Effectiveness*

Nachum Dershowitz and Evgenia Falkovich
School of Computer Science, Tel Aviv University, Tel Aviv, Israel

May 4, 2012

Abstract

We describe axiomatizations of several aspects of effectiveness: effectiveness of transitions; effectiveness relative to oracles; and absolute effectiveness, as posited by the Church-Turing Thesis.

Efficiency is doing things right;
effectiveness is doing the right things.
—Peter F. Drucker

1 Introduction

In 1900, David Hilbert posed, among other problems, the research challenge of how to effectively determine whether any given polynomial with rational coefficients has rational roots [25]:[1]

> [**Probleme**] **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

*This work was carried out in partial fulfillment of the requirements for the Ph.D. degree of the second author.

[1][**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.

welchem sich mittelst einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist."

In the same lecture, as his famous second problem, Hilbert asked for a proof of the consistency of (Peano) arithmetic.

Later, he and Wilhelm Ackermann underscored the importance of the decision problem for validity of formulaæ in (first-order predicate) logic, which they called the *Entscheidungsproblem* [26, pp. 73–74]:[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.

Hilbert was seeking an effective procedure that could solve every instance of the validity question, positively or negatively: "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" (quoted in [52]).

In 1936, Alonzo Church suggested that the recursive functions, or the computationally equivalent lambda-definable numeric functions, capture the intended concept of "effectively calculable" procedure [9, p. 356]. With his formalization of absolute effectivity in hand, he proceeded to demonstrate that no effective solution exists for the Entscheidungsproblem. When Church subsequently learned of Alan Turing's independent proof of undecidability [57], he conceded that Turing's machines have "the advantage of making the identification with effectiveness in the ordinary (not explicitly defined)

[2]The Entscheidungsproblem 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 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.

sense evident immediately" [10, p. 43]. Similarly, Kurt Gödel [19, pp. 369–370] realized that Turing's model of effective computation, which provides "a precise and unquestionably adequate definition of the general concept of formal system", strengthens his earthshaking incompleteness results and establishes that "the existence of undecidable arithmetical propositions and the non-demonstrability of the consistency of a system in the same system can now be proved rigorously for every consistent formal system containing a certain amount of finitary number theory". In short, Hilbert's dream of devising a complete and consistent finite axiomatization of mathematics, as expressed in his second problem, is inherently unattainable.

Stephen Kleene reformulated Church's contention that the recursive functions and the effective numeric functions are one and the same as a "thesis" ([30, p. 60], [31, p. 332], [32, p. 232]):

> **Thesis I.** Every effectively calculable function (effectively decidable predicate) is general recursive.
>
> **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 potentially partial recursive.
>
> Turing's and Church's theses are equivalent. We shall usually refer to them both as *Church's thesis*, or in connection with that one of its... versions which deals with "Turing machines" as *the Church-Turing thesis*.

Church's thesis asserted that the recursive functions are the only numeric functions that can be effectively computed. Turing's thesis staked the analogous claim that any function on strings that can be mechanically computed can be computed, in particular, by a Turing machine. Turing showed [57, Appendix] that with a suitable interpretation of strings as numbers, his machines compute exactly the recursive functions.

Three main lines of argument have been adduced in support of this thesis ([31, p. 320], [48, pp. 18–19], [31, p. 321]):

- Despite concerted efforts, no more powerful effective computational model has been devised.
- "By means of detailed combinatorial studies, 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," as have all subsequent effective models.

- 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 Turing machines.

Gödel is reported [12] 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 [49, 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).

This challenge of proving the Church-Turing Thesis is number one in Richard Shore's list of "pie-in-the-sky problems" for the twenty-first century [8]. Indeed, Harvey Friedman [16] has 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."

We discuss such an axiomatization of effectiveness in Sections 2–5. Unlike Turing's analysis [57], and subsequent generalizations [37, 38, 18, 50, 53, 51, 54], our axioms of effective computation are, at the same time, both formal and generic. They are formal, in that they may be cast as precise mathematical statements [4, 14]; they are generic, in that they apply to computations with arbitrary states (Section 3) and arbitrary programmable transitions (Section 4).

Computability is a more general notion than recursiveness or Turing computability. Just as Turing machines provide a computational model for strings and recursive functions for the natural numbers, there are comparable notions of effectiveness for other data types, as explained in Sections 3.2 and 5.3.

Beyond that, Turing extended the notion of computability to devices provided with oracles that "magically" provide answers to questions for which there may be no effective means of providing answers. See Sections 3.3 and 5.3.

We draw some conclusions in the final section.

2 Discrete Algorithms

By an algorithm, one invariably[3] means some type of state-transition system. As Donald Knuth writes [35], for example:

> 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.)

Classical algorithms proceed step by step, from state to next state. We formalize this as the first postulate.

Postulate (State Transition) *An* algorithm *determines a set (or class) of* states*, a subset (or subclass) of* initial *states, and a partial* next-state *transition function from states to states.* Terminal *states are those states for which no transition is defined.*

Having the transition depend only on the state means that states must store all the information needed to determine subsequent behavior. Prior history is unavailable to the algorithm unless stored in the current state.

