

# The Church-Turing Thesis over Arbitrary Domains<sup>\*</sup>

Udi Boker and Nachum Dershowitz

School of Computer Science, Tel Aviv University  
Ramat Aviv 69978, Israel  
email: [udiboker@tau.ac.il](mailto:udiboker@tau.ac.il),  
[nachum.dershowitz@cs.tau.ac.il](mailto:nachum.dershowitz@cs.tau.ac.il)

**Abstract.** The Church-Turing Thesis has been the subject of many variations and interpretations over the years. Specifically, there are versions that refer only to functions over the natural numbers (as Church and Kleene did), while others refer to functions over arbitrary domains (as Turing did). Our purpose is to analyze and formalize the thesis when referring to functions over arbitrary domains.

First, we must handle the issue of domain representation. We show that, *prima facie*, the thesis is not well defined for arbitrary domains, since the choice of representation of the domain might have a non-trivial influence. We overcome this problem in two steps: (1) phrasing the thesis for entire computational models, rather than for a single function; and (2) proving a “completeness” property of the recursive functions and Turing machines with respect to domain representations. In the second part, we propose an axiomatization of an “effective model of computation” over an arbitrary countable domain. This axiomatization is based on Gurevich’s postulates for sequential algorithms. A proof is provided showing that all models satisfying these axioms, regardless of underlying data structure, are equivalent to, or weaker than, Turing machines.

## 1 Introduction

*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 [22] suggested that his computational model, namely, Turing machines, could compute anything that might be mechanically computable, but his interests extended beyond numeric functions.

Church’s original thesis concerned functions over the natural numbers (with their standard interpretation) [4, p. 346, including fn. 3] (emphasis ours):

The purpose of the present paper is to propose a definition of effective calculability. As will appear, this definition of effective calculability can be stated in either of two equivalent forms, (1) that a *function of positive integers* will

---

<sup>\*</sup> This research was supported by the Israel Science Foundation (grant no. 250/05) and was carried out in partial fulfillment of the requirements for the Ph.D. degree of the first author.

be called effectively calculable if it is  $\lambda$ -definable in the sense of §2 below, (2) that a *function of positive integers* shall be called effectively calculable if it is recursive in the sense of §4 below.

Kleene, when speaking about Church's Thesis, also refers to functions over the natural numbers [10, pp. 58, 60] (emphasis ours):

We entertain various proposition *about natural numbers* . . . This heuristic fact [all recognized effective functions turned out to be general recursive], as well as certain reflections on the nature of symbolic algorithmic processes, led Church to state the following thesis. The same thesis is implicit in Turing's description of computing machines.

THEESIS I. Every effectively calculable function (effectively decidable predicate) is general recursive.

Turing, on the other hand, explicitly extends the notion of "effective" beyond the natural numbers [23, footnote on p. 166] (emphasis added):

We shall use the expression "computable function" to mean a function calculable by a machine, and we let "effectively calculable" refer to the intuitive idea without particular identification with one of these definitions. *We do not restrict the values taken by a computable function to be natural numbers.*

*The Problem.* Let  $f$  be some decision function over an arbitrary countable domain  $D$ . What does one mean by saying that " $f$  is computable"? One most likely means that there is a Turing machine  $M$ , such that  $M$  computes  $f$ , *using some string representation of the domain  $D$* . But what are the allowed string representations? Obviously, allowing any representation (any injection from  $D$  to  $\Sigma^*$ ) is problematic – it will make any decision function "computable". For example, the halting function is isomorphic to the simple parity function. Thus, under a "strange" representation it would become "computable", as we will see in Sect.2.1 below. Another approach is to allow only "natural" representations. However, in the context of defining computability, one is obliged to resort to a vague and undefined notion of "naturalness".

*Our Solution.* Our approach to overcoming the representation problem is to ask about effectiveness of a set of functions over the domain of interest, rather than of a single function. As Myhill observed [14]: undecidability is a property of classes of problems, not of individual problems. The Church-Turing Thesis, interpreted accordingly, asserts that there is no effective computational model that is more inclusive than Turing machines.

Nonetheless, there remains a serious problem. Let  $M$  be a computational model (computing a set of functions) over some countable domain  $D$ . Might it be the case that  $M$  is equal to the Turing-computable functions under some string representation, but strictly contains it under a different representation? Generally speaking, this could indeed be the case when comparing arbitrary computational models. For example, the standard two-counter machine model (2CM) is strictly contained in some models, while it also strictly contains them – all depending on the choice of domain representation.

Fortunately, this cannot be the case with Turing machines (nor with the recursive functions), as we have demonstrated in [3], where we proved that Turing machines are “complete” in the sense that if some model is equivalent to, or weaker than, Turing machines under one representation, then no other representation (no matter how “strange”) can make it stronger than Turing machines. Hence, the Church-Turing Thesis is well-defined for arbitrary computational models.

Due to this completeness of Turing machines, we can also define what it means for a string representation of an arbitrary domain to be “computable”.

