On the relation between representations and computability

Juan Casanova[1] and Simone Santini[3⋆]

[1] Centre for Intelligent Systems and their Applications, School of Informatics, University of Edinburgh, UK
[2] Departamento de Ingeniería Informática, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain

Abstract. One of the fundamental results in computability is the existence of well-defined functions that cannot be computed. In this paper we study the effects of data *representation* on computability; we show that, while for each possible way of representing data there exist incomputable functions, the computability of a specific abstract function is never an absolute property, but depends on the representation used for the function domain. We examine the scope of this dependency and provide mathematical criteria to favour some representations over others. As we shall show, there are strong reasons to suggest that computational enumerability should be an additional axiom for computation models. We analyze the link between the techniques and effects of representation changes and those of oracle machines, showing an important connection between their hierarchies. Finally, these notions enable us to gain a new insight on the Church-Turing thesis: its interpretation as the underlying algebraic structure to which computation is invariant.

1 Introduction

Both historically and conceptually, computability and decidability arise as a result of trying to formally deal with the way work in mathematics itself is performed. In an attempt to provide a formalization for the work of a mathematician proving theorems and deriving results, several authors proposed respective models, such as the Turing Machine [18] or Church's λ-calculus [3]. The arguably most important result deriving from this work is the proof that there are mathematically well defined functions which cannot be computed by any of these systems. The halting theorem [18] is one of the most well known forms of this statement, particularly proving the incomputability of the so-called halting problem. In close relation to this statement is Gödel's incompleteness theorem [6] proving that no formal system capable of expressing basic arithmetic can be

⋆ The work was carried out when Juan Casanova was with the Escuela Politécnica Superior, Universidad Autónoma de Madrid; Simone Santini was supported in part by the the Spanish *Ministerio de Educación y Ciencia* under the grant N. TIN2016-80630-P, *Recomendación en medios sociales: contexto, diversidad y sesgo algorítmico.*

both consistent and complete, a more general proof of the same underlying fact that the halting theorem proves.

The classic concept of "degree of recursive unsolvability" introduced by Post [13] appears as a result of extending the concept of reducibility, implicit in Turing's work [19], to incomputable functions. Formally, an oracle machine is a theoretical device that enables a Turing machine to compute an otherwise incomputable function (respectively, set) at any step of its computation, thus making a whole new set of previously incomputable functions (r., sets) computable under the new model, while leaving other functions (r., sets) incomputable. This introduces a preorder relation among functions (r., sets) that enables the definition of equivalence classes (the degrees of recursive unsolvability) and a partial ordering among them [8]. The structure of this set has been the subject of intensive study, much of it related to the properties of the *Turing jump*. The Turing jump is the extension of the basic halting problem to Turing machines with oracle, providing an effective method for obtaining an incomputable function (r., set) for any model of computation defined through a Turing machine with oracle. The classic 1954 paper by Kleene and Post [9] contains the fundamental results in this area: the structure of the Turing degrees is that of a join-semilattice, for any set A, there are $\aleph_0$ degrees between $\boldsymbol{a}$ and $\boldsymbol{a}'$. The global structure of the set of degrees of recursive unsolvability, $\mathcal{D}$, has been the object of a significant amount of work, especially in the period 1960–1990 [12,15,16,17]. Notable results include the fact that every finite lattice is isomorphic to an initial segment of $\mathcal{D}$ [10] and that every finite Boolean algebra [14] and every countable linear order with least element [7] can be embedded in an initial segment of $\mathcal{D}$. The strongest result in this area was obtained in [1]: every initial element of an upper semi-lattice of size $\aleph_1$ with the countable predecessor property occurs as an initial segment of $\mathcal{D}$.

A different set of questions arise if we pose some restrictions on the general Turing reducibility. By limiting the access to the oracle one obtains several types of *strong reducibilities* [11], among which the most significant are the truth-table [4] and enumeration [5] reducibilities.

However, while representation is an arguably omnipresent concept in all kinds of computability and computer science work, little specific research has been carried about its effects in computability of functions (respectively, sets) from a purely abstract and formalism-centred point of view. In this article, we examine this and related ideas, motivated by a small set of examples, introduced at the beginning of the text, that show that, without further formal constraints, the choice of the way data is represented in any computation formalism is in principle a trascendental decision that greatly affects the notion of computability.

Following the examples, we provide a short and straight to the point representation-free definition of the Turing machine formalism which allows us to restate some of the most common results in computability theory in absolutely formal terms, with no dependency on the representation chosen. The extremely formal character of these results makes them insufficient for answering the common questions

in computability, and thus, in the following section we introduce a formal definition of representation along with some basic derived concepts and results.

We then define two related but not identical partial ordering structures between representations (transformability and computational strength), which enable us to prove the two main results of the article. These results trace the boundaries on the effect that representation may have on computability and provide insight on the essential elements of computability that are in closest relation with the admittedly odd effects of representation changes shown in the introductory examples (namely, computational enumerability of sets and domain restriction of functions). From these results we conclude that some of the most counter-intuitive and consistency-defying issues raised by representation changes can be dealt with by restricting the definition of computability, however leaving some of the important effects of representation on computability inevitably remaining.

Finally, motivated by the resemblance of the effects of representations to those of oracle machines and of the relations between representations to the hierarchy of degrees of recursive unsolvability, we study the relation between these two concepts in two different ways. First, we show that the hierarchy of degrees of recursive unsolvability does not correspond to the hierarchy of the so-called representation degrees, but have a strong connection with them. Second, we show that all the definitions and results regarding representations introduced in this article can be naturally and easily extended to computability with oracle machines.

As a last consideration, we examine the relation that the Church-Turing thesis and related concepts have with the ideas introduced throughout the article.

2 Some definitions

Let $\mathcal{F}$ be the set of all functions defined on countable sets. Given a function f, we indicate as usual with $\mathrm{Dom}(f)$ the set of values for which f is defined and with $f(\mathrm{Dom}(f)) = \mathrm{Rg}(f)$ the set of values that are the image of elements of $\mathrm{Dom}(f)$. The restriction $f_{|A}$ of f to $A \subseteq \mathrm{Dom}(f)$ is the function defined on A such that for all $x \in A$, $f_{|A}(x) = f(x)$.

The set of functions $(A \to B)$ is defined in the usual way

$$(A \to B) = \{f \in \mathcal{F} \,\big|\, A \subseteq \mathrm{Dom}(f) \wedge \mathrm{Rg}(f) \subseteq B\} \tag{1}$$

We shall use a square bracket to indicate that a set coincides with the domain or range of a function, so we shall define

$$[A \to B] = \{f \in \mathcal{F} \,\big|\, A = \mathrm{Dom}(f) \wedge \mathrm{Rg}(f) = B\} \tag{2}$$

with the obvious meaning for $[A \to B)$ and $(A \to B]$. Note that, of course, $[A \to B] \subseteq (A \to B)$.

* * *

In order to ground the intuition that drives this article, namely that representations bear some relevance for computability, we shall present here two examples, which we shall use throughout the paper.

Example I:
Consider the function $eq \in [\mathbb{N} \times \mathbb{N} \to \{0,1\}]$ that compares its two argumens, defined as $eq(m,n) =$ if $m = n$ then 1 else 0.

Consider now two representations of pairs of natural numbers: in the first representation, one of the numbers is coded to the left of the initial head position (the least significant digit closest to the initial head position), and the other number is coded to the right of the initial head position, symmetrically, both in binary code, with no additional information. In the second representation, they are also represented one in each direction from the initial head position, but in unary code (n is coded as $n+1$ "1"s followed by zeroes).

It is easy to see that in the first representation, eq is not computable. For assume there exists a Turing machine ϕ which succesfully compares two numbers in binary representation. Let n be any natural number. Run ϕ with the binary representation of (n,n). Since it computes eq, it must halt with a positive result after a finite number of computation steps. Write $s_n \in \mathbb{N}$ for the number of steps ϕ executes before halting with input (n,n). Now consider any natural number m such that $m \neq n$ but $m = n \bmod 2^{s_n}$. That is, m and n have the same s_n least significant binary digits. An infinite amount of such numbers exist. If we run ϕ with (m,n) as input, it will execute the first s_n steps, in which the head cannot go any further from the initial position than s_n cell positions. Thus, all cells inspected by ϕ during this execution are equal to the ones inspected when running with input (n,n). Therefore, ϕ must necessarily halt after s_n steps, with a positive result. However, $m \neq n$. This proves that ϕ does not succesfully compare any two numbers.

Note that we are actually proving something stronger: no finite computability model (a model that can't do an infinite number of operations at the same time) can compare two numbers in finite time using the first representation.

However, in the latter representation, the function is trivially computable. Intuitively, the machine runs alternatively in both directions until it encounters the last "1" on one side. If it is also the last one on the other side, they are equal, otherwise, they are distinct. Note that the problem with the former representation disappears if the representation is slightly changed so that, as it happens in all practical computers, the set of possible tape configurations is finite or the boundaries of the tape are marked with a special symbol.

Example II:
Consider the halting problem: find a Turing machine ϕ_H such that, given any other Turing machine ϕ and an input tape τ, ϕ_H always halts and indicates whether ϕ with input tape τ halts or not. One of the fundamental and best known textbook theorems in computing science says that no such TM exist.

Consider the typical representation for which the halting theorem is proved: The TM is represented on one side of the tape, codified as a set of quintuples

depicting its transition function, and the input tape is on the other side, folded so that it can be represented on just one side of the global tape. In this representation, the halting problem is undecidable.

However, the halting problem, in its most general form, is an abstract problem about the properties of certain sets of Turing machines and input tapes. In order to consider its computability, we had to indicate that a Turing machine is represented by the quintuples of its transition function. But a TM is an abstract mathematical object that can be codified on a tape in different ways. For example, we can extend the previous representation by placing an additional symbol at the initial head position. This symbol will be 1 if the input Turing machine halts with the given input tape, and 0 otherwise.

In this representation, the TM ϕ_H exists trivially. Therefore the halting problem is decidable.

