A Formalization of the Church-Turing Thesis for State-Transition Models

Udi Boker and Nachum Dershowitz

School of Computer Science
Tel Aviv University
Ramat Aviv, Tel Aviv 69978, Israel
email: `udiboker@tau.ac.il`,
`nachum.dershowitz@cs.tau.ac.il`

Abstract. Our goal is to formalize the Church-Turing Thesis for a very large class of computational models. Specifically, the notion of an "effective model of computation" over an arbitrary countable domain is axiomatized. This is accomplished by modifying Gurevich's "Abstract State Machine" postulates for state-transition systems. A proof is provided that *all* models satisfying our axioms, regardless of underlying data structure—and including all standard state-transition models—are equivalent to (up to isomorphism), or weaker than, Turing machines. To allow the comparison of arbitrary models operating over arbitrary domains, we employ a quasi-ordering on computational models, based on their extensionality.

> LCMs can do anything that could be described as "rule of thumb" or "purely mechanical".... This is sufficiently well established that it is now agreed amongst logicians that "calculable by means of an LCM" is the correct accurate rendering of such phrases.
>
> —*Alan Turing* [21, p. 7]

1 Introduction

Motivation. The famous Church-Turing Thesis is inherently vague, relating to "effective" models in the broad, intuitive, sense. Though generally believed to be true, there have always been—especially recently—efforts to circumvent it under the banner of "hypercomputation." See, for example, [15, 5, 12, 18]. It is thus important to try to understand what the formal effectiveness constraints are that every such hypercomputational model must violate.

When designing a specific computational model, one first defines the model's domain and its manner of operation. For example, Turing machines operate over tapes of symbols, the λ-calculus over λ-terms, and counter machines over an n-tuple of natural numbers. In contrast, for axiomatizing effectiveness constraints, one ought to provide general axioms, not relating to a specific domain or operational mechanism.

Main Contribution. We provide four axioms for an "effective computational model." We believe these axioms to be general, both in the sense of capturing basic intuitive notions, and in the sense of encompassing a very large class of models, among which are all the standard state-transition models. A proof is provided (in the Appendix) that all models satisfying these axioms, regardless of underlying data structure, are equivalent to (up to isomorphism), or weaker than, Turing machines.

To achieve a wide effectiveness notion, we base our axioms on Gurevich's general definition of a "sequential algorithm" [10]. The models of the resulting class may operate over arbitrary domains, applying arbitrary internal mechanisms. In contrast to any particular model of computation, our axioms apply to models operating over *any* countable domain.

For comparing such a varied class of models with Turing machines, we employ a comparison notion based on the extensionality of models, a variation of the development in [2].

We believe that the axioms provided here may be a good starting point for further formalizations of effectiveness, as they demarcate a large *class* of computational models for which the Church-Turing Thesis is formally established.

Background. In 1936, Alonzo Church and Alan Turing each formulated a claim that a particular model of computation completely captures the conceptual notion of "effective" computability. Church [4, p. 356] proposed that effective computability of numeric functions be identified with Gödel and Herbrand's general recursive functions, or—equivalently—with Church and Kleene's lambda-definable functions of positive integers. Similarly, Turing [20] suggested that his computational model, namely, Turing machines, could compute anything that might be mechanically computable. As we know, Turing machines compute the same total functions and the same partial functions as do all other designs of practical universal models: Markov algorithms, Thue systems, Gödel's recursive functions, Church's lambda calculus, Post's string systems, Minsky's counter machines, random access machines, term rewriting systems, etc.

After Turing proved that his machines compute exactly the same numeric functions as does the lambda calculus, Kleene [13, p. 232] combined the two claims into one:

> So 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*.

The claim, then, is the following:

> **Church-Turing Thesis.** *All effective computational models are equivalent to, or weaker than, Turing machines.*

By 1948, Turing was convinced of the accuracy of his contention. See the opening quote, wherein he refers to his machine model as LCMs, or, "logical computing machines."

Goal. To formalize this thesis, we need to make precise what is meant by each of the terms: "effective," "computational model," and "weaker or equivalent." As suggested by Shoenfield [16, p. 26]:

> [I]t 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.

In fact, Gödel has also been reported (by Church in a letter to Kleene cited by Davis in [6]) to have thought "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."

This is the direction we follow.

Previous Work. Turing [20] already formulated some axioms for effective sequential deterministic symbol manipulation: finite internal states; finite symbol space; external memory that can be represented linearly; finite observability and local action.

Gandy [9], and later Sieg and Byrnes [17], define a model 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.

Whereas Turing machine states involve a linear sequence of symbols, and Gandy machine states are hereditarily finite sets, our axioms are meant to apply to arbitrary (countable) domains. For this reason, we use Gurevich's more general "Abstract State Machines" (ASMs) [10] as our starting point. (Some of the problems of incorporating the Gandy model under the abstract state machine rubric are dealt with in [1].) We also pay careful attention to questions of mappings between representations of data.

Approach. In what follows, we *axiomatize* a large class of computational models that includes all known Turing-complete state-transition algorithmic models, operating over any countable domain, and prove that they are all Turing-computable, up to isomorphism of domains.

We formalize the informal notions of the Church-Turing Thesis as follows:

- Since the comparison is meant to be extensional, we allow a "computational model" to be any set of partial functions over some domain. That is, a (finitary) partial algebra, with (normally) an infinite vocabulary.
- Since we are dealing with mechanisms that may operate on different physical media and employ very different data structures, we adopt a variation of the quasi-ordering on extensional models developed in [2].
- To capture what is intended by "effective," we propose a set of axioms for models, adapting the postulates developed for the "Sequential Algorithm Thesis" of Gurevich.