State-transitions are deterministic. Classical algorithms in fact never leave room for choices, nor do they involve any sort of interaction with the

[3]An exception is [41], which argues that algorithms should be equated with recursive definitions.

environment to determine the next step. To incorporate nondeterministic choice, probabilistic choice, or interaction with the environment, one would need to modify the above notion of transition.

This postulate is meant to exclude formalisms, such as [20, 44], in which the result of a computation—or the continuation of a computation—may depend on (the limit of) an infinite sequence of preceding (finite or infinitesimal) steps. Likewise, processes in which states evolve continuously (as in analog processes, like the position of a bouncing ball), rather than discretely, are eschewed.

Yuri Gurevich's "sequential postulates" [23] characterize algorithmicity in its classical sense. They assert that states are first-order structures and that transitions respect isomorphisms (see Section 3). An algorithm provides a prescription for updating states, that is, for changing some of the interpretations given to symbols by states. The essential idea is that there is a fixed finite set of terms that refer (possibly indirectly) to locations within a state and which suffice to determine what needs to be tested and how the state needs to change during any transition (see Section 4). This implies, as we will see, that it is possible to describe transitions by means of some finite text (see Section 5.1). These characteristics apply both to effective methods, such as factoring, and ideal ones, like inverting a matrix of arbitrary reals.

For an algorithm to be effective, there is one additional, crucial issue: it must be possible to describe (initial) states in some finite fashion.

3 States

States must be comprehensive: they need to incorporate *all* the relevant data that, when coupled with the program, completely determine the next state, and, hence, the whole future of the computation. For instance, the "instantaneous description" of a Turing-machine computation is just what is needed to pick up a machine's computation from where it has been left off; see [57]. Likewise, the state of a procedural language contains the values of all variables plus the "program counter", pointing to the current operation. Similarly, the "continuation" of a Lisp program contains all the state information needed to resume its computation.

3.1 Abstract States

In addition to storing mutable data, states of algorithms should incorporate the means to make changes. So, they (and, by the same token, states of non-algorithmic processes) may best be regarded as (first-order) logical structures with (finitely many) partial functions, relations, and constants. To simplify matters, relations can be treated as truth-valued functions and constants as nullary (scalar) functions. So, each state consists of a domain and interpretations for its symbols as partial functions over the domain. Structures, or (partial) algebras, as we sometimes refer to them, suffice to model all salient features of states. All relevant information about a state is given explicitly in the state by means of its interpretation of the symbols appearing in the vocabulary of the structure. Compare [43, pp. 420–429].

The values of programming variables, in and of themselves, are meaningless to an algorithm, which is implementation independent. The specific details of the implementation of the data types used by the algorithm should not matter. Rather, it is relationships between values that matter to the algorithm. It follows that an algorithm should work equally well in isomorphic worlds. Compare [18, p. 128]. In this sense states are "abstract". These considerations lead to the second postulate [23, 1]:

Postulate (State) *States of an algorithm are (first-order) structures over a finite vocabulary, closed under isomorphism of domains, such that initial states and terminal states are also closed under isomorphism. Furthermore, transitions preserve the domain of states, do not change any defined point of a function into undefined, respect isomorphisms, in the sense that non-terminal isomorphic states transition to corresponding isomorphic states.*

Qua structure, a state interprets each of the function symbols in its vocabulary. Each state interprets its function symbols as partial operations and, for every term over its vocabulary, either assigns it a domain value, or leaves it undefined if any of the operations involved is undefined for its arguments. States usually include equality, or at least a partially defined equality. (We presume that state structures are endowed with Boolean truth values and standard Boolean operations, and vocabularies include symbols for these.)

Vocabularies are finite, since an algorithm must be describable in finite terms, so can only refer explicitly to finitely many operations. Hence, an algorithm cannot, for instance, involve all of Knuth's [36] arrow operations, $\uparrow$, $\uparrow\uparrow$, $\uparrow\uparrow\uparrow$, etc. Instead one could employ a ternary operation $\lambda xyz.\ x \uparrow^z y$.

In restricting structures to be "first-order", we are limiting the *syntax* to be first-order. This precludes states with infinitary operations, like the supremum of infinitely many objects, which would not make sense from an algorithmic point of view. This does not, however, limit the semantics of algorithms to first-order notions. The domain of states may have sequences, or sets, or other higher-order objects, in which case, the state would also need to provide operations for dealing with those objects.

Closure under isomorphism ensures that the algorithm can operate on the chosen level of abstraction. The states' internal representation of data is invisible and immaterial to the program. This means that the behavior of an *algorithm*, in contradistinction with its "implementation" as, say, a C program—cannot, for example, depend on the memory address of some variable. If an algorithm does depend on such matters, then its full description must also include specifics of memory allocation.

It is possible to liberalize this postulate somewhat to allow the domain to grow or shrink, or for the vocabulary to be infinite or extensible, but such enhancements do not materially change the notion of algorithm.

