UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

Trabajo de Fin de Grado

On the Relation Between Representations and Computability

Autor: Juan Casanova Jaquete
Tutor: Simone Santini

JUNIO 2015

Abstract

Computability and decidability are intimately linked problems which have interested computer scientists and mathematicians for a long time, especially during the last century. Work performed by Turing, Church, Godel, Post, Kleene and other authors considered the questions "What is computable?" and "What is an algorithm?". Very important results with plenty of implications were obtained, such as the halting theorem [12], the several solutions to the Entscheidungsproblem [12, 5], the Church-Turing thesis [12] or Godel's incompleteness theorem. Further work was performed on topics which as of today have remained purely theoretical but which have offered us a great understanding of computability and related questions. Some of this work includes the one related to degrees of recursive unsolvability [1] [7] and Rice's theorem [11].

Several formalisms were described and compared, some of the most important ones being Turing machines and λ-calculus. These formalisms were mathematical constructions which allowed the study of the concept of computation or calculation and all of its related questions.

We have found that an often ignored detail and, as we show, important aspect of computability is related to representation. In particular, we show that the computability of an abstract problem can only be considered once a choice of representation has been made. We inquire to what extent this is essential and what effects it may have and in what manner. We offer a wide discussion on its implications, a formalisation of these considerations and some important results deriving from these formalisations. In particular, the main result of the work is a proof that computably enumerable representations cannot be strictly stronger or weaker than other representations. We also discuss the Church-Turing thesis with particular interest, inquiring about its deep meaning and the actual facts and false assumptions related to it.

Furthermore, we consider the relationship between representation and the so-called representation degrees and the degrees of recursive unsolvability derived from the concept of oracle machine. We show that these two concepts offer parallel hierarchies which are very similar in their construction but quite different in their essential meaning and properties.

Keywords: Computability, Turing machine, Turing degrees, Turing hierarchy, Representations

Resumen

La computabilidad y la decidibilidad son problemas estrechamente relacionados que han interesado ampliamente a informáticos y matemáticos, especialmente a lo largo del último siglo. Los trabajos realizados por Turing, Church, Godel, Post, Kleene y otros autores se planteaban las preguntas "¿Qué es computable?" y "¿Qué es un algoritmo?". Se lograron muchos resultados importantes con multitud de implicaciones, como el teorema de la parada [12], la solución al Entscheidungsproblem [12, 5], la hipótesis de Church-Turing [12] o el teorema de incompletidud de Godel. Gran cantidad del trabajo posterior se realizó en relación a otros temas que han permanecido hasta hoy en el campo de la teoría pero que nos han permitido entender en mayor medida la computabilidad y problemas relacionados. Por ejemplo, el relacionado con los grados de indecibilidad [1] [7] y el teorema de Rice [11].

Varios formalismos fueron descritos y comparados, algunos de los más importantes son las máquinas de Turing y el cálculo lambda. Estos formalismos constituían construcciones matemáticas que permitían el estudio del concepto de computacion o cálculo y todas las preguntas relacionadas.

Un aspecto comúnmente ignorado y relevante de la computabilidad está relacionado con la representacion. En particular, percatamos que la computabilidad de un problema abstracto sólo puede ser considerada una vez se ha producido una elección de representación. Nos preguntamos hasta qué punto esto es esencial y qué efectos puede tener y de qué manera. Ofrecemos una amplia discusión sobre sus implicaciones, una formalización de estas consideraciones y algunos resultados importantes derivados de las mismas. En particular, el resultado principal del trabajo es una demostración de que las representaciones computacionalmente enumerables no pueden ser más fuertes o más débiles que otras. Realizamos una discusion especialmente enfrascada en relación a la tesis de Church-Turing, su significado más profundo y los hechos y falacias que giran en torno a ella.

Además, consideramos la relación existente entre la representación y los llamados grados de representacion, y los grados de indecibilidad derivados del concepto de máquina oráculo. Demostramos que estos dos conceptos ofrecen jerarquías paralelas con una construcción muy similar pero notablemente distintas en su significado esencial y sus propiedades.

Palabras clave: Computabilidad, Máquina de Turing, Grados de decidibilidad, Jerarquía de Turing, Representaciones

Contents

List of Figures

List of Tables

1 Computability theory: Background

The concept of computability, with its many different facets and shapes, has interested logicians and generally mathematicians long before physical computers were even technically feasible; especially during the past century. Its close relationship with decidability and logic has been known for a very long time, and has been one of the main reasons (if not the main) for the interest devoted to it. An approach to the general problem is posing the question "What problems can be solved through mechanical procedures, that is, algorithms?". The dual version of this question in decidability terms could be formulated as "What mathematical truths can be proved?". A second question that arises naturally from the first one is "What can be considered a mechanical procedure?", or its decidability version "What can be considered a proof?". In these questions, "problem" means often (but not exclusively) "number theoretical problem", and "proofs" refers to proofs of theorems on sets of integers.

In this work, we shall approach computability mainly from the first point of view, namely, that of algorithms or (abstract) mechanical machines. Moreover, we shall focus on an often neglected aspect of computability: the representation of the problem in such a way that the device or other chosen formalism can work on it, and its possible influence on what the device can compute. Before we do that, we shall present in this section some general concepts and results on computability; and in the next one, the precise definitions and results in the adopted Turing machine formalism, all of which forms the conceptual basis on which we shall build our work.

1.1 Computation formalisms

Turing machines

One of the first and most relevant mathematicians to formalise the concept of computability was Alan Turing [12]. Turing gave a definition of computability, by way of his Turing machines, which he then used to answer a decidability question, known as the Entscheidungsproblem, one of Hillbert's problems. The Entscheidungsproblem asks for an algorithm that, given a statement of first order logic, determines whether the statement is true or not (or rather, if it is true in every structure satisfying a given set of axioms). Turing's answer to the problem is that it is impossible to build such an algorithm. Despite the negative answer to the Entscheidungsproblem, Turing

machines have since then been one of the most widely used formalisms of computation.

Turing machines were conceived as a mathematical representation of the formal work of a mathematician working on a proof of a theorem. The overall idea is that of a machine working on a tabulated tape written using a finite alphabet, with a finite set of states and a finite set of transitions which, depending on the current state and the symbol read on the current cell of the tape, write a possibly different symbol on the cell and move to an adjacent cell, changing the machine's state on the process. The machine continues this process iteratively, possibly coming to a halt, which represents the end of the algorithm.

A formal definition of Turing machine is given in a following section. There are plenty of introductory theoretical computer science books which provide a wide and deep analysis of Turing machines and related matters, in a way more similar to how they are understood today and arguably more useful than the original Turing paper, such as [6] or [3]

Register machines

A more modern formalism based on Turing machines is that of Register machines. Register machines consist of a finite number of registers (each storing a variable number, a set of states), a set of transitions between those states (based upon the current values of the registers) and a set of operations (addition, substraction) that can alter the values of the registers on the different states. Register machines model the work of a modern computer much more closely than the overly abstract Turing machine, and indeed most physical processors are based on them. They also give a simpler way of working in theoretical computability theory, particularly when working on arithmetic functions. However, the computational capabilities of Register Machines are proven to be the same as that of Turing machines. We shall see that this seemingly general statement is subject to certain conditions on the representations used. A precise definition and properties of register machines and the related counter machines can be found in [6]

Lambda calculus

Contemporary to Turing, Alonzo Church, in collaboration with others (from which Kleene deserves mention) developed the formalism known as λ-calculus, which was shown by Turing [12] to solve the same problems as Tur-

ing machines. In particular, Church independently proved that the Entscheidungsproblem is unsolvable using this formalism [5]. Lambda functions are typographical constructs which are "evaluated" using a series of reduction forms (namely, α-reduction, β-reduction and η-reduction), providing them with properties which resemble general mathematical functions, but with a much more restricted and, most importantly, constructive definition, in the sense that their typographical declaration is their actual definition and, at the same time, method of calculation. These characteristics make them a useful means of defining computation in an arguably more mathematically elegant form, but less powerful as a symbol of the actual physical process of computation, than Turing machines.

Next to Turing machines, λ-calculus is arguably the most widely spread formalism for computation, and a lot of common features and syntaxes of modern programming languages, such as anonymous functions, are based upon it. In fact, the family of programming languages known as functional is explicitly derived from this theoretical model of computation.

A good overview of modern type-free λ-calculus (which is the one considered here), can be found at [2].

Post canonical systems

Among other similar or equivalent formal language constructions, Post canonical systems provide a formalism for specifying recursive and recursively enumerable languages in an entirely syntactical way. Essentially, one provides a set of string-transformating rules and a set of initial words or axioms from which the rules are recursively applied. Unlike regular and context-free languages, the potency of post canonical systems is equivalent to that of the other formalisms described here.

Recursive and primitive recursive functions

Recursive functions (and their subset, primitive recursive functions) offer yet another equivalent formalism of computation defined as a set of functions on natural numbers. In particular, they are defined through a set of initial functions (namely, constant functions, successor function and projection functions) and a set of operations between functions under which recursive functions are closed (namely, composition, primitive recursion and minimisation). They provide a different formal approach (arguably more abstract, or at least, algebraic; and thus less clumsy, but also less symbolic), with practi-

cal advantages and disadvantages and the same background idea as register machines: Everything is essentially reducible to numbers.

