The Church-Turing Thesis over Arbitrary Domains*

Udi Boker and Nachum Dershowitz

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

For Boaz, pillar of a new discipline.

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 intended). Our purpose is to formalize and analyze 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 of equivalent computational power 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 [5, p. 356] proposed that effective computability of numeric functions be identified with Gödel and Herbrand's general recursive functions, or – equivalently, as it turned out [5] – with Church and Kleene's lambda-definable functions of positive integers. Similarly, Turing [31] suggested that his computational model, namely, Turing machines, could compute anything that might be mechanically computable, but his interests extended beyond numeric functions.

* 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.

A. Avron et al. (Eds.): Trakhtenbrot/Festschrift, LNCS 4800, pp. 199–229, 2008.
© Springer-Verlag Berlin Heidelberg 2008

Church's original thesis concerned functions over the natural numbers with their standard interpretation [5, 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 be called effectively calculable if it is λ-definable. . . , (2) that a *function of positive integers* shall be called effectively calculable if it is recursive. . . .

Kleene, when speaking about Church's Thesis, also refers to functions over the natural numbers [13, 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.
>
> THESIS 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 [32, fn. 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*; we may for instance have computable propositional functions.

But for Turing, even numerical calculations operate on their string representation.

Turing's model of computability was instrumental in the wide acceptance of Church's Thesis. As Trakhtenbrot explained [30]:

> This is the way the miracle occurred: the essence of a process that can be carried out by purely mechanical means was understood and incarnated in precise mathematical definitions.

The Problem. Let f be some decision function (a Boolean-valued 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 an arbitrary representation (any injection from D to Σ^*) is problematic – it will make any decision function "computable". For example, by permuting the domain of machine codes, the halting function can morph into the simple parity function, which returns true when the input number is even, representing a halting machine, and false otherwise). Thus, under a "strange" representation the function

becomes eminently "computable" (see Sect. 2.1). Another approach is to allow only "natural" or "effective" representations. However, in the context of defining computability, one is obliged to resort to a vague and undefined notion of "naturalness" or of "effectiveness", thereby defeating the very purpose of characterizing computability.

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 [19], undecidability is a property of *classes* of problems, not of individual problems. In this sense, the halting function is undecidable in conjunction with an interpreter (universal machine) for Turing machine programs that uses the same representation. 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 potentially serious problem. Let M be a computational model (computing a set of functions) over some countable domain D. Might it be the case that the set of functions that M computes is equal to the Turing-computable functions under one 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 [4], 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 sensibly define what it means for a string representation of an arbitrary domain to be "effective".

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 Yuri Gurevich's postulates for a sequential algorithm [11]. The thesis is then proved, in the sense that a proof is provided that all models satisfying these axioms are of equivalent power to, or weaker than, Turing machines.

Gurevich's postulates are a natural starting point for computing over arbitrary domains. They are applicable for computations over any mathematical structure and aim to capture any sequential algorithm. Nevertheless, while the computation steps are guaranteed to be algorithmic, that is, effective, the initial states are not. In addition, the postulates refer to a single algorithm, while effectiveness should consider, as explained above, the whole computational model. We address the effectiveness of the initial state by adding a fourth axiom

to the three of Gurevich. The effectiveness of an entire computational model is addressed by providing a minimal criterion for two sequential algorithms to be in the same model.

This direction of research follows Shoenfield's suggestion [25, 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 [7]) 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".

Thanks to Gurevich's Abstract State Machine Theorem, showing that sequential abstract state machines (ASMs) capture all (ordinary, sequential) algorithms (those algorithms that satisfy the three Abstract State Machine postulates), we get a third definition of an effective computational model: A model that consists of ASMs that share initial states satisfying the initial-state axiom.

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

The relationship of the three approaches to characterizing effectiveness over arbitrary domains is summarized in Sect. 3.4 and depicted in Fig. 1.

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 ("base structure"). 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, specified initial states, 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] and others. Time is discrete; transitions are deterministic; transfinite sequences are not relevant.
2. **Abstract State.** *Its states are (first-order) structures sharing 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 [11, p. 78]. This is meant to be fully general, allowing states to contain all salient features.