A fable To illustrate the importance of operating on the correct level of abstraction, consider the following story:[4] A student in an algebraic topology course did not hand in the assigned homework. "You know", said the student to the lecturer, "I was working hard on the homework. After a while, I felt hungry and decided to take a break for a cup of coffee and a donut. But then I spent the rest of the night trying to understand which one of them I should eat and which one I should drink." What actually happened to this poor fellow? There were two states to consider, one with a cup of coffee and no donut, the other with only a donut left. Both real-life situations comprise many properties, like color, temperature, material, recipe used to bake, shape, coordinates on the table, etc. But from the algebraic-topology point of view, one cares only for the genus of the surface and thus cannot distinguish between those two states. On the other hand, a proper algorithm for dealing with midnight hunger should be able to distinguish between food and drink. In other words, the "algebraic-topology algorithm" and the "midnight-

[4]Based on the famous quip of John Kelley, "A *topologist* is a man who doesn't know the difference between a doughnut and a coffee cup" [27, p. 88n.].

hunger algorithm" have different salient properties, a crucial factor, which the hapless student failed to account for.

3.2 Effective States

Already in 1922, Emil Post [43, pp. 427–428] noted the following about states of effective computations:

> 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]....

In other words, not only should states be symbolic and be represented by relational structures, but they need to be finitely representable if they are to be effective. Accordingly, we insist that effective states harbor no information beyond the means to reach domain values, plus anything that can be derived therefrom.

In general, then, the operations in states come in three flavors: domain constructors; defined functions; and black-box oracles. For a state to be effective, it should provide means to access all the elements of its domain and should not have any oracles.

Function symbols C *construct* a particular domain in a given state if the state assigns each value in the domain to exactly one term over C (so the terms over C form a free Herbrand algebra). Constructors are the usual way of thinking of the domain values of computational models. For example, strings over an alphabet {a,b,...} are constructed from a nullary constructor $\varepsilon()$ and unary constructors a($\cdot$), b($\cdot$), etc. The positive integers in binary notation are constructed out of the nullary ε and unary 0 and 1, with the constructed string understood as the binary number obtained by prepending the digit 1. A domain consisting of integers and Booleans can be constructed from TRUE, FALSE, 0, and a "successor" function that takes non-negative integers (n) to the predecessor of their negation ($-n-1$) and negative integers ($-n$) to their absolute value (n). To construct 0-1-2 trees, we would have three constructors, $k_0()$, $k_1(\cdot)$, and $k_2(\cdot,\cdot)$, for nodes of outdegree 0 (leaves), 1 (unary), and 2 (binary), respectively.

Definition 1 (Effective State)

- A state is *basic* if it includes constructors for its domain, plus totally undefined operations, meaning that they all always yield the same default value (UNDEF, say), and no oracles.
- Such states are *(absolutely) effective*.
- Moreover, a state is effective also if all its defined operations can be effectively computed (in a bootstrapped sense to be made precise below) from basic states and with the same constructors.

This effectiveness postulate excludes algorithms with ineffective oracles, such as the halting function, but allows one to be given effective operations, like equality of trees or division of integers. Having only free constructors at the foundation prevents the hiding of potentially uncomputable information by means of equalities between distinct representations of the same domain element. This is the approach to effectiveness advocated in [4], extended to include partial functions in states, as in [1].

3.3 Oracular States

Turing [58] introduced the powerful idea of computability relative to oracles, saying , "We shall not go any further into the nature of this oracle apart from saying that it cannot be a machine." We may think of a Turing machine that is equipped with a special tape for querying oracles and special states q_M and q_o for each oracle o in $\mathcal{O}$. When, during an execution, the machine enters state q_o, the oracle magically answers by replacing the string x on the query tape with the value $o(x)$ and reverts to state q_M.

In the presence of oracles, we still want the domain to be constructive, or else there may be no finite way of representing inputs and outputs, but now we allow basic operations that may not be effective. Accordingly, we speak, instead, of "relative" effectiveness.

Definition 2 (Relatively Effective State)

- A state is *basic* in oracles $\mathcal{O}$, if it includes constructors for its domain, totally undefined operations, plus oracles $\mathcal{O}$.
- Such states are *relatively effective.*
- Moreover, a state is relatively effective also if all its defined operations can be computed from basic states with the same constructors and oracles.

One can give an alternate characterization of effective state, one that is based on oracular Turing machines, extending a suggestion of Wolfgang Reisig [47].

Lemma *A state X is effective relative to a set of oracles if and only if there is a Turing machine with the same oracles that can semi-decide the congruence induced by X. In other words, given two terms over the vocabulary of X as input, the machine returns* TRUE *whenever both terms are defined and assigned the same values by X,* FALSE *when both are defined but not equal, and diverges otherwise. Input and output for the machines's oracles is via constructor terms.*

The proof is along the lines of the non-oracular one in [5].

4 Transitions

For a process, effective or not, to be deemed algorithmic, it must be possible to express the transition rules for going from state to state in some *finite* fashion. Kleene stressed this point repeatedly ([34, p. 17], [32, pp. 240–241n.], [33, 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.
>
> 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.

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.

So, algorithms need to be expressible by means of finite texts, making reference to only finitely many terms and relations among them. Indeed, an algorithm can only determine relations between values stored in an abstract state via terms in its vocabulary and equalities (and disequalities) between their values.