Effectivity axioms. We understand an "effective computational model" to be some set of "effective procedures". Since all procedures of a specific computational model should have some common mechanism, a minimal requirement is that they share the same domain representation. Any "effective procedure" should satisfy four postulates (formally defined as Axioms 1–4 in Sections 2–3):

1. **Sequential Time Axiom.** The procedure can be viewed as a set of states, a specified initial state, and a transition function from state to state.
   This postulate reflects the view of a computation as some transition system, as suggested by Knuth [14, p. 7]:
   > [D]efine a *computational method* to be a quadruple (Q, I, Ω, f), in which Q is a set containing subsets I and Ω, and f is a function from Q into itself.... The four quantities Q, I, Ω, f are intended to represent respectively the states of the computation, the input, the output, and the computational rule.
2. **Abstract State Axiom.** Its states are (first-order) structures of the same finite vocabulary. States are closed under isomorphism, and the transition function preserves isomorphism.
   Formalizing the states of the transition system as mathematical structures follows the proposal of Gurevich [10, p. 78]:
   > What would their states be? The huge experience of mathematical logic indicates that any kind of static mathematical reality can be faithfully represented as a first-order structure.
3. **Bounded Exploration Axiom.** There is a finite bound on the number of vocabulary-terms that affect the transition function.
   The postulate ensures that the transition system has effective behavior. One way to look at it is [10, p. 82]:
   > We assume informally that any algorithm A can be given by a finite text that explains the algorithm without presupposing any special knowledge.
4. **Initial Data Axiom.** The initial state comprises only finite data in addition to the domain representation; the latter is isomorphic to a Herbrand universe.
   Our fourth postulate restricts procedures to be wholly effective by insisting on the effectiveness of the initial data [7, p. 195]:
   > [I]f non-computable inputs are permitted, then non-computable outputs are attainable.

   To preclude this we insist that the initial data does not contain more than a finite amount of information. The underlying data type is required to be isomorphic to a Herbrand universe, so that nothing more than equality of data elements is given at the outset of a computation.

The freedom to add any finite data is obvious, but why do we limit the domain representation to be isomorphic to a Herbrand universe? There are two limitations here: a) every domain element has a name (a closed term); and b) the name of each element is unique. Were we to allow unnamed domain elements, then a computation cannot be referred to, nor repeated, hence would not be effective. As for the uniqueness of the names, allowing a built-in equality notion with an "infinite memory" of equal couples is obviously not effective. Hence, the equality notion should be the result of some internal mechanism, and thus needs to be a part of the computational model.

Proof sketch. To prove the thesis—under these assumptions—we prove that any procedure operating over an *arbitrary* countable structure, and satisfying the initial-data postulate, can be mapped (using a rigorous notion of mapping) to some **while**-like computer program, which in turn can be mapped to a Turing machine.

Hence, if one defines any model that satisfies the four axioms, this model is necessarily bounded by Turing computability, while if one claims that a model is hypercomputational, it must violate some of the axioms.

Overview. In the next two sections, we axiomatize "sequential procedures" and "effective models". In Section 4, we define an appropriate computational-power comparison notion, and use it in Section 5 to show that Turing machines, which constitute an effective model, are at least as powerful as any effective model (Theorem 2). We conclude with a brief discussion.

Proofs are given in the Appendix.

2 Sequential Procedures

We axiomatize "sequential procedures" along the lines of Gurevich's sequential algorithms [10]. We later axiomatize, in Section 3, "effective procedures" as their subclass. We start by providing the standard definition of mathematical structures, upon which sequential procedures are defined.

2.1 Structures

The states of a procedure should be a full instantaneous description of it. We represent them by (first order) *structures*, using the standard notion of structure from mathematical logic. For convenience, these structures will be *algebras*; that is, having purely functional vocabulary (without relations).

Definition 1 (Structures).

- *A* domain D *is a (nonempty) set of elements.*
- *A* vocabulary $\mathcal{F}$ *is a collection of function names, each with a fixed finite arity.*
- *A* term *of vocabulary* $\mathcal{F}$ *is either a nullary function name (constant) in* $\mathcal{F}$ *or takes the form* $f(t_1, \ldots, t_k)$*, where* f *is a function name in* $\mathcal{F}$ *of positive arity* k *and* $t_1, \ldots, t_k$ *are terms.*
- *A* structure S *of vocabulary* $\mathcal{F}$ *is a domain* D *together with interpretations* $[\![f]\!]_S$ *over* D *of the function names* $f \in \mathcal{F}$*.*
- *A* location *of vocabulary* $\mathcal{F}$ *over a domain* D *is a pair, denoted* $f(\overline{a})$*, where* f *is a* k*-ary function name in* $\mathcal{F}$ *and* $\overline{a}$ *is a k-tuple of elements of* D*. (If* f *is a constant, then* $\overline{a}$ *is the empty tuple.)*
- *The* value *of a location* $f(\overline{a})$ *in a structure* S*, denoted* $[\![f(\overline{a})]\!]_S$*, is the domain element* $[\![f]\!]_S(\overline{a})$*.*
- *It is often useful to indicate a location by a (ground) term* $f(t_1, \ldots, t_k)$*, standing for* $f([\![t_1]\!]_S, \ldots, [\![t_k]\!]_S)$*.*
- *Structures* S *and* S' *with vocabulary* $\mathcal{F}$ *over the same domain* coincide *over a set* T *of* $\mathcal{F}$*-terms if* $[\![t]\!]_S = [\![t]\!]_{S'}$ *for all terms* $t \in T$*.*

It is convenient to think of a structure S as a memory, or data-storage, of a kind. For example, for storing an (infinite) two dimensional table of integers, we need a structure S over the domain of integers, having a single binary function name f in its vocabulary. Each entry of the table is a location. The location has two indices, i and j, for its row and column in the table, marked $f(i, j)$. The content of an entry (location) in the table is its value $[\![f(i, j)]\!]_S$.