Projection functions entail the ability to work with tuples, whereas composition means chaining computations, primitive recursion is the basic iteration method (similar to for-loops) and minimisation is unbounded iteration (while-loops) or, equivalently, what is understood as recursion in common programming languages. Primitive recursive functions are a subset of recursive functions which do not make use of the minimisation operation. A more extensive introduction on recursive functions can be found on [3]

Given that in this work we are interested in the relations between representation and computability, it is interesting to look briefly at these models from this point of view. On that matter, it could be argued that, unlike the other formalisms presented here, recursive functions and register machines are more limited due to being defined specifically for natural numbers, whereas Turing machines, λ-calculus and Post canonical systems work, in principle, for any abstract set which can be represented on a tape, as a lambda function or as a formal language. However, it is essential to understand that this difference is only apparent, in a double-edged sense. On one hand, recursive functions can also compute more abstract concepts, by representing them as numbers (such as Godel numbering). On the other hand, the ability to represent becomes at the same time the need (or even, responsability) to choose one representation over others, and this opens up the question of up to what extent computability is independent of this choice, which is the main concern of this work.

1.2 Degrees of recursive unsolvability

It is a well known result of computability theory that there exist problems with a clear formulation and an unambiguous answer which cannot be resolved through mechanical procedures. The halting problem or the Entscheidungsproblem are some of the most well known examples of this. (A precise formulation of the halting problem will be included in a following section. For the current introductory discussion it is enough to know that there exist non-computable problems.) In fact, it can be easily shown through a standard diagonalisation argument that the set of problems which cannot be solved by any of the computation models previously mentioned (for instance, Turing machines) is uncountable. There is no point in engaging in a deep discussion about representations at this stage, but let us just point out that

we have not yet defined what it means to solve an abstract problem, and that a choice of representation must be present in this definition.

Even though there has been so far no real implementation of anything that could be considered a computer or a program with mathematically stronger computational capabilities than those described by any of the computation formalisms presented, it is interesting to ask what may happen if one extends the notion of computability. The theory of Turing degrees or degrees of recursive unsolvability, introduced by Post in [10], considers the so-called oracle machines, defines the degree of unsolvability of a problem using oracle machines and considers the structure of the resulting set of degrees. A thorough examination of the definitions and most important results of Turing degrees will be presented in a following section. We shall, however, offer an introductory discussion on the topic at this point.

An oracle machine is defined to be a Turing machine equipped with an *oracle*, which is a purely theoretical artifact that can answer an otherwise incomputable question (equivalently, decision problem). As the name suggests, an oracle can be thought to be some magical object or person with supernatural powers which can solve an otherwise unsolvable problem. It is important to understand that the oracle can answer one and only one type of questions. For example, one could consider an oracle which tells us whether someone is lying or not, and is never mistaken. We can ask this oracle as many times as we want for as many people as we want; however, this oracle can only tell us if somebody is lying on any given question at this particular time, and not whether this person has lied in the past, or how many times has he lied or if he will lie in the future. One could consider different oracles for these different questions, and they would absolutely make theoretical sense, but they would not be the same oracle. The Turing machine can "ask" this oracle at any step of its computation, thus being able to solve problems which would be otherwise unsolvable.

In a sense, given a problem X, a Turing machine with oracle X can solve any problem which is *reducible* to X, that is, any problem whose solution is contained in the solution of X. The set of problems reducible to X is called the *degree* of X. However, there might exist (and there are) other problems which are still unsolvable using an oracle for X. In particular, it is common to consider the problem X' (usually called the *Turing jump of* X), which takes as input a description of a Turing machine ϕ^X with oracle X and an input tape τ for ϕ^X and answers whether $\phi^X(\tau)$ halts or does not. This generalised version of the halting problem is proven to always be unsolvable

with an oracle for X, independent on the choice of the set X. This fact shows that the set of degrees of recursive unsolvability is infinite.

This theoretical construction has allowed us to understand the notion of computability much better, and its full applications might be yet to be discovered. In relation to this work, degrees of recursive unsolvability offer a very interesting perspective, and we will show that the considerations introduced by the representation problem are parallel and highly related to those introduced by the Turing jump.

1.3 Other related topics in theoretical computer science

Classically, computability and complexity have been highly related topics with common bases, parallel results and similar techniques. As a general description, complexity theory is the study of the variation of the size (in its many different senses) of an algorithm's execution as a function of the size of the input. Here, "size" may refer to execution time, memory usage or other aspects of the algorithm execution. Complexity has shown to be of extreme importance in almost all aspects of computer science, and it has many and very important implications on the implementation of algorithms. In particular, one of the most important contemporary unsolved problems is the P vs. NP problem, which considers a class of problems, in terms of complexity, known as NP (non-deterministic polynomial) problems, and asks whether these problems are in fact P(polynomial) problems. No polynomial algorithm has been found for an NP problem, and the general consensus is that P and NP must be different classes of problems. However, no proof of this fact has been found as of today. An introductory consideration on complexity, the P vs. NP problem and related topics can be found on [6].

It seems clear that the representation used must affect complexity of algorithms at least as much as it affects computability, and most likely more. As an example, consider representing natural numbers by their prime factor decomposition, expressing each prime number in binary. Obviously, given a number in this representation, the algorithm that provides its prime factor decomposition becomes trivial, and thus has very little complexity, whereas in a standard binary representation, it is in fact an NP problem. On the other hand, addition becomes a relatively complex problem on the new representation. The applications of complexity theory to new and unsolved problems

are very wide and extremely interesting; however, we have made no serious considerations on these matters in this work, mainly due to its much bigger difficulty. However, the study of the relation between representations and complexity is a promising direction for future work.

Some other classical topics in theoretical computer science on which the issue of representation might prove interesting are data structures, artificial intelligence and machine learning. Data structures and storage considers the problem of storing information in a computer system in a way that its reading, modification and verification are as efficient as possible. In this area, the idea of performing a purely theoretical and mathematical consideration on how a choice of representation affects the qualities and capabilities of databases is intruguing. For example, relational databases entail a partial choice of representation of data, in the sense that they have been chosen to be represented as tables with columns, and a full understanding, from a mathematical point of view, on how this choice affects the possibilities and all the different aspects of the database, in comparison to other database systems, would be highly valuable. On the other hand, artifical intelligence is the study of algorithms that perform what could be understood as intelligent tasks. In particular, machine learning considers somehow generic algorithms which can, in some sense, understand data and perform an analysis on it, extracting general invariants or summarized information about the assumed real world origin of that data. In other words, algorithms with the ability to learn. Once again, it might be very interesting to consider, from a purely theoretical point of view, what are the implications of a choice of data representation in an artificial intelligence system. Evidently, considerations about the relation between representations and computability, complexity, data storage and artifical intelligence are in no way a totally new perspective or an unstudied problem. In fact, considerations on how things are represented are omnipresent in computer science topics. However, we feel that a more abstract, generic and theoretical consideration on these topics has never been attempted. Approaches usually take representations as a particular, technical and highly specific aspect of that particular problem, and deem them uninteresting and effectless to the actual results on the topic being studied, once dealt with.

1.4 State of the art in computability theory

Computability theory was mostly developed as a continuation of Turing's, Church's, Godel's and other people's work during 1930-1940. Most of the

latest work in computability revolves around degrees of recursive unsolvability. In particular, Kleene and Post further studied the structure of this set [7], showing that it is a joint semi-lattice. Similar studies showed that every finite lattice is isomorphic to an initial segment of the set of degrees [8], and that every initial element of an upper semi-lattice of size $\aleph_1$ with the countable predecessor property occurs as an initial segment of the set of degrees [1].The idea of limiting access to the oracle during computation to only certain points or situations has been studied in [9].

H. G. Rice studied the properties of sets of partial functions and algorithms implementing them, and proved what has come to be known as Rice's theorem, which states that, given any non-trivial property of partial functions (that is, a function which is not true or false for every partial function), there is no algorithm which decides whether a given algorithm computes a function with said property [11].

2 Turing machine formalism

In this section we consider the Turing machine formalism in more depth, providing more precise and extensive definitions and theorems.

A small note on notation shall be included here. 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 elements for which f is defined and with $f(\mathrm{Dom}(f)) = \mathrm{Rg}(f)$ the set of elements 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 by

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

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, $[A \to B] \subseteq (A \to B)$.

2.1 Turing machines

Given a finite alphabet Σ, and a symbol $B \in \Sigma$, called the *blank symbol*, a Turing machine on Σ is usually thought of as a machine consisting on a finite

control and a tape, with a prescribed discrete operation over time (one instant at a time). The tape is divided in cells, which are placed linearly, from left to right, and is infinite on both sides. In each cell at each moment of the computation there is a symbol written, and at each moment of the computation only a finite number of cells of the tape may have symbols distinct from the blank symbol. This number needs not be bounded, though (the number of cells distinct from the blank symbol might grow unboundedly over time, but it must always be finite). The machine also has a head which points to a specific cell on the tape, and which might move, each instant, one position to the left or to the right. The finite control consists of a set of states, one and only one of which is active at each instant of the computation, and a transition function δ which takes as input the current state and the current symbol read (symbol written in the cell where the head is positioned) and outputs a new symbol to be written on such cell, a new state for the machine and the movement of the head (left or right). If, during any step of the computation, the machine reaches a state indicated as final, it halts.

More formally, a Turing machine is defined as follows.