*Axiomatization.* Equipped with a plausible interpretation of the Church-Turing Thesis over arbitrary domains, we investigate the general class of “effective computational models”. We proffer an axiomatization of this class, based on Gurevich’s postulates for a sequential algorithm [8]. The thesis is then proved, in the sense that a proof is provided that all models satisfying these axioms are equivalent to, or weaker than, Turing machines.

The specifics of our effectiveness axioms may perhaps be arguable. Nevertheless, it demonstrates the possibility of such an axiomatization of effectiveness *over arbitrary domains*, and provides some evidence for the correctness of the Church-Turing Thesis, regardless of underlying data structure and internal mechanism of the particular computational model.

This direction of research follows Shoenfield’s suggestion [19, 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”.

*Axioms of Effectiveness.* 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 Sects. 3.2–3.3):

1. **Sequential Time.** 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 [11, p. 7] and others.
2. **Abstract State.** 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 logical structures follows the proposal of Gurevich [8, p. 78].

3. **Bounded Exploration.** 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. Informally, this means that it can be described by a finite text that explains the algorithm without presupposing any special knowledge.
4. **Initial Data.** The initial state comprises only finite data in addition to the domain representation. The latter is isomorphic to a Herbrand universe.  
The fourth postulate restricts procedures to be wholly effective by insisting on the effectiveness of the initial data, in addition to the effectiveness of the algorithm.

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 pairs is obviously non-effective. Hence, the equality notion should be the result of some internal mechanism, and thus needs to be a part of the computational model.

*Previous Work.* Usually, the handling of multiple domains in the literature is done by choosing specific representations, like Gödel numbering, Church numerals, unary representation of numbers, etc. This is also true of the usual handling of representations in the context of the Church-Turing Thesis.

Richard Montague [13] raises the problem of representation when applying Turing’s notion of computability to other domains, as well as the circularity in choosing a “computable representation”.

Stewart Shapiro [18] raises the very same problem of representation when applying computability to number-theoretic functions. He suggests a definition of an “acceptable notation” (an acceptable string representation of natural numbers), based on some intuitive concepts. After we overcome the problem of defining the Church-Turing Thesis over arbitrary domains, we suggest, in Section 2.5, a definition of an “effective representation”, resembling Shapiro’s notion of “acceptable notation”.

Michael Rescorla claims in a recent paper [15] that the Church-Turing Thesis has inherent circularity because of the above problem of representing numbers by strings. He is not satisfied with Shapiro’s definition of an acceptable notation, finding it not general enough.

A more general approach for comparing the power of different computational models would be to allow any representation based on an injective mapping between their domains. This is done, for example, by Rogers [16, p. 27], Sommerhalder [21, p. 30], and Cutland [5, p. 24]. Our notion of comparing computational power is very similar to this.

To the best of our knowledge, [2, 3] are the first papers to point out and handle the possible influence of the representation on the extensionality of computational models.

As for the axiomatization of effectiveness, several different approaches have been taken over the years. Turing [22] already formulated some principles 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 [7], and later Sieg and Byrnes [20], define a model whose states are described by hereditarily finite sets. Effectiveness 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) [8] as our starting point. (Some of the problems of incorporating the Gandy model under the abstract state machine rubric are dealt with in [1].)

*Overview.* The first part of this paper, Sect. 2, deals with the issue of domain representation. In Sect. 2.1, we show that checking for the computability of a single function over an arbitrary domain is problematic due to the influence of the domain representation. As a result, we interpret the Church-Turing Thesis for entire computational models. In Sect. 2.2 we define the notion of power comparison between computational models, required for the above interpretation of the thesis. In Sect. 2.3 we show that the representation might generally have an influence even on entire computational models. In Sect. 2.4 we solve the above problem with relation to the Church-Turing Thesis, by proving some “completeness” property of the recursive functions and Turing machines with respect to domain representations. We conclude this part by discussing, in Sect. 2.5, what are “effective representations”.

The second part, Sect. 3, proposes an axiomatization of an “effective model of computation” over an arbitrary countable domain. In Sect. 3.2, we axiomatize “sequential procedures” along the lines of Gurevich’s postulates for a sequential algorithm. In Sect. 3.3, we axiomatize “effective models” on top of sequential procedures by adding a fourth axiom, requiring the effectiveness of the initial state. We then show, in Sect. 3.4, that Turing machines, which constitute an effective model, are at least as powerful as any effective model (Theorem 4). We conclude with a brief discussion. The proofs of this part are given in the Appendix.

*Notation.* When we speak of the recursive functions, denoted *Rec*, we mean the partial recursive functions. Similarly, the set of Turing machines, denoted *TM*, includes both halting and non-halting machines.

## 2 Arbitrary Domains

Simply interpreted, the Church-Turing Thesis is not well defined for arbitrary domains: the choice of domain representation might have a significant influence on the outcome. We explore below the importance of the domain representation and suggest how to overcome this problem.

## 2.1 Computational Model vs. Single Function

A single function over an arbitrary domain cannot be classified as computable or not. Its computability depends on the representation of the domain.<sup>1</sup> For example, as mentioned above, the (uncomputable) halting function over the natural numbers (sans the standard order) is isomorphic to the simple parity function; hence, we may map the natural numbers that normally encode halting Turing machines to strings starting with “0”, and the rest of the numbers as strings starting with “1”, yielding a computable “halting” function.