4.1 Effective Transitions

The actions taken by a transition are describable in terms of *updates* in which a *new* interpretation is given to function symbols by the next state. The set of updates encapsulates the state-transition relation of an algorithm by providing all the information necessary to change the current state into the next one. To determine the updates for any given state, the algorithm needs to evaluate some terms. The third postulate ([23], refined as in [1]) ensures that there is a finite description of the update process, and that its execution requires only a bounded amount of work. Simply stated, there is a fixed, finite set of (ground) terms that determines the stepwise behavior of an algorithm.

Postulate (Transition) *For every state of every algorithm, there is a set of* critical terms *over its vocabulary, of size up to some bound fixed by the algorithm, such that any set of states that all assign the same values to their shared critical terms all have the same critical terms and the same updates (if any).*

The intuition is that an algorithm must base its actions on the values contained at locations in the current state. Unless all states undergo the same updates unconditionally, an algorithm must explore one or more values at some accessible locations in the current state before determining how to proceed. The only means that an algorithm has with which to reference locations is via terms, since the values themselves are abstract entities. If every referenced location has the same value in more than one state, then the behavior of the algorithm must be the same for those states.

This postulate precludes programs of infinite size (like an infinite table lookup) or which are input-dependent.

On account of the presence of partial operations, we need to take into account which locations in the state are actually accessed by the given algorithm. Should an undefined location be accessed, the computation would go into limbo. That is why critical terms are individual to states. Partial operations are required for full generality of the formalization of effectiveness.

4.2 Classical Algorithms

Careful analysis of the notion of algorithm in [23] and an examination of the intent of the founders of the field of computability in [14] demonstrate that the postulates are in fact true of all ordinary, sequential algorithms, the (only) kind envisioned by the pioneers of the field. In this sense, the traditional notion of algorithm is precisely captured by these axioms. Accordingly, we refer to a process satisfying the above three postulates as a *classical algorithm*.

5 Effectiveness

Having axiomatized algorithmic processes, we turn to the question of how to describe them by finite means.

5.1 Algorithms

Gurevich [23] showed that his *abstract state machines* [22], constitute a most general model of computation, one that can precisely describe effective transitions of any classical algorithm, on any desired level of abstraction of data structures and native operations. Programs in this formalism may be built from just three components: (1) There are generalized assignments $f(s_1, \ldots, s_n) := t$, where f is any function symbol and the s_i and t are arbitrary (ground) terms. (2) Statements may be prefaced by a conditional test. (3) Program statements are composed in parallel. The semantics of assignment statements, parallel composition, and conditionals are as expected.

A program describes a single transition step; its statements are executed repeatedly, as a unit, until no assignments have their conditions enabled.

This very simple model of computation suffices to precisely capture the behavior of the whole class of classical algorithms over any domain, including those with partial operations, be they effective or oracular, that hang outside their domain of definition [1]. This model is not wedded to any particular data representation—in the way, say, that Turing machines manipulate strings using a small set of tape operations. In this sense, abstract state machines are the most generic of computational models.

A simple program in this framework is the following:

$$\textbf{if } |b-a| > \varepsilon \textbf{ then do } \begin{cases} \textbf{if } \operatorname{sgn} f((a+b)/2) = \operatorname{sgn} f(a) \textbf{ then } a := (a+b)/2 \\ \textbf{if } \operatorname{sgn} f((a+b)/2) = \operatorname{sgn} f(b) \textbf{ then } b := (a+b)/2 \end{cases}$$

The domain is the reals plus the Booleans; the operations for addition $(+)$, subtraction $(-)$, halving $(/2)$, equality $(=)$, greater than $(>)$, absolute value $(|\cdot|)$, and signum (sgn) are fixed in all states; the values of f and ε are given in the initial state as inputs; the values of the scalars a and b are also given as inputs, but they are changed by the transitions from state to state. The conditional is repeated over and over until the outer condition turns false, and no more assignments are made. Should the signs of a and b start out the same, then both inner conditions will hold, and both assignments will be performed forever. Also if ε is nonpositive, the program will never terminate.

The critical term is $|b-a| > \varepsilon$ in states that falsify this condition, and includes also $f((a+b)/2) = \operatorname{sgn} f(a)$ and $f((a+b)/2) = \operatorname{sgn} f(b)$ when the condition is true.

This program describes the standard bisection search for the root of a function, as described in [21, Algorithm #4]. The point is that this abstract formulation is, as the author of [21] wrote, "applicable to any continuous function" f over the reals—including ones that are not programmable. This program cannot be considered effective; indeed its domain is uncountable. See [46] for examples of geometric constructions with compass and straightedge.

5.2 Effective Algorithms

The sequential postulates limit transitions to be effective, in the sense of being amenable to finite description, but they place no constraints on the nature of the contents of states. In particular, states may contain ineffective oracles. To preclude that and ensure that an algorithm is effective, in an absolute sense, it suffices to place limits on initial states.

Postulate (Initial State) *Initial states of an* effective *algorithm are all (absolutely) effective (in the sense of Definition 1). Initial states of a* relatively effective *algorithm are all relatively effective (in the sense of Definition 2). In both cases, initial states are all identical, up to isomorphism, except for input values.*

Since transitions make only finitely many changes, once initial states are effective, so are all subsequent states.