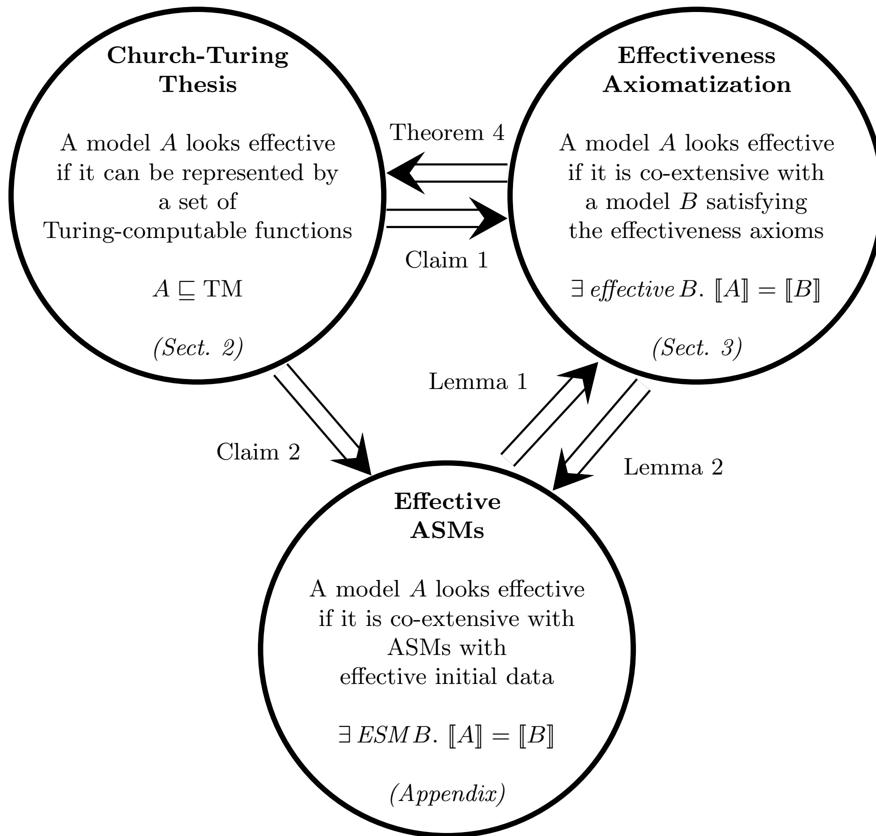

Fig. 1. Equivalent characterizations of an *extensional effectiveness* of a computational model over an arbitrary domain

3. **Bounded Exploration.** *There is a finite bound on the number of vocabulary-terms that affect the transition function.*
   This 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 could not 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, one that can be built up from scratch.

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 [18] 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 [24] 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. We discuss Shapiro's notion in Sects. 2.1 and 2.5.

Klaus Weihrauch [33,34] deals heavily with the representation of arbitrary domains by numbers and strings. He defines computability with respect to a representation, and provides justifications for the effectiveness of the standard representations. We elaborate on his justifications in Sect. 2.5.

After overcoming the problem of defining the Church-Turing Thesis over arbitrary domains, we suggest, in Sect. 2.5, a definition of an "effective representation", resembling Shapiro's notion of "acceptable notation" and along the lines of Weihrauch's justifications for the effectiveness of the standard representations.

Michael Rescorla claims in a recent paper [21] 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 insufficiently general.

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 [22, p. 27], Sommerhalder [29, p. 30], and Cutland [6, p. 24]. A similar approach is used for defining the effectiveness of an algebraic structure by Froehlich and Shepherdson [9], Rabin [20], and Mal'cev [16]. Our notion of comparing computational power is very similar to this.

To the best of our knowledge, our work in [2,4] was the first 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 [31] 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.

Robin Gandy [10], and later Sieg and Byrnes [28], 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.[1]

In [3], we proposed effectiveness axioms, but gave no proof that the axioms yield the same definition of effectiveness as does the Church-Turing Thesis. 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.

In [8], Gurevich and the second author provide an axiomatization of Church's Thesis based on the Abstract State Machine Thesis. They handle only numeric functions, ignoring the issue of effective computation over arbitrary domains, but allowing the use of domains richer than just the numbers.

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 taking advantage of a "completeness" property enjoyed by the recursive functions and Turing machines with respect to domain representations. We conclude this part by discussing, in Sect. 2.5, what are in fact "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. We conclude with a brief discussion. The proofs of this part are given in the Appendix.

We employ Gurevich's most general "Abstract State Machines" (ASMs) [11] as our programming paradigm.[2] Gurevich's ASM Theorem [11] shows that (sequential) ASMs capture (sequential) algorithms, the latter defined axiomatically. As a result, we get a third definition of an effective computational model over an arbitrary domain, namely programmable as an ASM satisfying the extra initial state axiom. See Sect. 3.4 and Fig. 1.

Terminology. When we speak of the recursive functions, denoted $\mathbb{REC}$, we mean the partial recursive functions. Similarly, the set of Turing machines, denoted TM, includes both halting and non-halting machines; we use $\mathbb{TM}$ for the set of

[1] The explicit bound on rank is removed in Sieg's more recent work [26,27].

[2] Some of the problems of incorporating the Gandy model under the abstract state machine rubric are dealt with in [1].

string functions computed by TMs. We use the term "domain" of a computational model and of a (partial) function to denote the set of elements over which it operates, not only those for which it is defined. By "image", we mean the values that a function actually takes: $\text{Im } f := \{f(x) \mid x \in \text{Dom } f\}$.

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 Versus 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.[3] For example, as mentioned above, the (uncomputable) halting function over the natural numbers (sans the standard order) is isomorphic to the simple parity function, under a permutation of the natural numbers that maps the usual codes of halting Turing machines to strings ending in "0", and the rest of the numbers to strings ending with "1". The result is a computable standalone "halting" function.

