MASTER'S THESIS

Challenging the frontiers of computability

On the legacy of the Church-Turing thesis and its significance in the twenty-first century

Author	Steven Veerbeek
Student number	3923363
Program	Artificial Intelligence
First examiner	Rosalie Iemhoff
Second examiner	Janneke van Lith

UTRECHT UNIVERSITY
The Netherlands

October 22, 2020

CONTENTS

Chapter 1

INTRODUCTION

The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.

— Kurt Gödel (1931)

[A] human calculator, provided with pencil and paper and explicit instructions, can be regarded as a kind of Turing machine.

— Alonzo Church (1937a)

As Kurt Gödel observed in the opening paragraph of his pioneering paper on incompleteness, the early twentieth century had seen an upsurge in projects that aimed to secure mathematical knowledge in formal axiomatic systems. Perhaps one of the most well-known attempts resulted in the comprehensive three-volume work *Principia Mathematica* (PM) by Bertrand Russell and Alfred North Whitehead in the years 1910–1913. But to strict formalists such as David Hilbert, PM lacked rigor and failed to adequately separate syntax from semantics. Hilbert believed that all mathematical thought should be entirely reduced to a "game" of mechanical manipulation of symbolic expressions, governed by a set of purely syntactical rules.

By the 1930s, a diversity of terminology had developed to describe the intuitive notion of finite and mechanical computation; but a precise mathematical definition was wanting. When in 1928, Hilbert and his student Wilhelm Ackermann posed

their famous *Entscheidungsproblem*, which asked for a general procedure that could decide within a finite amount of time whether any given formula of first-order predicate logic was valid or not, this catalyzed the search for a definition of computation or "effective calculation". Some eight years later, several authors more or less simultaneously proposed different formal models of computation that were subsequently proved to be equivalent. Two of these models, Alonzo Church's λ-*calculus* and Alan Turing's independently developed *Turing machines*, inspired what we know today as the *Church-Turing thesis*:

CHURCH-TURING THESIS (CTT) *Every function that can be computed by an idealized human being, provided with paper and pencil and explicit instructions, can be computed by a Turing machine.*[1]

What is often neglected in modern formulations of CTT is the explicit reference to the *human* aspect of computation. Remarkable as this may be to a twenty-first century reader, we should not forget that CTT originated in a time where the only association that people had with the word "computer" was that of a human worker performing manual calculations on paper. As a characterization of this original, human, interpretation of computation, CTT's validity is still virtually undisputed. However, much has changed in the world of computation since 1936 and so has our intuitive understanding of the concept.

The Church-Turing thesis was the first attempt at giving a precise and mathematical delineation of the absolute notion of computability. Furthermore, Turing showed that it should be possible to build one universal computing machine that embodied the most fundamental principles of computation and could be programmed to compute any computable function. This discovery would prove to be a pivotal moment in the history of computation, laying the foundations for entire new fields of study such as computer science and artificial intelligence. Once technological advancements had made it possible to turn the universal computer into a physical reality, we started liberating ourselves from tedious computational tasks by delegating them to our increasingly powerful and reliable electronic assistants.

As throughout the past decades the digital electronic computer gained ground at the expense of its human counterpart, an analogous shift took place in our minds: the concept of computation came to be associated primarily with machines rather than with humans. We could say that intensionally, the notion of computability has evolved significantly over the years. But what about its extension? Does the term "computable" still apply to the same class of functions as it did back in 1936?

Rapid increases in speed and efficiency have undeniably had a fundamental impact on the way we use and interact with computers. Through advancements in hardware technology and research in artificial intelligence, electronic computers have evolved from primitive number crunchers to increasingly complex and autonomous intelligent agents. However, despite speed and efficiency being important aspects of computation, we should not let these dramatic results draw our attention away

[1] Or, equivalently: is λ-definable.

from another fundamental question: can modern computers, apart from being much faster, compute essentially different things than we humans could theoretically do with paper and pencil? In fact, it is not at all implausible that the bounds within which modern computers operate are in essence still the ones found by Church and Turing in 1936. Even in the ever-changing world of computation, the Church-Turing has stood the test of time exceptionally well and is still used today as a benchmark for computational power. This makes the thesis relevant as ever, especially to cutting-edge fields such as artificial intelligence.

On the one hand, artificial intelligence is a groundbreaking field that has substantially changed our interaction with computers. Through new software approaches such as machine learning and natural language processing, behavior emerges that intuitively seems to transcend that of "classical" computers. On the other hand, this intelligent behavior ultimately relies on a set of elementary arithmetical and logical operations that can be realized by the most primitive Turing machine. As such, even the most sophisticated AI algorithms have brought us no closer to solving the halting problem than we were in 1936. It almost appears as if with their thesis, Church and Turing stumbled upon some magical barrier; a "law of nature" that is impossible to bypass.

Or could it be that this "Turing barrier" only exists in our minds? Perhaps there is a key waiting to be found that will unlock an entire new universe of possibilities. Perhaps more effort or new insights will one day lead us to this key and free us from the Turing tarpit. Would the discovery of super-Turing computation bring about an "intelligence explosion" that leads us to the legendary point of singularity? Does human intelligence even surpass the Turing barrier? If not, is it even possible for humans to create machine intelligence that does? Perhaps the Turing barrier is intrinsic to the human mind but not to nature in general, simply preventing us from recognizing the super-Turing intelligence that has been around us all along. It should be clear that the Church-Turing thesis is of fundamental importance to the field of artificial intelligence. The unsolved problem of whether super-Turing computation can ever be attained or not has deep implications for the future development of AI and should therefore be investigated within AI communities.

Since its inception, CTT has been the subject of ongoing debates in several scientific communities. Some feel that a critical border has already been crossed in the evolution of the computational paradigm since 1936, leaving CTT inadequate in today's context. As a result, several proposals have been made to "update" the original thesis to cover a wider and more modern notion of computation. Some of these proposals go as far as to question the adequacy of the Turing machine in modeling today's complex spectrum of computational processes, therefore replacing the model in its entirety or extending it to a possibly more powerful model of computation. Other, more moderate, proposals merely seek to extend the reach of the Turing machine by providing a more inclusive description of the intuitive notion of computation that, in addition to human computation, also applies to other types of computations, such as those performed by machines in general (e.g., Gandy 1980).

In this thesis, we will explore and discuss several challenges to the Church-

Turing thesis. First, we establish some terminology and notation in chapter 2. In chapter 3, we formalize the logical structure of the thesis and develop from it a framework that will serve as a guide for effectively analyzing and comparing different versions of the thesis in later discussions. In chapter 4, we further analyze the history leading up to the birth of CTT, followed by a discussion in chapter 5 of the responses that it has elicited from the scientific community. In chapter 6, we discuss three developments in modern computation that potentially pose a threat to CTT and analyze a model of computation and a corresponding extension of CTT that were proposed by Van Leeuwen and Wiedermann (2001a) in response to these developments. Finally, in chapter 7, we reflect on the contents of the present thesis, consider their implications for the status of CTT, and explore directions for further research.

Chapter 2

PRELIMINARIES

In this thesis, we will encounter a considerable number of mathematical notions, statements and definitions. I have chosen certain terminological and notational conventions for expressing mathematical concepts and objects, which I will explain in this chapter.

2.1 Terminology

In the literature on computability theory, it is not always clearly delineated what is meant by "(general) recursive functions". In certain contexts, the term is used to describe the class of *partial* recursive functions (also known as the μ-recursive functions). In other contexts it is used synonymously with *total* recursive functions, which are a subclass of the partial recursive functions. Likewise, the term "computable" is loosely used for both classes of functions. The difference between these classes is, however, absolutely crucial to the field of computability theory. Turing machines implementing the latter class of functions will halt on every input, while certain inputs may cause a Turing machine implementing a function of the former kind to loop forever. Therefore, in order to avoid any confusion, it is important to establish a clear and consistent terminology before starting our discussion.

In the remainder of this thesis, the terms "computable", "recursive", and "decidable" will be used exclusively to refer to those functions, numbers, sets, and procedures that can be computed or decided within a finite amount of time by some (total) Turing machine, or equivalently, can be computed by some *circle-free* Turing machine (see Turing 1936–7). On the other hand, there are functions that can be computed for only a subset of their domain. Analogously, there are sets of which membership can be proved for each member, but not necessarily disproved within a finite amount of time for each non-member. When describing these types of functions and sets we will use terms like "partially computable", "partial recursive", "recursively enumerable", "recognizable", and "semi-decidable". Table 2.1 presents a more comprehensive overview of this terminology.

	Functions	*Classes, relations*
PR	primitive recursive	primitive recursive
R	general recursive, (Turing) computable, (total) recursive	decidable, recursive
RE	partially computable, partial recursive, μ-recursive	semi-decidable, (Turing) recognizable, recursively enumerable (r.e.), computably enumerable (c.e.)
$\overline{\mathbf{RE}}$	non-computable, non-recursive	(Turing) unrecognizable

Table 2.1: Terminology used for describing levels of computability. Each row contains terms that are considered mutually equivalent in terms of computability. Note that $\mathbf{PR} \subset \mathbf{R} \subset \mathbf{RE}$; i.e., all total recursive functions are also partial recursive, and all decidable sets are also recognizable, etc.

2.2 Notation

When we speak of the set of natural numbers, written $\mathbb{N}$, we will usually mean the positive integers, i.e., excluding 0: $\{1, 2, 3, \ldots\}$. If we wish to make the exclusion or inclusion of 0 explicit, we will write $\mathbb{N}^+$ or $\mathbb{N}^0$, respectively. The ordinal number of $\mathbb{N}$ is written ω.

2.2.1 Models and instances

For some models of computation more than for others, the conventional terminology may not always clearly convey the distinction between the model as a general concept and its concrete instances that each perform a specific computational task. For example, the term "Turing machine" is used to denote the model as an abstract set of principles and rules, but it is also used to describe any concrete implementation of these principles and rules—i.e., an individual machine that is programmed to compute a particular function. In general, we will distinguish between *models* and *instances*. Thus, Turing machines, λ-calculus, and general recursive functions are models of computation whose instances are, respectively, Turing machines, λ-expressions, and Herbrand-Gödel style systems of equations. Models will be represented by regular uppercase Latin letters (M, N), while calligraphic uppercase Latin letters ($\mathcal{M}$, $\mathcal{N}$) will be used to denote individual instances of models. We may also regard a model as a class containing all its instances and write $\mathcal{M} \in M$ to say that $\mathcal{M}$ is an instance of the model M.