Definition 2.1. *A Turing machine is a 6-tuple* $\Phi = (Q, \Sigma, \delta, q_0, B, F)$*, where:*

- Q *is a finite set, called the* ***set of states***
- Σ *is a finite set, called the* ***alphabet***
- $\delta \in [Q \times \Sigma \to Q \times \Sigma \times \{left, right\})$ *is any function defined on the specified set, called the* ***transition function***
- $q_0 \in Q$ *is called the* ***initial state***
- $B \in \Sigma$ *is called the* ***blank symbol***
- $F \subset Q$ *is called the* ***set of final or accepting states***

Defining possible input tapes as functions $\tau \in [\mathbb{Z} \to \Sigma)$ such that the set $W \subset \mathbb{Z}$ of cell positions w where $\tau(w) \neq B$ is finite, and letting $\mathcal{T}$ be the set of possible input tapes, it is possible to derive a function $\Phi \in [\mathcal{T} \times \mathbb{N} \to \mathcal{T})$ from the Turing machine which, given (τ, i), provides the state of the tape $\Phi(\tau, i)$ on the i-th instant of the computation of the machine Φ given input tape τ.

A Turing machine Φ is said to halt with input tape τ if Φ reaches a final state during its computation with input τ. We will write $\Phi(\tau)\downarrow$ to indicate that Φ stops with input τ, and $\Phi(\tau)\uparrow$ to indicate that it does not. If $\Phi(\tau)\downarrow$, then there exists $i_0 \in \mathbb{N}$ such that for all $i \geq i_0$, $\Phi(\tau, i) = \Phi(\tau, i_0)$. In that case, we will write $\Phi(\tau) = \Phi(\tau, i_0)$. This is the common idea of $\Phi(\tau)$ being the *result* of the computation of Φ with input τ.

Based on the definition of Turing machine, it is common to do a few considerations, additional definitions and useful tools when working with Turing machines, such as a transition diagram representation of a Turing machine. However, this is not interesting for the topics considered in this work and will thus be skipped. A much more thorough explanation can be found in [6].

2.2 Computable functions and sets of tapes

While it is true that we are using the Turing machine formalism for this work, a more mathematically flexible approach is somehow taken by considering Turing computable functions and their properties instead of the basic mechanical properties of Turing machines, arguably harder to reason over. In this section we explore this approach and give the most essential and widely known results which allow us to better comprehend the structure of the sets of computable functions.

Thought of as an implementation of a function, a Turing machine (an algorithm) takes an input, performs a series of calculations on it and returns a result. Thus, even though any Turing machine could possibly take any tape as input and do something with it, when considering computable functions, we will consider partial functions, as it is common and necessary to restrict the possible inputs to those which are valid for said function. An example of this is that of the square root, which can only be calculated on positive numbers.

Definition 2.2. *Let $P, Q \subset \mathcal{T}$. Then, a function $f \in [P \to Q)$ is computable if there exists a Turing machine Φ such that for every $\tau \in P$,*

$$\Phi(\tau) = f(\tau) \tag{3}$$

The previous definition implies trivially that if f is computable, then for all $\tau \in P$, $\Phi(\tau)\downarrow$.

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, $f \in [P \twoheadrightarrow Q]$ entails that f is computable, that is:

$$[P \twoheadrightarrow Q] = \mathfrak{M} \cap [P \to Q] \tag{4}$$

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.

Computability is closed under composition.

Theorem 2.1. *Let $f \in [P \twoheadrightarrow Q)$ and $g \in (Q \twoheadrightarrow R]$, $Q' = Rg(f) \cap Dom(g)$, $P' = f^{-1}(Q')$ and $R' = g(Q')$, then the restriction of $g \circ f$ to P' is computable:*

$$(g \circ f) \in [P' \twoheadrightarrow R'] \tag{5}$$

In the following, we shall often be in the situation of having to represent different tapes on a single tape and on having to perform certain operations on these tapes. To this end, we consider a tape bijection:

$$\langle \_, \_ \rangle : \mathcal{T} \times \mathcal{T} \to \mathcal{T} \tag{6}$$

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:	$\alpha_1[f] : \langle \tau, \tau' \rangle \mapsto \langle f(\tau), \tau' \rangle$

It is a well known result that, using a proper representation, such a bijection exists. It is also possible to see, even though the proof will not be included here, that given any two such bijections, there exists a computable function which transforms one into the other.

The projection and application on the second element of a bijection can be defined in terms of the basic operations and the composition

$$\pi_2 = \pi_1 \circ \sigma \tag{7}$$

$$\alpha_2[f] = \sigma \circ \alpha_1[f] \circ \sigma \tag{8}$$

Other functions that we shall need later on can be defined in terms of the basic ones. In particular:

i) The *constant insertion*

$$\iota_1[q] : \tau \to \langle q, \tau \rangle$$

is defined as

$$\iota_1[q] = \alpha_1[\phi_q] \circ \delta \tag{9}$$

where ϕ_q is a Turing machine that produces the constant tape q; the constant insertion in the second element, $\iota_2[q]$ is defined in the obvious way using the swap.

ii The *left rotation*

$$\rho_l : \langle \tau, \langle \tau', \tau'' \rangle \rangle \mapsto \langle \langle \tau, \tau' \rangle, \tau'' \rangle$$

is defined as

$$\rho_l = \alpha_2[\pi_2 \circ \pi_2] \circ \alpha_1[\alpha_2[\pi_1]] \circ \delta \tag{10}$$

iii) The *right rotation*

$$\rho_r : \langle \langle \tau, \tau' \rangle, \tau'' \rangle \mapsto \langle \tau, \langle \tau', \tau'' \rangle \rangle$$

can be obtained from the left one.

iv) The *embeddings*

$$\epsilon_1 : \langle \langle \tau, \tau' \rangle, \tau'' \rangle \mapsto \langle \tau'', \tau' \rangle$$

defined as:

$$\epsilon_1 = \sigma \circ \alpha_1[\pi_2] \tag{11}$$

and

$$\epsilon_2 : \langle \langle \tau, \tau' \rangle, \tau'' \rangle \mapsto \langle \tau, \tau'' \rangle$$

Another useful and common tool used in computability are the yes and no tapes, that is, any two fixed, constant and different tapes, $\nu, o \in \mathcal{T}$ such that $\nu \neq o$, which represent "yes" and "no".

As explained in the introduction, computability of functions and decidability of sets are essentially the same problem. In fact, in computability theory it is common to avoid the choice of one of the two particular approaches but rather consider both at the same time, granting us the useful aspects of both. In that sense, it is possible to express decidability as a particular problem on the computability of a function.

Definition 2.3. *The* characteristic function $\chi_A \in [\mathcal{T} \to \mathcal{T})$ *of a set* $A \subseteq \mathcal{T}$ *is the function*

$$\chi_A(\tau) = \begin{cases} \nu & \text{if } \tau \in A \\ o & \text{if } \tau \notin A \end{cases} \tag{12}$$

Definition 2.4. *The set* $A \subseteq \mathcal{T}$ *is computable if* χ_A *is computable.*

A highly important class of sets which is closely related to computable sets are computably enumerable sets. There are two common definitions of a computably enumerable set, at first glance different, but equivalent after all. We offer both definitions here, as they are both useful.

Definition 2.5. *A set of tapes* $A \subseteq \mathcal{T}$ *is computably-enumerable-A if the restriction of* χ_A *to* A, $\chi_{A|A} \in [A \to \{\nu\}]$ *is partially computable, that is, if there is a Turing machine* ϕ *such that*

$$\begin{aligned} &\tau \in A \Rightarrow \phi(\tau) = \nu \\ &\tau \notin A \Rightarrow \phi(\tau)\uparrow \end{aligned} \tag{13}$$

Definition 2.6. *A set* $A \subseteq \mathcal{T}$ *is computably-enumerable-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* iterator *or* enumerator *of the set.*

Theorem 2.2. *A set* $A \subseteq \mathcal{T}$ *is computably-enumerable-A iff it is computably-enumerable-B*

The proof of this theorem is rather technical and tedious, and uninteresting for the purpose of this work, and will therefore not be included here. It is however simple in its concepts and a standard known result in computability theory. Because of this theorem, we can call sets with these properties simply *computably enumerable*, or *c.e.*

The following properties of computable and computably enumerable sets are known and easily derivable:

Lemma 2.1. *Let* $A_1 \subseteq \mathcal{T}$ *and* $A_2 \subseteq \mathcal{T}$*, then:*

i) *if* A_1 *is computable, then* A_1 *is c.e.;*

ii) *if* A_1 *and* A_2 *are c.e. then so are* $A_1 \cup A_2$ *and* $A_1 \cap A_2$*;*

iii) *if* A_1 *and* A_2 *are computable then so are* $A_1 \cup A_2$ *and* $A_1 \cap A_2$*;*

iv) *if A_1 is computable, then $\mathcal{T}\backslash A_1$ is computable;*

Theorem 2.3. *If $A \subseteq \mathcal{T}$ and $\mathcal{T}\backslash A$ are both c.e., then they are both computable.*

Proof. Let ϕ' be the Turing machine that decides A as in definition 2.5, and ϕ'' the machine that decides $\mathcal{T}\backslash A$. Since A and $\mathcal{T}\backslash A$ partition $\mathcal{T}$, for any tape τ, one and only one of the computations $\phi'(\tau)$ and $\phi''(\tau)$ will stop. A machine ϕ that simulates ϕ' and ϕ'' by interleaving steps can detect which one stops and consecuently decide whether $\tau \in A$. □

Corollary 2.1. *Let $P = \{A_1, \ldots, A_n\}$ be a finite partition of $\mathcal{T}$. If all A_i are c.e., then they are computable.*

Proof. Pick one of the A_i; let's assume, without loss of generality, that we pick A_1. The set is c.e. On the other hand, $\mathcal{T}\backslash A_1 = A_2 \cup A_3 \cup \cdots \cup A_n$ is the union of a finite number of c.e. sets and, by point ii) of lemma 2.1 is also c.e. Therefore, by theorem 2.3, A_1 is computable.