An analysis of the classes of number-theoretic functions that are computable relative to different notations (representations) is provided by Shapiro [24, 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 [18, 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

[3] There are functions that are inherently uncomputable, 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.

> 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", based on several intuitive concepts [24, p. 18]:

> This suggests two informal criteria on notations employed by algorithms:
> (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 [21]:

> 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 purpose lying behind this idea is to view the domain elements as arbitrary objects, deriving all their meaning from the model's functions. For example, it is obvious that the halting function has a meaning only if one knows the order of the elements of its domain. In that case, the successor function provides the meaning for the domain elements.

A variation of this solution is to allow any representation (injection between domains), provided that the image of the injection is computable. We consider both variations.

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

> **Thesis A.** *All "effective" computational models are of equivalent power to, or weaker than, Turing machines.*

By "effective", in quotes, we mean effective in its intuitive sense.

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.

For maximum generality, we do not want to limit computational models to any specific mechanism; hence, we allow a model to be any object, as long as it is associated with the set of functions that it implements. We consider only deterministic computations, as originally envisioned in Hilbert's program (see [8]). As models may have non-terminating computations, we deal with sets of partial functions. For convenience, we assume that the domain and range (co-domain) of functions are identical.

Definition 1 (Computational Model)

- *A* computational model B *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* $[\![B]\!]$.
- *We write* Dom B *for the domain over which model* B *operates.*

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 [2,4]. 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]). A variation of the above requires that the "stronger" model would also be able to compute the image of the representation. We formalize the comparison notion below.

A computational model is associated with a set of functions (see Definition 1), 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* $\rho : D_B \to D_A$ *(i.e.* ρ *is total and one-one). We write* $\operatorname{Im} \rho$ *for the image of the representation (the values in* D_B *that* ρ *takes).*

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

Model. *Representations also naturally extend to (the extensionalities of) 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 image of the representation.

Definition 3 (Restriction)

1. *A restriction of a function* f *over domain* D *to a subdomain* $C \subseteq D$*, denoted* $f\restriction_C$*, is the subset of tuples of* f *in which all elements are in* C*. That is,* $f\restriction_C := f \cap C^{n+1}$*, for* f *of arity* n.
2. *We write* $\rho(f) \in [\![A]\!]$ *as shorthand for* $\exists g \in [\![A]\!].\, \rho(f) = g\restriction_{\operatorname{Im} \rho}$*, meaning that the function* f *belongs to the (restriction of) the computational model* A *via representation* ρ.

We can now provide the appropriate comparison notion.

Definition 4 (Computational Power)

- *Model* A *is* (computationally) at least as powerful *as model* B*, denoted* $A \succsim B$*, if there is a representation* ρ *such that* $\rho(B) \subseteq \{f\restriction_{\operatorname{Im} \rho} \mid f \in [\![A]\!]\}$*. In such a case, we also say that model* A simulates *model* B *(via representation* ρ*).*
- *Models* A *and* B *are* (computationally) equivalent *if* $A \succsim B \succsim A$.

This is the notion of "implemented" used in [12, p. 52] and of "incorporated" used in [29, p. 29].

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

Turing-computable functions simulate the recursive functions via a unary representation of the natural numbers. The (untyped) λ-calculus (Λ) is computationally equivalent to the recursive functions ($\mathbb{REC}$), via Church numerals, on the one hand, and via Gödelization, on the other.

One may reasonably require that for a model A to be at least as powerful as a model B it should also be able to compute the image of the representation (see [2]). In such a case we get the following variation of the power comparison notion:

Definition 5 (Representational Power)

- *Model A is* (representationally) at least as powerful *as model B, denoted $A \sqsupseteq B$, if there is a representation ρ such that $\rho(B) \subseteq \{f\restriction_{\mathrm{Im}\,\rho} \mid f \in [\![A]\!]\}$ and there is a total function $f \in [\![A]\!]$, such that* $\mathrm{Im}\, f = \mathrm{Im}\, \rho$.
- *Models A and B are* (representationally) equivalent *if $A \sqsupseteq B \sqsupseteq A$.*

In what follows, we use both computational comparisons ($\succsim$) and representational comparisons ($\sqsupseteq$), preferring the more general one whenever possible.[4]

Our interpretation of the Church-Turing Thesis (Thesis A) agrees with Rabin's definition of a computable group [20, p. 343]:

> DEFINITION 3. An *indexing* of a set S is a one to one mapping $i : S \to I$ such that $i(S)$ is a recursive subset of I. ...
> DEFINITION 4. An indexing i of a group G is *admissible* if the function m from $i(G) \times i(G)$ into $i(G)$ ... is a computable function. ...
> DEFINITION 5. A group is *computable* if it possesses at least one *admissible* indexing.

Rabin defines computability for groups, fields, and rings; however, the idea naturally generalizes to any algebraic structure, as done by Lambert [15, p. 594]:

> Following Rabin ... we let an (*admissible*) *indexing* for structure $\mathfrak{U}$ be a 1-1 function $\kappa : A \to \omega$ such that
> (i) $K =$ range κ is recursive;
> (ii) each $\kappa^*(F_a)$ and $\kappa^*(R_a)$ are recursive relative to K ..., where κ^* applied to an (unmixed) operation or relation in A is the operation or relation in ω naturally induced by κ....
> $\mathfrak{U}$ is *computable* iff there is an indexing for $\mathfrak{U}$.