An analysis of the classes of number-theoretic functions that are computable relative to different notations (representations) is done by Shapiro [18, p. 15]:

It is shown, in particular, that the class of number-theoretic functions which are computable relative to every notation is too narrow, containing only rather trivial functions, and that the class of number-theoretic functions which are computable relative to some notation is too broad containing, for example, every characteristic function.

An intuitive approach is to restrict the representation only to “natural” mappings between the domains. However, when doing so in the scope of defining “effectiveness” one must use a vague and undefined notion.

This problem was already pointed out by Richard Montague on 1960 [13, pp. 430–431]:

Now Turing’s notion of computability applies directly only to functions on and to the set of natural numbers. Even its extension to functions defined on (and with values in) another denumerable set  $S$  cannot be accomplished in a completely unobjectionable way. One would be inclined to choose a one-to-one correspondence between  $S$  and the set of natural numbers, and to call a function  $f$  on  $S$  computable if the function of natural numbers induced by  $f$  under this correspondence is computable in Turing’s sense. But the notion so obtained depends on what correspondence between  $S$  and the set of natural numbers is chosen; the sets of computable functions on  $S$  correlated with two such correspondences will in general differ. The natural procedure is to restrict consideration to those correspondences which are in some sense ‘effective’, and hence to characterize a computable function on  $S$  as a function  $f$  such that, for some effective correspondence between  $S$  and the set of natural numbers, the function induced by  $f$  under this correspondence is computable in Turing’s sense. But the notion of effectiveness remains to be analyzed, and would indeed seem to coincide with computability.

Stewart Shapiro suggests a definition of “acceptable notation”, following some intuitive concepts [18, p. 18]:

This suggests two informal criteria on notations employed by algorithms:

---

<sup>1</sup> There are functions that are inherently incomputable, regardless of the domain representation. For example, a permutation of some countable domain, in which the lengths of the orbits are exactly the standard encodings of the non-halting Turing machines.

- (1) The computist should be able to *write* numbers in the notation. If he has a particular number in mind, he should (in principle) be able to write and identify tokens for the corresponding numeral.
- (2) The computist should be able to *read* the notation. If he is given a token for a numeral, he should (in principle) be able to determine what number it denotes.

It is admitted that these conditions are, at best, vague and perhaps obscure.

Michael Rescorla argues that the circularity is inherent in the Church-Turing Thesis [15]:

My argument turns largely upon the following constraint: a successful conceptual analysis should be *non-circular*. . . . I will suggest that purported conceptual analyses involving Church's thesis generate a subtle yet ineliminable circularity: they characterize the intuitive notion of computability by invoking the intuitive notion of computability. . . . So that syntactic analysis can illuminate the computable number-theoretic functions, we correlate syntactic entities with non-syntactic entities like numbers. We endow the syntax with a primitive semantics. I submit that, in providing this semantics, we must deploy the intuitive notion of computability. Specifically, we must demand that the semantic correlation between syntactic entities and non-syntactic entities itself be computable. But then the proposed analysis does not illuminate computability non-circularly.

A possible solution is to allow any representation (injection between domains), while checking for the effectiveness of an entire computational model, that is, to check for the computability of a function together with the other functions that are computable by that computational model. The reason behind this is to view the domain elements as arbitrary ones, getting all their meaning from the model's functions. For example, it is obvious that the halting function has a meaning only if you know the order of the elements of its domain. In that case, the successor function provides the meaning for the domain elements.

Adopting the above approach of checking for computability of an entire computational model, we interpret the Church-Turing Thesis as follows:

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

For maximum generality, 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. We consider only deterministic computational models. As models may have non-terminating computations, we deal with sets of partial functions. To simplify the development, we assume that the domain and range of functions are identical.

**Definition 1 (Computational Model).**

- A computational model  $A$  over domain  $D$  is any object associated with a set of partial functions  $f : D \rightarrow D$ . This set of functions is called the extensionality of the computational model, denoted  $\text{ext } A$ .

- 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  $f \in \text{ext } A$ ,  $A \subseteq B$  as short for  $\text{ext } A \subseteq \text{ext } B$ , and  $A = B$  as short for  $\text{ext } A = \text{ext } B$ .

To understand this thesis, it remains for us to define what it means to be “equivalent to, or weaker than”. That is, we must define a method by which to compare computational power of computational models.

## 2.2 Comparing Computational Power

Since we are dealing with models that operate over different domains, we adopt the quasi-ordering on extensional power developed in [3]. Basically, we say that model  $A$  is at least as powerful as model  $B$  if there is some representation via which  $A$  contains all the functions of  $B$ . A representation may be any injection between the domains (a generalization to mappings other than injections can be found in [2]). We formalize this comparison notion below.

A computational model is viewed as a set of functions, and its representation over a different domain is just the result of some renaming of the underlying domain elements.