Definition 2 (Structure Union). *A structure S of vocabulary $\mathcal{F}$ over domain D is the* union *of structures S' and S'' of vocabularies $\mathcal{F}'$ and $\mathcal{F}''$, respectively, over D, denoted $S = S' \uplus S''$, if $\mathcal{F} = \mathcal{F}' \uplus \mathcal{F}''$, $[\![l]\!]_S = [\![l]\!]_{S'}$ for every location l in S', and $[\![l]\!]_S = [\![l]\!]_{S''}$ for every location l in S''.*

Definition 3 (Update). *An* update *of location l over domain D is a pair, denoted $l := v$, where v is an element of D.*

Definition 4 (Structure Modification). *The* modification *of a structure S into a structure S' of the same vocabulary and domain, denoted $\Delta(S, S')$, is the set of updates $\{l := v' \mid [\![l]\!]_S \neq [\![l]\!]_{S'} = v'\}$.*

Definition 5 (Structure Mapping). *Let S be structure of vocabulary $\mathcal{F}$ over domain D and $\rho : D \to D'$ an injection from D to domain D'. A* mapping *of S by ρ, denoted $\rho(S)$, is a structure S' of vocabulary $\mathcal{F}$ over D', such that $\rho([\![f(\overline{a})]\!]_S) = [\![f(\rho(\overline{a}))]\!]_{S'}$ for every location $f(\overline{a})$ in S.*

Structures S and S' of the same vocabulary over domains D and D', respectively, are *isomorphic*, denoted $S \simeq S'$, if there is a bijection $\pi : D \leftrightarrow D'$, such that $S' = \pi(S)$.

2.2 Axiomatic Sequential Procedures

The axiomatization of a "sequential procedure" is very similar to that of Gurevich's sequential algorithm [10], with the following two main differences, allowing for the computation of a specific function, rather than expressing an abstract algorithm:

- The vocabulary includes special constants "*In*" and "*Out*."
- There is a single initial state, up to changes in *In*.

Axiom 1 (Sequential Time). *The procedure can be viewed as a collection $\mathcal{S}$ of* states*, a sub-collection $\mathcal{S}_0 \subseteq \mathcal{S}$ of* initial states*, and a* transition function *$\tau : \mathcal{S} \to \mathcal{S}$ from state to state.*

Axiom 2 (Abstract State).

- States. *All states are first-order structures of the same finite* vocabulary *$\mathcal{F}$.*
- Input. *There are nullary function names* In *and* Out *in $\mathcal{F}$. All initial states ($\mathcal{S}_0 \subseteq \mathcal{S}$) are over the same* domain *D, and are equal up to changes in the value of* In*. (The initial states may be referred to as a single state S_0).*

– Isomorphism Closure. *The procedure states are closed under isomorphism. That is, if there is a state $S \in \mathcal{S}$, and an isomorphism π via which S is isomorphic to a $\mathcal{F}$-structure S', then S' is also a state in $\mathcal{S}$.*
– Isomorphism Preservation. *The transition function preserves isomorphism. That is, if states S and S' are isomorphic via π, then $\tau(S)$ and $\tau(S')$ are also isomorphic via π.*
– Domain Preservation. *The transition function preserves the domain. That is, the domain of S and $\tau(S)$ is the same for every state $S \in \mathcal{S}$.*

Axiom 3 (Bounded Exploration). *There exists a finite set T of "critical" terms, such that $\Delta(S, \tau(S)) = \Delta(S', \tau(S'))$ if S and S' coincide over T, for all states $S, S' \in \mathcal{S}$.*

The isomorphism constraints reflects the fact that we are working at a fixed level of abstraction. See [10, p. 89]:

> A structure should be seen as a mere representation of its isomorphism type; only the isomorphism type matters. Hence the first of the two statements: distinct isomorphic structures are just different representations of the same isomorphic type, and if one of them is a state of the given algorithm A, then the other should be a state of A as well.

Domain preservation simply ensures that a specific "run" of the procedure is over a specific domain. The bounded-exploration axiom ensures that the behavior of the procedure is effective. This reflects the informal assumption that the program of an algorithm can be given by a finite text [10, p. 90].

Elements of a procedure A are indexed $\mathcal{F}_A$, τ_A, etc.

Definition 6 (Runs).