Similar notions were also presented by Froehlich and Shepherdson [9] and Mal'cev [16].

2.3 Influence of Representations

It turns out that even when dealing with entire computational models, we are not yet on terra firma. 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.

---

[4] Specifically, the "completeness" property (Definition 6) is defined using $\succsim$, which makes it stronger. Accordingly, the theorem that Turing machines and recursive functions are complete (Theorem 1) applies also to the analogous case with $\sqsupseteq$ instead. Likewise, when we show that Turing machines are at least as powerful as any effective model (Theorem 4, per Definition 17), we use the $\sqsupseteq$ notion, which provides a stronger result. Thus, the theorem applies also to the analogue $\succsim$. On the other hand, Claims 1 and 2 are stated with respect to $\sqsupseteq$, and do not necessarily hold for $\succsim$.

Consider two-counter machines. It is known that two-counter machines cannot compute the function $\lambda x.2^x$.[5] On the other hand, since two-counter machines can simulate all the recursive functions via some proper injective representation (viz. $n \mapsto 2^n$; see, for example, [17]), 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 [4]. 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 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 seen above, a model can be of equivalent power to its strict supermodel. There are, however, models that are not susceptible to such an anomaly; these are referred to as "complete" models, among which are Turing machines and the recursive functions.

Definition 6 (Completeness). *A model is* complete *if it is not of equivalent power to any of its strict supermodels. That is, A is complete if $A \gtrsim B$ and $[\![B]\!] \supseteq [\![A]\!]$ imply that $[\![A]\!] = [\![B]\!]$ for any B.*

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

Definition 7 (Hypercomputational Model). *A model H is* hypercomputational *if it simulates a model that strictly contains the recursive functions.*

Theorem 1 ([4]). *The recursive functions and Turing machines are complete. They cannot simulate any hypercomputational model.*

(The completeness of the recursive functions proved in [4] refers only to unary functions, but it is quite straightforward to extend it to any arity.)

Note that the completeness property is defined with computational comparison $\gtrsim$, which makes it a stronger property. Accordingly, Turing machines and the recursive functions are also complete with respect to representational comparison $\sqsupseteq$.

[5] This was shown by Rich Schroeppel in [23], and independently by Frances Yao and others.

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.[6]

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 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 Σ^*, 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 (isomorphic) function when numbers are unordered.

Hence, even when adopting the Church-Turing Thesis, a domain without any structure cannot have an effective 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 effective representations.

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

Definition 8. *An* effective representation *of the natural numbers by strings is an injection $\rho : \mathbf{N} \rightarrow \Sigma^*$, such that $\rho(s)$ is Turing-computable ($\rho(s) \in \mathbb{TM}$), where s is the successor function over $\mathbf{N}$.*

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

Remark 1. One may also require that the image of the representation ρ is totally Turing computable, meaning, that the question whether some string is in Im ρ is decidable.

[6] In fact, the lambda calculus also suffers from incompleteness in this sense, and would, therefore, not be a suitable candidate in terms of which to characterize generic effectivity.

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

Theorem 2

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

Proof Let $\xi : \mathbf{N} \to \Sigma^*$ be some standard bijective representation via which $\xi(\mathbb{REC}) = \mathbb{TM}$ (see, for example, [12, p. 131]). The point is that, once $\rho(s) \in \mathbb{TM}$, there are Turing-computable functions for switching between the ρ and the ξ representations. That is, $\rho \circ \xi^{-1}, \xi \circ \rho^{-1} \in \mathbb{TM}$. It can be done by a Turing machine that enumerates in parallel over both representations until reaching the required string.
(a) Since $f \in \mathbb{REC}$, it follows that there is a function $f' \in \mathbb{TM}$, 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, $\rho(f) \in \mathbb{TM}$ by the closure of TM under functional composition.
(b) Assume by contradiction that $g \notin \mathbb{REC}$ but $\rho(g) \in \mathbb{TM}$. Let g' be the corresponding function under the ξ 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 $g' \in \mathbb{TM}$. Since $\xi^{-1}(g') \in \mathbb{REC}$, 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. □

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

Our definition of an effective representation resembles Shapiro's notion of an "acceptable notation". He proposes three necessary "semi-formal" criteria for an acceptable notation [24, p. 19]:

(la) 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.
(lb) 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 (1b). It also goes along with Weihrauch's justifications for the effectiveness of the standard "numberings" (representation by natural numbers). He defines a standard numbering of a word set (the words over $\{a, b\}$, for example, are enumerated in the following order: $\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots$), and then proves three claims for justifying the effectiveness of the numbering [33, p. 80–81]:

> A numbering $\nu : \mathbf{N} \to W(\Sigma)$ is neither a word function nor a number function, hence neither of our two definitions of computability is applicable to ν. Nevertheless standard numberings $\nu : \mathbf{N} \to W(\Sigma)$ are intuitively effective. The following lemma expresses several effectivity properties of standard numberings of word sets.
>
> LEMMA (*effectivity of standard numberings of word sets*)
> Let Σ, Γ and Δ be alphabets with $\Delta = \Sigma \cup \Gamma$. Let ν_Σ (ν_Γ) be a standard numbering of $W(\Sigma)$ ($W(\Gamma)$).
>
> (1) Define $S, V : \mathbf{N} \to \mathbf{N}$ by $S(x) := x + 1$, $V(x) := x \dot{-} 1$.
> Define $S_\Sigma, V_\Sigma : W(\Sigma) \to W(\Sigma)$ by
> $S_\Sigma := \nu_\Sigma S \nu_\Sigma^{-1}$, $V_\Sigma := \nu_\Sigma V \nu_\Sigma^{-1}$.
> Then S_Σ and V_Σ are computable.
>
> (2) Let $b \in \Sigma$. Define $h^b : W(\Sigma) \to W(\Sigma)$, $S^b : W(\Sigma) \to \{1, 2\}$ and $pop : W(\Sigma) \to W(\Sigma)$ by
> $h^b(w) = wb$, $S^b(w) := (1$ if $w = xb$ for some $x \in W(\Sigma), 2$ otherwise$)$,
> and $pop(\varepsilon) := \varepsilon$, $pop(wc) := w$.
> Define $h^b_\Sigma : \mathbf{N} \to \mathbf{N}$, $S^b_\Sigma : \mathbf{N} \to \{1, 2\}$ and $pop_\Sigma : \mathbf{N} \to \mathbf{N}$ by
> $h^b_\Sigma := \nu_\Sigma^{-1} h^b \nu_\Sigma$, $pop_\Sigma := \nu_\Sigma^{-1} pop\, \nu_\Sigma$, $S^b_\Sigma := S^b \nu_\Sigma$.
> Then h^b_Σ, pop_Σ, and S^b_Σ are computable.
>
> (3) The following functions $p : W(\Delta) \to W(\Delta)$ and $q : \mathbf{N} \to \mathbf{N}$ are computable:
> $p(w) := (\nu_\Sigma \nu_\Gamma^{-1}(w)$ if $w \in W(\Gamma), \varepsilon$ otherwise$)$,
> $q(j) := (\nu_\Gamma^{-1} \nu_\Sigma(j)$ if $\nu_\Sigma(j) \in W(\Gamma), 0$ otherwise$)$.

Our notion resembles Weihrauch's first and second claims (the second claim concerns the construction of strings, and plays the rôle of the successor when reversing the "numbering" for representing numbers by strings).

3 An Axiomatization of Effective Models

Section 2 formalized the Church-Turing Thesis over arbitrary domains. We now provide additional 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 [11].

Gurevich's postulates are applicable for computations over any mathematical structure (of first order) and aim to capture any sequential algorithm. This makes them a natural candidate for axiomatizing effectiveness over arbitrary domains. Yet, there are several problems:

1. The postulates concern algorithms and not computations with input and output.
2. Initial states are not limited; thus, they might not be effective.
3. The postulates consider a single algorithm and not an entire computational model.

We address the first issue, in Sect. 3.2, by adding special input and output constants, and allowing a single initial state, up to differences in input. The second issue is addressed by adding Axiom 4, which limits the initial data. The third issue is addressed, in Sect. 3.3, by requiring all functions of the same model to share the same domain representation.

A proof is provided in the Appendix, showing that this axiomatization yields the same definition of effectiveness as the Church-Turing Thesis does. It is based on Gurevich's Abstract State Machine Theorem [11], showing that sequential abstract state machines (ASMs) capture sequential algorithm. As a result, we get three equivalent definitions of an effective computational model over an arbitrary domain. See Fig. 1.

We start in Sect. 3.2, with an axiomatization of "sequential procedures", along the lines of Gurevich's sequential algorithms [11]. Next, we axiomatize, in Sect. 3.3, "effective procedures" as a subclass, satifying an "effectivity axiom". 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 all its relevant features. 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 9 (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 $\mathcal{F}$ sharing 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 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 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 $S' = \pi(S)$.

3.2 Sequential Procedures

Our axiomatization of a "sequential procedure" is very similar to that of Gurevich's sequential algorithm [11], 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*".
- Initial states are identical, except for 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 ($\mathcal{S}_0 \subseteq \mathcal{S}$) share a* domain *$D$, and are equal up to changes in the value of* In*. (For convenience, the initial states can be referred to, collectively, as 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 $S, S' \in \mathcal{S}$, where $\Delta(S, S') = \{l := v' \mid [\![l]\!]_S \neq [\![l]\!]_{S'} = v'\}$ is a set of updates turning S into S'.*

The isomorphism constraints reflects the fact that we are working at a fixed level of abstraction. See [11, 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. (Should it be necessary, one could always combine many domains into one.) 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 [11, p. 90].