Applying the same argument to all the A_i we obtain the corollary. □

A very interesting question in computability is whether any injective algorithm can be effectively inverted. The answer is yes, provided that the domain is computably enumerable. The proof of this fact essentially consists on, given an element of the range, iterating over the domain (since it is computably enumerable) and executing the forward algorithm, until we find the element we were trying to invert. We then know that the left element that gave birth to it is its inverse image.

Theorem 2.4. *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.*

Proof. Since f is computable, there is a Turing machine ϕ_f that, given $\tau \in \mathrm{Dom}(f)$, stops with $f(\tau)$. Applying definition 2.6 to $\mathrm{Dom}(f)$, there is a machine ϕ_D that generates all elements of $\mathrm{Dom}(f)$ in a given order. Then, using 2.1 we can define a Turing machine that generates $f(\tau)$ for each $\tau \in \mathrm{Dom}(f)$. This proves that $f(\mathrm{Dom}(f))$ is c.e.

To prove that f^{-1} is computable, assume that we want to compute $f^{-1}(\tau)$, with $\tau \in \mathrm{Rg}(f)$. The tape contains initially τ; let τ_0 be the initial tape mentioned in definition 2.6. The idea is to start with τ_0 and apply the function f to all the tapes until we obtain τ as a result; the tape τ' such that $\tau = f(\tau')$ is clearly $f^{-1}(\tau)$. This algorithm can be implemented iterating a

basic operator on tapes:

$$\begin{array}{lll} \tau \overset{\iota_2[\tau_0]}{\longmapsto} \langle \tau, \tau_0 \rangle & \overset{\alpha_2[\delta]}{\longmapsto} & \langle \tau, \langle \tau_0, \tau_0 \rangle \rangle \\ & \overset{\alpha_2[\alpha_1[f]]}{\longmapsto} & \langle \tau, \langle f(\tau_0), \tau_0 \rangle \rangle \\ & \overset{\rho_r}{\longmapsto} & \langle \langle \tau, f(\tau_0) \rangle, \tau_0 \rangle \\ & & (\alpha_1[\text{eq}] \text{ tells us if we are done}) \\ & \overset{\alpha_1[\pi_1]}{\longmapsto} & \langle \tau, \tau_0 \rangle \\ & \overset{\alpha_2[\phi_D]}{\longmapsto} & \langle \tau, \phi_D(\tau_0) \rangle \end{array} \tag{14}$$

□

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

It is obvious that the set of tapes $\mathcal{T}$ is countable. What is more interesting, but trivial to verify (using definition A), is that it is computably enumerable. This enables us to specify a particularly interesting numeration of tapes. Thus, given the iterator $E \in [\mathcal{T} \twoheadrightarrow \mathcal{T}]$ and an initial tape τ_0, for a given tape τ, define $\#\tau = n$ ($n \in \mathbb{N}$) if $\tau = E^n(\tau_0)$.

In the following theorem we shall avoid the laborious and cumbersome construction of the Turing machines using the basic tape operations; most of the constructions that we shall use have already been used in the proofs of the previous theorems.

Turing machines are also countable. However, it is important to understand that, since they are not tapes, considering whether the set of Turing machines is computably enumerable would first require a choice of a representation of Turing machines (for example, the classical tuple description), which we will forgo. However, it is possible to arbitrarily numerate them without resorting to a representation of natural numbers. Thus, given a Turing machine ϕ_e, $e \in \mathbb{N}$, define

$$W_e = \{\tau \in \mathcal{T} | \phi_e(\tau)\downarrow\} \tag{15}$$

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

Theorem 2.5. *A set $A \subseteq \mathcal{T}$ is c.e. iff $A = \emptyset$ or A is the range of a computable function.*

Proof. Assume $A \neq \emptyset$ (if $A = \emptyset$ the theorem is trivially true).

Note that, by the definition, A is c.e. iff $A = W_e$ for some $e \in \mathbb{N}$.

Suppose $A = \mathrm{Rg}(f)$, with f computable. Build a Turing machine ϕ_e that, given an input τ, enumerates all tapes using E and, for each one, checks whether $f(\tau') = \tau$; if such a tape is found, then ϕ_e writes ν and stops. Clearly $A = W_e$.

Assume now that $A = W_e$ for some integer e. Find the tape τ_0 with the smallest value of $\#\tau_0$ such that $\tau_0 = \langle \tau_{01}, \tau_{02} \rangle$ and $\tau_{01} \in W_{e,\#\tau_{02}}$. Define the computable function f as

$$f(\langle \tau_1, \tau_2 \rangle) = \begin{cases} \tau_1 & \text{if } \tau_1 \in W_{e,\#\tau_2+1} - W_{e,\#\tau_2} \\ \tau_{01} & \text{otherwise} \end{cases} \tag{16}$$

Clearly $\mathrm{Rg}(f) \subseteq W_e$. Let $\tau \in W_e$; choose $\hat{\tau}$ such that $\#\hat{\tau}$ is the least integer for which $\tau \in W_{e,\#\hat{\tau}+1}$. Then $f(\langle \tau, \hat{\tau} \rangle) = \tau$. Therefore $W_e \subseteq \mathrm{Rg}(f)$. □

2.3 Degrees of recursive unsolvability

Halting problem

The halting problem has possibly become the most most famous computability problem of all. This achievement is not due to casuality nor unjustified, for the halting problem has the properties of answering a very natural question (Can it be checked whether an algorithm ends or not before executing it, to avoid infinite executions without mistaking any finite but long execution for an infinite one?), it is a paradigmatical example of how and why problems may fail to be computable and most of all, its definition implies a notion of self-understanding and recursion which is fascinating and a great source of interesting questions. The most common version of the halting problem states that it is not possible to build a Turing machine which, given a description of any other Turing machine and any input tape, determines correctly whether the given machine halts when given the tape as input or it does not. The question that the halting problem poses is well defined, in the sense that for every Turing machine and every input tape, the machine either halts or it does not when given the input tape, and it never does both. However, no Turing machine can correctly answer this question.

For a fully detailed and technical analysis, the original and commonly taught formulation of the halting problem and its proof are too heavy in explanations given in natural language, unexplicit assumptions and ambiguous reasonings. Therefore, a more sterile and straightforward version of the

halting problem and its proof is given in the following theorem. We hope this allows us to better perform our following discussion without technicism or the lack of it clouding it.

Theorem 2.6. *Let $K = \{\tau | \phi_{\#\tau}(\tau)\downarrow\}$. Then, K is not computable.*

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

$$f(\tau) = \begin{cases} E(\phi_{\#\tau}(\tau)) & \text{if } \chi_K(\tau) = \nu \\ o & \text{if } \chi_K(\tau) = o \end{cases} \tag{17}$$

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

Let us consider what the halting theorem actually states and what assumptions does it make. The halting problem asks if there is a Turing machine H that, given any other Turing machine ϕ and any input tape τ, determines whether $\phi(\tau)\downarrow$. However, for this hypothetical Turing machine H to work on this problem, the input Turing machine ϕ must be codified in the cells of the input tape for H, as a Turing machine works on symbols written on tapes and has no ability to interpret an abstract mathematical concept such as that of Turing machine without writing it down as symbols in a tape. Thus, what H really receives is a tape, τ_H, such that the information about ϕ and τ is in some way contained in τ_H. In this process of *representation* of the Turing machine ϕ and the tape τ as a tape τ_H, a choice on how to represent them must be made. The most common way of dealing with this is representing ϕ by a series of tuples which represent all the possibilities of the transition function δ. That is, a series of quintuples $({s_0}^i, {\sigma_0}^i, {s_f}^i, {\sigma_f}^i, left/right)$, where ${s_0}^i$ and ${\sigma_0}^i$ are the initial state and symbol read, ${s_f}^i$, ${\sigma_f}^i$ are the next state and the written symbol, and $left/right$ means whether the head moves left or right. We will call this representation $r_0(\phi, \tau)$.This representation is very natural, it is in fact useful in defining the universal Turing machine and it generally produces no discomfort when used in the halting problem's proof.

However, we see no way in which someone could argue, in technical terms, that another representation choice would be less valid. More importantly, realise that the process of taking the mathematically abstract concept of the Turing machine ϕ and writing down its quintuples *must* be done by an external human; or, more formally, must be done outside of the Turing machine

formalism, for if we were to consider another Turing machine T translating the Turing machine ϕ into its quintuple representation, we would once again need ϕ to be represented as a tape τ_T for T to be able to work on it, creating a circular argument.

Thus, any representation choice, so long as it allows us to distinguish between different inputs is in principle equally valid, independent of how the process of representing ϕ using this representation is performed. For example, consider that, knowing that in fact either $\phi(\tau)\downarrow$ or $\phi(\tau)\uparrow$, then a possible representation $r_1(\phi, \tau)$ for the Turing machine ϕ and the tape τ is

$$r_1(\phi,\tau) = \begin{cases} \langle r_0(\phi,\tau), \nu\rangle & \text{if } \phi(\tau)\downarrow \\ \langle r_0(\phi,\tau), o\rangle & \text{otherwise} \end{cases} \tag{18}$$

However, in this representation, the halting problem in its more unprecise formulation is trivially computable (the function π_2 is a computable function implementing it). This does not mean that the halting theorem is false or mistaken, it merely means that it is missing hypotheses on the representation which r_0 fulfills and r_1 does not. We will not engage in a long discussion about in what way the proof fails if these hypotheses are not met, such as in the case of r_1. Let us just point out that the most common proof assumes that at one point one can, from a representation of a Turing machine, copy it and hand it over as its own input. This is not a computable step in the representation r_1.