2.2.2 Strings

Strings will be typeset in a typewriter font. This allows us to easily distinguish between, for example, the integer 523021 and the string `523021`.

String representations

In discussions of computing machines or algorithms at higher levels of description, we are often not concerned with questions about data representation. It suffices to refer to abstract mathematical objects and operations on them without worrying about the implementational details. At lower levels of description, however, those details become more and more important. When developing a formal definition of a computing system, we will inevitably arrive at a point at which the use of conceptual descriptions such as "the integer 5" or "the square root of 5" simply no longer suffices. Ultimately, the operations of a computing machine or formal system only consist of the manipulation of symbolic expressions. These symbols and their concatenations into strings have no intrinsic meaning; meaning only comes into play when we start interpreting them.

The most common way to represent information in computing systems is by binary digits (or bits). In fact, in expounding his original theory of the Turing machine, Turing chose to encode information on the machine's tape in binary format. In principle we could use any set of symbols we like, but for convenience we will usually assume that a machine (either physical or theoretical) stores information in binary format, i.e., as strings over the alphabet $\{\texttt{0},\texttt{1}\}$. We can safely make these restrictive assumptions on the size of the alphabet without losing computational power, as was proved in Shannon (1956).

While algorithms usually live at a higher level of description than their implementations in specific machines, it is not uncommon in information processing tasks to address the problem of data representation already before writing the algorithm. As Marr (1982, p. 23) points out, the choice of algorithm depends critically on the particular representation of input and output. Furthermore, taking into account the representational aspect when specifying an algorithm may expose certain difficulties early on that could otherwise go unnoticed until they cause complications at a later stage. For example, while in a high-level specification of an algorithm there is no need to explicitly distinguish between different types of numbers, such as 5 and $\sqrt{5}$, at the representational and implementational level they require fundamentally different strategies. On the one hand, 5 is an integer and can perfectly and completely be represented by the binary string `101`. $\sqrt{5}$, on the other hand, is a real number and cannot be directly[1] represented in a finite way. One can represent a finite approximation, such as `10.00111100`, but this will always be imperfect.

Since a string does not have any intrinsic meaning, the behavior of one algorithm can have different interpretations depending on the way we choose to interpret

[1] Some real numbers—namely, those that we call computable—can however be represented indirectly in a finite way by using a string description of an algorithm that computes them.

strings. In fact, we can make algorithms manipulate any type of object or combination of objects, simply by representing it appropriately in string format so an algorithm can process it. Whenever we want to refer to the string representation of an object O, we will write $\langle O \rangle$. We can also represent multiple objects $O_1, \ldots, O_k$ in a single string by writing $\langle O_1, \ldots, O_k \rangle$.[2] In the context of a specific formal system, we may use the notation $\overline{x}$ to denote the formal numeral that, according to either the standard interpretation of that system or some explicitly specified interpretation, represents the number x in that system.

We are usually not concerned with the particular encoding that is used; when relevant this will be evident from the context. The only requirements we set for such encodings is that they are computable and injective (i.e., every object is associated with a unique string), such that a string representation is always decodable to an effective definition of the original object.

String operations

Given a string (or sequence) s of length $l \leq \omega$ and some $n \in \mathbb{N}$ such that $1 \leq n \leq l$, let s_n denote the n-th character (element) of s. Furthermore, let s_i^j denote the finite substring (subsequence) of s ranging from the i-th character (element) up to and including the j-th, where $1 < j \leq l$ and $1 \leq i < j$.

For a given finite alphabet Σ, we let Σ^* (the *Kleene closure* of Σ) and Σ^ω denote the sets of finite and infinite strings over Σ, respectively. Additionally, we let Σ^∞ denote the set $\Sigma^* \cup \Sigma^\omega$ that contains both all finite and all infinite strings over Σ. We can apply these operators not only to sets of symbols (alphabets), but also to sets of finite strings. $(\Sigma^*)^*$, for example, yields the set of finite sequences of finite strings over Σ, while $(\Sigma^*)^\infty$ yields the set of both finite and infinite sequences of finite strings over Σ.

As with individual objects, we will need to represent sets of objects as sets of strings before we can manipulate them in a useful way. We will represent a set S by the language L_S that consists of the finite string representations of the members of S:

$$L_S = \{ \langle x \rangle \mid x \in S \} \tag{2.1}$$

Since recognizer or decider algorithms can only be directly applied to languages, we will focus our discussion of decidability and recognizability on this specific kind of sets, while bearing in mind that languages can represent a much wider variety of sets.

2.2.3 Gödel numbers

As Kurt Gödel famously showed in the development of his incompleteness theorems (Gödel 1931), it is possible to associate each symbol of and each word over the alphabet of a formal language with a unique natural number, such that given this natural number, one can always reconstruct from it the original string. We

[2] Notation due to Sipser (2013, p. 185).

will use this principle in this thesis, writing $\ulcorner s \urcorner$ to denote the *Gödel number* of the expression s. We do not need to make any assumptions on the specific function that is used to realize a Gödel numbering, other than that it is computable and that there exists an effective procedure by which it can be determined for any natural number whether it is the Gödel number of some expression and, if so, of which expression.

The definition of a Gödel numbering requires two functions. The first associates each symbol of the alphabet with a unique natural number. Thus, strings over the alphabet will be represented by sequences of natural numbers. The second function maps these sequences of natural numbers again to unique natural numbers. By disregarding the first function we can directly apply Gödel numberings to sequences of natural numbers. We will write $\ulcorner x_1, \ldots, x_n \urcorner$ for the Gödel number of the sequence $(x_1, \ldots, x_n)$. Conversely, $\mathring{n}$ stands for the sequence of natural numbers of which n is the Gödel number. If n is not a Gödel number of any sequence, $\mathring{n}$ is the empty sequence and has length 0. It is possible by the methods of Gödel (1931, p. 182) to effectively obtain $\mathring{n}$ for any $n \in N$.

2.2.4 Enumerations

A set is *recursively* (or *computably*) *enumerable* if and only if it is the image of a computable function. For a given set S, this means that there exists a (total) computable function $\sigma\colon \mathbb{N} \longrightarrow S$ that is also surjective:

$$(\forall x \in S)(\exists n)\,\big[\sigma(n) = x\big]. \tag{2.2}$$

Listing the values of σ (i.e., producing the list $\sigma(1), \sigma(2), \ldots$) is an effective enumeration procedure for S. While such an enumeration may contain repetitions (as σ need not be injective), every element of S will eventually be listed by it. We will therefore say that σ is an enumeration of S. Often, we will treat enumeration functions as sequences, writing their arguments in subscript; i.e., $\sigma_1, \sigma_2, \ldots$ instead of $\sigma(1), \sigma(2), \ldots$.

2.2.5 Characteristic functions and sequences

The *characteristic* (or *indicator*) *function* of a set $S \subseteq A$ is the function $\mathbf{1}_S\colon A \longrightarrow \{0, 1\}$ that maps every member of A to 1 if it is in S, and to 0 if it is not:

$$\mathbf{1}_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases} \tag{2.3}$$

Let α be an enumeration of A. The binary sequence that is obtained by replacing all strings in this ordering by their image under $\mathbf{1}_S$ is called the *characteristic sequence* χ_S of S:

$$\chi_S = \big(\mathbf{1}_S(\alpha_1), \mathbf{1}_S(\alpha_2), \ldots\big) \tag{2.4}$$

The i-th digit of a characteristic sequence indicates whether α_i is in S or not. If, for example, we consider the set of prime numbers in their usual non-decreasing order (i.e., $\{2, 3, 5, 7, 11, \ldots\}$), its characteristic sequence would be $01101010001\ldots$.

A formal language being a special type of set, the characteristic function of a language L takes words over the language's alphabet Σ and maps these to 0 or 1 according as they belong to L or not.

A k-ary relation $R \subseteq S_1 \times \ldots \times S_k$ may be considered as a set of k-tuples, which lets us define the characteristic function[3] of R as follows:

$$\mathbf{1}_R(x_1, \ldots, x_k) = \begin{cases} 1 & \text{if } R(x_1, \ldots, x_k) \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

Given enumerations of the individual sets $S_1, \ldots, S_k$, we can define a function σ that is an enumeration of $S_1 \times \ldots \times S_k$. The characteristic sequence of R is then given by:

$$\chi_R = \big(\mathbf{1}_R(\sigma_1), \mathbf{1}_R(\sigma_2), \ldots\big) \tag{2.6}$$

Similarly, we define the characteristic function of a function to be the characteristic function of its graph:

$$\mathbf{1}_f(x_1, \ldots, x_k, y) = \begin{cases} 1 & \text{if } f(x_1, \ldots, x_k) = y \\ 0 & \text{otherwise} \end{cases} \tag{2.7}$$

The characteristic sequence of a function f is obtained by applying the same principles to the graph of f as were applied to R in eq. (2.6).

[3] Gödel (1934) uses the term "representing function" for the same concept (p. 2), with the conventional difference that the meanings of the output symbols 0 and 1 are precisely opposite to the ones assigned here.

Chapter 3

ANALYTICAL FRAMEWORK

Despite the strong position that it has acquired within the theory of computation since its inception in 1936, the Church-Turing thesis has received a fair share of criticism. It has been extended, complemented, reformulated and replaced, all of which resulted in related, but slightly different theses. Before we dive in and risk finding ourselves tangled up in the complex web of CTT statements, let us take a moment to ensure that we have a proper understanding of the terrain we are about to explore.

3.1 Logical structure

Often, there is more to a CTT version than is immediately observable from the concise statement by which it is introduced. Any claim relies on an implicit framework of assumptions and definitions. To properly appreciate a claim, it is therefore crucial to gain insight into the relevant considerations of the author. While superficial differences between CTT versions tend to catch the eye, a closer examination might reveal more fundamental differences due to subtle but critical nuances in these underlying assumptions and definitions. In order to facilitate an effective analysis and comparison of CTT versions, we will here develop a framework in which we can comprehensively yet concisely represent any version.