Definition 12 (Runs)

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

Definition 13 (Procedure Extensionality). *Let P be sequential procedure over domain D. The* extensionality *of P, denoted $[\![P]\!]$, is the partial function $f : D \to D$, such that $f(x) = [\![Out]\!]_{S'}$ whenever there's a run $S \rightsquigarrow^!_\tau S'$ 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 values (True, False) or connectives ($\neg, \wedge, \vee$) 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, as it compises infinite data. Nevertheless, since every domain element has a unique construction, it follows that an effective procedure may implement the equality notion with only finite initial data. A detailed description of this implementation is given in Sect. A.3.

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 initial states 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 begin, then, with sequential procedures, 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 "base structure".

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

Definition 14 (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 15 (Base Structure). *A structure S of finite vocabulary $\mathcal{F}$ over a domain D is a* base structure *if every domain element is the value of a unique $\mathcal{F}$-term. That is, for every element $e \in D$ there exists a unique $\mathcal{F}$-term t such that $[\![t]\!]_S = e$.*

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.

Definition 16 (Structure Union). *Let S' and S'' be two structures with domain D and with vocabularies $\mathcal{F}'$ and $\mathcal{F}''$, respectively. A structure S over D is the* union *of S' and S'', denoted $S = S' \uplus S''$, if its vocabulary is the disjoint union $\mathcal{F} = \mathcal{F}' \uplus \mathcal{F}''$, and if $[\![l]\!]_S = [\![l]\!]_{S'}$ for locations l in S' and $[\![l]\!]_S = [\![l]\!]_{S''}$ for locations in S''.*

Axiom 4 (Initial Data). *The initial state consists of:*

- *a fixed base structure BS (the domain representation);*
- *a fixed almost-constant AS structure (finite initial data); and*
- *a fixed effective structure ES over the base structure BS (effective oracles);*

in addition to an input value In *over BS that varies from initial state to initial state. That is, the initial state S_0 is the union $BS \uplus AS \uplus ES \uplus \{\mathrm{In}\}$, for some base structure BS, almost-constant structure AS, and effective structure ES.*

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 at any abstraction level, as long as we can assure that the underlying oracles are effective.

As already mentioned, there are two aspects to the requirement that the domain representation be isomorphic to a Herbrand universe: every domain element has a name, and names are unique. Were one 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 not effective. Hence, the equality notion should be the product of some internal effective mechanism, and thus needs to be a part of the computational model.

An *effective procedure* must satisfy Axioms 1–4.

Definition 17 (Effective Model). *An* effective model *is a set of effective procedures (objects satisfying Axioms 1–4) that share the same base structure.*

To sum up:

> **Thesis B.** *All "effective" computational models are effective models (per Definition 17).*

3.4 Effective Equals Computable

In the sense of our above definition of effectiveness (Definition 17) we have that:

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

Furthermore,

Theorem 4. *Turing machines are representationally at least as powerful as any effective model.*

That is, TM $\sqsupseteq E$ for every model E satisfying the effectiveness axioms.

Note that we use representational comparison $\sqsupseteq$, which provides a stronger result. Accordingly, Turing machines are also computationally at least as powerful ($\succsim$) as any effective model.

The proofs of Theorems 3 and 4 are quite straightforward but somewhat lengthy, so are relegated to the appendix. They make usage of Abstract State Machines, which operate over arbitrary domains, and are based on Gurevich's Abstract State Machine Theorem [11], showing that sequential abstract state machines (ASMs) capture sequential algorithms, defined axiomatically.

Definition 18 (Effective State Model). *An ASM model satisfying the initial data restrictions is called an* Effective State Model *(or* ESM*).*

This suggests the following variant thesis:

> **Thesis C.** *Every "effective" computational model is behaviorally equivalent to an ESM.*

If we adopt the variation of the comparison notion that requires the "stronger" model to be able to compute the image of the representation (Definition 5), we get a closer relationship between the three definitions of effectiveness (Theses A–C): When considering only the extensionality of computational models (that is, the set of functions that they compute) we have that the three effectiveness criteria (Theses A–C) are equivalent.

Definition 19 (Effective Looks). *A model A* looks effective *if the set of functions that it computes may be represented by Turing-computable functions. That is, if $A \sqsubseteq$ TM.*

Claim 1. *A model A looks effective if and only if there exists an effective model B, such that $[\![A]\!] = [\![B]\!]$.*

Thanks to Gurevich's Abstract State Machines Theorem [11], we have the analogous claim with respect to ASMs:

Claim 2. *A model A looks effective if and only if there is an ESM B, such that $[\![A]\!] = [\![B]\!]$.*

Claims 1 and 2 are not proved herein. (Their proofs are based on the proofs of Theorems 3 and 4, as well as aspects of the proof of Theorem 2.)

The resulting relationship between the different characterizations of effectiveness is depicted in Fig. 1.

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 members of a recursively-enumerable set.
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', such that $\tau(S) = S'$ 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 versus Model. In [11], 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 itself 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 and known effective operations 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.

Acknowledgement

The second author thanks Félix Costa for his gracious hospitality and for substantive comments on a draft of this work.