It is more interesting, from a technical point of view, to consider in what way does the previously proven formulation of the theorem fail, if it does. The answer is that the halting problem as considered when using representation r_1 does not fill the hypotheses of theorem 2.6, since theorem 2.6 only proves that K is not computable when K is defined using the numeration of tapes $\#\tau$ explained before, and using the fact that this enumeration corresponds to the *computable* enumerator E. The set of *tapes* K does not correspond to the representation, through r_1, of the set of Turing machines and tapes which halt, and thus theorem 2.6 does not proof that in representation r_1 the halting problem is not computable, because in fact it is computable.

This discussion is the central point of this work. However, in order to develop it in its full magnitude, more concepts must be introduced. For now, let us grab three main and undeniable ideas from the halting theorem. First, that there exist uncomputable problems. Second, that the halting problem,

in its precise formulation, is one of these. Finally, that the deep reason for which this problem fails to be computable is its self-referencing properties: It is because of the numeration $\#\tau$, associated with the computable enumerator E, that we can link Turing machines to tapes in a one-to-one and, in some informal and merely intuitive sense, computable way. From these ideas, we shall develop the concepts of oracle machine, Turing jump and degrees of recursive unsolvability in a little bit more of detail. This labour can be performed in a similar fashion to the previous subsection, considering only sets of tapes inside the Turing machine formalism and thus avoiding the need to pay attention to implications about representations just yet.

Oracle machines and Turing degrees

Given a set of tapes Q, an *oracle* for Q is the characteristic function for Q. A Turing machine with oracle Q, ϕ_e^Q is a Turing machine with one additional operation which can be used at any point of the computation: given a tape τ, the operation produces ν if $\tau \in Q$, and o if $\tau \notin Q$. Note that 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. However, this tape would not belong to our set of admissible tapes as it would have, in general, an infinite number of non-blank symbols. It is simply an artifact used to better understand the way to work with an oracle.

All the definitions from the previous subsection can be automatically extended to computable functions and sets with oracle Q, substituting Turing machine for Turing machine with oracle Q. In particular, the concepts of a computable function with oracle Q, a computable set with oracle Q and a computably enumerable set with oracle Q are defined in the obvious way. It is common to write that a set N is computable in Q if it is computable with oracle Q. Furthermore, the concept of computable function, computable set and computably enumerable set are a particular case of computable function, computable set and computably enumerable set with oracle O, where O is any computable set (for example, $\mathrm{O} = \mathcal{T}$). We will write $\mathfrak{M}^Q$ to refer to the set of functions computable with an oracle for Q.

Clearly, all Turing machines are also Turing machines with oracle Q, for any set Q. However, because the definition of Turing machine with oracle is constructive, it is clear that the set of Turing machines with oracle Q is still countable. It is also clear that Q is computably enumerable with oracle Q.

Lemma 2.2. *Q is computably enumerable with oracle Q.*

Proof. Let $Q \subset \mathcal{T}$. We know that E is a computable function that enumerates $\mathcal{T}$. Thus, E is also computable with oracle Q. Build a Turing machine with oracle Q which iterates over $\mathcal{T}$ using E and, for each iterated tape τ_i, uses the oracle for Q to verify whether $\tau_i \in Q$. If it is, stop. If it is not, continue iteration. This Turing machine with oracle Q enumerates Q. □

We will call this enumerator E^Q. Once again, E^Q provides an ordering for the tapes $q \in Q$, and we will write $\#^Q q$ to refer to the numbering, under this enumeration, of the tape q. This enumeration can be used to enumerate Turing machines with oracle Q. Thus, writing ${\phi^Q}_{\#^Q\tau}$ has the obvious meaning, and all Turing machines with oracle Q can be enumerated in such way.

Theorem 2.6 can be extended to Turing machines with oracles. This is called the *Turing jump*.

Definition 2.7. *Let $Q \subset \mathcal{T}$ be an infinite countable set of tapes. The* Turing jump *of Q is*

$$Q' = \left\{\tau \in Q | {\phi^Q}_{\#^Q\tau}(\tau)\downarrow\right\} \tag{19}$$

Theorem 2.7. *Q' is not computable in Q.*

Proof. Suppose Q' has a computable characteristic function with oracle Q, ${\chi_{Q'}}^Q$; define

$$f^Q(\tau) = \begin{cases} E^Q({\phi^Q}_{\#^Q\tau}(\tau)) & \text{if } {\chi_{Q'}}^Q(\tau) = \nu \\ o & \text{if } {\chi_{Q'}}^Q(\tau) = o \end{cases} \tag{20}$$

then $f^Q \in \mathfrak{M}^Q$. Thus, we know that there exists τ_0 such that for all $\tau \in Q$, $f^Q(\tau) = {\phi^Q}_{\#^Q\tau_0}(\tau)$ but, for all τ, $f^Q(\tau) = E^Q({\phi^Q}_{\#^Q\tau}(\tau)) \neq {\phi^Q}_{\#^Q\tau}(\tau)$. In particular, $f^Q(\tau_0) \neq {\phi^Q}_{\#^Q\tau_0}(\tau_0)$. This contradiction proves that Q' must be not computable in Q. □

This allows us to give a formal definition for Turing degrees.

Definition 2.8. *We shall say that $P \leq Q$ if P is computable in Q, and that $P \equiv Q$ if $P \leq Q$ and $Q \leq P$. This is an equivalence relation. We define*

$$deg(Q) = [Q]_{\equiv} = \{P | P \equiv Q\} \tag{21}$$

and this induces an order relation in the degrees. $deg(P) \leq deg(Q)$ if and only if $P \leq Q$.
Also, we recursively define

$$\emptyset^{(n)} = \{\tau \in \emptyset^{(n-1)} | \phi^{\emptyset^{(n-1)}}_{\#^{\emptyset^{(n-1)}}\tau}(\tau)\downarrow\} \tag{22}$$

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

$$\mathbf{0}^{(n)} = deg(\emptyset^{(n)}) \tag{23}$$

Clearly, $\mathbf{0}^{(0)} \leq \mathbf{0}^{(1)} \leq \cdots \leq \mathbf{0}^{(n)} \cdots$.

3 Representations

3.1 Church-Turing thesis

We started this work introducing computability as the theory built around the question "What can be done by an algorithm?" and "What can be defined as an algorithm?". The computation model provided by formalisms such as Turing machines or λ-calculus is broadly accepted to be a good definition of what can be defined as an algorithm. The idea that the notion of computability corresponds to that defined by Turing machines is called the Church-Turing thesis. Church-Turing 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 classical paper [12] that Turing machines and λ-calculus (as defined by Church) were equivalent.

However, there is a reason for which the Church-Turing *thesis* is thus named. The 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. Despite this undisputed fact, the almost unanimous view is that the equivalence of specific computation formalisms, for instance Turing machines and λ-calculus, establishes that the notion of whether a specific function is computable or not is independent of the formalism used, and that this is a proven and mathematically true fact. This is a perfectly compatible view with the acceptance that the Church-Turing thesis is unprovable, but in this work we will, however, provide a rather unorthodox discussion on this other matter. More precisely, Church-Turing thesis itself talks about the intuitive notion of computability, and is thus unprovable. On the other hand, the specific equivalences between

specific models of computation can and are proved. A common idea derived from this equivalence is that Turing machines and λ-calculus define the same functions to be computable, we will argue that this is not the case.

Assume it is a *mathematically proven* fact that Turing machines and *lambda*-calculus make the same functions computable. Then, it must have a mathematically precise and unambiguous formulation. We will progressively suggest possible formulations of this fact, and argue about their properties.

Statement 3.1. *For every Turing machine ϕ, there exists one and only one lambda function f such that for all x, $\phi(x) = f(x)$.*

This first attempted statement is clearly false, or rather, incorrectly formulated. Some of its problems are that Turing machines work on tapes, and lambda functions on other lambda functions; or the detail that it does not consider the cases of a Turing machine not halting or a lambda function not having a normal form.

Note: For this discussion, we will write $f(g)\downarrow$ to indicate that a lambda function f when provided g as argument has normal form (that is, it can be correctly reduced in a finite number of steps), and $f(g)\uparrow$ to indicate that it does not.

Statement 3.2. *There exists a bijection $\theta \in [\mathcal{T} \to \Lambda]$ between the set of possible tapes of a Turing machine, $\mathcal{T}$, and the set of lambda functions, Λ, such that for every Turing machine ϕ, there exists one and only one lambda function f such that for all $\tau \in \mathcal{T}$*

1. *$\phi(\tau)\downarrow$ if an only if $f(\theta(\tau))\downarrow$.*
2. *If $\phi(\tau)\downarrow$, then $\theta(\phi(\tau)) = f(\theta(\tau))$.*

This statement is also false. The problem now bears with the hypothesis that the equivalent lambda function is unique, and viceversa. Many Turing machines can implement exactly the same behaviour, and the same goes for lambda functions. In fact, it is easy to show that for every Turing machine (respectively, lambda function), there are infinitely many different Turing machines (respectively, lambda function) which implement exactly the same behaviour. The formulation can be corrected.

Statement 3.3. *There exists a bijection $\theta \in [\mathcal{T} \to \Lambda]$ between the set of possible tapes of a Turing machine, $\mathcal{T}$, and the set of lambda functions, Λ, such that for every Turing machine ϕ, there exists a lambda function f such that for all $\tau \in \mathcal{T}$*

1. *$\phi(\tau)\downarrow$ if an only if $f(\theta(\tau))\downarrow$.*

2. *If $\phi(\tau)\downarrow$, then $\theta(\phi(\tau)) = f(\theta(\tau))$.*

and viceversa (that is, for every lambda function f, there exists a Turing machine ϕ fulfilling the properties).