1. *A* run *of procedure A is a finite or infinite sequence $S_0 \leadsto_\tau S_1 \leadsto_\tau S_2 \leadsto_\tau \cdots$, where S_0 is an initial state and every $S_{i+1} = \tau_A(S_i)$.*
2. *A run $S_0 \leadsto_\tau S_1 \leadsto_\tau S_2 \leadsto_\tau \cdots$* terminates *if it is finite or if $S_i = S_{i+1}$ from some point on.*
3. *The* terminating state *of a terminating run $S_0 \leadsto_\tau S_1 \leadsto_\tau S_2 \leadsto_\tau \cdots$ is its last state if it is finite, or its stable state if it is infinite.*
4. *If there is a terminating run beginning with state S and terminating in state S', we write $S \leadsto^!_\tau S'$.*

Definition 7 (Procedure Extensionality). *Let A be sequential procedure over domain D. The* extensionality *of A, denoted* ext A, *is the partial function $f : D \to D$, such that $f(x) = [\![Out]\!]_{S'}$ whenever there's a run $S \leadsto^!_\tau S'$ with $[\![In]\!]_S = x$, and is undefined otherwise.*

Equality, Booleans and Undefined. In contradistinction with Gurevich's ASM's, we do not have built in equality, booleans, or undefined in the definition of procedures. That is, a procedure need not have boolean terms ('True' and 'False') or connectives ('and' and 'or') pre-defined in its vocabulary; rather, they may be defined like any other function. It also should not have a special term for undefined values, though the value of

the function implemented by the procedure is not defined when its run doesn't terminate. The equality notion is also not presumed in the procedure's initial state; furthermore, it cannot be a part of the initial state of an "effective procedure" (axiomatize later as a sequential procedure, satisfying Axiom 4). Nevertheless, the internal mechanism (transition function) of an effective procedure (e.g. a computer program) may perform equality checks, as the four axioms don't prevent it.

3 Effective Models

We are interested in sequential procedures that can be satisfying the "initial-data" postulate (Axiom 4, below), which states that it may only have finite initial data in addition to the domain representation ("base structure"). An "effective model" is some set of "effective procedures" that share the same domain representation.

We formalize the finiteness of the initial data by allowing the initial state to contain an "almost-constant structure."

Definition 8 (Almost-Constant Structure). *A structure F is* almost constant *if all but a finite number of locations have the same value.*

Since we are heading for a characterization of effectiveness, the domain over which the procedure actually operates should have countably many elements, which have to be nameable. Hence, without loss of generality, one may assume that naming is via terms. We limit the naming to be unique, thus demanding that any equality notion between different terms be part of the computational model's mechanism (see Section 1 for additional details).

Definition 9 (Base Structure). *A structure S of finite vocabulary $\mathcal{F}$ over a domain D is a* base structure *if all the domain elements are the value of a unique $\mathcal{F}$-term. That is, for every element $e \in D$ there exists a unique $\mathcal{F}$-term t s.t. $[\![t]\!]_S = e$.*

A base structure is isomorphic to the standard free term algebra (Herbrand universe) of its vocabulary.

Proposition 1. *Let S be a base structure over vocabulary G and domain D, then:*

- *The vocabulary G has at least one nullary function.*
- *The domain D is countable.*
- *Every domain element is the value of a unique location of S.*

Example 1. A structure over the natural numbers with constant *zero* and unary function *successor*, interpreted as the regular successor, is a base structure.

Example 2. A structure over binary trees with constant *nil* and binary function *cons*, interpreted as in Lisp, is a base structure.

Axiom 4 (Initial Data). *The initial state consists of an infinite base structure and an almost-constant structure. That is, the initial state S_0 is $BS \uplus AS \uplus \{In\}$ for some infinite base structure BS and almost-constant structure AS.*

An *effective procedure* must satisfy Axioms 1–4. An *effective model E* must be some set of effective procedures that share the same base structure.

Single Initial State. Axiom 4 above speaks of a single initial state, while there are actually many initial states in a sequential procedure. However, as defined in Axiom 2, all initial states are equal up to changes in the value of *In*, thus may be referred to as a single state S_0. This state is a structure union (see Definition 2) of a base structure and an almost-constant structure.

Infinite Base Structure. We require the initial state to include an infinite base structure (and not a finite one), for allowing the power comparison with Turing machines via a firm comparison notion, which demands a bijective mapping between domains (see Section 4). An initial state with a finite base structure is also effective, however will require a more permissive comparison notion, as the one developed in [2].

Big Steps and Small Steps. A computational model might have some predefined complex operations, as in a RAM model with built-in integer multiplication. A view of such a model as a sequential procedure allows the initial state to include these complex functions as oracles [10]. Since we are demanding effectiveness, we cannot allow arbitrary functions as oracles, and force the initial state to include only finite data over and above the domain representation (Axiom 4). Hence, the view of a model at the required abstraction level is done by "big steps," which may employ complex functions, while these complex functions are implemented by a finite sequence of "small steps" behind the scenes. That is, a function that is the extensionality of an effective procedure may be included (as an oracle) in the initial state of another effective procedure. (Cf. the "turbo" steps of [8].)

4 Computational Power

Since we are dealing with models that operate on different data structures, we adopt a variant of the quasi-ordering on extensional power developed in [2]. The notion of [2] is based on a surjective mapping between model domains, while here, to meet also more conservative comparison notions, we allow only bijective mappings. As a result, we consider Turing machines (TM) to be at least as powerful as a model A only if TM implements all functions implemented by A ($\mathrm{TM} \supseteq A$) or TM is isomorphic to such a model ($\mathrm{TM} \simeq B \supseteq A$).

We consider only deterministic computational models, hence their computational power can be viewed as a set of functions. As models may have non-terminating computations, we deal with sets of partial functions. To simplify the development, we will assume for now that the domain and range of functions are identical, except that the range is extended with $\perp$, representing "undefined." An extension of this assumption to the general case can be found in [2].