When we abstract away from the specifics of individual instances, we can identify a logical structure common to most CTT versions, as expressed in the following formula of first-order logic:

$$(\forall x \in D)\left[U(x) \rightarrow (\exists \mathcal{M} \in M)\left[C(\mathcal{M}, x)\right]\right] \tag{3.1}$$

Beside laying bare the basic reasoning structure of CTT, this "skeleton" provides placeholder predicates that represent the defining elements of any CTT version. The task of representing a specific version is then reduced to that of "filling in" the template by providing relevant definitions according to the following scheme:

- D represents the domain over which x ranges. Some CTT versions define computability of functions, while others focus their definition on formal languages, real numbers, relations, algorithms, or some type of computational process. Each choice is modeled by a different instantiation of D.
- U stands for some property defined over D. The specificity and degree of formality in which this property is defined can vary from some vague, intuitive notion (e.g., "effective calculability") to a more precise, mathematical characterization.
- M represents some model of computation, or, equivalently, the class of all instances of that model (e.g., the class of all Turing machines, or the class of all lambda expressions).
- $C(\mathcal{M}, x)$ defines the relation "$\mathcal{M}$ computes x". For a given CTT version, C reflects the author's account of computation with respect to any object x from the domain D, but is indifferent to the specific model M of which $\mathcal{M}$ is an instance.

The common part of most versions of CTT is that the computational power of some (more or less intuitive) notion of computation is identified with that of some formal model of computation. The above formula will serve throughout this thesis as a "skeleton" CTT that captures this basic reasoning structure and can be differentiated into any concrete CTT version by assigning specific interpretations to the predicates. This method of carefully distilling and isolating each aspect of a given CTT version provides an accurate and systematic way of comparing it to other versions. It allows us to easily pinpoint the exact aspect(s) at which two versions differ (i.e., at D, U, M, and/or C). In the following sections, the significance of each of the variables of our framework is discussed.

3.2 Domains

The domain D on which a computability statement is based can differ between versions of CTT. Some versions express the computational power of a model in terms of the class of formal languages that it can decide. Others characterize this property by the class of functions the model can compute, or the set of real numbers of which it can produce the decimals. How do these statements relate to each other? Do they concern different types of computability, or are they reconcilable? In this section, we will examine the role of the domain in a CTT version and try to determine the relationship between CTT versions with different domains.

3.2.1 θ-translatability

While at a superficial level, an algorithm that computes a real number differs considerably from a procedure that decides a formal language, the principles that guide both processes are, as we will soon see, essentially the same. In fact, there

often exist very natural ways of associating different CTT domains with one another in such a way, that a computable (decidable) object in one domain always has a unique computable (decidable) counterpart in the other domain. Such associations allow us to indirectly decide formal languages using models of computation whose domain is the class of integer functions or that of the real numbers, and vice versa. In literature, these relationships are often simply assumed without explicit verification. By taking the effort to make these intuitions more precise, we will see that they are not always entirely trivial and may require explicit proof.

For this purpose, we will make adaptations to two existing definitions and introduce two new definitions. First, whereas the existing notions of *Turing reducibility* (Post 1944) and *Turing equivalence* are traditionally defined between two objects of the same class (e.g., two formal languages), we will for our purposes define generalizations of these concepts that can exist between objects of different types. Our definition, for example, allows for a formal language to be Turing reducible to a function of natural numbers. Second, we introduce the notion of *θ-translatability* (and the derived notion of *θ-equivalence*), a type of relative computability defined "en masse" between the members of two classes of mathematical objects rather than between individual objects. We will for example see that the *class* of all formal languages is θ-equivalent to the *class* of all functions on natural numbers.

In our definitions we make use of the notion of *oracles*, which were introduced by Turing (1939). A P-oracle is a hypothetical problem solver that for a given computational problem[1] P (either computable or not) correctly responds to all queries regarding P. An *oracle machine* is a Turing machine that has an oracle at its disposal, which it may consult at any time during computation and whose answers it may use freely. Since an oracle may introduce non-computable information into the process of computation, the input-output relation of an oracle machine need not be computable. Apart from these oracle calls however, the machine behaves like a regular Turing machine.

Furthermore, since an oracle is an external entity whose behavior is not determined by the program of the machine, the input-output behavior of one machine may vary depending on which oracle is assigned to it. If $\mathcal{M}^{\varnothing}$ denotes an oracle machine with an "empty" oracle (i.e., a complete Turing machine whose program specifies all interactions with an oracle but where the oracle itself has yet to be specified), $\mathcal{M}^{P}$ and $\mathcal{M}^{Q}$ denote two concrete machines that result from assigning to $\mathcal{M}$ a P-oracle and a Q-oracle, respectively.

Note that in this section we may use the terms "compute" and "computable" in a broad sense to include the meanings of a number of related terms, such as "decide", "decidable", "recursive". When a model has the computational power to compute all *computable* (not necessarily *all*) objects from some domain D, we will refer to such a model as a "D-model". When calling some model a D-model, this will always be under the assumption that its procedures are effective, in the sense

[1] In this context, by "computational problem" we mean problems that ask for a general procedure rather than a specific solution; e.g., the general problem of deciding a set S or that of computing a function f, as opposed to deciding propositions "$x \in S$" for specific objects x or computing specific values $f(x)$, respectively.

that they can be executed in practice and do not make use of any purely theoretical tools such as oracles.

We will now present a generalized definition of Turing reductions.

DEFINITION 1 *Turing reduction*
A *Turing reduction* from a problem P to a problem Q is a machine $\mathcal{M}^Q$ (i.e., an oracle machine with a Q-oracle) that computes P.

From this definition, we derive the following definition of Turing reducibility and Turing equivalence.

DEFINITION 2 *Turing reducibility and Turing equivalence*
A problem P is *Turing reducible* to a problem Q, written $P \leq_T Q$, if there exists a Turing reduction from P to Q.
If both $P \leq_T Q$ and $Q \leq_T P$, the problems are *Turing equivalent*, written $P \equiv_T Q$.

Instead of saying that "the decision problem of a set S" is Turing reducible to "the problem of computing a function f" we will often simply state that S is Turing reducible to f, or alternatively that S is f-computable. An S-oracle, for a given set $S \subseteq \mathbb{N}$, is then an oracle that on input $n \in \mathbb{N}$ responds with "yes" or "no" (or, equivalently, `1` or `0`) according as $n \in S$ or not. An f-oracle, for a given function $f\colon \mathbb{N}^k \longrightarrow \mathbb{N}$, is an oracle that for every input tuple $(n_1, \ldots, n_k)$ returns the value of $f(n_1, \ldots, n_k)$. As a final example, a γ-oracle, for a given infinite sequence $\gamma \in \{\mathtt{0}, \mathtt{1}\}^\omega$, is an oracle that on input $i \in \mathbb{N}$ returns the digit γ_i, or, alternatively, the entire sub-sequence γ_1^i.

We will now introduce θ-translations, which establish a "deep" one-to-one correspondence between the objects in one class and a subset of the objects in another class, such that one universal series of instructions converts any computational procedure for an object in the first class to a computational procedure for its image in the second class, and vice versa.

DEFINITION 3 *θ-translation*
A *θ-translation* is an injective mapping $\theta\colon A \longrightarrow B$ between two classes of mathematical objects A and B, satisfying the following inverse conditions:

1. There exists an oracle machine $\mathcal{U}^\varnothing$ such that $\forall a \in A$, $\mathcal{U}^a$ computes $\theta(a)$.
2. There exists an oracle machine $\mathcal{R}^\varnothing$ such that $\forall a \in A$, $\mathcal{R}^{\theta(a)}$ computes a.

In simple terms, a θ-translation together with the definitions of two "reduction machines" $\mathcal{U}^\varnothing$ and $\mathcal{R}^\varnothing$ explains how, by fixed effective[2] procedures, every object in

[2] That is, the programs of the reduction machines $\mathcal{U}^\varnothing$ and $\mathcal{R}^\varnothing$ are effective. Of course, it still depends on the degree of computability of a and $\theta(a)$ whether the global input-output relations of $\mathcal{U}^a$ and $\mathcal{R}^{\theta(a)}$ are computable or not.

one class A can be (i) matched with a unique object in a second class B, and (ii) computed relative to this object in B. One of the consequences of conditions 1 and 2 is that the objects that a θ-translation associates with one another will always be Turing equivalent, i.e., $a \equiv_T \theta(a)$ for every $a \in A$. However, the two conditions go further than that. While Turing equivalence is already satisfied when for every $a \in A$ there exists *some* Turing reduction to $\theta(a)$ (and vice versa), it is essential to the definition of a θ-translation that two "universal" Turing reductions $\mathcal{U}^{\varnothing}$ and $\mathcal{R}^{\varnothing}$ suffice to reduce every object to its image and vice versa.

Why would it be undesirable to allow the Turing equivalences between objects and their images to be realized by a diverse, non-uniform set of Turing reductions? This becomes particularly clear in cases where the objects in question are computable. The relation $a \leq_T b$, and thus the existence of an oracle machine $\mathcal{M}^b$ that computes a, holds for any b if a is computable. Instead of relying critically on its oracle for computing a, such a machine can simply ignore its oracle altogether and compute a "from scratch". While this technically qualifies as a reduction from a to b, in reality the procedure is entirely independent from b. In other words, this approach potentially leaves the computation of A-objects decoupled from that of B-objects, telling us nothing about how computability statements regarding both classes relate to each other.

By demanding the existence of two universal reduction machines, we enforce that θ-translations do not just associate objects with random objects of the same Turing degree, but that a sort of isomorphism exists between every object and its image and that the reduction procedure for each object truly depends on an oracle of its image, with the procedure itself being nothing more than a universal "conversion shell".

DEFINITION 4 *θ-translatability and θ-equivalence*

A class A is *θ-translatable* to a class B, written $A \leq_\theta B$, if there exists a θ-translation from A to B.
If both $A \leq_\theta B$ and $B \leq_\theta A$, the classes are *θ-equivalent*, written $A \equiv_\theta B$.