This statement is true, and it is what Turing proved in [12]. Furthermore, the following corollary can be proved.

Corollary 3.1. *Let $\theta_1, \theta_2 \in [\mathcal{T} \to \Lambda]$ be two correspondences between tapes and lambda functions for which the above properties hold. Then, the function ${\theta_2}^{-1} \circ \theta_1 \in [\mathcal{T} \to \mathcal{T}]$ must be computable.*

Proof. Assume ${\theta_2}^{-1} \circ \theta_1$ was not computable. We know that for every Turing machine ϕ, there exists a lambda function f, and transitively another Turing machine $\overline{\phi}$ such that

1. $\phi(\tau)\downarrow$ if and only if $f(\theta_1(\tau))\downarrow$ if and only if $\overline{\phi}({\theta_2}^{-1} \circ \theta_1(\tau))\downarrow$.

2. If $\phi(\tau)\downarrow$, then $\theta_1(\phi(\tau)) = f(\theta_1(\tau))$, and then ${\theta_2}^{-1} \circ \theta_1(\phi(\tau)) = \overline{\phi}({\theta_2}^{-1} \circ \theta_1(\tau))$.

And we have that $\phi = ({\theta_2}^{-1} \circ \theta_1)^{-1} \circ \overline{\phi} \circ ({\theta_2}^{-1} \circ \theta_1)$, where ϕ and $\overline{\phi}$ are implemented by Turing machines. It is clear that this bears a contradiction.

□

However, Turing's theorem does not imply that a function's computability is independent of the computation formalism. To see this, reconsider one of the adaptations we had to make to the statement to make it become true (statement 3.2). We had to include a bijection between tapes and lambda functions because Turing machines work on tapes and lambda functions work on other lambda functions. A function's computability cannot be independent of the computation formalism because computable functions in different computation formalisms are different functions defined on different sets. At least two objections could be made to this conclusion. First, it could be argued that when we consider Turing machines and lambda functions as an implementation of the same abstract function in a different set (for instance, a function in natural numbers $g \in [\mathbb{N} \to \mathbb{N}]$) then this theorem implies that the function will be computable in one formalism if and only if it is computable in the other formalism. Second, it could be argued that the proven corollary ensures that if we "adapted" lambda functions to work on tapes,

then they would inevitably be equal to the Turing machine formalism. Both of these objections rely on false assumptions, and to see this we will refer to the discussion on the halting problem, in which we showed that different representations of the problem changed the computability of the problem.

A more abstract and algebraic analysis, which might be more enlightening, goes as follows. Mathematically talking, objects can only be compared in regards to some common structure. It makes no sense, for instance, to consider an isomorphism between a group and a vector space. There are groups which are also vector spaces, and thus are isomorphic to other vector spaces; but this comparison will only make sense once we have a common structure on both sides. Moreover, it is imprecise to say that a group is *also a vector space*. It would be much more formal to state that we can give the underlying set of this group a vector space structure which relates to the group structure in a specific way. A common and somehow philosophical question in abstract algebra asks what is the difference between equality and isomorphism. The most correct answer is that, as far as that structure (group, vector space, computation formalism...) goes, they are the same. However, when granted additional structures, they might not be the same. An important point on this discussion is that mathematical objects don't have underlying structures which must be somehow discovered, it is after granting them a structure that we can make considerations on them; or rather, mathematics talk only about the structure we had previously granted them, or even, about empty structures which can then be applied to "real world objects". This does not mean that, for example, the group and manifold structures of a Lie group are unrelated. But they are only a Lie group so long as these structures are related in a specific way. Any manifold can also be made a group, and viceversa, but they will only be a Lie group so long as there exists a link between both structures, granting them a more complex structure than the simple addition of the two independent structures (Lie group structure).

Bringing this idea back to computability, considering whether Turing machines and lambda functions make the same functions computable "in some sense" only becomes meaningful so long as we precise what that sense is. From the theorem on the equivalence of both formalisms provided by Turing we can extract that the structure they share (computation model) does not specify what functions shall be computable, only their relative structure (closed through composition, constant functions being computable, etc). As an example, consider the function $1 \in [\mathcal{T} \to \{\nu, o\}]$ which outputs ν if the symbol on the initial head position is a blank symbol, and o if it is not. From a structural point of view, outside of the Turing machine formalism,

this function is isomorph to the halting problem's function. They both have an image of size 2 (halt, don't halt), the preimage of both elements in the image is infinite (there are infinite tapes which have a blank symbol on the starting position, and infinite tapes that don't; and there are infinite Turing machines that halt and infinite Turing machines that don't). However, in the Turing machine formalism, 1 is computable and the halting problem is not. This could be deemed similar to the realisation that in the integers ring, -1 has an inverse and 2 does not. We could relabel all numbers or (from a different point of view) grant the set a different ring structure, in which 2 behaved like -1 and viceversa (that is, $2 \cdot 2 = 1$ and $1+1 = -1$), and this new ring would be isomorph to integers. Once again, -1 and 2 are mathematical objects and thus do not refer to some specific and fully detailed reality, -1 only means "the opposite of the neutral element through multiplication" and 2 only means "the number obtained by adding the neutral element through multiplication with itself", and it does not matter if we label them as -1 and 2 or as *potatoe* and *carrot*. A further and enlightening example of this situation comes when we consider i and $-i$ in the complex field $\mathbb{C}$. It is common knowledge that there is no mathematical difference between these two elements, except that they must be two different ones; but selecting one to be the positive and the other one to be the negative is an absolutely arbitrary choice with no consequences.

3.2 Formalising representations

What we hope to have achieved through the discussion performed so far is to justify a need to consider the question of representation in computation and what its implications are. From this point onwards, we offer what we feel is a rational formalisation of representations and some interesting results derived from this formalism. These results can be found in [4].

Definition 3.1. *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 3.2. *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*

$$\begin{array}{ccc} A & \xrightarrow{\quad f \quad} & B \\ {\scriptstyle \mathfrak{a}}\downarrow & & \downarrow{\scriptstyle \mathfrak{b}} \\ \mathfrak{a}(A) & \xrightarrow[f_{\mathfrak{ab}}]{} & \mathfrak{b}(B) \end{array}$$

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 $(\mathfrak{a}, \mathfrak{b})$.*

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 $\mathfrak{M}(\mathfrak{a})$.

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.

We assume that the representation has certain minimal properties that allow us to work with tuples of elements of A. In particular, let

$$\langle \_, \_ \rangle \in [A \times A \to A] \tag{24}$$

be a bijection.

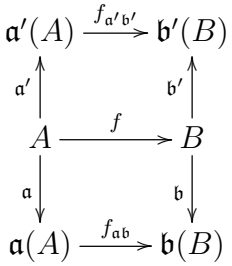
Definition 3.3. *A representation of a set A is* well founded *if the duplicate, swap, projection, partial application are computable in it, and it allows the definition of the* eq *machine (see p. 14).*

If we need to iterate the use of this bijection, we shall use the shortcuts

$$\begin{aligned} (n_1, \ldots, n_k) &= (n_1, (n_2, (\ldots, (n_{k-1}, (n_k, o))) \ldots) \\ \pi_i &= \pi_1 \circ \underbrace{\pi_2 \circ \cdots \circ \pi_2}_{i-1} \end{aligned} \tag{25}$$

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

f. We then have



The compositions $\mathfrak{x} = \mathfrak{a}' \circ \mathfrak{a}^{-1}$ and $\mathfrak{y} = \mathfrak{b}' \circ \mathfrak{b}^{-1}$ are endorepresentations: representations of $\mathcal{T}$ into $\mathcal{T}$: $f_{\mathfrak{x}\mathfrak{y}}$ is a representation of $f_{\mathfrak{a}\mathfrak{b}}$[1]. Note that if $f \in \mathfrak{M}(\mathfrak{a}, \mathfrak{b})$ and $f \in \mathfrak{M}(\mathfrak{a}', \mathfrak{b}')$, then $f_{\mathfrak{a}\mathfrak{b}} \in \mathfrak{M}(\mathfrak{x}, \mathfrak{y})$. The reverse is also true: if $f \in \mathfrak{M}(\mathfrak{a}, \mathfrak{b})$ and $f_{\mathfrak{a}\mathfrak{b}} \in \mathfrak{M}(\mathfrak{x}, \mathfrak{y})$, then $f \in \mathfrak{M}(\mathfrak{a}', \mathfrak{b}')$.

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 that constant functions are computable in all representations.

Definition 3.4. *The* domain *of the Turing machine ϕ_e in a representation $\mathfrak{x}$ is*

$$W_e^{\mathfrak{x}} = \{a | a \in A \wedge \phi_e(\mathfrak{x}(a))\downarrow\} \tag{26}$$

Computability and enumerability of sets are extended in the obvious way through the computability of their characteristic functions.

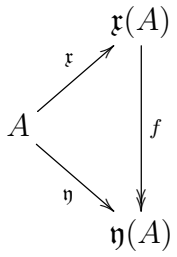
3.3 Relations between representations

One of the most interesting ways to study representations is by comparing them. That is, considering what abstract functions become computable in different representations and what properties of representations might be linked with these changes in computation power.

Definition 3.5. *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 $f \in [\mathfrak{x}(A) \twoheadrightarrow \mathfrak{y}(A)]$*

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

such that $\mathfrak{y} = f \circ \mathfrak{x}$*, viz. such that*



(27)

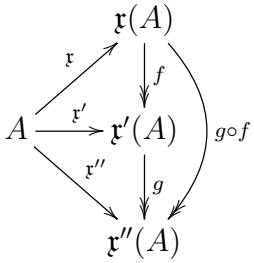
commutes.