### Definition 2 (Representation).

**Domain.** Let  $D_A$  and  $D_B$  be two domains (arbitrary sets of atomic elements). A representation of  $D_B$  over  $D_A$  is an injection  $\rho : D_B \rightarrow D_A$  (i.e.  $\rho$  is total and one-one). We write  $\text{rng } \rho$  for the actual range of the representation.

**Function.** Representations naturally extend to relations, which are sets of tuples of domain elements:  $\rho(f) := \{\langle \rho(x_1), \dots, \rho(x_n) \rangle \mid \langle x_1, \dots, x_n \rangle \in f\}$ .

**Model.** Representations also naturally extend to computational models, which are sets of functions:  $\rho(B) := \{\rho(f) \mid f \in B\}$ .

Since representations are allowed to be arbitrary injections, they might not cover the target domain. Hence, when we compare a model  $A$  with a representation of some model  $B$ , we should restrict  $A$  to the range of the representation.

### Definition 3 (Restriction).

1. A restriction of a function  $f$  over domain  $D$  to a subdomain  $C \subseteq D$ , denoted  $f \upharpoonright_C$ , is the subset of tuples of  $f$  in which all elements are in  $C$ . That is,  $f \upharpoonright_C := \{\langle x_1, \dots, x_n \rangle \in f \mid x_i \in C, 1 \leq i \leq n\}$
2. The restriction naturally extends to models:  $A \upharpoonright_C := \{f \upharpoonright_C \mid f \in A\}$
3. We usually write  $\rho(f) \in A$ , as a shortcut for  $\rho(f) \in A \upharpoonright_{\text{rng } \rho}$ , meaning that the function  $f$  belongs to the (restriction of) the computational model  $A$  via representation  $\rho$ .

We can now provide the appropriate comparison notion.

### Definition 4 (Computational Power).

- Model  $A$  is (computationally) at least as powerful as model  $B$ , denoted  $A \succeq B$ , if there is a representation  $\rho$  such that  $A|_{\text{rng } \rho} \supseteq \rho(B)$ . In such a case, we also say that model  $A$  simulates model  $B$  via the representation  $\rho$ .
- Model  $A$  is (computationally) more powerful than  $B$ , denoted  $A \succ B$ , if  $A \succeq B$  but  $B \not\succeq A$ .
- Models  $A$  and  $B$  are (computationally) equivalent if  $A \succeq B \succeq A$ , in which case we write  $A \sim B$ .

This is the notion of “implemented” used in [9, p. 52] and of “incorporated” used in [21, p. 29].

**Proposition 1.** *The computational power relation  $\succeq$  between models is a quasi-order. Computational equivalence  $\sim$  is an equivalence relation.*

Turing-computable functions simulate the recursive functions via a unary representation of the natural numbers. The (untyped)  $\lambda$ -calculus ( $\Lambda$ ) is computationally equivalent to the recursive functions (Rec), via Church numerals, on the one hand, and via Gödelization, on the other.

### 2.3 Influence of Representations

It turns out that even when dealing with entire computational models, we are not yet “on safe ground”. The representation of the domain still allows for the possibility that a model be equivalent to one of its strict supermodels. That is, a representation might allow to “enlarge” a model, adding some “new” functions to it.

Consider two-counter machines. It was shown by Rich Schroepel [17], and independently by Frances Yao, that two-counter machines cannot compute the function  $\lambda n.2^n$ . Since two-counter machines can simulate all the recursive functions via some proper injective representation (see, for example, [12]), it follows that two-counter machines can “enlarge” their computational power via some representations.

A reasonable direction might have been to restrict the representation to bijections between domains. However, while it works for this example, it turns out that there are models equivalent to some of their supermodels even with bijective representations [3]. Hence, there are models isomorphic to some of their strict supermodels.

This places a question mark on the definition of Turing-computability and on the meaning of the Church-Turing Thesis. Can it be that the recursive functions are isomorphic to a larger set of functions?! Can we find a string representation of the natural numbers via which we have Turing machines to compute all the recursive functions *plus* some additional functions?

In the next section, we put a firmer ground beneath the definition of the Church-Turing Thesis, by showing that Turing machines, as well as the recursive functions, enjoy a special “completeness” property.

### 2.4 Completeness

As shown above, a model can be of equivalent power to its strict supermodel. There are, however, models that are not susceptible to such an anomaly, denoted “complete” models, among which are Turing machines and the recursive functions.

**Definition 5 (Completeness).** *A model is complete if it is not of equivalent power to any of its strict supermodels. That is,  $A$  is complete if  $A \lesssim B \supseteq A$  implies  $A = B$  for all  $B$ .*

A supermodel of the recursive functions (or Turing machines) is a “hypercomputational” model.

**Definition 6 (Hypercomputational Model).** *Model  $H$  is hypercomputational if it simulates a model that strictly contains the recursive functions. That is, if there is a model  $A$ , such that  $A \supsetneq \text{Rec}$  and  $H \lesssim A$ .*

**Theorem 1 ([3]).** *The recursive functions (Rec) and Turing machines are complete. That is, they cannot simulate any hypercomputational model.*

(The completeness of the recursive functions proved in [3] refers only to unary functions, but it is quite straightforward to extend it to any arity.)

