.ucla.edu/~asl/bsl/0501/0501-001.ps` (viewed Apr. 21, 2008).

[96] ———, *Calculations by man and machine*: *Conceptual analysis*, ***Reflections on the foundations of mathematics: Essays in honor of Solomon Feferman*** (W. Sieg, R. Sommer, and C. Talcott, editors), Lecture Notes in Logic, vol. 15, A. K. Peters, Natick, MA, 2002, pp. 390–409.

[97] ———, *Calculations by man and machine*: *Mathematical presentation*, ***In the scope of logic, methodology and philosophy of science, vol. 1, Proceedings of the eleventh international congress of Logic, Methodology and Philosophy of Science, Cracow, Poland,*** August 1999 (P. Gärdenfors, J. Woleński, and K. Kijania-Placek, editors), Synthese Library, vol. 315, Kluwer Academic, Dordrecht, 2003, pp. 247–262.

[98] ———, *Gödel on computability*, ***Philosophia Mathematica, Series III***, vol. 14 (2007), no. 2, pp. 189–207.

[99] ———, *Church without dogma—Axioms for computability*, ***New computational paradigms: Changing conceptions of what is computable*** (S. B. Cooper, B. Löwe, and A. Sorbi, editors), Springer-Verlag, Berlin, 2008, pp. 139–152. Available at `http://www.hss.cmu.edu/philosophy/sieg/Church%20without%20dogma.pdf` (viewed Nov. 28, 2007).

[100] ———, *Computability: Emergence and analysis of a mathematical notion*, ***Handbook of the philosophy of mathematics*** (A. Irvine, editor), to appear.

[101] Wilfried Sieg and John Byrnes, *K-graph machines: Generalizing Turing's machines and arguments*, ***Gödel 96: Logical Foundations of Mathematics, Computer Science, and Physics*** (P. Hájek, editor), Lecture Notes in Logic, vol. 6, Springer-Verlag, Berlin, 1996, pp. 98–119.

[102] ———, *An abstract model for parallel computations: Gandy's thesis*, ***The Monist***, vol. 82 (1999), no. 1, pp. 150–164.

[103] Robert I. Soare, *Computability and recursion*, this Bulletin, vol. 2 (1996), no. 3, pp. 284–321, available at `http://www.math.ucla.edu/~asl/bsl/0203/0203-002.ps` (viewed Nov. 28, 2007).

[104] ———, *Computability and incomputability*, ***Proceedings of the third conference on Computability in Europe (CiE 2007) Siena, Italy*** (S. B. Cooper, B. Löwe, and A. Sorbi, editors), Lecture Notes in Computer Science, vol. 4497, Springer-Verlag, Berlin, June 2007, pp. 705–715, draft available at `http://www.people.cs.uchicago.edu/~soare/siena502m.pdf` (viewed Nov. 28, 2007). Longer version to appear.

[105] Robert Stärk, Joachim Schmid, and Egon Börger, ***Java and the Java Virtual Machine: Definition, verification, validation***, Springer-Verlag, Berlin, 2001.

[106] Alfred Tarski, *Der Wahrheitsbegriff in den formalisierten Sprachen*, ***Studia Philosophica***, vol. 1 (1936), pp. 261–405, in German. English version: *The concept of truth in formalized languages* (J. H. Woodger, translator), ***Logic, semantics, methamathematics*** (J. Corcoran, editor), Hackett, Indianapolis, IN, 1983, pp. 152–277.

[107] Alan M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, ***Proceedings of the London Mathematical Society, Series 2***, vol. 42, parts 3 and 4 (1936), pp. 230–265, Corrigenda in vol. 43 (1937), pp. 544–546. Reprinted in [27], pp. 116–154; also in ***The essential Turing: Seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life, plus the secrets of Enigma*** (B. Jack Copeland, editor), Oxford University Press, 2004, pp. 58–90; and in ***Collected works of A. M. Turing***, vol. 4, ***Mathematical logic*** (R. O. Gandy and C. E. M. Yates, editors), North-Holland, Amsterdam, 2001, pp. 18–56. Available at `http://www.abelard.org/turpap2/tp2-ie.asp` (viewed Nov. 27, 2007).

[108] ———, *Computability and λ-definability*, ***The Journal of Symbolic Logic***, vol. 2 (1937), pp. 153–163, reprinted in ***Collected works of A. M. Turing***, vol. 4, ***Mathematical logic*** (R. O. Gandy and C. E. M. Yates, editors), North-Holland, Amsterdam, 2001, pp. 59–69.

[109] ———, *Computing machinery and intelligence*, ***Mind***, vol. 59 (1950), pp. 433–460, reprinted in ***Collected works of A. M. Turing***, vol. 1, ***Mechanical intelligence*** (D. C. Ince, editor), North-Holland, Amsterdam, 1992, pp. 133–160; and in ***The essential Turing*** (B. Jack Copeland, editor), Oxford University Press, 2004, pp. 433–464. Available at `http://web.comlab.ox.ac.uk/oucl/research/areas/ieg/e-library/sources/t_article.pdf` (viewed Nov. 28, 2007).

[110] ———, *Solvable and unsolvable problems*, ***Science News***, vol. 31 (1954), pp. 7–23, reprinted in ***Collected works of A. M. Turing***, vol. 1: ***Mechanical intelligence*** (D. C. Ince, editor), North-Holland, Amsterdam, 1992, pp. 187–204.

[111] Vladimir A. Uspensky and Alexei L. Semenov, ***Algorithms: Main ideas and applications***, Kluwer, Norwell, MA, 1993.

[112] HAO WANG, ***From mathematics to philosophy***, Routledge and Kegan Paul, London, 1974.

[113] RICHARD ZACH, *Completeness before Post*: *Bernays, Hilbert, and the development of propositional logic*, this BULLETIN, vol. 5 (1999), no. 3, pp. 331–366, available at `http://www.math.ucla.edu/~asl/bsl/0503/0503-003.ps` (viewed Nov. 28, 2007).

MICROSOFT RESEARCH
REDMOND, WA 98052, USA
and
SCHOOL OF COMPUTER SCIENCE
TEL AVIV UNIVERSITY
RAMAT AVIV 69978, ISRAEL
E-mail: nachum.dershowitz@cs.tau.ac.il
URL: www.cs.tau.ac.il/~nachum

MICROSOFT RESEARCH
REDMOND, WA 98052, USA
E-mail: gurevich@microsoft.com
URL: research.microsoft.com/~gurevich