This corresponds to the intuitive idea of being able to translate from one representation to the other. However, this only comes after an initial representation has been settled, that is, abstract concepts cannot be computationally translated.

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

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.



(28)

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

Theorem 3.1. *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 2.4, 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.

Example I:
We provide here a formalisation of the issue with representations in the case of the halting problem.

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

$$f(n) = \begin{cases} 1 & \text{if } \phi_n(\mathfrak{c}(n))\downarrow \\ 0 & \text{if } \phi_n(\mathfrak{c}(n))\uparrow \end{cases} \tag{29}$$

Define

$$\mathfrak{y}(n) = \langle \mathfrak{c}(n), \mathfrak{c}(f(n)) \rangle \tag{30}$$

in the sense of coding multiple tapes onto one on the standard way, or any other which allows us to compute the basic operations considered at the beginning of the text, for that matter.
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

$$\mathfrak{y}(n) \mapsto \begin{cases} \langle \mathfrak{c}(0), \mathfrak{c}(f(0)) \rangle & \text{if } \pi_2(\mathfrak{y}(n)) = \mathfrak{c}(0) \\ \langle \mathfrak{c}(1), \mathfrak{c}(f(1)) \rangle & \text{if } \pi_2(\mathfrak{y}(n)) = \mathfrak{c}(1) \end{cases} \tag{31}$$

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

One question that comes naturally is whether the transformability hierarchy corresponds to an effective increase in computing power. In order to answer this question, we introduce a second relation among representations.

Definition 3.7. *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 3.8. *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 3.9. *Two representations* $\mathfrak{x}$ *and* $\mathfrak{y}$ *are* incomparable *(*$\mathfrak{x}||\mathfrak{y}$*) if neither* $\mathfrak{x} \subseteq \mathfrak{y}$ *nor* $\mathfrak{y} \subseteq \mathfrak{x}$.

It could maybe seem somehow insubstantial to consider an actual increase in computing power produced merely by a change in representation, considering that computable functions in tapes are always the same and unaffected by representation. Notice, though, that on one hand, representations not only map abstract concepts into tapes, but they also restrict the set of tapes which are considered a valid representation. That is, any tape with only a finite number of non-blank symbols is a valid tape, but for certain representations, some other invariants might be necessary, such as having delimiter characters, starting on a blank symbol or having a bounded number of written symbols. These restrictions or invariants allow us to build Turing machines which assume them and work as if they were granted. These Turing machines, if applied to tapes which did not fulfill these invariants, would most likely not halt or at best, produce an irrational result. However, as a partial function on the represented set, they might be able to perform computations otherwise impossible if these invariants where not met. On the other hand, infinite sets are differentiated from finite sets mainly by their ability to contain proper subsets which are bijective to them. As such, even if we restrict the set of tapes, allowing Turing machines to work in a different way as they used to before, since representations only require to be injective, we might be able to represent equally big sets in a "smaller" tape set, gaining invariants which enhance the computation power, without losing representation power.

Example II:

Consider the function $c \in [\mathbb{N} \times \mathbb{N} \to \mathbb{N})$ such that $c(n, m) =$ *if* $n =$ m *then* 1 *otherwise* 0. A comparator of numbers. Consider now a representation $\mathfrak{b}$ of pairs of natural numbers in which they are coded one to the left of the initial head position (least significant digit closer to the initial head position), one to the right of the initial head position, in binary code; with

no additional information. Consider, on the other hand, another representation of pairs of natural numbers in which they are coded as well one to the left, one to the right, but in unary code (to code n, we would write n "1"'s, followed by zeroes).

Lemma 3.1. *c is not computable in $\mathfrak{b}$.*

Proof. Assume it was computable. Then $c_{\mathfrak{b}} \in \mathfrak{M}$, and is thus implemented by at least one Turing machine $\phi_{c_{\mathfrak{b}}}$. Let $n \in \mathbb{N}$ be any natural number. If we execute $\phi_{c_{\mathfrak{b}}}(\mathfrak{b}(n,n))$, it halts with a positive result after a finite number of steps. Write $s_n \in \mathbb{N}$ for the number of steps $\phi_{c_{\mathfrak{b}}}$ executes when given $\mathfrak{b}(n,n)$. Now consider any natural number $m \in \mathbb{N}$ such that $m \neq n$ and $m = n \ mod \ 2^{s_n}$, that is, m and n have the same least significant s_n binary digits. An infinite amount of such numbers exist. If we execute $\phi_{c_{\mathfrak{b}}}(\mathfrak{b}(n,m))$, in the first s_n steps it will only be able to read at most s_n tape positions, and since machines move one position at a time, it would never be able to read any digit of n or m other than the s_n least significant digits. Since they are equal, this execution would be exactly the same as the one performed when running $\phi_{c_{\mathfrak{b}}}(\mathfrak{b}(n,n))$ and would thus halt with a positive result, indicating that $n = m$, which is incorrect. Thus, $\phi_{c_{\mathfrak{b}}}$ does not correctly implement c. □

However, in $\mathfrak{x}$, c is trivially computable. Intuitively, the machine just runs both directions at the same time until it encounters a 0. If it encounters a 0 on both sides, they are equal. Otherwise, they are not.

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

Lemma 3.2. *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 $\mathfrak{x} \subseteq \mathfrak{y}$. 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 before the computation and to interpret after the computation.

We have delimited the implications on one direction between the two defined hierarchies (transformability hierarchy and computational strength hierarchy). We know that two transformationally equivalent representations must be computationally equivalent, and that it is not necessary that a representation transformable into another is neither computationally stronger

nor weaker. We now ask if any implications on the opposite direction can be made. Must computationally stronger or equivalent representations imply anything about their transformability? A very satisfactory answer to this question, which also answers some yet unformulated questions, arises when we add the additional hypothesis that the represented sets be c.e.

Theorem 3.2. *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 $\mathfrak{x}(u) = e'^k(t_0)$. That is, q is an enumerator of A and $e' = q_{\mathfrak{x}}$ is its computable representation.

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 a computable function. Therefore $\mathfrak{y}(A)$ is not only c.e., but also with the same enumeration function (in A) as $\mathfrak{x}(A)$. □

This theorem is even stronger than it looks at first sight. The fact that $\mathfrak{y}$ is not only computably enumerable, but also with the exact same enumeration as $\mathfrak{x}$ yields us an immense power, summarized in the following theorem.

Theorem 3.3. *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 3.2 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 Turing machines 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 Turing machine that computes f.

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

We can reformulate these results to show all of its implications.

Theorem 3.4. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of A; if $\mathfrak{x}(A)$ or $\mathfrak{y}(A)$ 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 3.5. *Let A be a set with an enumerator e, and let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of A; then if $e \in \mathfrak{M}(\mathfrak{x})$ or $e \in \mathfrak{M}(\mathfrak{y})$, it is then either $\mathfrak{x} \overset{t}{\sim} \mathfrak{y}$ or $\mathfrak{x}||\mathfrak{y}$.*

4 Turing hierarchy and representations

The hierarchy induced by the computationally stronger relation and the Turing hierarchy seem, at first sight, to be somewhat related. When we presented oracle machines, we explicitly defined them to be an increase in computing power. On the other hand, while oracle machines are an extension of Turing machines, we have shown that representations are something necessary in order to be able to use the Turing machine formalism to consider computability on abstract sets, and thus it would seem inappropriate that they happened to implement the same situations. In this section we consider the relationships between both hierarchies.

4.1 Represented oracles

We will first extend the ideas of oracle machines to represented sets. 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 $\mathfrak{x}(N)$. The Turing machine ϕ_e^N with oracle N is defined in the obvious way. We shall indicate with $\mathfrak{M}^N(\mathfrak{x})$ the set of functions in A computable with an $\mathfrak{x}$-oracle for N.

Definition 4.1. *Given two sets $B, C \subseteq A$ and a representation $\mathfrak{x}$ of A, B is* $\mathfrak{x}$-computable in C *if $\chi_{B|A} \in \mathfrak{M}^C(\mathfrak{x})$.*

We will now provide an analogue definition of Turing jump and Turing degrees for represented sets. For this, a computable enumeration is needed. Consider $\mathfrak{x}$ a representation of a set A such that $\mathfrak{x}(A)$ is computably enumerable. Then, C is trivially computable using an $\mathfrak{x}$-oracle for C, and as such,

it is computably 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 computably enumerable if A is not computably enumerable. In general, a represented set which is computable need not be computably enumerable if the representation is not computably enumerable. This provides further evidence that representations which are not computably enumerable do not seem very appropriate.