We will say that an algorithm *computes* a partial function f over a domain D if there are *input* terms such that their values in all initial states with domain D cover all possible input values. We also demand that those states otherwise agree on the values of all terms, so no information is hidden in individual states. Given values $\bar{a}$ for the input terms, the corresponding input state leads, via a sequence of transitions specified by the algorithm, to a terminal state in which the value of some designated *output* term is $f(\bar{a})$ whenever the latter is defined, and leads to an infinite computation whenever it is not.

When we spoke earlier (Definitions 1 and 2) of "bootstrapping", we meant that there is a way of programming the defined operations, using constructors and oracles, if any. And if there is any way of programming them, then there is an abstract-state-machine program that fits the bill. For example, with 0 and successor (add 1), one can program addition (+), starting from basic states, so addition my be included in the initial states of (absolutely) effective algorithms over the natural numbers. Multiplication is also effective, since there is a program for multiplication that makes use of addition.

We are requiring that all elements of an algorithm's domain be accessible via terms in initial states (inaccessible superfluous elements may be removed with no ill effect). But note that a transition may cause accessible elements to become inaccessible in later states [47].

5.3 Relatively Effective Algorithms

Just like Turing extended his machines to incorporate oracles, the notion of recursive functions has been extended to allow oracles, for total functions by Turing [58, p. 175] and for partial ones later by Kleene [31, p. 178].

One form of this generalization is as follows: The *partial-recursive functions relative to oracles* $\mathcal{O}$ is the class of partial functions over the naturals, $\mathbb{N}$, that includes the constant zero (nullary) function, successor, all the projections, plus the operations in $\mathcal{O}$, and is closed under composition, primitive recursion, and minimization. We say that an algebra (with finitely or infinitely many partial functions) over the naturals is *recursive in* $\mathcal{O}$ if all its functions are.

Another extension of recursion theory applies it to domains other than the naturals. For this, we need the concept of "simulation" under encodings. An algebra $\mathcal{A}$ with domain D *simulates* an algebra $\mathcal{B}$ with domain E if there is an injective encoding ρ of E into D such that for every partial function g of $\mathcal{B}$ there is a partial function f of $\mathcal{A}$, such that $\rho \circ g = f \circ \rho$. A detailed discussion of simulations may be found in [2].

So, a state X over vocabulary F and arbitrary domain D is *computable over oracles* $\mathcal{O}$ if there is an encoding of D into the naturals and a recursive structure Y with domain $\mathbb{N}$ over oracles $\rho \circ o \circ \rho^{-1}$ for all $o \in \mathcal{O}$ that simulates X via ρ. An algorithm is *relatively computable* if all its initial states are computable all over the same oracle. And a model is *relatively computable* if all its algorithms are, via the same encoding and same oracle. Sans oracles, we call it *computable*. This is akin to a *computable algebra*, as in [17, 40, 45, 56], but we are not placing restrictions on the injective encoding.

Were we not to require the encoding to be an injection, we could trivially simulate everything by encoding everything by a single constant. One may ask whether the allowance of any injective encoding between the arbitrary domain and the natural numbers is sensible. But it turns out that, as long as all domain elements are reachable by ground terms, any arbitrary injective representation implies the existence of a bijection between the domain and the natural numbers [5, Lemma 1]. Hence, the initial functions of a computable algorithm are isomorphic to some partial-recursive functions, which makes their effectiveness hard to dispute.

For example, one standard injective encoding of lists, with nullary "ε" and binary ":" as constructors, is given by $\rho(\varepsilon) = 0$ and $\rho(x : y) = 2^{\rho(x)}3^{\rho(y)}$. The standard bijective encoding is $\rho(\varepsilon) = 0$ and $\rho(x : y) = 2^{\rho(x)}(2\rho(y) + 1)$.

These two notions, effective relative to oracles and computable over oracles, are coextensional (cf. the non-oracle case proved in [4]).

An alternative An equivalent definition—along the lines of Gödel's [19] original definition of recursive equations—is to say that an algebra over domain D, with finitely many operations F, is computable relative to $\mathcal{O}$ if there exist constructors C for D and a finite set E of equations defining F. Each equation in E is of the form $f(\bar{s}) = t$, where f is a symbol for an operation in F, $\bar{s}$ is a tuple of constructor terms built from C and variables, and t is an arbitrary term built from F, C, and variables. The equations define an operation f in F relative to $\mathcal{O}$ if for all tuples $\bar{c}$ of ground constructor terms, one can deduce (by substitution of equals for equals) $E \cup \mathcal{O} \vdash f(\bar{c}) = d$ for *at most one* ground constructor term d, where $\mathcal{O}$ is now an infinite set of (ground) equations giving the (defined) values of the oracular functions in constructor terms.

For example, a computable algebra of lists with an append operation $\star$ is defined by $\varepsilon \star z = z$ and $(x : y) \star z = x \cdot (y \star z)$. With $\star$ as the (in this case, computable) oracle, one can define list reversal using just $r(\varepsilon) = \varepsilon$ and $r(x : y) = r(y) \star (x : \varepsilon)$.