The Church-Turing Thesis, as interpreted in Sect. 2.1, matches the intuitive understanding only due to this completeness of the recursive functions and Turing machines. Were the thesis defined in terms of two-counter machines (2CM), for example, it would make no sense: a computational model is not necessarily stronger than 2CM even if it computes strictly more functions.

## 2.5 Effective Representations

What is an effective representation? We argued above that a “natural representation” must be a vague notion when used in the context of defining effectiveness. We avoided the need of restricting the representation by checking the effectiveness of entire computational models. But what if we adopt the Church-Turing Thesis; can we then define what is an effective (or computable) string representation?

Simply put, there is a problem here. Turing machines operate only over strings. Thus a string representation, which is an injection from some domain  $D$  to  $\Sigma^*$ , is not itself computable by a Turing machine. All the same, when we consider, for example, string representations of natural numbers, we can obviously say regarding some of them that they are effective. How is that possible? The point is that we look at a domain as having some structure. For the natural numbers, we usually assume their standard order. A function over the natural numbers without their order is not really well-defined. As we saw, the halting function and the simple parity function are exactly the same function when numbers are unordered.

Hence, even when adopting the Church-Turing Thesis, a domain without any structure cannot have a computable representation. It is just a set of arbitrary elements. However, if the domain comes with a generating mechanism (as the natural numbers come with the successor) we can consider computable representations.

Due the completeness of the recursive functions and Turing machines, we can define what is a computable string representation of the natural numbers (with their standard structure). A similar definition can be given for other domains, rather than the natural numbers, provided that they come with some finite means of generating them all.

**Definition 7.** A computable representation of the natural numbers by strings is an injection  $\rho : \mathbf{N} \rightarrow \Sigma^*$ , such that  $\rho(s)$  is Turing-computable ( $\rho(s) \in \text{TM}$ ), where  $s$  is the successor function over  $\mathbf{N}$ .

That is, a representation of the natural numbers is computable if the successor function is Turing-computable via this representation.

*Remark 1.* One may also require that the range of the representation  $\rho$  is totally Turing computable, meaning, that the question whether some string is in  $\text{rng } \rho$  is decidable. This is the general approach of [2] with respect to power comparison, declaring that model  $A$  is at least as powerful as model  $B$  if it simulates it via some representation  $\rho$  and also decides the range of  $\rho$ .

We justify the above definition of a computable representation by showing that (a) every recursive function is Turing-computable via any computable representation; (b) every non-recursive function is not Turing-computable via any computable representation; and (c) for every incomputable representation there is a recursive function that is not Turing-computable via it.

**Theorem 2.**

- (a) Let  $f$  be a recursive function and  $\rho : \mathbf{N} \rightarrow \Sigma^*$  a computable representation. Then  $\rho(f) \in \text{TM}$ .
- (b) Let  $g$  be a non-recursive function and  $\rho : \mathbf{N} \rightarrow \Sigma^*$  a computable representation. Then  $\rho(g) \notin \text{TM}$ .
- (c) Let  $\eta : \mathbf{N} \rightarrow \Sigma^*$  be an incomputable representation. Then there is a recursive function  $f$ , such that  $\eta(f) \notin \text{TM}$ .

*Proof.* Let  $\xi : \mathbf{N} \rightarrow \Sigma^*$  be some standard bijective representation via which  $\xi(\text{Rec}) = \text{TM}$  (see, for example, [9, p. 131]). The point is that once  $\rho(s) \in \text{TM}$  there are Turing computable functions for switching between the  $\rho$  and the  $\xi$  representations. That is,  $\rho \circ \xi^{-1}, \xi \circ \rho^{-1} \in \text{TM}$ . It can be done by a Turing machine that enumerates in parallel over both representations until reaching the required string.

(a) Since  $f \in \text{Rec}$ , it follows that there is a function  $f' \in \text{TM}$ , such that  $f = \xi^{-1}(f') = \xi^{-1} \circ f' \circ \xi$ . Thus,  $\rho(f) = \rho \circ f \circ \rho^{-1} = \rho \circ \xi^{-1} \circ f' \circ \xi \circ \rho^{-1}$ . Hence,  $\rho(f) \in \text{TM}$  by the closure of TM under functional composition.

(b) Assume by contradiction that  $g \notin \text{Rec}$  but  $\rho(g) \in \text{TM}$ . Let  $g'$  be the corresponding function under the  $\xi$  representation. That is,  $g' = \xi \circ \rho^{-1} \rho(g) \circ \rho \circ \xi^{-1}$ . We have by the closure of TM under functional composition that  $g' \in \text{TM}$ . Since  $\xi^{-1}(g') \in \text{Rec}$ , it is left to show that  $\xi^{-1}(g') = g$  for getting a contradiction:  $\xi^{-1}(g') = \xi^{-1} \circ g' \circ \xi = \xi^{-1} \circ \xi \circ \rho^{-1} \rho(g) \circ \rho \circ \xi^{-1} \circ \xi = \rho^{-1} \rho(g) \circ \rho = \rho^{-1} \rho \circ g \circ \rho^{-1} \circ \rho = g$ .