* * *

Remark 1: One possible objection to the previous example is that we have obtained a solution to the halting problem only because we have a "non-computable" representation, that is, a representation in which a non-computable quantity is computed and made explicit.

We must stress, however, that computability is defined only for quantities represented on tape, that is, only once the abstract entities involved have undergone the process of representation and its computability is thus a strictly formulable fact in the Turing machine formalism (or any other computation formalism). To say that a representation is non-computable (or, for that matter, that it is is computable) is meaningless, as representation is the prerequisite necessary so that one can define computability.

The purpose of this article is to study the mathematical aspects of representation and their relation to computability, in the light of the two examples above and related facts. We will offer a theoretical justification as to why the first representation offered for the halting problem is more "reasonable" than the second. This justification will allow us to acquire an insight of the mathematical properties of computation formalisms and to better understand how representations affect the notion of computability.

3 Definitions and formalism

Throughout most of this paper we shall consider representations in the context of computation with Turing Machines, as it is arguably the best known computing model, and one in which the explicit separation between the computing device (the machine) and the input data (the tape) makes the representation problem clearer and easier to work on. Towards the end of the paper we will offer a discussion about how the choice of formalism affects the discussions and conclusions offered here and discuss briefly how formalism is related to a choice of representation.

Many interesting results in computability can be formulated meaningfully as statements on sets of tapes or strings, without the need for representations. We will discuss these results later in this section. Most of the results of this section are very well known facts whose proofs can be found in textbooks. We shall therefore simply remind the results, skipping the proofs.

We shall indicate with $\mathcal{T}_\Sigma$ the set of tapes on a finite alphabet Σ with a finite number of non-blank symbols; whenever the alphabet in question is clear from the context, we shall omit the subscript Σ; tapes will be indicated as τ, τ' (or τ_1), etc. As usual, given a tape τ, we shall indicate with $\phi(\tau)\downarrow$ the fact that the machine ϕ stops on input τ and with $\phi(\tau)\uparrow$ the fact that ϕ doesn't stop.

Functions from tapes to tapes are especially interesting here as they are the only ones for which Turing computability can be defined, as observed in Remark 1. A Turing machine, *qua* machine can take any tape as input and do something with it (possibly never stopping, of course) but in general a function $f \in (\mathcal{T} \to \mathcal{T})$ will be defined only for a subset of tapes. Therefore, a Turing machine *qua* implementation of a function f is defined only in the domain of that function.

Representations often come with a restriction in the set of "valid" tapes. In example 1, any tape can be interpreted as the binary representation of two numbers, but only tapes consisting in a collection of consecutive "1", with "0" everywhere else can be interpreted as the unary representation of numbers. A TM ϕ_{eq} that implements a comparator on this representation would of course work on any tape, in the sense that whatever may be the input tape the TM ϕ_{eq} would operate, but on these tapes ϕ_{eq} would not be an implementation of the comparator function.

Definition 1. *Let $Q, P \subseteq \mathcal{T}$; a function $f \in [Q \to P)$ is* computable *if there is a TM ϕ such that*

$$\tau \in Q \Rightarrow \phi(\tau) = f(\tau) \tag{3}$$

Definition 2. *Let $Q, P \subseteq \mathcal{T}$; a function $f \in [Q \to P)$ is* partially computable *(p.c.) if it is computable and*

$$\tau \notin Q \Rightarrow \phi(\tau)\uparrow \tag{4}$$

Definition 3. *A function $f \in [Q \to P)$ is* total computable *(t.c.) if it is p.c. and $Q = \mathcal{T}$.*

We shall indicate with $\mathfrak{M}$ the set of computable functions, and with a doubly pointed arrow the fact that a specific function is computable, that is, $f \in [Q \twoheadrightarrow P]$ entails that f is computable, that is:

$$[Q \twoheadrightarrow P] = \mathfrak{M} \cap [Q \to P] \tag{5}$$

whenever such an arrow appears in a diagram, the diagram will be said to commute if, for each doubly pointed arrow there is a computable function that makes the diagram commute in the traditional sense. The following theorem is the functional formulation of the standard result on the composability of Turing machines.

Theorem 1. *Let $f \in [Q \twoheadrightarrow P)$ and $g \in (P \twoheadrightarrow R]$, $P' = Rg(f) \cap Dom(g)$, $Q' = f^{-1}(P')$ and $R' = g(P')$, then the restriction of $g \circ f$ to Q' is computable:*

$$(g \circ f) \in [Q' \twoheadrightarrow R'] \tag{6}$$

Let $\nu, o \in \mathcal{T}$ be any two tapes with $\nu \neq o$; we shall call these the *yes* and *no* tapes.

Definition 4. *The* characteristic function $\chi_A \in [\mathcal{T} \to \mathcal{T})$ *of a set $A \subseteq \mathcal{T}$ is the function*

$$\chi_A(\tau) = \begin{cases} \nu & \text{if } \tau \in A \\ o & \text{if } \tau \notin A \end{cases} \tag{7}$$

Definition 5. *The set $A \subseteq \mathcal{T}$ is computable if χ_A is computable.*

Definition 6. *A set of tapes $A \subseteq \mathcal{T}$ is computationally-enumerable-A (c.e.-A) if the restriction of χ_A to A, $\chi_{A|A} \in [A \to \{\nu\}]$ is p.c., that is, if there is a TM ϕ such that*

$$\begin{aligned} &\tau \in A \Rightarrow \phi(\tau) = \nu \\ &\tau \notin A \Rightarrow \phi(\tau)\uparrow \end{aligned} \tag{8}$$

Definition 7. *A set $A \subseteq \mathcal{T}$ is computationally-enumerable-B (c.e.-B) if there is a partially computable function $f \in [A \to A]$ and a tape $\tau_0 \in A$ such that for each $\tau \in A$ there is $i \in \mathbb{N}$ such that $\tau = f^i(\tau_0)$; the function f is called the* enumerator *of the set.*

Theorem 2. *A set $A \subseteq \mathcal{T}$ is c.e.-A iff it is c.e.-B*

Because of this theorem, we can call sets with these properties simply *computationally enumerable*, or *c.e.*

* * *

So far, we have considered TMs working on arbitrary sets of tapes. However, example 1 shows that we should exert caution in choosing our sets of tapes, lest we be unable to compute very fundamental operations, such as comparing the representations of two numbers.

At this point, we are still considering tapes *qua* tapes, without assuming that they are the representation of anything else. Even so, however, it seems obvious that there is a certain number of properties that a "reasonable" set of tapes must satisfy if we want to do some reasonable computation with them. If we do not limit the set of tapes that we are considering, example 1 shows that we can't even determine whether two parts of a tape are equal: trying to do some meaningful computation in these circumstances would be a daunting task, exacerbating the pointlessness of doing so. It is beyond the scope of this paper to offer a theoretical justification of the properties that we assume to hold for a set of tapes, but some partial pragmatic justification can be derived from the way in which we shall use these properties in the remainder of this paper.

Some fundamental operations, such as comparison and copy, require the capacity to represent pairs of tapes (or, more in general, tuples of tapes) on a single tape. To this end, we consider a tape bijection:

$$\langle \_, \_ \rangle : \mathcal{T} \times \mathcal{T} \to \mathcal{T} \tag{9}$$

such that the following operations are computable:

Duplicate:	$\delta : \tau \mapsto \langle \tau, \tau \rangle$
Swap:	$\sigma : \langle \tau, \tau' \rangle \mapsto \langle \tau', \tau \rangle$
Projection:	$\pi_1 : \langle \tau, \tau' \rangle \mapsto \tau$
Partial application:	$\alpha_1[f] : \langle \tau, \tau' \rangle \mapsto \langle f(\tau), \tau' \rangle$

Any two such bijections are computationally equivalent under the definitions given in section 4.1.

We assume that we can combine TMs in such a way that we can detect when one of them enters an accepting state and continue the computation consequently. In particular, we assume that we can define a TM that recognizes whether its input is the same as a constant tape τ (encoded in the stucture of the machine) and a TM (called *eq*) that, given a tape containing $\langle \tau, \tau' \rangle$ accepts if $\tau = \tau'$ and rejects if $\tau \neq \tau'$.

The second projection π_2 and the partial application on the second element of a bijection $\alpha_2(f)$ can be defined in terms of the basic operations and the composition.

All the operations that we need in order to prove the results of this paper can be defined in terms of these. The proof of this fact is easy (one only has to show how to build the operations) but technical and we omit it, as it is peripheral to the contents of the paper.

Under these assumptions, but conditioned to them, more of the typical results can be proven.

Theorem 3. *If $f \in \mathfrak{M}$ and $Dom(f)$ is c.e. then $Rg(f)$ is c.e. Furthermore, if f is injective, then $f^{-1} \in [Rg(f) \twoheadrightarrow Dom(f)]$ is computable.*

Note that the function f^{-1} is computable only on $\mathrm{Rg}(f)$: the theorem doesn't guarantee that this set be computable.

The set of tapes $\mathcal{T}$ is computationally enumerable. Given an enumerator $E \in [\mathcal{T} \twoheadrightarrow \mathcal{T}]$ and an initial tape τ_0, for a given tape τ, define $\#\tau = n$ ($n \in \mathbb{N}$) if $\tau = E^n(\tau_0)$. As we will see in the following, this is equivalent to considering a representation of natural numbers which allows the computation of the successor function.

The set of TMs is also countable. One can thus consider the mth Turing machine ϕ_m, for $m \in \mathbb{N}$ under a certain enumeration. In order to allow the use of a universal Turing Machine, the enumeration that we use must allow us, given a tape with the representation of a number m on it, to execute ϕ_m. Unless we

state the contrary, the standard enumeration of Turing machines which we will consider will have this universality property.

Given a TM ϕ_e (under a particular enumeration of Turing machines), $e \in \mathbb{N}$, define

$$W_e = \{\tau \in \mathcal{T} | \phi_e(\tau)\downarrow\} \tag{10}$$

We indicate with $W_{e,n}$, $e, n \in \mathbb{N}$ the set of tapes that ϕ_e accepts after n steps.