We do not want to limit computational models to any specific mechanism, hence we allow it to be any object, as long as it is associated with the set of functions that it implements.

Definition 10 (Computational Model).

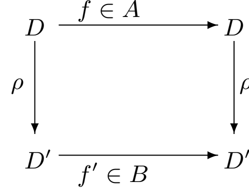

Fig. 1. *Model mapping:* Model B computes all functions of model A via mapping ρ.

- *A* computational model A *over domain* D *is any object associated with a set of partial functions* $f : D \to D$*. This set of functions is called the* extensionality *of the computational model, denoted* $\text{ext } A$*. The partial functions may also be interpreted as total functions* $f : D \to D \cup \{\bot\}$.
- *We write* $\text{dom } A$ *for the domain over which model* A *operates.*
- *For models* A *and* B*, and a function* f *we shall write* $f \in A$ *as shorthand for* $f \in \text{ext } A$*, and* $A \subseteq B$ *as short for* $\text{ext } A \subseteq \text{ext } B$.

To deal with models operating over different domains it is incumbent to map the domain of one model to that of the other. As a result, we get *model mapping*, which is a structure mapping of the model's extensionality, as defined in Definition 5. Model mapping is illustrated in Fig. 1.

Definition 11 (Computational Power).

1. *Model* B *is* (computationally) at least as powerful *as model* A*, denoted* $B \gtrsim A$*, if there is a bijective model mapping* $\pi : \text{dom } B \to \text{dom } A$ *such that* $\pi(B) \supseteq A$.
2. *Models* A *and* B *are* computationally equivalent *if* $A \gtrsim B \gtrsim A$*, in which case we write* $A \approx B$.

Proposition 2. *The computational power relation* $\gtrsim$ *between models is a quasi-order. Computational equivalence* $\approx$ *is an equivalence relation.*

Transitivity of $\gtrsim$ is because the composition of bijections is a bijection.

The above notion, when replacing the requirement for a bijective mapping with an injective mapping, is the notion of "implemented" used in [11, p. 52] and of "incorporated" used in [19, p. 29].

It should be mentioned that, generally, a model might be isomorphic to one computing more functions [3]. Fortunately, TM are an exception and are not susceptible to this anomaly [3].

5 Effective Equals Computable

Theorem 1. *Turing machines are an effective model.*

Theorem 2. *Turing machines* (TM) *are at least as powerful as any effective model. That is,* $\text{TM} \gtrsim E$ *for every model* E *satisfying the effectivity axioms.*

Proofs of Theorems 1 and 2 are relegated to the Appendix.

6 Discussion

Postulate minimality. An effective procedure should satisfy, by our definitions, Axioms 1–4. In the introduction, we argued for the necessity of the postulates from the intuitive point of view of effectiveness. Moreover, omitting any of them allows for models that compute more than Turing machines:

1. The Sequential Time Axiom is necessary if we wish to analyze computation, which is a step-by-step process. Allowing for transfinite computations, for example, would allow a model to precompute all values of a recursively-enumerable function.
2. In the context of effective computation, there is no room for infinitary functions, for example. Without closure under isomorphism there would be no value to the Bounded-Exploration Axiom, allowing the assigning of any desired value to the *Out* location.
3. By omitting the Bounded-Exploration Axiom, a procedure need not have any systematical behavior, hence may "compute" any function by simply assigning the desired value at the *Out* location. That is, for each initial state S there is a state S', s.t. $\tau(S) = S'$ and $[\![Out]\!]_{S'}$ is the 'desired' value.
4. Omitting the Initial-Data Axiom, one may "compute" any function (e.g. a halting oracle), by simply having all its values in the initial state. Such functions could also be encoded in equalities between locations, were the initial data not (isomorphic to) a free term algebra.

Algorithm vs. model. In [10], Gurevich proved that any algorithm satisfying his postulates can be represented by an Abstract State Machine. But an ASM is designed to be "abstract," so is defined on top of an arbitrary structure that may contain *non-effective* functions. Hence, it may compute non-effective functions. We have adopted Gurevich's postulates, but added an additional postulate (Axiom 4) for effectivity: an algorithm's initial state may contain only finite data in addition to the domain representation. Different runs of the same procedure share the same initial data, except for the input; different procedures of the same model share a base structure. We proved that—under these assumptions—the class of all effective procedures is of equivalent computational power to Turing machines.

Constructivity. With our definition of base structure (Definition 9) domain elements are of no importance, as long as there is a way to construct them. Hence, the domain is effective up to isomorphism. One may wish to require that domain elements themselves be effective (not up to isomorphism), requiring additional axioms.

References

1. A. Blass and Y. Gurevich. Background, reserve, and Gandy machines. In *Proceedings of CSL'2000, Peter Clote and Helmut Schwichtenberg, eds.*, volume 1862 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2000.