(c) By the definition of recursive representation, the successor is such a function.  $\square$

To see the importance of the completeness property for the definition of a computable representation, one can check that an analogous definition cannot be provided with two-counter machines as the yardstick.

Our notion of an effective representation resembles Shapiro's notion of an "acceptable notation". He proposes three necessary "semi-formal" criteria for an acceptable notation [18, p. 19]:

- (1a) If the computist is given a finite collection of distinct objects, then he can (in principle) write and identify tokens for the numeral which denotes the cardinality of the collection.
- (1b) The computist can *count* in the notation. He is able (in principle) to write, in order, tokens for the numerals denoting any finite initial segment of the natural numbers.
- (2a) If the computist is given a token for a numeral  $p$  and a collection of distinct objects, then he can (in principle) determine whether the denotation of  $p$  is smaller than the cardinality of the collection and, if it is, produce a subcollection whose cardinality is the denotation of  $p$ .

Our notion coincides with Shapiro’s second criterion.

### 3 An Axiomatization of Effective Models

Section 2 formalized the Church-Turing Thesis over arbitrary domains. We now provide some evidence for the thesis by validating it against a class of “effective computational models”, axiomatized on top of Gurevich’s postulates for a sequential algorithm [8].

We start by providing the standard definition of mathematical structures, upon which “sequential procedures” are defined. We axiomatize, in Sect. 3.2, “sequential procedures” along the lines of Gurevich’s sequential algorithms [8]. We later axiomatize, in Sect. 3.3, “effective procedures” as their subclass. We then show, in Sect. 3.4, the equivalence of Turing machines to the class of effective models. We conclude this part with a brief discussion.

#### 3.1 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 8 (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  $f(t_1, \dots, t_k)$ , where  $f$  is a function name in  $\mathcal{F}$  of positive arity  $k$  and  $t_1, \dots, t_k$  are terms.
- A structure  $S$  of vocabulary  $\mathcal{F}$  is a domain  $D$  together with interpretations  $\llbracket f \rrbracket_S$  over  $D$  of the function names  $f \in \mathcal{F}$ .
- A location of vocabulary  $\mathcal{F}$  over a domain  $D$  is a pair, denoted  $f(\bar{a})$ , where  $f$  is a  $k$ -ary function name in  $\mathcal{F}$  and  $\bar{a}$  is a  $k$ -tuple of elements of  $D$ . (If  $f$  is a constant, then  $\bar{a}$  is the empty tuple.)
- The value of a location  $f(\bar{a})$  in a structure  $S$ , denoted  $\llbracket f(\bar{a}) \rrbracket_S$ , is the domain element  $\llbracket f \rrbracket_S(\bar{a})$ .

- It is often useful to indicate a location by a (ground) term  $f(t_1, \dots, t_k)$ , standing for  $f(\llbracket t_1 \rrbracket_S, \dots, \llbracket t_k \rrbracket_S)$ .
- Structures  $S$  and  $S'$  with vocabulary  $\mathcal{F}$  over the same domain coincide over a set  $T$  of  $\mathcal{F}$ -terms if  $\llbracket t \rrbracket_S = \llbracket t \rrbracket_{S'}$  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  $\llbracket f(i, j) \rrbracket_S$ .

**Definition 9 (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  $\mathcal{F} = \mathcal{F}' \uplus \mathcal{F}''$ ,  $\llbracket l \rrbracket_S = \llbracket l \rrbracket_{S'}$  for every location  $l$  in  $S'$ , and  $\llbracket l \rrbracket_S = \llbracket l \rrbracket_{S''}$  for every location  $l$  in  $S''$ .

**Definition 10 (Update).** An update of location  $l$  over domain  $D$  is a pair, denoted  $l := v$ , where  $v$  is an element of  $D$ .

**Definition 11 (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  $\{l := v' \mid \llbracket l \rrbracket_S \neq \llbracket l \rrbracket_{S'} = v'\}$ .

**Definition 12 (Structure Mapping).** Let  $S$  be structure of vocabulary  $\mathcal{F}$  over domain  $D$  and  $\rho : D \rightarrow D'$  an injection from  $D$  to domain  $D'$ . A mapping of  $S$  by  $\rho$ , denoted  $\rho(S)$ , is a structure  $S'$  of vocabulary  $\mathcal{F}$  over  $D'$ , such that  $\rho(\llbracket f(\bar{a}) \rrbracket_S) = \llbracket f(\rho(\bar{a})) \rrbracket_{S'}$  for every location  $f(\bar{a})$  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  $S' = \pi(S)$ .

### 3.2 Sequential Procedures

Axiomatization of a “sequential procedure” is very similar to that of Gurevich’s sequential algorithm [8], 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  $\tau : \mathcal{S} \rightarrow \mathcal{S}$  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 ( $S_0 \subseteq \mathcal{S}$ ) 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  $\pi$  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  $\pi$ , then  $\tau(S)$  and  $\tau(S')$  are also isomorphic via  $\pi$ .*
- **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 [8, 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 [8, p. 90].