References

1. Blass, A., Gurevich, Y.: Background, Reserve, and Gandy Machines. In: Clote, P.G., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, Springer, Heidelberg (2000)

2. Boker, U., Dershowitz, N.: How to Compare the Power of Computational Models. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) CiE 2005. LNCS, vol. 3526, pp. 54–64. Springer, Heidelberg (2005)
3. Boker, U., Dershowitz, N.: Abstract effective models. In: Fernández, M., Mackie, I. (eds.) New Developments in Computational Models: Proceedings of the First International Workshop on Developments in Computational Models (DCM 2005), Lisbon, Portugal (July 2005), Electronic Notes in Theoretical Computer Science, 135(3), 15–23 (2006)
4. Boker, U., Dershowitz, N.: Comparing computational power. Logic Journal of the IGPL 14(5), 633–648 (2006)
5. Church, A.: An unsolvable problem of elementary number theory. American Journal of Mathematics 58, 345–363 (1936)
6. Cutland, N.: Computability: An Introduction to Recursive Function Theory. Cambridge University Press, Cambridge (1980)
7. Davis, M.: Why Gödel didn't have Church's Thesis. Information and Control 54(1/2), 3–24 (1982)
8. Dershowitz, N., Gurevich, Y.: A natural axiomatization of Church's Thesis. Bulletin of the ASL (to appear), available as Technical report MSR-TR-2007-85, Microsoft Research, Redmond, WA (July 2007)
9. Froehlich, A., Shepherdson, J.: Effective procedures in field theory. Philosophical Transactions of the Royal Society of London 248, 407–432 (1956)
10. Gandy, R.: Church's thesis and principles for mechanisms. In: Barwise, J., et al. (eds.) The Kleene Symposium. Studies in Logic and The Foundations of Mathematics, vol. 101, pp. 123–148. North-Holland, Amsterdam (1980)
11. Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. ACM Transactions on Computational Logic 1, 77–111 (2000)
12. Jones, N.D.: Computability and Complexity from a Programming Perspective. The MIT Press, Cambridge, MA (1997)
13. Kleene, S.C.: Recursive predicates and quantifiers. Transactions of the American Mathematical Society 53(1), 41–73 (1943)
14. Knuth, D.E.: The Art of Computer Programming. Fundamental Algorithms, vol. 1. Addison-Wesley, Reading, MA (1968)
15. Lambert Jr., W.M.: A notion of effectiveness in arbitrary structures. The Journal of Symbolic Logic 33(4), 577–602 (1968)
16. Mal'cev, A.: Constructive algebras I. Russian Mathematical Surveys 16, 77–129 (1961)
17. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, NJ (1967)
18. Montague, R.: Towards a general theory of computability. Synthese 12(4), 429–438 (1960)
19. Myhill, J.: Some philosophical implications of mathematical logic. Three classes of ideas 6(2), 165–198 (1952)
20. Rabin, M.O.: Computable algebra, general theory and theory of computable fields. Transactions of the American Mathematical Society 95(2), 341–360 (1960)
21. Rescorla, M.: Church's thesis and the conceptual analysis of computability. Notre Dame Journal of Formal Logic 48(2), 253–280 (2007)
22. Rogers Jr., H.: Theory of Recursive Functions and Effective Computability. McGraw-Hill, New York (1966)

23. Schroeppel, R.: A two counter machine cannot calculate 2^N. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory (1972) (viewed November 28, 2007), `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-257.pdf`
24. Shapiro, S.: Acceptable notation. Notre Dame Journal of Formal Logic 23(1), 14–20 (1982)
25. Shoenfield, J.R.: Recursion Theory. Lecture Notes in Logic, vol. 1. Springer, Heidelberg (1991)
26. Sieg, W.: Church without dogma—Axioms for computability. In: Löwe, B., Sorbi, A., Cooper, S.B. (eds.) New Computational Paradigms: Changing Conceptions of What is Computable, pp. 18–44. Springer, Heidelberg (2007)
27. Sieg, W.: Computability: Emergence and analysis of a mathematical notion. In: Irvine, A. (ed.) Handbook of the Philosophy of Mathematics (to appear)
28. Sieg, W., Byrnes, J.: An abstract model for parallel computations: Gandy's thesis. The Monist 82(1), 150–164 (1999)
29. Sommerhalder, R., van Westrhenen, S.C.: The Theory of Computability: Programs, Machines, Effectiveness and Feasibility. Addison-Wesley, Workingham, England (1988)
30. Trakhtenbrot, B.A.: Comparing the Church and Turing approaches: Two prophetical messages. In: Herken, R. (ed.) The Universal Turing Machine: A half-century survey, pp. 603–630. Oxford University Press, Oxford (1988)
31. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42, 230–265 (1936), Corrections in vol. 43, pp. 544–546 (1937), Reprinted in Davis, M. (ed.), The Undecidable, Raven Press, Hewlett, NY (1965)
32. Turing, A.M.: Systems of logic based on ordinals. Proceedings of the London Mathematical Society 45, 161–228 (1939)
33. Weihrauch, K.: Computability. EATCS Monographs on Theoretical Computer Science, vol. 9. Springer, Berlin (1987)
34. Weihrauch, K. (ed.): Computable Analysis – An introduction. Springer, Berlin (2000)

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) [11]. We then show that sequential procedures and programmable 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 20 (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 21 (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 12 and 13), 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 (satisfying the three axioms), and every sequential procedure is programmable. This result is derived directly from the main lemma of [11].