2. U. Boker and N. Dershowitz. How to compare the power of computational models. In *Computability in Europe 2005: New Computational Paradigms (Amsterdam), S. Barry Cooper, Benedikt Löwe, Leen Torenvliet, eds.*, volume 3526 of *Lecture Notes in Computer Science*, pages 54–64, Berlin, Germany, 2005. Springer-Verlag.
3. U. Boker and N. Dershowitz. Comparing computational power. *Logic Journal of the IGPL*, 2006, to appear. A preliminary version is available at: `http://arxiv.org/abs/cs.LO/0510069`.
4. A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.
5. B. J. Copeland. Hypercomputation. *Minds and Machines*, 12:461–502, 2002.
6. M. Davis. Why Gödel didn't have Church's Thesis. *Information and Control*, 54(1/2):3–24, 1982.
7. M. Davis. The myth of hypercomputation. In C. Teuscher, editor, *Alan Turing: Life and Legacy of a Great Thinker*, pages 195–212. Springer, 2003.
8. N. G. Fruja and R. F. Stärk. The hidden computation steps of Turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines — Advances in Theory and Applications, 10th International Workshop, ASM 2003, Taormina, Italy*, pages 244–262. Springer-Verlag, Lecture Notes in Computer Science 2589, 2003.
9. R. Gandy. Church's thesis and principles for mechanisms. In *The Kleene Symposium, J. Barwise, D. Kaplan, H. J. Keisler, P. Suppes, A. S. Troelstra, eds.*, volume 101 of *Studies in Logic and The Foundations of Mathematics*, pages 123–148. North-Holland, 1980.
10. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.
11. N. D. Jones. *Computability and Complexity from a Programming Perspective*. The MIT Press, Cambridge, Massachusetts, 1997.
12. T. D. Kieu. Quantum algorithm for Hilbert's Tenth Problem. *International Journal of Theoretical Physics*, 42:1461–1478, 2003.
13. S. C. Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.
14. D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, 1968.
15. T. Ord. Hypercomputation: Computing more than the Turing machine. *Technical report, University of Melbourne, Melbourne, Australia*, 2002.
16. J. R. Shoenfield. *Recursion Theory*, volume 1 of *Lecture Notes In Logic*. Springer-Verlag, Heidelberg, New York, 1991.
17. W. Sieg and J. Byrnes. An abstract model for parallel computations: Gandy's thesis. *The Monist*, 82(1):150–164, 1999.
18. H. T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit*. Birkhäuser, Boston, 1998.
19. R. Sommerhalder and S. C. van Westrhenen. *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility*. Addison-Wesley, Workingham, England, 1988.
20. A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936–37.
21. A. M. Turing. Intelligent machinery. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7. 1969. Unpublished 1948 report for National Physical Laboratory.

A Proofs of Theorems

We provide here proofs of Theorems 1 and 2. First, we require some additional definitions and lemmata.

A.1 Programmable Machines

In Section 2, we axiomatized sequential procedures. To link these procedures with Turing machines, we define some mediators, named "programmable procedures," along the lines of Gurevich's Abstract State Machines (ASMs) [10]. We then show that sequential procedures and programable procedures are equivalent (Lemma 1).

A "programmable procedure" is like a sequential procedure, with the main difference that its transition function should be given by a finite "flat program" rather than satisfy some constraints.

Definition 12 (Flat Program). *A* flat program P *of vocabulary* $\mathcal{F}$ *has the following syntax:*

if $x_{11} \doteq y_{11}$ and $x_{12} \doteq y_{12}$ and $\ldots$ $x_{1k_1} \doteq y_{1k_1}$
  then $l_1 := v_1$

if $x_{21} \doteq y_{21}$ and $x_{22} \doteq y_{22}$ and $\ldots$ $x_{2k_2} \doteq y_{2k_2}$
  then $l_2 := v_2$

$\vdots$

if $x_{n1} \doteq y_{n1}$ and $x_{n2} \doteq y_{n2}$ and $\ldots$ $x_{nk_n} \doteq y_{nk_n}$
  then $l_n := v_n$

where each $\doteq$ *is either '$=$' or '$\neq$',* $n, k_1, \ldots, k_n \in \mathbf{N}$*, and all the* x_{ij}*,* y_{ij}*,* l_i*, and* v_i *are* $\mathcal{F}$*-terms.*

Each line of the program is called a rule*. The part of a rule between the* if *and the* then *is the* condition*,* l_i *is its location, and* v_i *is its value.*

The activation *of a flat program* P *on an* $\mathcal{F}$*-structure* S*, denoted* $P(S)$*, is a set of updates* $\{l := v \mid$ *there is a rule in* P*, whose condition holds (under the standard interpretation), with location* l *and value* $v\}$*, or the empty set if the above set includes two values for the same location.*

Coding style. To make flat programs more readable, let

```
% comment
if cond-1
   stat-1
   stat-2
else
   stat-3
```

stand for

```
if cond-1 then stat-1
if cond-1 then stat-2
if not cond-1 then stat-3
```

and, similarly, for other such abbreviations.

Definition 13 (Programmable Procedure). *A* programmable procedure *is composed of:* $\mathcal{F}$, In, Out, $D, \mathcal{S}, \mathcal{S}_0$*, and* P*, where all but the last component is as in a sequential procedure (see Section 2.2), and* P *is a flat program of* $\mathcal{F}$*.*

The run *of a programmable procedure and its* extensionality *are defined as for sequential procedures (Definitions 6 and 7), where the transition function τ is given by $\tau(S) = S' \in \mathcal{S}$ such that $\Delta(S, S') = P(S)$.*

A.2 Sequential Equals Programmable

We show that every programmable procedure is sequential, and every sequential procedure is programmable. This result is derived directly from the main lemma of [10].