Note that a θ-equivalence can only exist between two classes of the same cardinality. However, to prove a θ-equivalence between two classes it is not required that a bijective θ-translation exists between them. In section 3.2.2 we will see examples where θ-equivalences are proved by demonstrating the existence of two distinct non-surjective θ-translations.

THEOREM 1

If $A \leq_\theta B$, any B-model can be converted into an A-model.

Proof. We make the following observations:

1. Since $A \leq_\theta B$, there exists a θ-translation $\theta\colon A \longrightarrow B$ and an oracle machine $\mathcal{R}^{\varnothing}$ that can be made to compute any $a \in A$ by "plugging in" a $\theta(a)$-oracle.

2. If some model of computation M is a B-model, this means that for every computable $b \in B$ there exists an instance $\mathcal{M}_b \in M$ that computes b.

3. By definition, $\theta(a) \in B$ is computable if and only if a is computable.

Now define the following class of machines:

$$N = \{\mathcal{R}^{\mathcal{M}} \mid \mathcal{M} \in M\},$$

where $\mathcal{R}^{\mathcal{M}}$ is the reduction machine described in observation 1, but with $\mathcal{M}$ incorporated as an actual subroutine rather than an oracle. As a result, the machines in N all represent procedures that can be effectively[3] carried out.

From observations 2 and 3 it follows that for every computable $a \in A$ there exists an instance $\mathcal{M}_{\theta(a)} \in M$ that computes $\theta(a)$. This, in combination with the definition of N, means that for every computable $a \in A$ there exists a machine $\mathcal{R}^{\mathcal{M}_{\theta(a)}} \in N$ that computes a. To sum up: by injecting each procedure from M into a uniform "conversion shell", we have obtained a new class of procedures N that constitutes an A-model. □

Based on theorem 1, we can draw the following conclusion about the role of the domain in a CTT version: for any two classes of mathematical objects A and B where $A \equiv_T B$, it does not essentially matter whether a CTT version's domain is A or B, as any A-model can be converted into a B-model, and vice versa.

3.2.2 Some proofs of θ-equivalence

In this section we prove the mutual θ-equivalence of the following three domains:

$$\Gamma := \{0,1\}^{\omega} \qquad \text{(infinite binary sequences)}$$

$$\Phi := \{f : \mathbb{N}^k \longrightarrow \mathbb{N} \mid k \in \mathbb{N}\} \qquad \text{(total functions on natural numbers)}$$

$$\Pi := \mathcal{P}(\mathbb{N}) \qquad \text{(subsets of the natural numbers)}$$

The proof consists of defining a θ-translation $\theta^A_B : A \longrightarrow B$ between each ordered pair (A, B) of these domains.[4] We will only do this for pairs where $A \neq B$, since any domain is trivially θ-equivalent with itself. The relational structure of the statements below is visualized in fig. 3.1.

---

1. $\Phi \equiv_\theta \Pi$.

   Proof. We will prove both $\Phi \leq_\theta \Pi$ and $\Pi \leq_\theta \Phi$.

[3] Depending on whether the architecture of the model M is Turing machine-based or not, the subroutine $\mathcal{M}$ may or may not be directly executable as an integral part of $\mathcal{R}$. However, even if the execution of the subroutine requires a temporal transfer of control to an external procedure that realizes $\mathcal{M}$, we may, since M was defined as an A-model, assume that its procedures are effective and, thus, that every procedure $\mathcal{R}^{\mathcal{M}}$ as a whole is effective.

[4] When it is clear from context which particular mapping is denoted, we will omit the subscript and superscript and simply write θ.

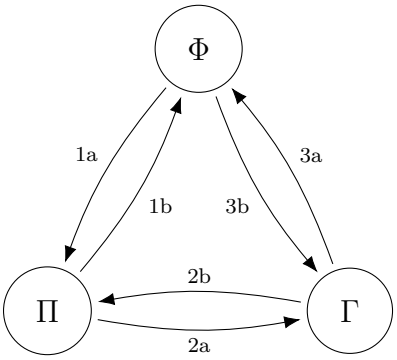

Figure 3.1: An arrow from class A to class B expresses the proposition $A \leq_\theta B$. Each arrow is labeled with a number, referring to the corresponding proof.

a) $\Phi \leq_\theta \Pi$.
Define the θ-translation θ^{Φ}_{Π} from total functions of natural numbers to sets of natural numbers:

$$\theta^{\Phi}_{\Pi}(f) = \{\ulcorner n_1, \ldots, n_k, m \urcorner \mid f(n_1, \ldots, n_k) = m\} \tag{3.2}$$

where k is the arity of f.

There exists an oracle machine $\mathcal{U}^{\varnothing}$ that for every k-ary function $f \in \Phi$, when supplied with an f-oracle, decides the set $\theta(f)$ as follows:

$$\mathcal{U}^{f}(n) = \begin{cases} \textit{accept} & \text{if } |\mathring{n}| = k+1 \wedge f(\mathring{n}_1, \ldots, \mathring{n}_k) = \mathring{n}_{k+1} \\ \textit{reject} & \text{otherwise} \end{cases} \tag{3.3}$$

Here, $\mathring{n}$ denotes the sequence of natural numbers of which n is the Gödel number (see section 2.2.3).

Conversely, an oracle machine $\mathcal{R}^{\varnothing}$ can be constructed that, when supplied with a $\theta(f)$-oracle, computes f as follows:

$$\mathcal{R}^{\theta(f)}(n_1, \ldots, n_k) = \mu m \left[\ulcorner n_1, \ldots, n_k, m \urcorner \in \theta(f)\right], \tag{3.4}$$

where μ is the least number (or minimization) operator. This characterization is guaranteed to yield a value for every k-tuple of natural numbers within a finite amount of steps and is therefore effective,[5] since, owing to the fact that f is a total function, the following holds:

$$(\forall n_1, \ldots, n_k) \left[(\exists m) \left[\ulcorner n_1, \ldots, n_k, m \urcorner \in \theta(f)\right]\right]. \tag{3.5}$$

[5] Of course, these and the following oracle machine definitions are only effective *relative* to the oracle in question. Only in cases where the object represented by the oracle is itself computable, will the procedure as a whole be computable (since the oracle would be unnecessary and could be replaced with a subroutine that actually computes the object).

b) $\Pi \leq_\theta \Phi$.
Define the mapping θ^{Π}_{Φ} from sets of natural numbers to total functions of natural numbers:

$$\theta^{\Pi}_{\Phi}(S) = \mathbf{1}_S. \tag{3.6}$$

Every set $S \subseteq \mathbb{N}$ is simply associated with its characteristic function. I trust that the immediate and intuitive nature of this particular reducibility relation allows me to leave the rest of its proof implicit.

□

---

2. $\Pi \equiv_T \Gamma$.

Proof. We will prove both $\Pi \leq_\theta \Gamma$ and $\Gamma \leq_\theta \Pi$.

a) $\Pi \leq_\theta \Gamma$.
Define the mapping θ^{Π}_{Γ} from sets of natural numbers to infinite binary sequences:

$$\theta^{\Pi}_{\Gamma}(S) = \big(\mathbf{1}_S(1), \mathbf{1}_S(2), \ldots\big). \tag{3.7}$$

Every set $S \subseteq \mathbb{N}$ is associated with its *characteristic sequence* χ_S (see section 2.2.5).

An oracle machine $\mathcal{U}^{\varnothing}$ exists that for every set $S \subseteq \mathbb{N}$, when supplied with an S-oracle, computes the sequence $\theta(S)$ by an infinite process that, starting with $i = 1$, consists of printing a `1` or a `0` on the i-th square of the machine's tape according as its oracle confirms or denies that $i \in S$, incrementing i by 1, and repeating the process forever.

Conversely, an oracle machine $\mathcal{R}^{\varnothing}$ can be constructed that, when supplied with a $\theta(S)$-oracle, decides S in the following manner:

$$\mathcal{R}^{\theta(S)}(n) = \begin{cases} \textit{accept} & \text{if } \theta(S)_n = \mathtt{1} \\ \textit{reject} & \text{otherwise} \end{cases} \tag{3.8}$$

b) $\Gamma \leq_\theta \Pi$.
Define the mapping θ^{Γ}_{Π} from infinite binary sequences to sets of natural numbers:

$$\theta^{\Gamma}_{\Pi}(\gamma) = \{\, i \in \mathbb{N} \mid \gamma_i = \mathtt{1} \,\}, \tag{3.9}$$

For every binary sequence γ, the set $\theta(\gamma)$ consists of all indices of γ that hold a `1`.

There exists an oracle machine $\mathcal{U}^{\varnothing}$ that for every sequence $\gamma \in \Gamma$, when supplied with a γ-oracle, decides the set $\theta(\gamma)$ as follows:

$$\mathcal{U}^{\gamma}(n) = \begin{cases} \textit{accept} & \text{if } \gamma_n = \mathtt{1} \\ \textit{reject} & \text{otherwise} \end{cases} \tag{3.10}$$

Conversely, we can construct an oracle machine $\mathcal{R}^{\varnothing}$ that, when supplied with a $\theta(\gamma)$-oracle, computes γ by an infinite process that, starting with $i = 1$, consists of printing a `1` or a `0` on the i-th square of the machine's tape according as its oracle confirms or denies that $i \in \theta(\gamma)$, incrementing i by 1, and repeating the process forever.

□

---

3. $\Gamma \leq_T \Phi$.