6 Conclusion

To summarize, we have seen that a model of computation is effective relative to oracles (and can be programmed by abstract state machines over the domain in question) if and only if the congruences of its states are semi-decidable by oracular Turing machines if and only all its (partial) functions are (partially) computable over those oracles (vis-à-vis some encoding).

Theorem *Every relatively effective algorithm can be simulated by an oracular Turing machine.*

The fact that these three prima facie different definitions of relative effectiveness over arbitrary domains, building on competing suggestions in [4, 14, 47], comprise exactly the same functions, strengthens our conviction that the essence of the underlying notion of effectiveness has in fact been captured.

In the special case of no oracles, this proves (a formalization of) what Church and Turing have claimed:

Theorem (Church-Turing Thesis [4]) *Every absolutely effective algorithm can be simulated by a Turing machine.*

In fact [5, Theorem 4], the set of Turing-computable string functions (and likewise the set of partial recursive functions each algorithm in the model, there is a partial-recursive function) is the unique maximal effective model, up to isomorphism, over any countable domain. By "maximal", we mean that adding any function would make it impossible to show the model to be computable (via simulation).

Moreover, we have recently demonstrated the validity of the widely believed (classical; non-physical) *Extended Church-Turing Thesis*:

Theorem (Extended Church-Turing Thesis [13]) *Every effective algorithm can be polynomially simulated by a Turing machine.*

It follows from all the above that any model purporting to be hypercomputational model, that computes all the Turing-computable functions and then some, be they (idealized) humans (as claimed, for example, in [39, 42, 6]), theoretical contrivance (e.g. [20, 44, 7]), or hypothetical (or idealized) physical apparatus (as proposed, for instance, in [15, 28, 55]), must violate one of our postulates. Note that, to be truly hypercomputational, it is crucial that a model that encodes strings in some way also be capable of computing the ordinary computable functions. It is not sufficient to merely compute one additional function—as explained in [3].[5]

References

[1] Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. Exact exploration and hanging algorithms. In *Proceedings of the 19th EACSL Annual Conferences on Computer Science Logic (Brno, Czech Republic)*, volume 6247 of *Lecture Notes in Computer Science*, pages 140–154, Berlin, Germany, August 2010. Springer. `doi:10.1007/978-3-642-15205-4_14`. Available at `http://nachum.org/papers/HangingAlgorithms.pdf` (viewed June 3, 2011); longer version at `http://nachum.org/papers/ExactExploration.pdf` (viewed May 27, 2011).

[2] Udi Boker and Nachum Dershowitz. Comparing computational power. *Logic Journal of the IGPL* 14(5):633–648, 2006. `doi:10.1007/978-3-540-78127-1`.

[5]Cf. [11, p. 1]: "Hypercomputation is the computation of functions or numbers that cannot be computed in the sense of Turing. . . ."

[3] Udi Boker and Nachum Dershowitz. A hypercomputational alien, *Applied Mathematics and Computation* 178(1):44–57, 2006. `doi:10.1016/j.amc.2005.09.069`

[4] Udi Boker and Nachum Dershowitz. The Church-Turing thesis over arbitrary domains. In Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich, editors, *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 199–229. Springer, 2008. `doi:10.1007/978-3-642-15205-4_14`. Available at `http://nachum.org/papers/ArbitraryDomains.pdf` (viewed Dec. 13, 2011).

[5] Udi Boker and Nachum Dershowitz. Three paths to effectiveness. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 36–47, Berlin, Germany, August 2010. Springer. `doi:10.1007/978-3-642-15025-8_7`. Available at `http://nachum.org/papers/ThreePathsToEffectiveness.pdf` (viewed Dec. 13, 2011).

[6] Selmer Bringsjord, Owen Kellett, Andrew Shilliday, Joshua Taylor, Bram van Heuveln, Yingrui Yang, Jeffrey Baumes, and Kyle Ross: A new Gödelian argument for hypercomputing minds based on the busy beaver problem. *J. Applied Mathematics and Computation* 176(2): 516–530 (2006) `doi:10.1016/j.amc.2005.09.071`

[7] Mark Burgin (2005), *Super-Recursive Algorithms*, Monographs in computer science, Springer. `doi:10.1016/j.tcs.2003.12.001`

[8] Samuel R. Buss, Alexander A. Kechris, Anand Pillay, and Richard A. Shore, The prospects for mathematical logic in the twenty-first century, *Bulletin of Symbolic Logic*, vol. 7, no. 2, June 2001, pp. 169–196. Available at `http://www.math.ucla.edu/~asl/bsl/0702/0702-001.ps` (viewed Dec. 13, 2011). `doi:10.2307/2687773`

[9] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936. `doi:10.2307/2371045`

[10] Alonzo Church, review of Alan M. Turing, On computable numbers, with an application to the Entscheidungsproblem (*Proceedings of the London Mathematical Society*, vol. 2, no. 42, 1936, pp. 230–265), *Journal of Symbolic Logic*, vol. 2, 1937, pp. 42–43. `doi:10.2307/2268810`

[11] B. Jack Copeland. Hypercomputation. *Minds and Machines*, 12:461–502, 2002. `doi:10.1016/j.tcs.2003.12.014`

[12] Martin Davis. The myth of hypercomputation. In Christof Teuscher, editor, *Alan Turing: Life and Legacy of a Great Thinker*, pages 195–212. Springer, 2003.