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

**Definition 13 (Runs).**

1. *A run of procedure  $A$  is a finite or infinite sequence  $S_0 \rightsquigarrow_{\tau} S_1 \rightsquigarrow_{\tau} S_2 \rightsquigarrow_{\tau} \dots$ , where  $S_0$  is an initial state and every  $S_{i+1} = \tau_A(S_i)$ .*
2. *A run  $S_0 \rightsquigarrow_{\tau} S_1 \rightsquigarrow_{\tau} S_2 \rightsquigarrow_{\tau} \dots$  terminates if it is finite or if  $S_i = S_{i+1}$  from some point on.*
3. *The terminating state of a terminating run  $S_0 \rightsquigarrow_{\tau} S_1 \rightsquigarrow_{\tau} S_2 \rightsquigarrow_{\tau} \dots$  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  $S \rightsquigarrow_{\tau}^! S'$ .*

**Definition 14 (Procedure Extensionality).** *Let  $A$  be sequential procedure over domain  $D$ . The extensionality of  $A$ , denoted  $\text{ext } A$ , is the partial function  $f : D \rightarrow D$ , such that  $f(x) = \llbracket \text{Out} \rrbracket_{S'}$  whenever there's a run  $S \rightsquigarrow_{\tau}^! S'$  with  $\llbracket \text{In} \rrbracket_S = x$ , 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. 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.

### 3.3 Effective Models

A sequential procedure may be equipped with any oracle, given as an operation of the initial state. Hence, the extensionality of such a procedure might not be effective. As a result, we are interested only in sequential procedures that use effective oracles. Since we are defining effectiveness, we get an inductive definition, allowing the initial state to include functions that are the extensionality of effective procedures. The starting point must be operations that are very simple and inherently effective. These basic operations must then be finite. We get that the starting point are sequential procedure, in which the initial state has finite data in addition to the domain representation (“base structure”). This constraint is formalized in Axiom 4, below.

Different procedures of the same computational model have some common mechanism. The level of shared configuration between the model’s procedures may vary, but they must obviously share the same domain representation. Hence, we define an “effective model” to be 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 15 (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.

**Definition 16 (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.  $\llbracket t \rrbracket_S = e$ .*

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

**Proposition 2.** *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.

We are now in position to formalize the fourth axiom, requiring the effectiveness of the initial state. It is an inductive definition, allowing any function that can be implemented by a (simpler) effective procedure.

**Axiom 4 (Initial Data)** *The initial state may consist of:*

- A base structure *BS* (The domain representation)
- An almost-constant AS structure (Finite initial data)
- An effective structure *ES* over the base structure *BS* (Effective oracles)

*That is, the initial state  $S_0$  is  $BS \uplus AS \uplus ES \uplus \{In\}$  for some base structure *BS*, an almost-constant structure *AS*, and an effective structure *ES*.*

The axiom 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  $S_0$ .

The effective structure contains finite many functions that are the extensionality of effective procedures over the same domain representation. This allows the procedure to use an algorithm of any abstraction level, as long as we can assure that the underlying oracles are effective.

There are two aspects to the requirement that the domain representation is isomorphic to a Herbrand universe : 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.

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.

### 3.4 Effective Equals Computable

**Theorem 3.** *Turing machines are an effective model.*

**Theorem 4.** *Turing machines (TM) are at least as powerful as any effective model. That is,  $TM \succeq E$  for every model *E* satisfying the effectiveness axioms.*

Proofs of Theorems 3 and 4 are quite straightforward but somewhat lengthy, so are relegated to an appendix.

### 3.5 Discussion

*Necessity.* 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 systematic 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  $\llbracket Out \rrbracket_{S'}$  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 [8], 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 effectiveness: 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.

## A Proofs of Two Theorems

We provide here proofs of Theorems 3 and 4. First, we require some additional definitions and lemmata.

### A.1 Programmable Machines

In Sect. 3.2, 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) [8]. 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 17 (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 ...  $x_{1k_1} \doteq y_{1k_1}$   
then  $l_1 := v_1$

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

⋮

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

where each  $\doteq$  is either ‘=’ or ‘≠’,  $n, k_1, \dots, k_n \in \mathbf{N}$ , 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  $\{l := v \mid \text{there is a rule in } P, \text{ whose condition holds (under the standard interpretation), with location } l \text{ and value } v\}$ , 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 18 (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 Sect. 3.2), 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 13 and 14), where the transition function  $\tau$  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 [8].