Lemma 1. *Every programmable procedure is sequential. That is, let A be a programmable procedure with states $\mathcal{S}$ and a flat program P, then there exists a sequential procedure B with the same elements of A, except for having a transition function τ instead of the program P, such that $\Delta(S, \tau(S)) = P(S)$ for every $S \in \mathcal{S}$.*

Proof. Let $A = \langle \mathcal{F}, \textit{In}, \textit{Out}, D, \mathcal{S}, \mathcal{S}_0, P \rangle$ be an arbitrary programmable procedure. Define the finite set of critical $\mathcal{F}$-terms T to include all terms and subterms of P. Define a transition function $\tau : \mathcal{S} \to \mathcal{S}$ by $\tau(S) = S'$ s.t. $\Delta(S, S') = P(S)$. To show that $B = \langle \mathcal{F}, \textit{In}, \textit{Out}, D, \mathcal{S}, \mathcal{S}_0, \tau \rangle$ is a sequential procedure such that $\Delta(S, \tau_B(S)) = P_A(S)$ for every $S \in \mathcal{S}$ it remains to show that B satisfies the constraints defined for τ in a sequential procedure. Since the flat program P includes only terms in T (and doesn't refer directly to domain elements), it obviously follows that τ satisfies the isomorphism constraint. Since T includes all the terms of P, as well as the subterms of the location-terms of P, it obviously follows that states that coincide over T have the same set of updates by τ. Thus, τ satisfies the bounded-exploration constraint. □

Lemma 2. *Every sequential procedure is programmable. That is, let B be a sequential procedure with states $\mathcal{S}$ and a transition function τ, then there exists a programmable procedure A with the same elements of B, except for having a flat program P instead of τ, such that $\Delta(S, \tau(S)) = P(S)$ for every $S \in \mathcal{S}$.*

This follows directly from Gurevich's analogous proof that for every sequential algorithm there exists an equivalent sequential abstract state machine [10, Lemma 6.11].

A.3 Effective Equals Computable

We prove now that Turing machines are of equivalent computational power to all effective models.

Turing Machines are Effective. First, we show that the class of effective procedures is at least as powerful as Turing machines, as the latter is an effective model.

Proof (of Theorem 1). We consider Turing machines (TM) with two-way infinite tapes. The tape alphabet is $\{0, 1\}$. So domain elements are comprised of an internal machine state and an infinite tape, containing finitely many 0's and 1's, and the rest blank, and a read/write head somewhere along the tape.

As is usual, this Turing machine state (instantaneous description) may be described as a triple $\langle \textit{Left}, q, \textit{Right} \rangle$, where *Left* is a finite string containing the tape section left of

the reading head, q is the internal state of the machine, and *Right* is a finite string with the tape section to the right to the read head. The read head points to the first character of the *Right* string. The two edges of the tape are marked by a special $\$$ sign.

TMs can be viewed as an effective model with the following components:

Domain: The domain consists of all finite strings over $0, 1$ with a suffix of a $\$$ sign. That is the domain $D = \{0, 1\}^*\$$.

Base structure: Constructors for the finite strings ($name/arity$): $\$/0$, $Cons_0/1$, and $Cons_1/1$.

Almost-constant structure:

- Input and Output (nullary functions): *In*, *Out*. The value of *In* at the initial state is the content of the tape, as a string over $\{0, 1\}^*$ ending with a $\$$ sign.
- Constants for the alphabet characters and TM-states (nullary): 0, 1, q_0, q_1, $\ldots$, q_k. Their values are of no importance as long as it is a different value for each of them.
- Variables to keep the current status of the Turing machine (nullary): *Left*, *Right*, and q. Their initial values are: *Left* $= \$$, *Right* $= \$$, and $q = q_0$.
- Functions to examine the tape (unary functions): *Head* and *Tail*. Their initial value, at all locations, is $\$$.

Transition function: By Lemma 1, every programmable procedure is a sequential procedure. Thus, a programmable procedure that satisfies the initial-data postulate is an effective procedure. Every Turing machine $m \in \mathrm{TM}$, is an effective procedure with a flat program looking like this:

```
if q = q_0  % TM's state q_0
   if Head(Right) = 0
      % write 1, move right, switch to q_3
      Left := Cons_1(Left)
      Right := Tail(Right)
      q := q_3
      % Internal operations
      Tail(Cons_1(Left)) := Left
      Head(Cons_1(Left)) := 1
   if Head(Right) = 1
      % write 0, move left, switch to q_1
      Left := Tail(Left)
      Right := Cons_0(Right)
      q := q_1
      % Internal operations
      Tail(Cons_0(Right)) := Right
      Head(Cons_0(Left)) := 0
```

```
if q = q_1   % TM's state q_1
        ...
if q = q_k   % the halting state
   Out := Right
```

The updates for `Head` and `Tail` are bookkeeping operations that are really part of the "behind-the-scenes" small steps.