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 $A = \langle \mathcal{F}, \mathit{In}, \mathit{Out}, D, \mathcal{S}, \mathcal{S}_0, P\rangle$ 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 $\tau(S) = S'$ such that $\Delta(S, S') = P(S)$. To show that $B = \langle \mathcal{F}, \mathit{In}, \mathit{Out}, D, \mathcal{S}, \mathcal{S}_0, \tau\rangle$ is a sequential procedure such that $\Delta(S, \tau(S)) = P(S)$ 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. $\square$

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 proof that for every sequential algorithm there exists an equivalent sequential abstract state machine [11, 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 with two-way infinite tapes. By way of example, let the tape alphabet be $\{0, 1\}$. 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.

A Turing machine state (instantaneous description) contains three things: *Left*, a finite string containing the tape section left of the reading head; *Right*, a finite string with the tape section to the right to the read head; and q, the internal state of the machine. The read head points to the first character of *Right*.

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

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

Base structure: Constructors for the finite strings: the constant symbol @ and unary function symbols *Cons_0* and *Cons_1*. Thus, @ has the empty string, ε, as its permanent value.

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 $\{0, 1\}^*$.
- Constants for the alphabet characters and TM-states (nullary): 0, 1, *q_0*, *q_1*, ..., *q_k*. Their actual values are of no significance, as long as they are all different.
- Variables to keep the current status of the Turing machine (nullary): *Left*, *Right*, and *q*. Their initial values are: $Left = \varepsilon$, $Right = \varepsilon$, and $q = q_0$.

Effective structure:

- Functions to examine the tape (unary functions): *Head* and *Tail*. Their initial values are as in the standard implementation of *Head* and *Tail*. Their effective implementation is given below, after the description of the Turing machine model.
- The Boolean equality notion =. Note that the standard equality notion contains infinite data, thus cannot be contained in the almost-constant structure, nor in the base structure. Nevertheless, since every domain element has a unique construction, the equality notion can be effectively implemented with only finite initial data. This implementation is explained after the implementation of *Head* and *Tail*.

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 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
   if Head(Right) = 1
      % write 0, move left, switch to q_1
      Left := Tail(Left)
      Right := Cons_0(Right)
      q := q_1
   if Right = @
      % write 0, move left, switch to q_2
```

```
      Left := Tail(Left)
      Right := Cons_0(Right)
      q := q_2
if q = q_1   % TM's state q_1
       ...
if q = q_k   % the halting state
   Out := Right
```

In the above description of Turing machines as an effective model we've used the functions *Head* and *Tail*. We show now their effectiveness.

The implementation sequentially enumerates all strings, assigning their *Head* and *Tail* values, until encountering the input string. Note that it uses the equality notion, which is shown to be effective afterwards.

It uses the same base structure and almost-constant structure described above, with the addition of the following nullary functions (Name = initial value): `New` $= \varepsilon$, `Backward` $= 0$, `Forward` $= 1$, `AddDigit` $= 0$, and `Direction` $= \varepsilon$.

```
% Sequentially constructing the Left variable
% until it equals to the input In, for filling
% the values of Head and Tail.
% The enumeration is: empty string, 0, 1, 00, 01, ...
if Left = In % Finished
   Right := Left
   Left := @
else % Keep enumerating
   if Direction = New % default val
      if 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 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 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
```

The equality notion. The standard equality notion has infinite data, thus cannot be given in the initial state. However, since the domain elements are uniquely constructed, it follows that it can be effectively implemented using only finite initial data. The implementation scheme is quite similar to the above implementation of *Head* and *Tail*. Initially, the value of the equality function is $\perp$ at all locations. The implementation sequentially enumerates the strings, assigning *True* as the value of equality of each string with itself and *False* for comparisons with all preceeding strings. This continues until the process gives one of the defined values to the equality operation applied to the inputs. □

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 $f(\overline{a})$ 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 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 $\varphi^{-1}(t) = [\![t]\!]_S$ 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, representationally, as any effective model.*

Proof. We show that for every effective model E over domain D there is a bijection $\pi : D \to \mathbf{N}$ such that CP $\sqsupseteq_\pi E$.

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 $[\![P'_e]\!] = \pi([\![e]\!])$. 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 $[\![P'_e]\!] = \pi([\![e]\!])$. Hence, CP $\sqsupseteq E$. □

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

Proof (of Theorem 4). By Lemma 4, computer programs (CP) are representationally at least as powerful as any effective model, while Turing machines (TM) are of equivalent power to computer programs. (There are standard bijections between Σ^* and $\mathbf{N}$.) □