**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  $\tau$  instead of the program  $P$ , such that  $\Delta(S, \tau(S)) = P(S)$  for every  $S \in \mathcal{S}$ .*

*Proof.* Let  $A = \langle \mathcal{F}, \text{In}, \text{Out}, D, \mathcal{S}, \mathcal{S}_0, P \rangle$  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  $\tau : \mathcal{S} \rightarrow \mathcal{S}$  by  $\tau(S) = S'$  s.t.  $\Delta(S, S') = P(S)$ . To show that  $B = \langle \mathcal{F}, \text{In}, \text{Out}, D, \mathcal{S}, \mathcal{S}_0, \tau \rangle$  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  $\tau$  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  $\tau$  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  $\tau$ . Thus,  $\tau$  satisfies the bounded-exploration constraint.  $\square$

**Lemma 2.** *Every sequential procedure is programmable. That is, let  $B$  be a sequential procedure with states  $\mathcal{S}$  and a transition function  $\tau$ , then there exists a programmable procedure  $A$  with the same elements of  $B$ , except for having a flat program  $P$  instead of  $\tau$ , 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 [8, 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 3).* 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 usual, this Turing machine state (instantaneous description) may be described as a triple  $\langle \text{Left}, q, \text{Right} \rangle$ , where  $\text{Left}$  is a finite string containing the tape section left of the reading head,  $q$  is the internal state of the machine, and  $\text{Right}$  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  $\text{Right}$  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,  $q\_0$ ,  $q\_1$ , ...,  $q\_k$ . 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  $m \in \text{TM}$ , 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” 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 operates over the natural numbers and includes the syntax and semantics of flat programs.

**Lemma 3.** *Every infinite 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  $f(\bar{a})$  of  $S$  we have that  $\llbracket f(\bar{a}) \rrbracket_S = \pi^{-1}(\llbracket f(\pi(\bar{a})) \rrbracket_{S'})$ .*

*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 2, 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  $\varphi^{-1}(t) = \llbracket t \rrbracket_S$  for every  $t \in \mathcal{T}$ . Hence,  $S$  and  $\tilde{S}$  are isomorphic via  $\varphi$ . 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:  $\llbracket f(n_1, \dots, n_k) \rrbracket_{S'} = \eta(f(\eta^{-1}(n_1), \dots, \eta^{-1}(n_k)))$ . That is, for computing the value of a function  $f$  on a tuple  $\bar{n}$  the program should recursively find the terms of  $\bar{n}$ , and then compute the enumeration of the combined term. By the construction of  $S'$  we have that  $S'$  and  $\tilde{S}$  are isomorphic via  $\eta$ . Hence,  $S'$  and  $S$  are isomorphic via  $\varphi \circ \eta$ .  $\square$

**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 an injection  $\rho : D \rightarrow \mathbf{N}$  such that  $\text{CP} \succeq E$  via  $\rho$ .*

*Proof.* When the effective model  $E$  has a finite base structure, then the computability is obvious due to the finite number of possible procedures. We consider then the infinite case; let  $E$  be an effective model over a domain  $D$  with an infinite 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  $AS'_e$ . Analogously, we have by induction, a computer program  $P_{ES_e}$  implementing the effective structure of  $e$ . 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  $P'_e = P_{BS} \cup P_e \cup P_{AS_e} \cup P_{ES_e}$ . Since  $BS' = \pi(BS)$ ,  $AS'_e = \pi(AS_e)$  and  $ES'_e = \pi(ES_e)$  it follows that  $\text{ext } P'_e = \pi(\text{ext } e)$ . Therefore, there is a bijection  $\pi : D \leftrightarrow \mathbf{N}$ , such that for every effective procedure  $e \in E$  there is a computer program  $P'_e \in \text{CP}$  such that  $\text{ext } P'_e = \pi(\text{ext } e)$ . Hence,  $\text{CP} \succeq E$ .  $\square$

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

*Proof (of Theorem 4).* We need to show that  $\text{TM} \simeq E$  for every model  $E$  satisfying the effectiveness 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 ( $\text{TM} \approx \text{CP}$ ).  $\square$

## 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*, 14(5):633–648, 2006.
4. A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.
5. N. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, Cambridge, 1980.
6. M. Davis. Why Gödel didn't have Church's Thesis. *Information and Control*, 54(1/2):3–24, 1982.
7. 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.
8. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.
9. N. D. Jones. *Computability and Complexity from a Programming Perspective*. The MIT Press, Cambridge, Massachusetts, 1997.
10. S. C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943.
11. D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, 1968.
12. M. L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, N.J., 1967.
13. R. Montague. Towards a general theory of computability. *Synthese*, 12(4):429–438, 1960.
14. J. Myhill. Some philosophical implications of mathematical logic. three classes of ideas. *The Review of Metaphysics*, 6(2):165–198, 1952.
15. M. Rescorla. Church's thesis and the conceptual analysis of computability. *Notre Dame Journal of Formal Logic*, 48(2):253–280, 2007.
16. H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York, 1966.
17. R. Schroeppel. A two counter machine cannot calculate  $2^n$ . Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1972. available at: <ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-257.pdf>.
18. S. Shapiro. Acceptable notation. *Notre Dame Journal of Formal Logic*, 23(1):14–20, 1982.
19. J. R. Shoenfield. *Recursion Theory*, volume 1 of *Lecture Notes In Logic*. Springer-Verlag, Heidelberg, New York, 1991.

20. W. Sieg and J. Byrnes. An abstract model for parallel computations: Gandy's thesis. *The Monist*, 82(1):150–164, 1999.
21. R. Sommerhalder and S. C. van Westrhenen. *The Theory of Computability: Programs, Machines, Effectiveness and Feasibility*. Addison-Wesley, Workingham, England, 1988.
22. A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936–37.
23. A. M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 45:161–228, 1939.