Theorem 4. *A set $A \subseteq \mathcal{T}$ is c.e. iff $A = \emptyset$ or A is the range of a computable function.*

Finally, we give a version of the halting theorem which can be formulated in exclusive terms of Turing machines.

Theorem 5. *Let $K = \{\tau | \phi_{\#\tau}(\tau)\downarrow\}$. K is not computable*

Proof. Suppose K has a computable characteristic function χ_K; define

$$f(\tau) = \begin{cases} E(\phi_{\#\tau}(\tau)) & \text{if } \chi_K(\tau) = \nu \\ o & \text{if } \chi_K(\tau) = o \end{cases} \tag{11}$$

then $f \in \mathfrak{M}$. Thus, we know that there exists τ_0 such that for all $\tau \in \mathcal{T}$, $\phi_{\#\tau_0}(\tau)\downarrow$ and $f(\tau) = \phi_{\#\tau_0}(\tau)$ but, for all τ such that $\phi_{\#\tau}(\tau)\downarrow$, $f(\tau) = E(\phi_{\#\tau}(\tau)) \neq \phi_{\#\tau}(\tau)$. In particular, $\phi_{\#\tau_0}(\tau_0)\downarrow$ and thus $f(\tau_0) \neq \phi_{\#\tau_0}(\tau_0)$. This contradiction proves that K must be not computable.

This theorem shows that not all undecidability can be whisked away with a suitable choice of representation, the way we have done in example 2. Undecidable problems can be defined based purely on tape computation, without assuming that the tapes are representations of abstract sets.

4 Representations

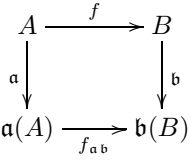
Having established a computation formalism and some basic results, we can now offer a formal definition of representation.

Definition 8. *A* representation *of an abstract set A into the set of tapes $\mathcal{T}$ is an injective function $\mathfrak{a} \in (A \to \mathcal{T})$*

The *image* of A under $\mathfrak{a}$ is the set of tapes $\mathfrak{a}(A) \subseteq \mathcal{T}$. Note that if $\mathfrak{a}$ is a representation of A and $A' \subseteq A$, then $\mathfrak{a}$ is also a representation of A' but its properties as a representation of A might be quite different from its properties as a representation of A'. For example, the set $\mathfrak{a}(A)$ might be c.e. while $\mathfrak{a}(A')$ may fail to be. As we'll see in the following, this fact has quite far-reaching consequences.

The set $\mathcal{T}$ is countable and, since representations are required to be injective, the abstract set A is also countable: we can't represent any set with cardinality higher than $\aleph_0$.

Definition 9. *Given a function $f \in [A \to B]$ and two representations $\mathfrak{a}$ and $\mathfrak{b}$ of A and B, respectively, a* representation *of f is the function $f_{\mathfrak{ab}} : [\mathfrak{a}(A) \to \mathfrak{b}(B)]$ such that the following diagram commutes*



If $f_{\mathfrak{ab}} \in \mathfrak{M}$, then we say that f is computable *in the pair of representations $\mathfrak{a}$, $\mathfrak{b}$. We shall indicate with $\mathfrak{M}(\mathfrak{a}, \mathfrak{b})$ the set of functions computable in $(\mathfrak{a}, \mathfrak{b})$.*

Remark 2: Note that even if $A = B$ we don't assume necessarily that $\mathfrak{a} = \mathfrak{b}$: the same set can be represented in two different ways as arguments and as result of the function. If $A = B$ and $\mathfrak{a} = \mathfrak{b}$, we can abbreviate $\mathfrak{M}(\mathfrak{a}, \mathfrak{a})$ as $\mathfrak{M}(\mathfrak{a})$.

On the other hand, we can assume that $A = B$ and use only one representation. If $A \neq B$, and given representations $\mathfrak{a}$ and $\mathfrak{b}$ we can always consider the set $C = A \sqcup B$ (disjoint union) and a representation $\mathfrak{c}$ such that $\mathfrak{c}(a) = \mathfrak{a}(a)$ and $\mathfrak{c}(b) = \mathfrak{b}(b)$, if $a \in A$ and $b \in B$. If for certain $a \in A$ and $b \in B$, $\mathfrak{a}(a) = \mathfrak{b}(b)$, then we can consider a partially disjoint union of A and B in which we consider that $a = b$. Using just one set and one representation for input and output becomes particularly natural when considering that most computations are to be composed, and they cannot be composed if they are on different sets. In other words, usual computation is defined for functions $(\mathbb{N} \to \mathbb{N})$ which can be chained.
Remark 3: The representation of a function is always defined: $\mathfrak{a}$ is injective in $\mathfrak{a}(A)$ and therefore invertible, so we have $f_{\mathfrak{ab}} = \mathfrak{b} \circ f \circ \mathfrak{a}^{-1}$. The representation, however, may fail to be computable.

Computability is not an immediately transferred property. Consider two functions $f_1, f_2 \in [A \to B]$ in $\mathfrak{M}(\mathfrak{a}, \mathfrak{b})$, and assume that $f_1 \in \mathfrak{M}(\mathfrak{a}', \mathfrak{b}')$. This doesn't entail that $f_2 \in \mathfrak{M}(\mathfrak{a}', \mathfrak{b}')$. To see this, consider examples 1 and 2. The identity function is computable in both representations (it is computable in any representation), but the comparator in example 1 and the halting decision function in example 2 are computable only in one of them.

Lemma 1. *The identity function of any set is computable in any representation.*

Proof. The Turing machine with just one accepting state computes the identity.

Lemma 2. *Let A be a finite set. Let $f \in [A \to A)$ be any function on this set. Then, f is computable in any representation a of A.*

Proof. For each $a \in A$, we can build a Turing machine ϕ_a such that, given the representation of another element $\mathfrak{a}(a') \in \mathfrak{a}(A)$, checks whether $\mathfrak{a}(a) = \mathfrak{a}(a')$. Since A is finite, we can build a Turing machine which executes all machines ϕ_a on any input a'. Since $a' \in A$, then one and only one of those machines will have a positive result. If the Turing machine with positive result was ϕ_a, then $a' = a$ and thus $f(a') = f(a)$. Write $f(a)$ in the result.

Lemma 3. *Let $f \in [A \to A)$ be a constant function. Then, f is computable in any representation $\mathfrak{a}$ of A.*

Proof. Let $\{x\} = Rg(f)$. Consider $\tau = \mathfrak{a}(x)$. There exists a Turing machine which replaces any input tape for the constant tape τ.

Remark 4: Lemma 3 requires the assumption that the comparator be computable or at least that there exists a way to verify when all the input has been read. Consider the binary representation considered in example 1. Replacing the input tape with a constant tape is not a computable problem in such representation. Even if the boundaries of the constant tape are previously know, the boundaries of the input tape are not, and thus it is not possible to decide when all of the input tape's extra data have been erased.

Lemma 2 does not require these assumptions, however, as we are not implementing a general comparator but rather a comparator for a constant tape, which can be implemented in any representation.

With these definitions at hand, consider example 2 again. The usual proof of the halting problem fails in this case because the new representation doesn't necessarily allow us to carry on the manipulations that the proof of the theorem requires. In particular, given the pair $\langle \phi, \tau \rangle$, the theorem assumes, by *reductio ad absurdum*, the existence of a TM ϕ_h implementing a function h such that

$$h(\langle \phi, \tau \rangle) = \begin{cases} 1 & \text{if } \phi(\tau)\downarrow \\ 0 & \text{if } \phi(\tau)\uparrow \end{cases} \tag{12}$$

The diagonalization argument then assumes that given a tape τ, we execute $h(\langle \tau, \tau \rangle)$; the argument therefore assumes that the representation used for the pair $\langle \phi, \tau \rangle$ allows us the computation of the function

$$\tau \mapsto \langle \tau, \tau \rangle \tag{13}$$

but if we represent the pair as $\langle \langle \phi, \tau \rangle, b \rangle$, where b is the solution of the halting problem for ϕ and τ, then the function

$$\tau \mapsto \langle \langle \tau, \tau \rangle, b \rangle \tag{14}$$

is computable only by solving the halting problem, and the argument becomes circular.

Most of the well known results in computability theory are expressed in terms of sets of natural numbers. We are now in the condition to express them again but, this time, relative to the representation used. We shall consider here only the concepts that we shall use in the following. We shall consider the set A that we are representing as fixed.

Definition 10. *The* domain *of the TM ϕ_e in a representation $\mathfrak{x}$ is*

$$W_e^{\mathfrak{x}} = \{a | a \in A \wedge \phi_e(\mathfrak{x}(a))\downarrow\} \tag{15}$$

Definition 11. *If $U \subseteq A$, and $\mathfrak{a}$ and $\mathfrak{b}$ are representations of A and B, respectively; a function $f \in [U \to B]$ is p.c. in the representations $\mathfrak{a}$, $\mathfrak{b}$ if there is $e \in \mathbb{N}$ such that*

$$\begin{aligned} x \in U &\Rightarrow \phi_e(\mathfrak{a}(x)) = \mathfrak{b}(f(x)) \\ x \notin U &\Rightarrow \phi_e(\mathfrak{a}(x))\uparrow \end{aligned} \tag{16}$$

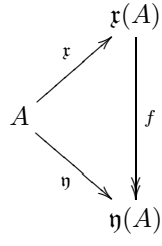
Clearly, partially computable functions are computable functions.

Definition 12. *A function $f \in [A \to B]$ is total computable in the representations $\mathfrak{a}$, $\mathfrak{b}$ if it is p.c. and $\mathfrak{a}(A) = \mathcal{T}$.*

Computability and enumerability of sets are extended in the obvious way through the computability of their characteristic functions. However, note that given a set A and a representation $\mathfrak{a}$ of A, $\mathfrak{a}(A)$ need not be necessarily c.e.. This makes some basic and intuitive results become false when extended, as they rely on the enumerability of $\mathcal{T}$. For example, if a set $B \subset A$ is computable under representation $\mathfrak{a}$ (that is, the function $\chi_B \in [A \to \{\nu, o\} \subset A]$ is computable under representation $\mathfrak{a}$); then B need not be necessarily computationally enumerable. This suggests that enumerability might be (along with comparation, duplication and the other aforementioned elemental functions) another reasonable hypothesis to require for a representation. We will come back to this idea further on.