Definition 4.2. *Let A, C be two infinite countable sets with $C \subset A$. Let $\mathfrak{x}$ be a computably enumerable representation of A. The* Turing jump *of C in the representation $\mathfrak{x}$ is*

$$C'_{\mathfrak{x}} = \left\{ c \in C | \phi_{{\#_{\mathfrak{x}}}^C c}(\mathfrak{x}(c)) \downarrow \right\} \tag{32}$$

Theorem 4.1. *$C'_{\mathfrak{x}}$ 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

$$ {f^C}_{\mathfrak{x}}(\mathfrak{x}(c)) = \begin{cases} {E^C}_{\mathfrak{x}}({\phi^C}_{{\#_{\mathfrak{x}}}^C c}(\mathfrak{x}(c))) & \text{if } {\chi_{C'_{\mathfrak{x}}}}^C(c) = \nu \\ o & \text{if } {\chi_{C'_{\mathfrak{x}}}}^C(c) = o \end{cases} \tag{33}$$

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

Definition 4.3. *Let A be a set and $\mathfrak{x}$ a computably enumerable representation of A. We shall say that $P \leq_{\mathfrak{x}} Q$ if P is $\mathfrak{x}$-computable in Q, and that $P \equiv_{\mathfrak{x}} Q$ if $P \leq_{\mathfrak{x}} Q$ and $Q \leq_{\mathfrak{x}} P$. This is an equivalence relation. We define*

$$deg_{\mathfrak{x}}(Q) = [Q]_{\equiv_{\mathfrak{x}}} = \{P | P \equiv_{\mathfrak{x}} Q\} \tag{34}$$

and this induces an order relation in the degrees. $deg_{\mathfrak{x}}(P) \leq_{\mathfrak{x}} deg_{\mathfrak{x}}(Q)$ if and only if $P \leq_{\mathfrak{x}} Q$.
Also, we recursively define

$$\emptyset^{(n)}_{\mathfrak{x}} = \{a \in \emptyset^{(n-1)}_{\mathfrak{x}} | {\phi^{\emptyset^{(n-1)}_{\mathfrak{x}}}}_{{\#_{\mathfrak{x}}}^{\emptyset^{(n-1)}_{\mathfrak{x}}} a}(a) \downarrow\} \tag{35}$$

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

$$\mathbf{0}^{(n)}_{\mathfrak{x}} = deg_{\mathfrak{x}}(\emptyset^{(n)}_{\mathfrak{x}}) \tag{36}$$

Each representation $\mathfrak{x}$ induces a hierarchy

$$\mathbf{0}_{\mathfrak{x}} \leq_{\mathfrak{x}} \mathbf{0}'_{\mathfrak{x}} \leq_{\mathfrak{x}} \mathbf{0}''_{\mathfrak{x}} \leq_{\mathfrak{x}} \cdots \leq_{\mathfrak{x}} \mathbf{0}^{(n)}_{\mathfrak{x}} \leq_{\mathfrak{x}} \cdots \tag{37}$$

4.2 Representation degrees

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".

The connection between Turing degrees and representations comes through the so-called *representation degrees*, which apparently mimic Turing degrees, achieving the capacity to compute new functions through representation instead of oracles. In particular, we use the same general construction used in the presented alternative halting theorem representation. That is, we somehow include the answer to a specific problem we would like to compute in the very same representation.

Let $\chi_{\mathfrak{x}}^{(n)}$ be the characteristic function of the set $\emptyset_{\mathfrak{x}}^{(n)}$, and $\mathfrak{c}$ the standard representation. Define the class of representations $\mathfrak{u}^{(k)} : \mathbb{N} \to \mathcal{T}$ as

$$\mathfrak{u}^{(k)}(n) = \langle \mathfrak{c}(\chi_{\mathfrak{c}}^{(k)}(n)), \ldots, \mathfrak{c}(\chi'_{\mathfrak{c}}(n)), \mathfrak{c}(n) \rangle \tag{38}$$

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. We call this the *representation degree* of $\mathfrak{u}^{(k)}$:

$$\boldsymbol{\mathfrak{u}^{(k)}} = \mathrm{rdg}(\mathfrak{u}^{(k)}) = \mathfrak{M}([\mathfrak{u}^{(k)}]_{\overset{t}{\sim}}) \tag{39}$$

Lemma 4.1.

$$\boldsymbol{\mathfrak{u}^{(k)}} \subseteq \boldsymbol{\emptyset^{(k)}} \tag{40}$$

Proof. Let $f \in \boldsymbol{\mathfrak{u}^{(k)}}$; then there is a Turing machine 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) $\chi^{(k)} \in \boldsymbol{\emptyset^{(k)}}$, there is a Turing machine with oracle $\emptyset^{(k)}$ that can compute $\mathfrak{c}(\chi^{(k)})$ and, by the transitivity of the relation $\leq_{\mathfrak{c}}$, there are Turing machines with oracle $\emptyset^{(k)}$ that can compute $\emptyset', \ldots, \emptyset^{(k-1)}$; the bijection $\langle , \rangle$ is also computable, therefore there is a Turing machine $\overline{\phi^{\emptyset^{(k)}}}$ 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 $\chi^{(k)} \in \boldsymbol{\mathfrak{u}^{(k)}}$, but $\chi^{(k)} \notin \boldsymbol{\emptyset^{(k-1)}}$

Lemma 4.2.

$$\boldsymbol{\mathfrak{u}^{(k)}} - \boldsymbol{\emptyset^{(k-1)}} \neq \emptyset \tag{41}$$

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.

4.3 Relations between represented oracles

Consider the two representations $\mathfrak{u}$ and $\mathfrak{u}'$: they are representations of $\mathbb{N}$, which is c.e. under $\mathfrak{u}$, and they are not equivalent, since $\chi'_{\mathfrak{c}}$ is computable in the latter but not in the former. Therefore, by theorem 3.5, it must be $\mathfrak{u}||\mathfrak{u}'$–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 Turing machines with oracles.

Definition 4.4. *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 $\mathfrak{x} \overset{c}{\underset{R}{\sim}} \mathfrak{y}$, $\mathfrak{x} \overset{t}{\underset{R}{\sim}} \mathfrak{y}$ and the incomparability $\mathfrak{x}||\mathfrak{y}$ are defined in the obvious way.*

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

Theorem 4.2. *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) *$\mathfrak{x} \overset{t}{\underset{R}{\sim}} \mathfrak{y}$;*

iii) *$\mathfrak{x} \overset{c}{\underset{R}{\sim}} \mathfrak{y}$.*

From this we derive

Theorem 4.3. *Let $\mathfrak{x}$ and $\mathfrak{y}$ be two representations of a set A, then*

i) *if $\mathfrak{x}(A)$ or $\mathfrak{y}(A)$ are R-c.e., then either $\mathfrak{x} \underset{R}{\overset{c}{\sim}} \mathfrak{y}$ or $\mathfrak{x} \underset{R}{||} \mathfrak{y}$;*

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

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 both representations, therefore, by theorem 4.3, it must be $\mathfrak{u}^{(k)} \underset{\emptyset^k}{||} \mathfrak{u}^{(k+1)}$.

5 Conclusions

In this work we have paid attention to the fact that in order to answer the question of what is computable, one must create a computation formalism and must decide a way to represent the abstract elements on which the computability question was asked into the formalism. We have shown that this choice is not trivial, we have formalised this notion and considered and answered several questions regarding up to what point and in what manner this choice affects computability notions. In particular, the essential result of this work proves that representations which allow the computation of an enumerator of the set can never be strictly better or worse than any other representation, they must either be equivalent or incomparable.

We have also shown that any problem can be made computable by maing the right choice of representation, but in doing so, other problems may become incomputable. Consequently, we argue that a more rational approach than asking about the computability of a problem is asking about the co-computability of several problems (that is, what problems can be simultaneously computed in the same representation). In particular, the main result of the work can be thought to say that enumerators have unique co-computability. That is, given an enumerator, any representation which allows its computation will allow the computation of exactly the same problems.

The classic degrees of recursive unsolvability, one of the main topics of study in computability theory, are built on the idea of oracle machines, and offer a rich and deeply studied algebraic structure which we have shown to be mirrored, but not reached, by what we have called representation degrees.

In particular, there is an analogue hierarchy of representation degrees, totally included inside the original Turing machine formalism, which are built by forcing the same incomputable problems to be computable. However, while degrees of recursive unsolvability do not lose any computation power in the process, thus creating strictly greater degrees, representation degrees lose some computational power when forcing new problems to be computable.

An idea which goes one step further in relation with representation degrees and co-computability is the study of computability of functions and their relationship to representations (or, alternatively, co-computability with other functions) from a fully algebraic point of view. This point of view, unexplored in this work, might consider the definition of an abstract and precise concept of computational model and derive general results in a similar fashion to the study of groups, rings or vector spaces. For example, we believe that there exists some kind of duality between the specification of a representation and the choice of some basic functions (like the enumerator) to be computable, in the sense that the choice of one or the other are essentially the same thing, even when one is a specification on the basic symbols used and the other is a specification on the operations performed with those symbols. Therefore, the results presented in this work and the results in favour of the Church-Turing thesis could all be contained under the same general theory. We believe that research on these ideas would be highly valuable and would allow for a better scaffolding of computability theory that would make different points of view and different problems reunite and allow for a better overall understanding of the concept of computability.

Another highly interesting related topic of study is the effects of representation on complexity. This possible future work could as well be made on a common basis such as the algebraic definition suggested in the previous paragraph. We believe that, at least, this would allow for a better and more centralised understanding of these topics, and possibly help in the resolution of some of today's most challenging problems in theoretical computer science.

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] George S. Boolos, John P. Burgess, and Richard C. Jeffrey. *Computability and logic*. 1974.

[4] Juan Casanova and Simone Santini. On the relation between representations and computability (submitted). *International workshop on Mathematical foundations*, 40, 2015.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. 2002.

[7] S. C. Kleene and Emil L. Post. The upper semi-lattice of degrees of recursive unsolvability. *Annals of Mathematics*, 59(3):379–407, 1954.

[8] M. Lerman. Initial segments of the degrees of unsolvability. *Annals of Mathematics*, 93:365–89, 1971.

[9] P. Odifreddi. Reductibilities. In E. R. Griffor, editor, *Handbook of Computability Theory*, pages 89–119. Amsterdam:North-Holland, 1999.

[10] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50(641–2), 1944.

[11] H.G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[12] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society II*, 43:230–265, 1936.