Proof. We will prove both $\Gamma \leq_\theta \Phi$ and $\Phi \leq_\theta \Gamma$.

a) $\Gamma \leq_\theta \Phi$.
An adequate mapping θ^{Γ}_{Φ} from infinite binary sequences to total functions of natural numbers can be obtained by composition of the previously defined functions θ^{Γ}_{Π} and θ^{Π}_{Φ}:

$$\theta^{\Gamma}_{\Phi} = \theta^{\Pi}_{\Phi} \circ \theta^{\Gamma}_{\Pi}. \tag{3.11}$$

There exists an oracle machine $\mathcal{U}^{\varnothing}$ that for every sequence $\gamma \in \Gamma$, when supplied with a γ-oracle, computes the function $\theta(\gamma)$ as follows:

$$\mathcal{U}^{\gamma}(n) = \gamma_n. \tag{3.12}$$

Conversely, an oracle machine $\mathcal{R}^{\varnothing}$ can be defined that, when supplied with a $\theta\gamma$-oracle, computes γ by an infinite process that, starting with $i = 1$, consists of printing on the i-th square of the machine's tape the value of $\theta(\gamma)$ that it receives from its oracle for argument i, incrementing i by 1, and repeating the process forever.

b) $\Phi \leq_\theta \Gamma$.
An adequate mapping θ^{Φ}_{Γ} from total functions of natural numbers to infinite binary sequences can be obtained by composition of the previously defined functions θ^{Φ}_{Π} and θ^{Π}_{Γ}:

$$\theta^{\Phi}_{\Gamma} = \theta^{\Pi}_{\Gamma} \circ \theta^{\Phi}_{\Pi}. \tag{3.13}$$

There exists an oracle machine $\mathcal{U}^{\varnothing}$ that for every k-ary function $f \in \Phi$, when supplied with an f-oracle, computes the sequence $\theta(f)$ by an infinite process that, starting with $i = 1$, consists of printing on the i-th square of the machine's tape a digit c_i as determined by the formula in eq. (3.14), incrementing i by 1, and repeating the process forever.

$$c_i = \begin{cases} \mathtt{1} & \text{if } |\mathring{i}| = k+1 \wedge f(\mathring{i}_1, \ldots, \mathring{i}_k) = \mathring{i}_{k+1} \\ \mathtt{0} & \text{otherwise} \end{cases} \tag{3.14}$$

Conversely, an oracle machine $\mathcal{R}^{\varnothing}$ can be constructed that, when supplied with a $\theta(f)$-oracle, computes f as follows:

$$\mathcal{R}^{\theta(f)}(n_1, \ldots, n_k) = \mu m \left[\theta(f)_{\ulcorner n_1, \ldots, n_k, m \urcorner} = \mathtt{1}\right] \tag{3.15}$$

As was the case with eq. (3.4), the fact that f is a total function guarantees the effectiveness of this definition.

□

The mutual associability among these classes shows us that the notion of computability defined by a version of CTT is not necessarily restricted to any particular domain. The notions of decidability of a set, computability of an integer function, and computability of an infinite binary sequence are all translatable into each other and are therefore merely different manifestations of one common, universal notion of computability. The particular type of mathematical objects in terms of which an author chooses to express the computational power of a model often depends on the relevant context. As different as the resulting CTT versions may look on the surface, they may still express the same notion of computability. This can however only be properly assessed if we manage to reconcile the different domains by establishing an appropriate mapping between them, as we did for the three domains above by means of θ-translations.

3.2.3 Further extensions of the equivalence class

Of course, the three classes whose θ-equivalence we just proved are by no means the only members of their equivalence class. While these θ-equivalences are particularly intuitive and easy to prove, we may come up with equivalence proofs for several other classes of mathematical objects. We will briefly discuss a number of candidates here.

Sets of finite sequences of natural numbers

First, consider the class Υ of all sets of finite natural number sequences:

$$\Upsilon = \mathcal{P}\Big(\big\{(n_1, \ldots, n_k) \;\big|\; k \in \mathbb{N} \wedge n_i \in \mathbb{N}\big\} \cup \{\varnothing\}\Big). \tag{3.16}$$

It is easily seen that $\Upsilon \equiv_\theta \Pi$. By associating every set of natural number sequences with the set of Gödel numbers of these sequences we obtain a θ-translation that proves $\Upsilon \leq_\theta \Pi$. Conversely, $\Pi \leq_\theta \Upsilon$ can be proved by associating every subset S of the natural numbers with the set of singleton sequences $\{(n) \mid n \in S\}$.

Formal languages

Intuitively, a natural extension can then be made from sets of natural number sequences to formal languages by associating symbols with natural numbers and words with sequences of natural numbers. This however raises a problem: while an individual formal language is usually defined over a finite alphabet, there is no upper bound to the size of an alphabet and thus we must assume that the set of all distinct symbols one can choose from—or the union of all possible alphabets—is infinite in size. In order to be able to successfully associate each symbol with a

unique natural number, this set must be recursively enumerable. As far as the author is aware, no algorithm is available that produces such an enumeration.

We could of course adopt a more pragmatic strategy and choose to consider only the class of formal languages whose alphabet is (or can be replaced with) a subset of, say, the (finite) set of Unicode characters. Let us write Λ_U to denote this class. By associating each Unicode character with a unique natural number and thus each Unicode string with a unique natural number sequence, and conversely each natural number sequence with a unique Unicode string (e.g., the sequence $(129, 5, 70, 82654)$ with the string $(\texttt{129}, \texttt{5}, \texttt{70}, \texttt{82654})$), we can easily prove that $\Lambda_U \equiv_\theta \Upsilon$ (and thus that $\Lambda_U \equiv_\theta X$ for all $X \in \{\Gamma, \Phi, \Pi, \Upsilon\}$).

Real numbers

Another intuitive direction for expansion of the equivalence class is from infinite binary sequences to real numbers. In fact, Alan Turing used binary sequences to represent real numbers on the tape of his Turing machine. Strictly speaking however, Turing's way of associating binary sequences with real numbers lacks the uniqueness that is required of a θ-translation—in either direction. First, Turing only represented the fractional parts of real numbers, meaning that many real numbers were associated with the same binary sequence. This can easily be resolved by adopting a different strategy for associating real numbers with binary sequences such that the integer part of a number is included in its representation. Such a mapping allows us to prove $\mathbb{R} \leq_\theta \Gamma$. In practice of course, Turing's notation often suffices for the computation of real numbers in the form of infinite binary sequences.

Conversely, Turing's representation does not associate every binary sequence with a unique real number, in the sense that every real number is represented by two distinct sequences. As counter-intuitive as it may have appeared to us when we were first confronted with it, we have all come to accept that the numerals $\texttt{0.}\overline{\texttt{9}}$ (where $\overline{\texttt{9}}$ stands for an infinite sequence of $\texttt{9}$s) and $\texttt{1}$ denote the same number. Analogously, Turing's conventions associate any two sequences $p\texttt{0}\overline{\texttt{1}}$ and $p\texttt{1}\overline{\texttt{0}}$ (with $p \in \{\texttt{0},\texttt{1}\}^*$) with the exact same set of numbers. For Turing this was no serious problem as his paper only concerned the representation of real numbers by binary sequences, and not vice versa. For our purpose however of proving not only $\mathbb{R} \leq_\theta \Gamma$, but also $\Gamma \leq_\theta \mathbb{R}$ and thus $\mathbb{R} \equiv_\theta \Gamma$, we would like for every pair of infinite binary sequences to be associated with two distinct real numbers. To achieve this, we may define a θ-translation $\theta^{\Gamma}_{\mathbb{R}}$ such that it maps the infinite binary sequences onto a *nowhere dense* subset of the real numbers such as the *Cantor ternary set*, as shown in the following example:

$$\theta^{\Gamma}_{\mathbb{R}}(\gamma) = \sum_{i=1}^{\infty} \frac{2\gamma_i}{3^i} \tag{3.17}$$

Such a mapping has the desirable property that no two binary sequences map to the same real number, i.e., it is injective—even bijective when we consider the Cantor set as the codomain. For any two binary sequences $p\texttt{0}\overline{\texttt{1}}$ and $p\texttt{1}\overline{\texttt{0}}$ (with $p \in \{\texttt{0},\texttt{1}\}^*$), there is an entire interval between the two real numbers they represent.

Of course the mapping is far from surjective, since it reaches only a fraction of the real numbers in the unit interval; but this is irrelevant in the present situation. What we were looking for was a mapping from binary sequences to real numbers that is injective, which is the case fore the given definition of $\theta^{\Gamma}_{\mathbb{R}}$.

3.3 Intuitive notion of computability

In his renowned paper on computable numbers, Turing asserted that the numbers that can be computed by his machine include "all numbers which would naturally be regarded as computable" (Turing 1936–7, pp. 230, 249). While the former class of numbers can be effectively and unambiguously defined (namely, using the definition of the Turing machine), the composition of the latter is left entirely to the intuition of the reader. To assert a mathematical relation between these classes, as Turing did, produces an epistemologically interesting situation. Different interpretations of the phrase "which would naturally regarded as computable" could obviously lead to different evaluations of this assertion. To further complicate the situation, many alternative terms circulated that aimed to describe the elusive concept of computability, such as Church's "effective calculability", and Gödel's "mechanical procedure", each with their own subtleties. These ambiguities led to interesting philosophical discussions on the epistemological status of CTT and the nature of computation, as we will see later in section 5.3.

As with Turing's original formulation, most CTT versions follow a pattern where some (more or less) intuitive notion of computability is formalized in terms of a formal model of computation. We will model the intuitive notion using the U predicate. Beside viewing U as a predicate, we will also use it to refer to its extension, i.e., the subclass of the domain D whose members this notion applies to. In effect, when modeling Turing's thesis, we will let D denote the set of all (real) numbers, $U(x)$ will mean "x would naturally be regarded as computable", and additionally, U will refer to those real numbers "which would naturally be regarded as computable".

3.4 Models of computation

One of the central components of any CTT version is the model of computation. A model of computation can be viewed as an abstract and idealized computer, which is used to facilitate reasoning about the computability of functions and the complexity of algorithms. The computational power[6] of a model can be characterized in terms of the class of functions (or, as we saw in the previous section, languages, numbers, etc.) that it can compute. The extent of this class depends on the capabilities and limitations of the model. The significance of a CTT version then lies in asserting that some intuitively defined class of objects U is included in the computational power of a certain model M, i.e., that M is powerful enough to compute U. Note

[6] In the context of computability theory, we use the term *computational power* to refer to *what* a model or machine can compute, not to how fast or efficiently this can be done.

that, given a CTT version and the corresponding model of computation, we will use the symbol M to denote that model as a general concept, as well as the class of all instances of that model.