4.1 Relations between representations

Definition 13. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of a set A: $\mathfrak{x}$ is (computationally) transformable in $\mathfrak{y}$ ($\mathfrak{y} \preceq \mathfrak{x}$) if there is a function $f \in [\mathfrak{x}(A) \twoheadrightarrow \mathfrak{y}(A)]$ such that $\mathfrak{y} = f \circ \mathfrak{x}$, viz. such that*



(17)

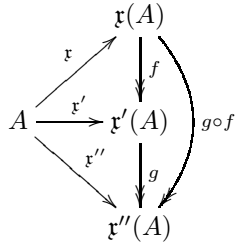
commutes.

Remark 5: If we drop the computability requirement, then the function f always exists and is unique in $[\mathfrak{x}(A) \to \mathfrak{y}(A)]$, since the representations are one-to-one and onto. The previous definition, therefore, is tantamount to requiring that $\mathfrak{y} \circ \mathfrak{x}^{-1} \in \mathfrak{M}$.

* * *

Definition 14. *Two representations* $\mathfrak{x}$ *and* $\mathfrak{y}$ *are* transformationally equivalent *(*$\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$*) if* $\mathfrak{x} \preceq \mathfrak{y}$ *and* $\mathfrak{y} \preceq \mathfrak{x}$.

Remark 6: Transformability is transitive, as can be seen from the commutativity of the following diagram and by the fact that $\mathfrak{M}$ is closed under composition.



(18)

Therefore $\overset{t}{\sim}$ is an equivalence relation. Moreover, $\preceq$ induces a partial ordering on the equivalence classes.

Theorem 6. *Let* $\mathfrak{x}, \mathfrak{y}$ *be representations of a set* A*. If* $\mathfrak{y} \preceq \mathfrak{x}$ *and* $\mathfrak{x}(A)$ *is c.e., then* $\mathfrak{y} \overset{t}{\sim} \mathfrak{x}$.

Proof. Since $\mathfrak{y} \preceq \mathfrak{x}$, there is a transformation $f \in [\mathfrak{x}(A) \twoheadrightarrow \mathfrak{y}(A)]$ and since $\mathfrak{x}(A)$ is c.e., by theorem 3, there is $f^{-1} \in [\mathfrak{y}(A) \twoheadrightarrow \mathfrak{x}(A)]$ (also computable). Therefore $\mathfrak{x} \preceq \mathfrak{y}$ and $\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$.

This theorem gives us some indications on the hierarchy induced by the relation $\preceq$; it tells us that all representations whose range is c.e. are at the bottom of the hierarchy: they are computationally transformable only in relations equivalent to them. This is no longer the case if we drop the c.e. requirement.

While transformability is an easy to understand and sound concept, it fails to express all the concerns on computability of representations we are trying to consider here. We introduce here another relation which is neither generally stronger nor weaker than the transformability relation.

Definition 15. *Let* $\mathfrak{x}$ *and* $\mathfrak{y}$ *be two representations of a set* A*;* $\mathfrak{x}$ *is* computationally stronger *than* $\mathfrak{y}$ *(written* $\mathfrak{y} \subseteq \mathfrak{x}$*) if* $\mathfrak{M}(\mathfrak{y}) \subseteq \mathfrak{M}(\mathfrak{x})$.

That is, $\mathfrak{x}$ is computationally stronger than $\mathfrak{y}$ if all functions computable in the representation $\mathfrak{y}$ are also computable in the representation $\mathfrak{x}$. This induces, of course, another equivalence relation and a partial ordering among equivalence classes.

Definition 16. *Two representations* $\mathfrak{x}$ *and* $\mathfrak{y}$ *are* computationally equivalent *(*$\mathfrak{x} \overset{c}{\sim} \mathfrak{y}$*) if* $\mathfrak{x} \subseteq \mathfrak{y}$ *and* $\mathfrak{y} \subseteq \mathfrak{x}$.

Definition 17. *Two representations* $\mathfrak{x}$ *and* $\mathfrak{y}$ *are* incomparable *(*$\mathfrak{x}||\mathfrak{y}$*) if neither* $\mathfrak{x} \subseteq \mathfrak{y}$ *nor* $\mathfrak{y} \subseteq \mathfrak{x}$.

Example III:
We will provide a new, more technical formulation of example 2, now that we have the necessary concepts. Let $\mathfrak{c}$ be the standard representation of $\mathbb{N}$, in which a number n is represented by a string of $n+1$ symbols "1" followed by a "0". Build a representation $\mathfrak{y}$ as follows. Define the function $f(n)$ as

$$f(n) = \begin{cases} 1 & \text{if } \phi_n(n)\downarrow \\ 0 & \text{if } \phi_n(n)\uparrow \end{cases} \tag{19}$$

Define

$$\mathfrak{y}(n) = \langle \mathfrak{c}(n), \mathfrak{c}(f(n)) \rangle \tag{20}$$

It is clear that $\mathfrak{y}(\mathbb{N})$ is not c.e. and that $\mathfrak{c} \preceq \mathfrak{y}$, since $\mathfrak{c} = \pi_1 \circ \mathfrak{y}$ and π_1 is computable in $\mathfrak{y}$. On the other hand, it is not $\mathfrak{c} \overset{t}{\sim} \mathfrak{y}$, as in $\mathfrak{y}$ one can compute $f(n)$ simply as

$$\mathfrak{y}(n) \mapsto \begin{cases} \langle \mathfrak{c}(0), \mathfrak{c}(f(0)) \rangle & \text{if } \pi_2(\mathfrak{y}(n)) = \mathfrak{c}(0) \\ \langle \mathfrak{c}(1), \mathfrak{c}(f(1)) \rangle & \text{if } \pi_2(\mathfrak{y}(n)) = \mathfrak{c}(1) \end{cases} \tag{21}$$

while in $\mathfrak{c}$ this is not possible (as proved by the undecidability of the halting problem).

* * *

The previous example shows that the hierarchy established by the relation $\preceq$ is not trivial.

Example IV:
As we have already mentioned, this paper is framed mostly in the context of the Turing model of computation, that is, using Turing machines. However, the concept of representations and the issues that derive from it are not limited to Turing Machines. Here we give an example that hints at the generality of representation by discussing briefly the halting problem in the context of Church's λ-calculus.

While the Turing model has two elements (namely, the machine, which implements the computing algorithm and the tape, which contains the input and output data), λ-calculus defines a single concept: the λ-funtion. Data must be represented as λ-functions on which other λ-functions operate. Computation is done through function application, and it stops (*if* it stops) when the expression is reduced to a normal form (see [2] for details). An expression that cannot be reduced to normal form is in a sense the equivalent of a Turing machine that never stops. The halting problem in the context of λ-calculus can therefore be expressed as follows:

Is there a λ-function H (the halt detector) such that, given any λ-function L (algorithm) and another λ-function I (input), H detects whether L, when given I as input, has a normal form (halts)?

Here, "detecting" means that H gives two possible results, one being the 'yes' answer (a λ-function which we will label ν) and the other the 'no' answer (o). The function H must work on two parameters, namely L and I. These two parameters will be expressed by considering the λ-function $\lambda a.aLI$ as the *representation* of the tuple (L, I), which is then evaluated by feeding it the function which must be evaluated on L and I. That is, for any lambda function F, $F(L)(I) = (\lambda a.aLI)(F)$.

Under this representation, the halting problem is known to be undecidable, a fact derived from the equivalence between λ-calculus and Turing machines. Consider, however, an alternative representation.

We shall represent the tuple (L, I) as the lambda function $\lambda a.aLIh$, where h is ν if L evaluated on I has normal form, and o if it does not. This is, of course, the same 'trick' used in the Turing machine formalism.

In this representation, H can easily be computed with the function

$$\lambda m.m(\lambda x.\lambda y.\lambda z.z).$$

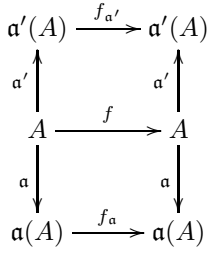
In particular,

$$\begin{aligned}(\lambda m.m(\lambda x.\lambda y.\lambda z.z))(\lambda a.aLIh) &= (\lambda a.aLIh)(\lambda x.\lambda y.\lambda z.z)\\ &= (\lambda x.\lambda y.\lambda z.z)(L)(I)(h)\\ &= h.\end{aligned}$$

* * *

The examples that we have presented so far show a common pattern: a representation is improved not by a *semantic* change that uses additional properties of the abstract set hitherto not used, but by a change internal to the space of representations that can be carried out independently of the abstract set that we are representing (adding delimiting characters, including the result of certain functions, etc.). This pattern is general, and reveals a fundamental property of representation improvement.

Let $\mathfrak{a}$, $\mathfrak{a}'$ be two representations of A. Let $f \in [A \to A]$ and $f_{\mathfrak{a}}$, $f_{\mathfrak{a}'}$ be the two corresponding representations of f. We then have



The composition $\mathfrak{x} = \mathfrak{a}' \circ \mathfrak{a}^{-1}$ is an endorepresentation: a representation of $\mathcal{T}$ into $\mathcal{T}$: $f_{\mathfrak{x}}$ is a representation of $f_{\mathfrak{a}}$[3]. Note that if $f \in \mathfrak{M}(\mathfrak{a})$ and $f \in \mathfrak{M}(\mathfrak{a}')$, then $f_{\mathfrak{a}} \in \mathfrak{M}(\mathfrak{x})$. The reverse is also true: if $f \in \mathfrak{M}(\mathfrak{a})$ and $f_{\mathfrak{a}} \in \mathfrak{M}(\mathfrak{x})$, then $f \in \mathfrak{M}(\mathfrak{a}')$.