[13] Nachum Dershowitz and Evgenia Falkovich. A formalization and proof of the Extended Church-Turing Thesis. In *Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011)*, Zurich, Switzerland, July 2011. Available at `http://nachum.org/papers/ECTT.pdf` (viewed Dec. 9, 2011).

[14] Nachum Dershowitz and Yuri Gurevich. A natural axiomatization of computability and proof of Church's Thesis. *Bulletin of Symbolic Logic*, 14(3):299–350, September 2008. `doi:10.2178/bsl/1231081370`. Available at `http://nachum.org/papers/Church.pdf` (viewed Dec. 13, 2011).

[15] Gábor Etesi and István Németi, 2002, Non-Turing computations via Malament-Hogarth space-times, *Int. J. Theor. Phys.* 41(2), 2002, 341–370. `doi:10.1023/A:1014019225365`. Available at `http://lanl.arxiv.org/pdf/gr-qc/0104023v2` (viewed Dec. 8, 2011).

[16] Harvey M. Friedman. Mathematical logic in the 20th and 21st centuries. FOM mailing list. April 27, 2000. Available at `http://cs.nyu.edu/pipermail/fom/2000-April/003913.html` (viewed December 6, 2011).

[17] 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. `doi:10.1098/rsta.1956.0003`

[18] Robin Gandy. Church's thesis and principles for mechanisms. In *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, pages 123–148. North-Holland, 1980.

[19] Kurt Gödel, On undecidable propositions of formal mathematical systems, Lecture notes by S. C. Kleene and J. B. Rosser, Inst. for Advanced Study, Princeton, 1934. Reprinted with corrections and postscriptum in M. Davis (ed.): *The Undecidable – Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions*, Raven Press, 1965, pp. 39–74. The postscriptum is also reprinted in Godel's *Collected Works*, vol. I, pp. 369–371.

[20] E. Mark Gold. Limiting recursion. *J. Symbolic Logic*, 30(1):28–48, 1965. `doi:10.2307/2270580`

[21] Saul Gorn. Algorithms: Bisection routine. *Communications of the ACM*, 3(3):174, 1960. `doi:10.1145/367149.367173`

[22] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995. Available at `http://research.microsoft.com/~gurevich/opera/103.pdf` (viewed Apr. 15, 2009).

[23] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000. `doi:10.1145/343369.343384`. Available at `http://research.microsoft.com/~gurevich/opera/141.pdf` (viewed Apr. 15, 2009).

[24] David Harel. On folk theorems. *Communications of the ACM*, 23(7):379–389, July 1980. `doi:10.1145/358886.358892`

[25] David Hilbert, Mathematische Probleme: Vortrag, gehalten auf dem internationalen Mathematiker-Kongreß zu Paris 1900 (in German). Available at `http://wikilivres.info/wiki/Mathematische_Probleme` (viewed Dec. 1, 2011).

[26] 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.

[27] John L. Kelley, *General Topology*, van Nostrand, New York, 1955.

[28] Tien D. Kieu. Quantum algorithm for Hilbert's Tenth Problem. *International Journal of Theoretical Physics*, 42:1461–1478, 2003. `doi:10.1103/PhysRevE.77.036708`.

[29] Stephen C. Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936. `doi:10.1215/S0012-7094-36-00227-2`.

[30] Stephen C. Kleene, Recursive predicates and quantifiers, *Transactions of the American Mathematical Society*, vol. 53, no. 1, 1943, pp. 41–73. Reprinted in M. Davis (ed.), *The Undecidable*, Raven Press, Hewlett, NY, 1965, pp. 255–287. `doi:10.2307/1990131`.

[31] Stephen C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, New York, 1952. `doi:10.1007/978-0-8176-4769-8`.

[32] Stephen C. Kleene. *Mathematical Logic*. Wiley, New York, 1967. `doi:10.2307/2268665`.

[33] Stephen C. Kleene. Reflections on Church's thesis. *Notre Dame Journal of Formal Logic*, 28(4):490–498, 1987. `doi:10.1305/ndjfl/1093637645`.

[34] Stephen C. Kleene. Turing's analysis of computability, and major applications of it. In *A Half-Century Survey on The Universal Turing Machine*, pages 17–54, New York, NY, 1988. Oxford University Press. `doi:10.1007/978-3-7091-6597-3_2`.

[35] Donald E. Knuth, Algorithm and program; information and data, *Communications of the ACM*, vol. 9, no. 9, Sept. 1966, p. 654.

[36] Donald E. Knuth, 1976, Mathematics and computer science: Coping with finiteness". *Science* 194 (4271): 1235–1242. `doi:10.1126/science.194.4271.1235`.