Two models are considered computationally equivalent if they compute the same class of functions. In the 1930s, it was discovered that several independently developed models of computation—among which the Turing machine and Alonzo Church's lambda calculus—were in fact computationally equivalent. This remarkable equivalence arose in a time where mathematicians and logicians were desperate to formalize the notion of "computability" or "effective calculability", which until then only existed as a vague, intuitional concept. Whether or not satisfactory as a definition (see also section 5.3.1), the Turing machine and equivalent models came to be the benchmark of computability, the corresponding class of functions generally being referred to as the "computable" or "recursive" functions. The challenge to breach the magical Turing barrier inspired many new models of computation, some of which were adaptations of existing models, while others sprang from fundamentally new ideas. In this section we will have a quick glance at the most common types of computational models.

3.4.1 Inter-model differences

While it was only after developing his computing machine that Turing became aware of Church's work, he was quick to realize that the classes of Turing-computable functions and lambda-definable functions coincided. The shared conjecture by Turing and Church that this class of functions represented a very fundamental notion of computability came to be known as the Church-Turing thesis. Earlier, it had already been established that Kurt Gödel's notion of "general recursive functions" (Gödel 1934) corresponded with lambda-definability, and in the same year that Turing developed his machine, Emil Post published a very similar and computationally equivalent system which he referred to as "formulation 1" (Post 1936). Due to their mutual equivalence, these models can in theory be used interchangeably to express a CTT version, although depending on the context, one may be more convenient to use than another.

The equivalence between these and many other models is truly remarkable, especially when considering how much their definitions vary. There are models, such as the Turing machine and the random-access machine (RAM, Cook and Reckhow 1973), which are conceived of as actual (yet idealized) machines, their computational process being described in terms of interacting hardware mechanisms. Other models, including lambda calculus, general recursive functions, and combinatory logic (Curry 1930; Schönfinkel 1924) are defined at a more abstract, functional, level. Then there exist models, like the cellular automaton "Game of Life" by John Conway (Gardner 1970), in which multiple operations are executed in parallel, as opposed to the sequential character of the previously discussed models. Arguably, recurrent neural networks (RNN) can be used as Turing-complete models of computation as well (Siegelmann and Sontag 1992), although this view is controversial (see Graves, Wayne, and Danihelka 2014; Weiss, Goldberg, and Yahav

2018).

3.4.2 Intra-model differences

Apart from differences *between* models, many variations can exist *within* one model. In particular the Turing machine, being arguably the most well-studied of all models of computation, has been the subject of numerous modifications, extensions, and alternative definitions. We will here discuss some common variants that can be shown to be equivalent in computing power to the classical Turing machine.

Whereas Turing's original work implies a one-way infinite tape, we could just as easily imagine the machine operating on a tape that extends infinitely in both directions. Of course this does not make any difference in terms of computational power: we can represent any two-way infinite tape on a one-way infinite tape, for example by "folding" the tape at a certain square, as demonstrated in Davis, Sigal, and Weyuker (1994, pp. 163–164).

It can sometimes be very convenient to define a Turing machine as having multiple tapes, as this enables one to keep input values, intermediate computations, and output values clearly separated. This too, however, does not affect the computational power of the machine, as is proved by Hartmanis and Stearns (1965, p. 293). In the same paper (p. 297) it is shown that a two-dimensional tape (i.e., a grid of squares) similarly has no effect in terms of computability.

Lastly, we will explore the consequences of allowing non-deterministic elements in the definition of a Turing machine. In his 1936 paper, Turing repeatedly stresses that the behavior of the machine at any moment is fully determined by its configuration and the symbol that is read from the tape at that moment. Let us now consider a non-deterministic variant of the Turing machine that for any configuration-symbol combination makes an arbitrary choice between several actions. In effect, a computation of such a machine forms a tree, each path of which represents one possible course of the process. A non-deterministic Turing machine is then said to accept its input if any of these paths terminates in an accepting state. Lewis and Papadimitriou (1998) provide a detailed discussion on non-deterministic Turing machines, including a proof that these machines, too, are no more powerful than ordinary Turing machines.

3.5 Formal definition of computation

As modeled in 3.1, the computational power of a model M is usually expressed in terms of some intuitively defined class $U \subseteq D$,[7] by asserting that for every object $x \in U$ there exists an instance $\mathcal{M}$ of M that computes x. But what do we really mean when we say that "$\mathcal{M}$ computes x"?[8] While the meaning of such a phrase would appear to be quite straightforward, we will see that subtle differences

[7] Here, we use the symbol U to denote the extension of the predicate U. Also, we assume that the universe of discourse is D, and as such that $U \subseteq D$ (i.e., U has no members outside of D).

[8] Note that, depending on the context, "computes" may be replaced with "decides", "simulates", etc.

in intuitions can have significant consequences. It is therefore important to take account of the particular definition of *computation* on which an author bases his or her characterization of *computability*. When representing a CTT version in our analytical framework, we will use the binary C predicate to represent the definition of computation that underlies that particular version. In this section we will discuss the relevance of this definition to the appraisal of a CTT version, and how we can find it for a given version.

3.5.1 Relevance of the C predicate

Given a CTT version with domain D, our objective is to formalize the author's conception of the relation "computes" with respect to D. Note that here we do not pursue a definition of computation in general, but one that specifies what it means to compute a specific type of objects, namely that which is defined in D. On the other hand, we are looking for a definition that is as general as possible, in the sense that it cannot use any model-specific terms or mechanisms ("tape", "halt", etc.), as to not gratuitously preclude any models from implementing it. Our definition of C should thus specify a minimum set of sufficiently abstract conditions that, when all satisfied by some model instance $\mathcal{M}$ with respect to an object $x \in D$, guarantee the truth[9] of the proposition "$\mathcal{M}$ computes x". In simpler terms, we are looking for those criteria that eventually lead the author to judge one model adequate to compute some class U, and another not.

In fact, divergences of judgments about CTT versions easily arise as a result of discrepancies between these criteria. While some authors maintain liberal criteria and require only that a model can realize a certain extensional input-output relation, others may involve additional factors and set further requirements on the course of a computation. For example, one may impose complexity bounds on the process, as a means to ensure that computation is not only possible in theory, but also feasible in practice. Restrictions like this can be relevant in complexity-theoretic approaches to CTT, such as the *Extended Church-Turing thesis*, which states that "any 'reasonable' model of computation can be efficiently simulated on a probabilistic Turing machine (an efficient simulation is one whose running time is bounded by some polynomial in the running time of the simulated machine)" (Bernstein and Vazirani 1997).

Another interesting question is to which degree a simulation of a computational process should be able to reproduce the technical aspects of the atomic operations involved in that process. It seems reasonable not to be too demanding in this respect, given the multitude of possible computational techniques and hardware mechanisms, which cannot reasonably be expected to be captured in one single model. In this context, Copeland and Shagrir (2019, p. 68) point out that

> [a] thesis aiming to limit the scope of algorithmic computability to Turing computability should thus not state that every possible algorithmic

[9] That is, the truth of this proposition as interpreted by the author of the relevant CTT version.

> process can be performed by a Turing machine. The way to express the thesis is to say the extensional input-output function $\iota\alpha$ associated with an algorithm α is always Turing-computable; $\iota\alpha$ is simply the extensional mapping of α's inputs to α's outputs. The algorithm the Turing machine uses to compute $\iota\alpha$ might be very different from α itself.

But what if an algorithm or physical process does not have clearly delineated inputs and outputs? How should we for example model an interactive computation that consists of continuous and overlapping streams of inputs and outputs? We will return to these questions in section 6.1.

A last example concerns the representation of objects from the domain. While total functions from integers to integers are relatively easy to represent using a finite model—one that can transform the (finite) representation of an input integer into the (finite) representation of its image under that function would do—we will sometimes encounter domains whose representations pose a greater challenge. For example, how should we represent values that are drawn from a continuum, like real numbers? Such values are often not easily captured in finite representations. Does it suffice when they can be approximated up to any desired precision? It seems that Turing (1936–7) would have answered this question in the affirmative—at least for values on the output side of a computation—given his elaboration of a machine that "computes" a real number by successively printing its decimals. Deutsch (1985), on the other hand, clearly distinguishes between "simulation" and "perfect simulation", where the former concerns itself with discrete approximation of real variables and is therefore considered insufficient for modeling continuous systems in classical physics.

3.5.2 Finding a definition

Unfortunately, authors do not typically state the exact definitions of computation that inspired their computability statements. It can however be expected that a statement of this kind, especially when it deviates considerably from traditional theories, is accompanied by a detailed commentary. We will often need to infer an author's views on computation from this commentary. Both low-level and high-level discussions of a model can offer valuable clues that may help us reconstruct the implicitly present definition of computation.

For a given CTT version that involves a model M, a domain D, and some subclass U of D, the author will most likely provide a more or less formal description of how the model works, and what the computation of an object from U would look like. Unless the author maintains an extremely narrow definition of computation, this does not mean that the given description was intended to outline the *only* possible way in which such an object can be computed. Rather, it will probably have been presented to exemplify a more general quality of the model, leaving room for many equally adequate alternative implementations. Consider, for example, the question of whether to represent integers in binary or decimal notation, which in many contexts will make no meaningful difference and is merely a matter of choice.

Moreover, the author might not even consider the ability to compute U to be an exclusive quality of M. It could be that a fundamentally differently defined model that somehow allows the same abstract principle to be implemented, would have met with similar approval by the author. It is therefore this more fundamental principle at the root of an author's understanding of computation which we would like to express in our definition, abstracting away from any distracting implementational details.

The challenge is then to identify which elements exactly are those implementational details that can be chipped away, and which elements we should preserve and incorporate in the definition. Let us consider the following example. An author presents a model M that is supposed to simulate all algorithms. To demonstrate this ability, the author provides a definition of an instance $\mathcal{M}$ of M that simulates John Conway's Game of Life (Gardner 1970). While Game of Life explicitly requires all cells to be updated simultaneously in a single time step, $\mathcal{M}$ is defined to simulate these updates in a serial manner over the course of multiple time steps. This last fact is significant, since it tells us that an exact replication of every atomic step of an algorithm is, as it appears not essential to the author's concept of "simulation". As it appears from this minimal example, the author considers the result of a process more relevant to the concept of computation or simulation than the process itself.