Theorem 7. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of a set A. Then, $\mathfrak{x}$ is computationally stronger than $\mathfrak{y}$ if and only if $\mathfrak{x} \circ \mathfrak{y}^{-1} \in [\mathfrak{y}(A) \to \mathfrak{x}(A)]$, as an endorepresentation of $\mathfrak{y}(A) \subset \mathcal{T}$ is computationally stronger than the trivial representation of $\mathfrak{y}(A)$ (the identity function $id_{\mathfrak{y}(A)}$).*

Proof. Let $\mathfrak{y} \subseteq \mathfrak{x}$. Let $f \in [\mathfrak{y}(A) \to \mathfrak{y}(A))$ such that $f \in \mathfrak{M}(id_{\mathfrak{y}(A)}) \subset \mathfrak{M}$. Consider the function $g = \mathfrak{y}^{-1} \circ f \circ \mathfrak{y} \in [A \to A)$, $f = g_{\mathfrak{y}} \in \mathfrak{M}$. Therefore, $g \in \mathfrak{M}(\mathfrak{y})$. Since $\mathfrak{y} \subseteq \mathfrak{x}$, then $g \in \mathfrak{M}(\mathfrak{x})$, that is, $g_{\mathfrak{x}} = \mathfrak{x} \circ g \circ \mathfrak{x}^{-1} = \mathfrak{x} \circ \mathfrak{y}^{-1} \circ f \circ \mathfrak{y} \circ \mathfrak{x}^{-1} = f_{\mathfrak{x} \circ \mathfrak{y}^{-1}} \in \mathfrak{M}$, and $f \in \mathfrak{M}(\mathfrak{x} \circ \mathfrak{y}^{-1})$.

Now let $id_{\mathfrak{y}(A)} \subseteq \mathfrak{x} \circ \mathfrak{y}^{-1}$, and let $f \in [A \to A)$ such that $f \in \mathfrak{M}(\mathfrak{y})$. Consider $g = f_{\mathfrak{y}} \in \mathfrak{M} \cap [\mathfrak{y}(A) \to \mathfrak{y}(A))$. Thus, $g \in \mathfrak{M}(id_{\mathfrak{y}(A)})$, and since $id_{\mathfrak{y}(A)} \subseteq \mathfrak{x} \circ \mathfrak{y}^{-1}$, then $g \in \mathfrak{M}(\mathfrak{x} \circ \mathfrak{y}^{-1})$. Then, $g_{\mathfrak{x} \circ \mathfrak{y}^{-1}} = \mathfrak{x} \circ \mathfrak{y}^{-1} \circ g \circ \mathfrak{y} \circ \mathfrak{x}^{-1} = \mathfrak{x} \circ \mathfrak{y}^{-1} \circ \mathfrak{y} \circ f \circ \mathfrak{y}^{-1} \circ \mathfrak{y} \circ \mathfrak{x}^{-1} = \mathfrak{x} \circ f \circ \mathfrak{x}^{-1} = f_{\mathfrak{x}} \in \mathfrak{M}$, and $f \in \mathfrak{M}(\mathfrak{x})$.

Corollary 1. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of a set A. Then, $\mathfrak{x}$ and $\mathfrak{y}$ are computationally equivalent if and only if $\mathfrak{x} \circ \mathfrak{y}^{-1}$ and $\mathfrak{y} \circ \mathfrak{x}^{-1}$ are computationally equivalent to $id_{\mathfrak{y}(A)}$ and $id_{\mathfrak{x}(A)}$ respectively.*

Proof. $\mathfrak{x}$ and $\mathfrak{y}$ are computationally equivalent if and only if $\mathfrak{x} \subseteq \mathfrak{y}$ and $\mathfrak{y} \subseteq \mathfrak{x}$.

$\mathfrak{x} \subseteq \mathfrak{y}$ if and only if $id_{\mathfrak{x}(A)} \subseteq \mathfrak{y} \circ \mathfrak{x}^{-1}$ and $\mathfrak{y} \subseteq \mathfrak{x}$ if and only if $id_{\mathfrak{y}(A)} \subseteq \mathfrak{x} \circ \mathfrak{y}^{-1}$.

On the other hand, let $f \in [\mathfrak{y}(A) \to \mathfrak{y}(A))$, $f \in \mathfrak{M}(\mathfrak{x} \circ \mathfrak{y}^{-1})$ if and only if $f_{\mathfrak{x} \circ \mathfrak{y}^{-1}} = \mathfrak{x} \circ \mathfrak{y}^{-1} \circ f \circ \mathfrak{y} \circ \mathfrak{x}^{-1} \in \mathfrak{M}$. Because $f_{\mathfrak{x} \circ \mathfrak{y}^{-1}} \in [\mathfrak{x} \circ \mathfrak{y}^{-1}(A) \to \mathfrak{x} \circ \mathfrak{y}^{-1}(A)) \subset [\mathfrak{x}(A) \to \mathfrak{x}(A))$, then this implies that $f_{\mathfrak{x} \circ \mathfrak{y}^{-1}} \in \mathfrak{M}(id_{\mathfrak{x}(A)})$. Now, $\mathfrak{x} \subseteq \mathfrak{y}$ if and only if $id_{\mathfrak{x}(A)} \subseteq \mathfrak{y} \circ \mathfrak{x}^{-1}$, and then $f_{\mathfrak{x} \circ \mathfrak{y}^{-1}} \in \mathfrak{M}(\mathfrak{y} \circ \mathfrak{x}^{-1})$, if and only if $(f_{\mathfrak{x} \circ \mathfrak{y}^{-1}})_{\mathfrak{y} \circ \mathfrak{x}^{-1}} = \mathfrak{y} \circ \mathfrak{x}^{-1} \circ f_{\mathfrak{x} \circ \mathfrak{y}^{-1}} \circ \mathfrak{x} \circ \mathfrak{y}^{-1} = \mathfrak{y} \circ \mathfrak{x}^{-1} \circ \mathfrak{x} \circ \mathfrak{y}^{-1} \circ f \circ \mathfrak{y} \circ \mathfrak{x}^{-1} \circ \mathfrak{x} \circ \mathfrak{y}^{-1} = f \in \mathfrak{M}$, and thus $f \in \mathfrak{M}(id_{\mathfrak{y}(A)})$.

Symmetrically prove that for every $f \in [\mathfrak{x}(A) \to \mathfrak{x}(A))$, $f \in \mathfrak{M}(\mathfrak{y} \circ \mathfrak{x}^{-1})$ implies $f \in \mathfrak{M}(id_{\mathfrak{x}(A)})$.

According to these theorems improvements in representation of any abstract set come through changes internal to the tape representation, a change in the way the tapes are written. Conversely, every change that represents an improvement in the space of representation corresponds to an improvement in the representation of the abstract set. This theorem allows us to push all considerations about representation improvement to within the space of tapes, independently of the properties of the set that we are representing.

Remark 7: It is important to note that all relations between representations are relative to their use *qua* representations of a specific set. So, when we state that

[3] Formally $f_{\mathfrak{x}} = f_{\mathfrak{a}'}$, a function in $a'(A)$. Nevertheless, we use two different symbols because the interpretation of the two functions is not the same: $f_{\mathfrak{a}'}$ is a representation of f, while $f_{\mathfrak{x}}$ is a representation of $f_{\mathfrak{a}}$.

$id_{\mathfrak{y}(A)} \subseteq \mathfrak{x} \circ \mathfrak{y}^{-1}$, the relation holds only when the representations are intended as representations of the set $\mathfrak{y}(A)$. If we consider them as representation of a different set (for example of a set $B \subset A$), the relation may fail to hold.

For example, in lemma 2 we proved that all representations of finite sets are computationally equivalent. This, and the fact that there exist endorepresentations of $\mathcal{T}$ which are not computationally equivalent to the identity would seem to lead us to a contradiction. We could, the argument would go, strictly improve a representation of a finite set by using this improvement in the set of tapes. However, these endorepresentations of $\mathcal{T}$ which are not computationally equivalent to the identity as representations of $\mathcal{T}$ are computationally equivalent to the identity as representations of the finite set.

* * *

5 Representations and the Turing hierarchy

5.1 Oracles

Given a set of tapes Q, an *oracle* for Q is the characteristic function for Q. A TM with oracle Q, ϕ_e^Q is a TM with one additional operation: given a tape τ, the operation produces ν if $\tau \in Q$, and o if $\tau \notin Q$. Note that, as in the standard definition, it is easy to implement Q as an additional infinite tape that in the position $\#\tau$ has a "1" if $\tau \in Q$ and a "0" otherwise.

Given a set A, a representation $\mathfrak{x}$ of A, and a set $N \subseteq A$, an $\mathfrak{x}$-oracle for N is a function that, given the tape $\mathfrak{x}(n)$, produces ν if $n \in N$, and o otherwise; that is, it is an oracle for $\mathfrak{x}(N)$ inside $\mathfrak{x}(A)$. The TM ϕ_e^N with oracle N is defined in the obvious way.

Given a set $Q \subset \mathcal{T}$ we shall indicate with $\mathfrak{M}^Q$ the set of functions on tapes computable using an oracle for Q. Analogously, given a representation $\mathfrak{a}$ of A, and a set $N \subseteq A$, we shall indicate with $\mathfrak{M}^N(\mathfrak{a})$ the set of functions in $[A_1 \to A_2]$ for some $A_1, A_2 \subset A$ which are computable with an $\mathfrak{x}$-oracle for N.

Definition 18. *Given two sets $B, C \subseteq A$ and a representation $\mathfrak{x}$ of A, B is* $\mathfrak{x}$-computable in C *if $\chi_B \in \mathfrak{M}^C(\mathfrak{x})$.*

Consider a representation $\mathfrak{x}$ of a set A such that $\mathfrak{x}(A)$ is computationally enumerable. Then, any set $C \subseteq A$ is trivially computable using an $\mathfrak{x}$-oracle for C, and as such, it is computationally enumerable using an $\mathfrak{x}$-oracle for C. Use this enumerator to provide a standard numbering $\#_{\mathfrak{x}}^C$ of C. Note that even if C is always computable using an $\mathfrak{x}$-oracle for C, it need not be computationally enumerable if $\mathfrak{x}(A)$ is not computationally enumerable.