[37] Andreĭ N. Kolmogorov, O ponyatii algoritma [On the concept of algorithm], *Uspekhi Matematicheskikh Nauk* [*Russian Mathematical Surveys*], vol. 8, no. 4, 1953, pp. 175–176 (in Russian). English version in: Vladimir A. Uspensky and Alexei L. Semenov, *Algorithms: Main Ideas and Applications*, Kluwer, Norwell, MA, 1993, pp. 18–19.

[38] Andreĭ N. Kolmogorov and Vladimir A. Uspensky, K opredeleniu algoritma, *Uspekhi Matematicheskikh Nauk* [*Russian Mathematical Surveys*], vol. 13, no. 4, 1958, pages 3–28 (in Russian). English version: On the definition of an algorithm, *American Mathematical Society Translations*, ser. II, vol. 29, 1963, pp. 217–245.

[39] 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, no. 4, Dec. 1982, pp. 441–444.

[40] Anatoliĭ I. Mal'cev, Konstruktivnyye algyebry. 1, *Uspekhi Matematicheskikh Nauk*, vol. 16, no. 3, 1961, pp. 3–60. English version: Constructive algebras, I, The meta-mathematics of algebraic systems (K. A. Hirsch, translator), *Russian Mathematical Surveys*, vol. 16, no. 3, 1961, 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.

[41] Yiannis N. Moschovakis, 2001, What is an algorithm? In: *Mathematics Unlimited — 2001 and Beyond*, B. Engquist and W. Schmid, editors, Springer, Berlin, pp. 929–936. Available at `http://www.math.ucla.edu/~ynm/papers/eng.pdf` (viewed May 4, 2012).

[42] Roger Penrose. *Shadows of the Mind: A Search for the Missing Science of Consciousness.* Oxford University Press, Oxford, 1994.

[43] Emil L. Post. Absolutely unsolvable problems and relatively undecidable propositions: Account of an anticipation. In M. Davis, editor, *Solvability, Provability, Definability: The Collected Works of Emil L. Post*, pp. 375–441. Birkhaüser, Boston, MA, 1994. Unpublished paper, 1941.

[44] Hilary Putnam. Trial and error predicates and the solution to a problem of Mostowski. *J. Symbolic Logic*, 30(1):49–57, 1965. `doi:10.2307/2270581`.

[45] Michael O. Rabin, Computable algebra, general theory and the theory of computable fields, *Transactions of the American Mathematical Society*, vol. 95, no. 2, May 1960, pp. 341–360. `doi:10.2307/1993295`.

[46] Wolfgang Reisig. On Gurevich's theorem on sequential algorithms. *Acta Informatica*, 39(4):273–305, April 2003. Available at `http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2003_ai395.pdf` (viewed Dec. 13, 2011). `doi:10.1007/s00236-002-0106-3`.

[47] Wolfgang Reisig. The computable kernel of Abstract State Machines. *Theoretical Computer Science*, 409(1):126–136, December 2008. `doi:10.1016/j.tcs.2008.08.041`. Draft available at `http://www2.informatik.hu-berlin.de/top/download/publications/Reisig2004_hub_tr177.pdf` (viewed Dec. 13, 2011). `doi:10.1007/3-540-36498-6`.

[48] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1966.

[49] Joseph R. Shoenfield. *Recursion Theory*, volume 1 of *Lecture Notes In Logic.* Springer, Heidelberg, 1991. `doi:10.1090/S0273-0979-1993-00346-X`.

[50] Wilfried Sieg, Mechanical procedures and mathematical experiences, in: *Mathematics and Mind* (A. George, editor), Oxford University Press, Oxford, 1994, pp. 71–117.

[51] Wilfried Sieg, Step by recursive step: Church's analysis of effective calculability, *Bulletin of Symbolic Logic* 3(2), June 1997. `doi:10.2307/421012`.

[52] Wilfried Sieg, Hilbert's programs: 1917–1922, *Bulletin of Symbolic Logic*, vol. 5, no. 1, Mar. 1999, pages 1–44. Available at `http://www.math.ucla.edu/~asl/bsl/0501/0501-001.ps` (viewed Dec. 13, 2011). `doi:10.2307/421139`.

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

[54] Wilfried Sieg and John Byrnes, An abstract model for parallel computations: Gandy's thesis, *The Monist*, vol. 82, no. 1, 1999, pages 150–164.

[55] Warren D. Smith. (July 2006). Church's thesis meets the N-body problem. *Applied Mathematics and Computation*, 178(1):154–183. doi:10.1.1.54.3570.

[56] John V. Tucker and Jeffery I. Zucker. Abstract versus concrete computation on metric partial algebras. *ACM Transactions on Computational Logic*, 5(4):611–668, 2004. doi:10.1145/1024922.1024924.

[57] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936–37. Corrections in vol. 43 (1937), pages 544–546. Reprinted in M. Davis (ed.), *The Undecidable*, Raven Press, Hewlett, NY, 1965. Available at http://www.abelard.org/turpap2/tp2-ie.asp. doi:10.1112/plms/s2-42.1.230.

[58] Alan M. Turing, Systems of logics based on ordinals. *Proceedings of the London Mathematical Society*, 45:161–228, 1939. doi:10.1112/plms/s2-45.1.161.

Index