The procedure also requires some initialization, so as to fill the internal functions *Head* and *Tail* with their values for all strings up to the given input string. It sequentially enumerates all strings, assigning their *Head* and *Tail* values, until encountering the input string. The following internal variables (nullary functions) are used in the initialization (Name = initial value): `New` = \$, `Backward` = 0, `Forward` = 1; `AddDigit` = 0, and `Direction` = \$.

```
% Sequentially constructing the Left variable
% until it equals to the input In, for filling
% the values of Head and Tail.
% The enumeration is $, 0$, 1$, 00$, 01$, ...
if Left = In % Finished
   Right := Left
   Left := $
else % Keep enumerating
   if Direction = New % default val
      if Head(Left) = $ % $ -> 0$
         Left := Cons_0(Left)
         Head(Cons_0(Left)) := 0
         Tail(Cons_0(Left)) := Left
      if Head(Left) = 0 % e.g. 110$ -> 111$
         Left := Cons_1(Tail(Left))
         Head(Cons_1(Tail(Left)) := 1
         Tail(Cons_1(Tail(Left)) := Tail(Left)
      if Head(Left) = 1 % 01$->10$; 11$->000$
         Direction := Backward
         Left := Tail(Left)
         Right := Cons_0(Right)
   if Direction = Backward
      if Head(Left) = $ % add rightmost digit
         Direction := Forward
         AddDigit := True
      if Head(Left) = 0 % change to 1
         Left := Cons_1(Tail(Left))
         Direction := Forward
      if Head(Left) = 1 % keep backwards
         Left := Tail(Left)
         Right := Cons_0(Right)
   if Direction = Forward % Gather right 0s
      if Head(Right) = $ % finished gathering
         Direction := New
         if AddDigit = 1
            Left := Cons_0(Left)
```

```
        Head(Cons_0(Left)) := 0
        Tail(Cons_0(Left)) := Left
        AddDigit = 0
else
        Left := Cons_0(Left)
        Right := Tail(Right)
        Head(Cons_0(Left)) := 0
        Tail(Cons_0(Left)) := Left
```

□

Effective Procedures are Computable. Next, we show that all effective models are equal to or weaker than Turing machines by mapping every effective model to a **while**-like computer program (CP). The computer program may be of any programming language known to be of equivalent power to Turing machines, as long as it includes the syntax and semantics of flat programs.

Lemma 3. *Every base structure S of vocabulary $\mathcal{F}$ over a domain D is isomorphic to a computable structure S' of the same vocabulary over $\mathbf{N}$. That is, there is a bijection $\pi : D \leftrightarrow \mathbf{N}$ such that for every location $f(\overline{a})$ of S we have that $[\![f(\overline{a})]\!]_S = \pi^{-1}([\![f(\pi(\overline{a}))]\!]_{S'})$.*

Proof. Let S be a base structure of vocabulary $\mathcal{F}$ over a domain D. Let $\mathcal{T}$ be the domain of all $\mathcal{F}$-terms, and $\tilde{S}$ the standard free term algebra (structure) of $\mathcal{F}$. Since all structure functions are total, it follows that every $\mathcal{F}$-term has a value in D, and by Proposition 1, every element $e \in D$ is the value of a unique $\mathcal{F}$-term. Therefore, there is bijection $\varphi : D \leftrightarrow \mathcal{T}$, such that $\varphi^{-1}(t) = [\![t]\!]_S$ for every $t \in \mathcal{T}$. Hence, S and $\tilde{S}$ are isomorphic via φ. Since $\mathcal{F}$ is finite, it follows that its set of terms $\mathcal{T}$ is recursive. Define a computable enumeration $\eta : \mathcal{T} \leftrightarrow \mathbf{N}$. Define a structure S' of vocabulary $\mathcal{F}$ over $\mathbf{N}$ by the following computable recursion: $[\![f(n_1, \ldots, n_k)]\!]_{S'} = \eta(f(\eta^{-1}(n_1), \ldots, \eta^{-1}(n_k)))$. That is, for computing the value of a function f on a tuple $\overline{n}$ the program should recursively find the terms of $\overline{n}$, and then compute the enumeration of the combined term. By the construction of S' we have that S' and $\tilde{S}$ are isomorphic via η. Hence, S' and S are isomorphic via $\varphi \circ \eta$. □

Lemma 4. *Computer programs (*CP*) are at least as powerful as any effective model. That is, for every effective model E over domain D there is a bijection $\pi : D \to \mathbf{N}$ such that* CP $\succsim$ *E via π.*

Proof. Let E be an effective model over a domain D with base structure *BS*. By Lemma 3 there is a bijection $\pi : D \leftrightarrow \mathbf{N}$, such that the structure $BS' := \pi(BS)$ is computable. Let P_{BS} be a computer program implementing BS'. For each effective procedure $e \in E$, let AS_e be its almost-constant structure. Since AS_e is almost constant, it follows that $AS'_e := \pi(AS_e)$ is computable; let P_{AS_e} be a computer program implementing AS'_e. By Lemma 2, the transition function of every effective procedure $e \in E$ can be defined by a flat program P_e. For every effective procedure $e \in E$, define a computer program $P'_e = P_e \cup P_{AS_e} \cup P_{BS}$. Since $BS' = \pi(BS)$ and $AS'_e = \pi(AS_e)$ it follows that ext $P'_e = \pi(\text{ext } e)$. Therefore, there is a bijection $\pi : D \leftrightarrow \mathbf{N}$, such that

for every effective procedure $e \in E$ there is a computer program $P'_e \in CP$ such that $\text{ext } P'_e = \pi(\text{ext } e)$. Hence, CP $\succsim E$. $\square$

We are now in position to prove our main theorem, namely that Turing machines (TM) are at least as powerful as any effective model.

Proof (of Theorem 2). We need to show that $\text{TM} \succsim E$ for every model E satisfying the effectivity axioms. We have, by Lemma 4, that computer programs (CP) are at least as powerful as any effective model, while TMs are of equivalent power to computer programs $(\text{TM} \approx \text{CP})$. $\square$