Definition 19. *Let A, C be two infinite countable sets with $C \subset A$. Let $\mathfrak{x}$ be a computationally enumerable representation of A. The* Turing jump *of C in the representation $\mathfrak{x}$ is*

$$C'_{\mathfrak{x}} = \left\{ c \in C | \phi^{C}{}_{\#_{\mathfrak{x}}^C(c)}(\mathfrak{x}(c)) \downarrow \right\} \tag{22}$$

The definition, per se, would not require that C be countable nor infinite, but the infinite and countable case is the one for which all the interesting results are derived so, rather than adding the restriction to the discussions that follow, we have preferred to make it explicit in the definition.

Theorem 8. *$C'_{\mathfrak{x}}$ is not $\mathfrak{x}$-computable in C.*

Proof. Let $E^C \in [C \to C]$ be the enumerator of C. Suppose $C'_{\mathfrak{x}}$ has a computable characteristic function with $\mathfrak{x}$-oracle C, ${\chi_{C'_{\mathfrak{x}}}}^C$; define $f^C \in [C \to C]$ such that

$$ {f^C}_{\mathfrak{x}}(\mathfrak{x}(c)) = \begin{cases} {E^C}_{\mathfrak{x}}({\phi^C}_{\#_{\mathfrak{x}}{}^C(c)}(\mathfrak{x}(c))) & \text{if } {\chi_{C'_{\mathfrak{x}}}}^C(c) = \nu \\ o & \text{if } {\chi_{C'_{\mathfrak{x}}}}^C(c) = o \end{cases} \tag{23} $$

then $f^C \in \mathfrak{M}^C(\mathfrak{x})$. Thus, we know that there exists $c_0 \in C$ such that for all $c \in C$, ${\phi^C}_{\#_{\mathfrak{x}}{}^C(c_0)}(\mathfrak{x}(c))\downarrow$ and ${f^C}_{\mathfrak{x}}(\mathfrak{x}(c)) = {\phi^C}_{\#_{\mathfrak{x}}{}^C(c_0)}(\mathfrak{x}(c))$ but, for all c such that ${\phi^C}_{\#_{\mathfrak{x}}{}^C(c)}(\mathfrak{x}(c))\downarrow$, ${f^C}_{\mathfrak{x}}(\mathfrak{x}(c)) = {E^C}_{\mathfrak{x}}({\phi^C}_{\#_{\mathfrak{x}}{}^C(c)}(\mathfrak{x}(c))) \neq {\phi^C}_{\#_{\mathfrak{x}}{}^C(c)}(\mathfrak{x}(c))$. In particular, ${\phi^C}_{\#_{\mathfrak{x}}{}^C(c_0)}(\mathfrak{x}(c_0))\downarrow$, and so ${f^C}_{\mathfrak{x}}(\mathfrak{x}(c_0)) \neq {\phi^C}_{\#_{\mathfrak{x}}{}^C(c_0)}(\mathfrak{x}(c_0))$. This contradiction proves that $C'_{\mathfrak{x}}$ must be not $\mathfrak{x}$-computable in C.

Definition 20. *We shall say that $C \leq_{\mathfrak{x}} B$ if C is $\mathfrak{x}$-computable in B, and that $C \equiv_{\mathfrak{x}} B$ if $C \leq_{\mathfrak{x}} B$ and $B \leq_{\mathfrak{x}} C$. We define*

$$ deg_{\mathfrak{x}}(C) = [C]_{\equiv_{\mathfrak{x}}} = \{B | B \equiv_{\mathfrak{x}} C\} \tag{24} $$

also, we set

$$ \emptyset^{(n)}_{\mathfrak{x}} = \{a | {\phi^{\emptyset^{(n-1)}_{\mathfrak{x}}}}_{\#_{\mathfrak{x}}(a)}(\mathfrak{x}(a))\downarrow\} \tag{25} $$

where $\emptyset^{(0)} = \emptyset$ is a computable set. Also, set

$$ \mathbf{0}^{(n)}_{\mathfrak{x}} = deg_{\mathfrak{x}}(\emptyset^{(n)}_{\mathfrak{x}}) \tag{26} $$

In the following, we shall indicate with $\mathfrak{c}$ the standard representation of $\mathbb{N}$, that in which the number n is represented as $n+1$ symbols "1" followed by one "0". Then $\mathbf{0}_{\mathfrak{c}} = \mathbf{0}$, the degree of standard computable sets, and

$$ \mathbf{0}_{\mathfrak{c}} \leq_{\mathfrak{c}} \mathbf{0}'_{\mathfrak{c}} \leq_{\mathfrak{c}} \mathbf{0}''_{\mathfrak{c}} \leq_{\mathfrak{c}} \cdots \leq_{\mathfrak{c}} \mathbf{0}^{(n)}_{\mathfrak{c}} \leq_{\mathfrak{c}} \cdots \tag{27} $$

is the standard Turing hierarchy. On the other hand, each representation $\mathfrak{x}$ induces a hierarchy

$$ \mathbf{0}_{\mathfrak{x}} \leq_{\mathfrak{x}} \mathbf{0}'_{\mathfrak{x}} \leq_{\mathfrak{x}} \mathbf{0}''_{\mathfrak{x}} \leq_{\mathfrak{x}} \cdots \leq_{\mathfrak{x}} \mathbf{0}^{(n)}_{\mathfrak{x}} \leq_{\mathfrak{x}} \cdots \tag{28} $$

5.2 Representation hierarchy

Representations and their hierarchies have a connection with the Turing hierarchy. Let $\chi^{(n)}_{\mathfrak{x}}$ be the characteristic function of the set $\emptyset^{(n)}_{\mathfrak{x}}$, and $\mathfrak{c}$ the standard representation. Define the class of representations $\mathfrak{u}^{(k)} : \mathbb{N} \to \mathcal{T}$ as

$$ \mathfrak{u}^{(k)}(n) = \langle \mathfrak{c}(\chi^{(k)}_{\mathfrak{c}}(n)), \ldots, \mathfrak{c}(\chi'_{\mathfrak{c}}(n)), \mathfrak{c}(n) \rangle \tag{29} $$

We have $\mathfrak{u}^{(k-1)} = \pi_2 \circ \mathfrak{u}^{(k)}$, therefore $\mathfrak{u}^{(k-1)} \preceq \mathfrak{u}^{(k)}$; on the other hand, $\chi_{\mathfrak{c}}^{(k)}$ is computable in $\mathfrak{u}^{(k)}$, but not in $\mathfrak{u}^{(k-1)}$, so $\mathfrak{u}^{(k)} \overset{t}{\not\sim} \mathfrak{u}^{(k-1)}$. We can consider the equivalence classes $[\mathfrak{u}^{(k)}]_{\overset{t}{\sim}}$. Clearly, if $\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$ it is $\mathfrak{M}(\mathfrak{x}) = \mathfrak{M}(\mathfrak{y})$, so the set $\mathfrak{M}([\mathfrak{u}^{(k)}]_{\overset{t}{\sim}})$ is well defined, and within it are the set of characteristic functions of sets which are computable by the class of representations transformationally equivalent to $\mathfrak{u}^{(k)}$. We call this the *representation degree* of $\mathfrak{u}^{(k)}$:

$$\mathbf{u^{(k)}} = \mathrm{rdg}(\mathfrak{u}^{(k)}) = \mathfrak{M}([\mathfrak{u}^{(k)}]_{\overset{t}{\sim}}) \tag{30}$$

Lemma 4.

$$\mathbf{u^{(k)}} \subseteq \mathbf{0}^{(k)} \tag{31}$$

Proof. Let $f \in \mathbf{u^{(k)}}$; then there is a TM ϕ such that, for each $n \in \mathbb{N}$, $(\mathfrak{u}^{(k)} \circ f)(n) = (\phi \circ \mathfrak{u}^{(k)})(n)$. Consider a tape with the representation $\mathfrak{c}(n)$. Since (trivially) $\chi_{\mathfrak{c}}^{(k)} \in \mathbf{0}^{(k)}$, there is a TM with oracle $\emptyset^{(k)}$ that can compute $\chi_{\mathfrak{c}}^{(k)}$ and, by the transitivity of the relation $\leq_{\mathfrak{c}}$, there are TM with oracle $\emptyset^{(k)}$ that can compute $\emptyset', \ldots, \emptyset^{(k-1)}$; the bijection $\langle , \rangle$ is also computable, therefore there is a TM $\overline{\phi^{\emptyset^{(k)}}}$ with oracle $\emptyset^{(k)}$ that, given $\mathfrak{c}(n)$ can compute $\mathfrak{u}^{(k)}(n)$. Applying $\overline{\phi^{\emptyset^{(k)}}}$ followed by ϕ we can compute f with oracle $\emptyset^{(k)}$.

The following property derives trivially from the observation that $\chi_{\mathfrak{c}}^{(k)} \in \mathbf{u^{(k)}}$, but $\chi_{\mathfrak{c}}^{(k)} \notin \mathbf{0}^{(k-1)}$

Lemma 5.

$$\mathbf{u^{(k)}} - \mathbf{0}^{(k-1)} \neq \emptyset \tag{32}$$

So, there is a hierarchy of representations that in a sense mirrors the Turing hierarchy. One question that comes naturally is whether this hierarchy corresponds to an effective increase in computing power. The following results will allow us to answer this question.