As was mentioned before, further clues with respect to the pursued definition may be sought in higher-level discussions of the model. Considering the extensive corpus of existing literature on computability theory, one will need to provide a new theory with proper justification in order to warrant its legitimacy in relation to established or competing theories. It is often in such arguments about the legitimacy of a CTT version that one's conception of computation most notably manifests itself. Given definitions of U and M, a corresponding CTT statement in the form of eq. (3.1) might be accepted by one person while rejected by another. Similarly, an author may contend that (and hopefully explain why) while one model suffices for the purpose of computing D, another does not. Such differentiations help us pinpoint the pivotal principles that motivate the definition of C that we are after.

Note that the level of detail at which a definition of computation can be reconstructed depends on the amount of information that can be extracted from an author's account. While some CTT versions are accompanied by exhaustive reflections, others are defined in very general terms, limiting the rigor with which the C predicate can be defined. Our analytical framework thus furthermore helps expose differences in degree of detail between CTT versions that might not be visible on initial inspection.

Chapter 4

HISTORICAL BACKGROUND

The genesis of the Church-Turing thesis was by no means an isolated event, but rather the culmination of a momentous historical process that gripped mathematical communities over the early decades of the twentieth century. Hence, if we only focused on CTT's technical aspects without taking note of the relevant circumstances from which it originated, our analysis would surely be deficient. In this chapter, we place CTT in a historical perspective by reviewing the events that led to its original formulation and discussing the responses that it elicited from the scientific world.

4.1 A foundational crisis

The beginning of the twentieth century was a turbulent yet exciting time in the world of mathematics. The field found itself in a "foundational crisis" following the discovery of a number of unsettling paradoxes, among which was Russell's paradox in 1901 (communicated to Gottlob Frege in Russell 1902). Motivated by a pressing need to secure the status of mathematics as a legitimate science, several ambitious projects were launched to cure the field of its flaws. One such effort by Alfred North Whitehead and Bertrand Russell brought forth the highly influential work *Principia Mathematica* (PM), published in the years 1910–1913. The aim of the authors was to formalize all of classical mathematics in a system of symbolic logic using a minimal set of axioms and inference rules, while avoiding the problematic paradoxes and contradictions.

Despite the great impact that PM has had on the development of mathematics, it was not embraced by all as a solution to the crisis. Among the skeptics was the German mathematician David Hilbert, one of the most respected mathematicians of the time. While being a fierce advocate of the axiomatic approach, to Hilbert a system that merely avoids existing paradoxes was not enough. He insisted that in order to restore the reputation of mathematics, we should build our theories on axioms from which it can be shown to be *impossible at all* ("überhaupt unmöglich", Hilbert 1917, p. 411) to derive contradictions. In other words, Hilbert required

that the solution be accompanied by a consistency proof. A formal theory T is (syntactically) *consistent* if there is no formula ϕ for which both ϕ itself and its negation $\neg\phi$ can be proved from T (or, if there is no ϕ such that $T \vdash \phi \wedge \neg\phi$). Earlier attempts had only resulted in relative consistency proofs, reducing for example the consistency of Euclidean geometry to that of analysis (alternatively referred to in literature as "arithmetic of real numbers" or "second-order arithmetic"). However, a further reduction seemed impossible and thus Hilbert pinned his hopes on finding a direct proof of consistency for analysis, a problem that had remained unsolved since the turn of the century, when it was Hilbert himself who presented it as the second of his famous 23 problems at the International Congress of Mathematicians in Paris (Hilbert 1900, pp. 264–266).

In 1921, he further specified his proposal for the "refounding of mathematics" in a series of lectures that he held at the University of Hamburg (Hilbert 1922). In these lectures, he expressed a strong confidence in the possibility of restoring the unimpeachable reputation of mathematics through proving the consistency of analysis (Hilbert 1922, p. 162; translation from Ewald 1996, p. 1121):

> [A] satisfactory conclusion to the research into [the foundations of mathematics] can only be attained by the solution of the problem of the consistency of the axioms of analysis. If we can produce this proof, then we can say that mathematical statements are in fact incontestable and ultimate truths—a piece of knowledge that (also because of its general philosophical character) is of the greatest significance for us.

In fact, this was not the first time that Hilbert made an effort to solve his second problem. Much earlier, in 1905, he had already sketched a consistency proof for the axioms of analysis. It was however not long before the French mathematician Henri Poincaré discovered that Hilbert's proof relied on a circular reasoning structure, a flaw that took Hilbert a while to come to terms with. To make matters worse, Hilbert's ambitions were further plagued by the emergence of a rivaling mathematical philosophy known as *intuitionism*, founded by the Dutch mathematician L.E.J. Brouwer and supported by Hilbert's own former student Hermann Weyl. A radical critique of classical mathematics, intuitionism rejected several well-established principles in mathematics and logic, such as Aristotle's *law of the excluded middle* (i.e., $A \vee \neg A$). Particularly disturbing to Hilbert was Brouwer's aversion to Georg Cantor's theory of *transfinite numbers* and the concept of *actual* or *completed* infinity, achievements that Hilbert greatly cherished.[1]

In his bold new attempt, Hilbert set out to salvage existing mathematics—including the theories that were under attack from the intuitionistic camp—by proving its consistency using methods so uncontroversial that even his adversaries would have no other option than to agree. In order to ban the circularities that Poincaré discovered in his earlier attempt, Hilbert introduced a rigid distinction

[1] See, for example, Hilbert 1926, p. 167, where he calls Cantor's theory "the most admirable flower of the mathematical intellect and in general one of the highest achievements of purely rational human activity" (Van Heijenoort 1967, p. 373).

between two types of mathematics. First, he would develop mathematics and logic together in a purely formal system, yielding *mathematics proper* ("die eigentliche Mathematik", Hilbert 1922, p. 174; Ewald 1996, p. 1131)—a formal language consisting of all axioms and theorems of existing mathematics. Then, the consistency proof of the axioms of mathematics proper was to be constructed within a new kind of mathematics which Hilbert called *metamathematics* ("Metamathematik") or *proof theory* ("Beweistheorie").

To eliminate each and every possibility of Brouwer and his followers rejecting his consistency proof, Hilbert chose to restrict himself to those basic modes of inference whose validity appeared to him as immediate, intuitive, and indisputable. This *finitary standpoint*[2] ("finiter Standpunkt"), presented most comprehensively in Hilbert (1926), ruled out the use of controversial modes of inference—such as the application of quantifiers to infinite totalities—at the level of metamathematics, while imposing no such restrictions on the methods of mathematics proper. By following this strategy, Hilbert hoped to conclusively establish the legitimacy of transfinite reasoning within mathematics on a finitary (and thus intuitionistically admissible) basis (Hilbert 1923, p. 156; Ewald 1996, p. 1140):

> We therefore see that, if we wish to give a rigorous grounding of mathematics, we are not entitled to adopt as logically unproblematic the usual modes of inference that we find in analysis. Rather, our task is precisely to discover why and to what extent we always obtain correct results from the application of transfinite modes of inference of the sort that occur in analysis and set theory. The free use and the full mastery of the transfinite is to be achieved on the territory of the finite!

In fact, though Hilbert never gave a precise delineation of which methods were to be considered "finitary", it is argued that they were even more restrictive than would have been required by Brouwer (e.g., Davis 2018, p. 94; Ewald 1996, pp. 1116, 1168).

Together with his assistant Paul Bernays and his student Wilhelm Ackermann, and in collaboration with John von Neumann and Jacques Herbrand, Hilbert would devote much of the 1920s to the development of this grand project, which came to be known as "Hilbert's program". As of 1928, substantial progress had been made and Hilbert was confident that victory was within reach.

In that same year, Hilbert and Ackermann published a textbook called *Grundzüge der theoretischen Logik*, in which they presented two related problems for first-order predicate logic (FOL), referred to as the "restricted functional calculus" ("engere Funktionenkalkül," Hilbert and Ackermann 1928). First, the authors presented a proof system and asked whether it is *semantically complete*: can all semantically (or universally) valid formulas of first-order logic (i.e., those formulas that are true under every possible interpretation of the system) be syntactically derived from its axioms (p. 68)? For the more elementary propositional calculus

[2] An alternative translation for the German "finit" that is commonly found in literature is "finitistic".

found in Principia Mathematica, completeness had already been confirmed several years earlier, first by Emil Post and later by Paul Bernays. For FOL, however, no such proof existed yet.

The second problem concerned the *decidability* of FOL and other logical systems. Aptly named "das Entscheidungsproblem"[3], it asked for a procedure that, within a finite number of steps, decides for any given logical expression whether that expression is universally valid or not (p. 73). Note that the Entscheidungsproblem is in principle not concerned with provability (or derivability), which is a strictly syntactic notion, but rather with the semantic notion of validity. However, for systems that are sound and (semantically) complete, these two notions coincide. As with the problem of completeness, the Entscheidungsproblem is relatively easy to solve for propositional logic: the validity of propositions can easily be tested using truth tables. For FOL, however, the Entscheidungsproblem did not have an obvious solution. In their textbook, Hilbert and Ackermann stress that "the Entscheidungsproblem should be considered the main problem of mathematical logic" (p. 77).

Striking in Hilbert and Ackermann's formulation is the apparent absence of restraint in assuming the existence of such a procedure. It is characteristic of Hilbert's uncompromisingly optimistic attitude toward the acquisition of mathematical knowledge, which he most famously expressed at the conclusion of a speech in his birthplace Königsberg, shortly after he retired in 1930 (Hilbert 1930a, p. 387; Ewald 1996, p. 1165):

> For the mathematician there is no *ignorabimus*, nor, in my opinion, for any part of natural science. ... The real reason why Comte was unable to find an unsolvable problem is, in my opinion, that there are absolutely no unsolvable problems. Instead of the foolish *ignorabimus*, our answer is on the contrary:
>
> *Wir müssen wissen,*
> *Wir werden wissen.*
>
> (We must know,
> We shall know.)

[3] While Hilbert and Ackermann's formulation of 1928 is often presented as the first introduction of the Entscheidungsproblem, Hilbert's student Heinrich Behmann had formulated the problem in a general form as early as 1921 in a lecture entitled "Entscheidungsproblem und Algebra der Logik" (Mancosu and Zach 2015, p. 176):

> [Axiomatizations of symbolic logic show] us, like the rules of chess, *only what one may do*, and *not what one should do*. The latter remains—in the one as in the other case—a question of inventive *thinking*, of lucky *combination*. We, however, require a lot more: not only the individual operations but also the *path of calculation* as a whole should be specified by rules, in other words, *an elimination of thinking in favor of mechanical calculation*. If a logical or mathematical statement is given, the required procedure should give complete instructions for determining whether the statement is *correct or false by a deterministic calculation after finitely many steps*. The problem thus formulated I want to call the *general decision problem* [*das allgemeine Entscheidungsproblem*].