Theorem 9. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of a set A, with $\mathfrak{x} \subseteq \mathfrak{y}$. Then, if $\mathfrak{x}(A)$ is c.e. so is $\mathfrak{y}(A)$.*

Proof. Let $\mathfrak{x}_{|A} \in [A \to \mathfrak{x}(A)]$; note that $\mathfrak{x}$ is injective and therefore $\mathfrak{x}^{-1}$ exists in $\mathfrak{x}(A)$. The set $\mathfrak{x}(A)$ is c.e., therefore there are $e' \in [\mathfrak{x}(A) \twoheadrightarrow \mathfrak{x}(A)]$ and $\tau_0 \in \mathfrak{x}(A)$ such that for each $\tau \in \mathfrak{x}(A)$ there is k such that $\tau = e'^k(\tau_0)$.

Consider now the function $q = \mathfrak{x}^{-1} \circ e' \circ \mathfrak{x}_{|A} \in [A \to A]$. It is obvious that given $u_0 = \mathfrak{x}^{-1}(\tau_0)$ and $u \in A$ it is $u = q^k(u_0)$ where k is the number such that $\mathfrak{x}(u) = e'^k(\tau_0)$. That is, q is an enumerator of A and $e' = q_{\mathfrak{x}}$.

By a similar argument, it can be seen that $e = \mathfrak{y} \circ q \circ \mathfrak{y}^{-1} = q_{\mathfrak{y}}$ is an enumerator of $\mathfrak{y}(A)$ and, since q is computable in $\mathfrak{x}$ and $\mathfrak{x} \subseteq \mathfrak{y}$, q is also computable in $\mathfrak{y}$, that is, e is computable. Therefore $\mathfrak{y}(A)$ is not only c.e., but also with the same enumeration function (in A) as $\mathfrak{x}(A)$.

Theorem 10. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of a set A. If $\mathfrak{x} \subseteq \mathfrak{y}$ and $\mathfrak{x}(A)$ is c.e. then $\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$*

Proof. From theorem 9 we know that $\mathfrak{y}(A)$ is also c.e. and that there is an enumerator e of A that is computable in both representations, that is, $e \in \mathfrak{M}(\mathfrak{x}) \cap \mathfrak{M}(\mathfrak{y})$.

Let us begin by showing that $\mathfrak{y} \preceq \mathfrak{x}$; in order to do this, we must find a computable function f such that $\mathfrak{y} = f \circ \mathfrak{x}$. Let $a \in A$, and $\mathfrak{x}(a)$ its $\mathfrak{x}$-representation. If a_0 is the start value of e, initialize two tapes with $\mathfrak{x}(a_0)$ and $\mathfrak{y}(a_0)$, then simulate the two TM that compute $e_\mathfrak{x}$ and $e_\mathfrak{y}$ until we reach an iteration i such that $e^i_\mathfrak{x}(\mathfrak{x}(a_0)) = \mathfrak{x}(a)$, then $e^i(a_0) = a$, and $e^i_\mathfrak{y}(\mathfrak{y}(a_0)) = \mathfrak{y}(a)$, that is, on the second tape we have the $\mathfrak{y}$-representation of a. We therefore have a TM that computes f.

In a similar way we can build a TM that computes g such that $\mathfrak{x} = g \circ \mathfrak{y}$.

The following lemma is a direct consequence of the definition of equivalence by transformability

Lemma 6. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of A; if $\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$ then $\mathfrak{x} \overset{c}{\sim} \mathfrak{y}$.*

Note, however, that equivalence is necessary here. It is generally not true that if $\mathfrak{x} \preceq \mathfrak{y}$ then $\mathfrak{x} \subseteq \mathfrak{y}$ nor $\mathfrak{y} \subseteq \mathfrak{x}$. The intuitive idea is that transformability gives us a one way translation capability, but in order to compute we need to translate both ways, to represent and to interpret.

From these results, we easily derive the following theorem:

Theorem 11. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of A; if $\mathfrak{x}(A)$ and $\mathfrak{y}(A)$ are c.e. then either $\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$ or $\mathfrak{x} || \mathfrak{y}$.*

The proof of all these properties boils down in practice to the possibility of computing an enumerator of A. We can therefore formulate the previous result in the following guise:

Theorem 12. *Let A be a set with an enumerator e, and let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of A; then if $e \in \mathfrak{M}(\mathfrak{x})$ and $e \in \mathfrak{M}(\mathfrak{y})$, then $\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$.*

The proven results effectively prevent the creation of a hierarchy of degrees based on representations. Consider the two representations $\mathfrak{u}$ and $\mathfrak{u}'$: they are representations of $\mathbb{N}$, which is c.e. under $\mathfrak{u}$ so it is not $\mathfrak{u} \subseteq \mathfrak{u}'$, and it is not $\mathfrak{u}' \subseteq \mathfrak{u}$, since $\chi'_\mathfrak{c}$ is computable in $\mathfrak{u}'$ but not in $\mathfrak{u}$. Therefore, $\mathfrak{u} || \mathfrak{u}'$–that is, in order to gain the possibility of computing $\chi'_\mathfrak{c}$, we must give up the computability of some of the functions that are computable in $\mathfrak{u}$. Specifically, we lose the possibility of computing the iterator itself: given $\mathfrak{u}'(n) = \langle \mathfrak{c}(\chi'_\mathfrak{c}(n)), \mathfrak{c}(n) \rangle$, we can't compute $\mathfrak{u}'(n+1) = \langle \mathfrak{c}(\chi'_\mathfrak{c}(n+1)), \mathfrak{c}(n+1) \rangle$.

On the other hand, the theorem doesn't tell us anything about the other degrees of the hierarchy, since none of the range $\mathfrak{u}^{(k)}(\mathbb{N})$, $k \geq 1$ are c.e. and none of them allows the computation of the iterator. This leads to the idea of "relativizing" the properties seen so far through the use of TMs with oracles. Let $\mathfrak{M}^R(\mathfrak{x})$ be the set of functions that can be computed in a representation $\mathfrak{x}$ using a TM with oracle R. A set $Q \subseteq \mathcal{T}$ is computationally enumerable in R (R-c.e.) if its enumerator can be implemented by a TM with oracle R.

Definition 21. *A representation* $\mathfrak{y}$ *is R-better than* $\mathfrak{x}$ *(*$\mathfrak{x} \underset{R}{\subseteq} \mathfrak{y}$*) if* $\mathfrak{M}^R(\mathfrak{x}) \subseteq \mathfrak{M}^R(\mathfrak{y})$*; the equivalences* $\mathfrak{x} \underset{R}{\overset{c}{\sim}} \mathfrak{y}$, $\mathfrak{x} \underset{R}{\overset{t}{\sim}} \mathfrak{y}$ *and the incomparability* $\mathfrak{x} \underset{R}{||} \mathfrak{y}$ *are defined in the obvious way.*

The following theorem can be proved in the same way as the preceeding theorems, simply by replacing all TMs with a TM with the suitable oracle.

Theorem 13. *Let* $\mathfrak{x}$ *and* $\mathfrak{y}$ *be two representations of a set* A *with* $\mathfrak{x} \underset{R}{\subseteq} \mathfrak{y}$*; if* $\mathfrak{x}(A)$ *is R-c.e. then:*

i) $\mathfrak{y}(A)$ *is R-c.e.;*
ii) $\mathfrak{x} \underset{R}{\overset{t}{\sim}} \mathfrak{y}$*;*
iii) $\mathfrak{x} \underset{R}{\overset{c}{\sim}} \mathfrak{y}$.

From this we derive

Theorem 14. *Let* $\mathfrak{x}$ *and* $\mathfrak{y}$ *be two representations of a set* A*, then*

i) *if* $\mathfrak{x}(A)$ *and* $\mathfrak{y}(A)$ *are R-c.e., then either* $\mathfrak{x} \underset{R}{\overset{t}{\sim}} \mathfrak{y}$ *or* $\mathfrak{x} \underset{R}{||} \mathfrak{y}$*;*
ii) *if* A *is R-c.e. with enumerator* e*,* $e_{\mathfrak{x}} \in \mathfrak{M}^R(\mathfrak{x})$ *and* $e_{\mathfrak{y}} \in \mathfrak{M}^R(\mathfrak{y})$*, then* $\mathfrak{x} \underset{R}{\overset{t}{\sim}} \mathfrak{y}$.

Consider now the representations $\mathfrak{u}^{(k)}$ and $\mathfrak{u}^{(k+1)}$; note that $\chi^{(k+1)} \in \mathfrak{M}^{\emptyset^k}(\mathfrak{u}^{(k+1)})$, but $\chi^{(k+1)} \notin \mathfrak{M}^{\emptyset^k}(\mathfrak{u}^{(k)})$, so $\mathfrak{u}^{(k)} \underset{\emptyset^k}{\overset{c}{\not\sim}} \mathfrak{u}^{(k+1)}$. On the other hand the enumerator of $\emptyset^{(k)}$ is computable in $\mathfrak{u}^{(k)}$, therefore, it must be $\mathfrak{u}^{(k)} \underset{\emptyset^k}{||} \mathfrak{u}^{(k+1)}$.

* * *

6 Church-Turing thesis

The Church-Turing thesis is arguably one of the most famous statements in computability theory. The notion of representation in the Turing machine formalism and its properties, suggest us a new point of view on the Church-Turing thesis, one that we hope may lead us to a better understanding of it by seeing it in a somewhat different light. In a nutshell, the Church-Turing thesis states that the notion of computability in any formalism essentially corresponds to that defined by Turing machines. The thesis' name is due to Turing's work on providing evidence that this thesis was most likely true. More precisely, Turing provided a proof in his classic paper [18] that Turing machines and λ-calculus (defined by Church [3]) were equivalent.

There are two clarifications to be made vis-a-vis the Church-Turing thesis and related ideas. On the one hand, there is a reason for which Church-Turing

thesis is thus named. Church-Turing thesis talks about the intuitive notion of computability, and thus it is not a mathematical statement which can be proven. One can only provide evidence that suggests that any reasonable model of computability is equivalent to Turing machines. On the other hand, the Church-Turing thesis, even if considered true (in its informal sense), still does not provide an unambiguous definition of whether a given mathematical function (alternatively, set) is computable or it is not. The reason for this is that all existing mathematical models for computation are defined using representative but specific sets (strings or tapes for Turing machines, lambda functions for λ-calculus, natural numbers for (primitive) recursive functions...). Using these formalisms to consider whether a function outside the formalism is computable or it is not requires a process of representation of external elements into the formalism.

This makes our consideration on representations relevant vis-a-vis the Church-Turing thesis.

Property 1:

> For each possible mathematical set, there exists a choice of representation which makes that set computable.

The representation choice is analogue to the one provided in example 2. This fact prevents us from giving an absolute statement about the computability of functions outside of formalisms. In the case of Turing machines, we can only give absolute results for functions defined on tapes, in the case of λ-calculus for functions on λ-expressions, and so on. This makes it impossible to compare, in absolute terms, two formalisms, as they operate on different sets.

The Turing equivalence theorem, on the other hand, is a formal comparison of two formalisms, each taking as operating on a different set. In light of the apparent ambiguity that representations give to the concept of computable function, the question then becomes: What kind of equivalence exists between the Turing machine formalism and the λ-calculus formalism? The equivalence provided by Turing's theorem is *structural*: it shows that the set of Turing-computable functions (which are functions on strings or tapes) and the set of computable functions in λ-calculus (which are functions on lambda functions) are isomorphic.

In order to provide a full statement of the theorem, then, it would be necessary to specify what kind of structure they are equivalent under. This is not explicitly stated in the original Turing theorem, since the equivalence is proven by showing the somehow obvious equivalence of the basic functions and using operations such as composition to build the two isomorphic sets of computable functions in both formalisms[4]. Looking at the details of the proof, the structure

[4] It could be argued that the reason these precisions were not considered by Turing is simply because he was not interested at the moment in doing these abstract considerations, but rather was more concerned with the computability of certain functions on natural numbers with respect to the basic arithmetic operations.

for which the equivalence is proven can be given explicitly. The basic operations are considered equivalent because they are undistinguishable in set-theoretic terms. This corresponds to the notions of injectivity (cardinality) and composability, and derived ones. That is, there is a bijection between strings or tapes and λ-functions such that for each computable function and each subset, the cardinalities of the images and preimages are preserved (if one string can be obtained from two possible different strings applying a certain Turing machine, then the corresponding lambda function must be obtainable from two different λ-functions applying the corresponding lambda function, etc); and at the same time, composability is preserved (if Turing machines f and g are composed, the resulting Turing machine corresponds to the λ-function resulting from composing those corresponding to f and g).

This is not different from any other mathematical structure. For instance, sets are not group-isomorphic until they are given a group structure, and so long as cardinalities (set-theoretic structure) correspond, any two sets can be made group-isomorph by making the right choice of group structure. Computability is usually only considered for countable sets, so cardinality is not an issue. These considerations allow us to restate Property 1 in the following guise.

Property 2:

> For each possible mathematical set, there exists a choice of a computation formalism which is structurally equivalent to Turing machines such that the set is computable in such formalism.

This is somewhat the heart of the issue of representations. A change of representation and a change of formalism are two faces of the same coin: computability, the same way as groups, rings, vector spaces or any other algebraic structure, only can be endowed with after we have specified what structure we are working in. Thus, the usual notion of computability of problems of natural numbers corresponds to a computability structure specified on natural numbers and which relates to its ring structure in a particular way, namely, in that it enables the computation of the successor function. Other equivalent but incompatible structures could be considered on the same set, providing an equivalent to considering different representations of natural numbers.

7 Some Concluding Remarks

A Turing machine is an abstract device that implements syntactic functions, that is, functions that transform finite sequences of symbols on a tape into sequences of symbols on the same tape. We have shown that using this formalism to define the computability of functions specified on arbitrary sets requires a process of representation that cannot, *a priori*, be ignored. The same happens with any other model of computability that is defined on a representative set, such as

λ-functions for λ-calculus, recursive functions or natural numbers for register machines.

The way these formalisms are defined is historically motivated by the resemblances they exhibit with the mental and physical processes carried by a mathematician when working on a problem using pen and paper. This metaphor makes the computability of certain functions and operators evidently desirable: composition and partial application of computable functions, computability of the identity function and of a comparator function. While the computability of an enumerator is intuitively assumed, it is not derivable from the other properties in a general computability model, and it is generally not included as an additional axiom for the model. We have shown that there are theoretical reasons that compel the inclusion of the computability of an enumerator as an additional axiomatic capability of common computability models or, alternatively, of the representations carried when using these models. Theorem 12 of this paper proves that any computability model with the usually assumed properties can never be more powerful than another one with the same properties plus enumerability.

This result, however, gives rise to new questions. The central one is related to the fact that there are uncountably many enumerators of a countable set, and that they are obviously not all equivalent. When we are not thinking of a specific purpose for our computability model, there is no principled reason to favour the use of one of these enumerators over others. This creates a whole (uncountable) set of possible computability models over the same set, all of them mutually incomparable. As an example, consider two representations of the cartesian product of Turing machines and input tapes. The first one is the usual representation used for the classical universal Turing machine, and it allows the computation of the typical enumerator of Turing machines and tapes, and thus is a computationally enumerable representation. The second one allows the computation of another enumerator which enumerates Turing machines and tapes in a way such that odd numbers correspond to halting Turing machines and tapes, while even numbers correspond to non-halting Turing machines and tapes. Inside each of these subsets, the machines are enumerated using the enumeration provided in the first representation. As we showed, the implementation of this enumeration using a representation is trivial, as it is on the mapping function, outside all formalism, where we can include all the "magic" necessary. So much that in practice, the Turing machine implementing both enumerators can actually be the same one (the general enumerator of tapes). It is obvious that these two enumerators are incomparable, since the latter allows the computation of the halting problem while the former does not, while both being computationally enumerable, and the proven theorem can then be applied to prove the incomparability.

The kind of problems that we want to solve, or more generally, the kind of mathematical structure for which we want to create a computability model (e.g., natural numbers with successor function) will indicate which enumerator we should use. We can see this as thinking that instead of considering the

absolute computability of arbitrary abstract functions, we can consider their co-computability (or co-decidability respectively for set membership). For example, no computability model for natural numbers that allows the computation of addition also allows the computation of the halting problem (as typically formulated for natural numbers): these problems are incompatible. The general Turing jump definition included in the text generalizes the halting theorem's traditional statement that there are absolutely incomputable problems over representations to provide a new statement that in any representation, there are some incomputable problems.

We have also shown that the structure of the family of models generated by changes in representations is inherently different from that provided by oracle machines. While the introcuction of oracles strictly increases the computation power of a model, a change of representation is not guaranteed to increase or even retain its computation power. Moreover, while oracle machines cannot be produced simply by cleverly exploiting the typical definition of Turing machines, instead requiring an extension on the definitions, representations can. We have proven that the "degrees of representation" originated from using representations are related but not equal to the classical hierarchy of degrees of recursive unsolvability provided by oracle machines.

In the section on the Church-Turing thesis we suggest that computability could be defined as a mathematical structure rather than as a representative model defined on a representative set, thus transforming the problem of representation into a problem of specifying the appropriate structure in the represented set. We believe that this approach is more in line with the usual work in mathematics, while at the same time avoiding some ambiguities and apparent uncertainties present when using models such as Turing machines or λ-calculus. We believe that trying to define this algebraic structure in the same terms as groups, vector fields or topological spaces is a promising direction for future work.

Finally, a very promising line of future work is the study of the relation between representation and computational complexity. For example, while multiplication of integers represented in unary is an $O(n^2)$ problem and an $O(log^2(n))$ problem in binary, if we represent integers by their decomposition in prime factors, it becomes an O(1) problem, however making addition a potentially exponential problem. We believe that a generic approach to the problem of the effect of representation on complexity under an abstract formalism similar to the one proposed in this text could be useful and possibly provide new directions for work on usual problems in complexity.

References

1. U. Abraham and R. A. Shore. Initial segments of the degrees of size $\aleph_1$. *Israel Journal of Mathematics*, 53(1):1–51, 1986.
2. H. P. Barendregt. *The lambda calculus, its syntax and semantics*. 1984.
3. Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

4. S. Feferman. Degrees of unsolvability associated with classes of formalized theories. *Journal of Symbolic Logic*, 22:161–75, 1957.
5. R. M. Friederberg and H. Rogers. Reducibility and completeness for sets of integers. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 5:117–25, 1959.
6. Kurt Gödel. Uber formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik*, 38:173–198, 1931.
7. D. F. Hugill. Initial segments of Turing degrees. *Proceedings of the London Mathematical Society*, 19:1–16, 1969.
8. S. C. Kleene. *Introduction to Metamathematics*. New York:Van Nostrand, 1952.
9. S. C. Kleene and Emil L. Post. The upper semi-lattice of degrees of recursive unsolvability. *Annals of Mathematics*, 59(3):379–407, 1954.
10. M. Lerman. Initial segments of the degrees of unsolvability. *Annals of Mathematics*, 93:365–89, 1971.
11. P. Odifreddi. Reductibilities. In E. R. Griffor, editor, *Handbook of Computability Theory*, pages 89–119. Amsterdam:North-Holland, 1999.
12. P.D. Posner and R. W. Robinson. Degrees joining to $\emptyset'$. *Journal of Symbolic Logic*, 46(4):714–22, 1981.
13. Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50(641–2), 1944.
14. J. G. Rosenstein. Initial segments of degrees. *Pacific Journal of Mathematics*, 24:163–72, 1968.
15. G. E. Sacks. A minimal degree less than $\emptyset'$. *Bulletin of teh American Mathematical Society*, 67:416–9, 1961.
16. G. E. Sacks. The recursively enumerable degrees are dense. *Annals of Mathematics*, 80:300–12, 1964.
17. T. A. Slaman and R. I. Soare. Extensions of embeddings in the computably enumerable degrees. *Annals of Mathematics*, 154:1–43, 2001.
18. Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society II*, 43:230–265, 1936.
19. Alan Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society II*, 45:161–228, 1939.