4.2 Discovery of the undecidable

Around 1930, a new player entered the scene. Kurt Gödel, a young Austrian[4] logician, chose Hilbert and Ackermann's first problem—that of proving the semantic completeness of first-order logic—as the subject of his doctoral dissertation. Under the supervision of Hans Hahn he succeeded in proving that indeed, all universally valid formulas of FOL are theorems of Hilbert and Ackermann's proof system—i.e., can be proved from the axioms using the inference rules (Gödel 1929, 1930). Beside being an significant result in its own right, Gödel's *completeness theorem* was also a major advancement toward a solution to the Entscheidungsproblem (of first-order logic). By showing that all valid formulas of FOL were provable, half of the problem had been solved: what remained was to verify that conversely, *non*-validity—or, which by the same completeness result had been confirmed equivalent, unprovability—of formulas could be effectively determined as well. In modern terminology, Gödel had shown that validity (provability) in FOL is *semi-decidable*, but it had yet to be proved that it is indeed *decidable*.

In contrast to the purely logical system considered for the Entscheidungsproblem, the aim of Hilbert's program was to formalize logic together with mathematics in one all-encompassing system. For such a mathematical system, a proof of mere semantic completeness did not suffice. Whereas semantic completeness requires only that a system can prove all *logically valid* sentences of a formal language, for a mathematical theory it was deemed desirable that for *every* sentence ϕ of the language, either ϕ or its its negation could be proved. The latter type of completeness is generally referred to as *syntactic completeness*[5], as it is defined solely in terms of formal derivability and does in principle not involve the notions of truth or validity. Only when a theory is interpreted in terms of some model, its sentences come to express meaningful propositions about the domain of the particular model. Hilbert himself maintained an equivalent notion of syntactic completeness, but characterized it slightly differently—here in relation to number theory as the intended model (Hilbert 1929, p. 140; Mancosu 1998, p. 232):

> The assertion of the completeness of the axiom system for number theory can also be stated in this way: If a formula belonging to number theory, but not provable in it, is added to the axioms of number theory, then a contradiction can be derived from the extended axiom system.

While work on Hilbert's program by Ackermann and Von Neumann was producing promising results, Gödel chose to join the cause and adopt the problem of proving the consistency of analysis as his next challenge. Tragically however, in the process he made an unnerving but fundamental discovery that relentlessly crippled Hilbert's

[4] Gödel was born in 1906 in the Austro-Hungarian town of Brünn. When after World War I Czechoslovakia declared independence from the defeated Austro-Hungarian empire, Brünn became Brno and Gödel officially became a Czechoslovak citizen. In 1924 Gödel moved to Vienna where received Austrian citizenship in 1929.

[5] Syntactic completeness is also called "negation completeness", "formal completeness", "deductive completeness", and "maximal completeness".

program instead. That is to say, he stumbled across an inherent limitation of formal systems that simply rules out the possibility of formalizing all of mathematics in a system that is both syntactically complete and consistent—which happened to be the very aim of Hilbert's program.

Gödel (1931) discovered that, curiously, in every consistent[6] axiomatic system that is powerful enough to support basic arithmetic, it is possible to formulate propositions that can neither be proved nor disproved using the rules of the system. In his paper he developed a formal system P which corresponds with the arithmetical part of Russell and Whitehead's Principia Mathematica. For this system and all extensions of it with recursive classes of axioms, Gödel proved the existence of undecidable formulas.

At the time that PM was written, it was widely recognized that many problematic paradoxes and contradictions in mathematics arose from use of self-referential constructions. As such, Russell and Whitehead had gone to great lengths to make sure that their system was free from all forms of self-reference. Gödel invented an ingenious mechanism by which he was able to bypass this principle and introduce a "hidden" form of self-reference into ordinary arithmetical sentences of P. Known today as "Gödel numbers", he associated each symbol, formula, and proof of the system with a unique natural number. Under this mapping, each formula, which by the usual interpretation of the system asserts certain properties of natural numbers, receives an alternative interpretation as expressing propositions about other formulas of the same system—it could even be made to refer to itself. Exploiting this principle, Gödel created a sentence similar in spirit to the well-known Liar's paradox, which could be read as saying "I am unprovable in P". In a way, Gödel had blurred the line between metamathematics and ordinary mathematics by showing that the former can be brought into the realm of the latter.

The significance of Gödel's self-referential sentence—we will call it G_P, the "Gödel sentence" of P—becomes evident when we consider its provability in P. If we assume that G_P is provable, then we must accept the proposition that it expresses, namely that G_P is unprovable—a direct contradiction with our initial assumption. If, on the other hand, we assume that $\neg G_P$ is provable, we obtain a proposition that says "G_P is not unprovable in P", i.e., that G_P is provable in P—which leaves us with a syntactically inconsistent system, since both G_P and its negation are provable in it. Thus, neither G_P nor $\neg G_P$ can be provable in P, making G_P a *formally undecidable* ("formal unentscheidbare") proposition of P, and P itself a syntactically incomplete system.

Unfortunately for Hilbert, Gödel showed that undecidable sentences not only

[6] Gödel's original proof of his first incompleteness theorem only concerns so-called "ω-consistent" systems, a notion stronger than "simple" consistency. Whereas simple consistency only requires that a system S does not prove two formulas that directly contradict each other (i.e., $(\forall\phi)\,[\neg(S \vdash \phi \wedge \neg\phi)]$), ω-consistency requires that there is no property P of the natural numbers such that

$$(\forall n)\,[S \vdash P(n)] \wedge S \vdash (\exists n)\,[\neg P(n)].$$

Rosser (1936) proved by constructing his own undecidable "Rosser sentence" that Gödel's incompleteness result can be generalized to simply consistent systems. When speaking of Gödel's first theorem, we will tacitly assume the inclusion of Rosser's improvement.

exist in the specific system P, but in *every*[7] consistent formal system that is capable of representing a certain basic part of arithmetic. Moreover, in addition to this first *incompleteness theorem* he proved a second theorem, stating that any consistent mathematical theory that can represent the addition and multiplication of integers cannot prove its own consistency—let alone that severely restricted finitary methods suffice for proving the consistency of infinitary mathematics. Suddenly, things looked very bad for Hilbert's program.

It is not surprising that Gödel's incompleteness theorems have become widely recognized as one of the most significant results ever achieved in mathematical logic. Although conjectures concerning the possibility of incompleteness had been made by several mathematicians in the preceding years, Gödel was the first to provide a rigorous proof that, in defiance of Hilbert's "Wir müssen wissen, wir werden wissen", mathematical problems exist that are unsolvable.

However, we must note that the kind of (un)decidability that Gödel speaks of is, as the title of the work suggests, a strictly *formal* one: given a mathematical statement ϕ, it does not concern the existence of a procedure for determining the truth value of ϕ, nor is it about the possibility of deciding whether ϕ is a theorem of a certain formal system S. Rather, Gödel considers a statement ϕ decidable in a formal system S if the axioms and rules of inference of S suffice to derive either ϕ or $\neg\phi$ as a theorem. As such, this type of decidability can only be expressed *relative* to a certain formal system. A sentence that is undecidable in one system might very well be decided by a stronger system or, as Gödel points out, by metamathematical considerations.

After proving his first incompleteness theorem, Gödel carries the discussion of (formal) decidability from individual sentences to classes and relations. An n-ary relation[8] R between natural numbers is said to be *strongly representable* ("entscheidungsdefinit")[9] in a system S if there is a formula ρ in the language of S with n free variables, such that for every n-tuple of natural numbers $(x_1, \ldots, x_n)$ the following holds:

$$\begin{aligned} R(x_1, \ldots, x_n) &\Rightarrow S \vdash \rho(\overline{x_1}, \ldots, \overline{x_n}), \\ \neg R(x_1, \ldots, x_n) &\Rightarrow S \vdash \neg\rho(\overline{x_1}, \ldots, \overline{x_n}), \end{aligned} \tag{4.1}$$

where $\overline{x}$ denotes the formal numeral that represents the natural number x in S. By associating any n-ary function f with an equivalent $(n+1)$-ary relation F, such that $f(x_1, \ldots, x_n) = y \Leftrightarrow F(x_1, \ldots, x_n, y)$, we have also defined strong representability for functions.

---

[7] Actually, the general form in which Gödel's results are expressed here was adopted by Gödel only after Turing's construction of the Turing machine, which Gödel called "a precise and unquestionably adequate definition of the general concept of formal system". In his original 1931 paper, the result was only stated for a more definite, yet very comprehensive class of systems.

[8] Gödel treats classes as unary relations.

[9] The word "entscheidungsdefinit" has been translated in many different ways in scientific literature. The most commonly found translations include "numeralwise expressible", "(numeralwise) decidable", "binumerable", and "strongly representable".

4.3 General recursive functions

Earlier in his paper, Gödel defined an extensive class of functions which we know today as the *primitive recursive* functions.[10] This class consists of functions that are either a constant function or the successor function, or can be derived from these by simple forms of substitution and recursion (induction). A relation is then primitive recursive if its characteristic function (see eq. (2.5)) is primitive recursive. After introducing the notion of strong representability, Gödel notes that, by a proof given earlier in the paper, all primitive recursive relations (and as a consequence all primitive recursive functions) are strongly representable in his arithmetical system P.

Due to the fact that P is recursively axiomatizable (i.e., it can be decided within a finite amount of time whether some well-formed formula ϕ is an axiom of P), it is possible to enumerate all of its proofs. Whereas the property of strong representability (in P) of a primitive recursive function f in itself only asserts the *existence* of a proof for every computation of f, the recursive axiomatizability of P additionally ensures that this proof can be found within a finite number of steps. During a lecture in the spring of 1934 at the Institute for Advanced Study in Princeton, New Jersey, Gödel remarked (Gödel 1934, p. 3):

> [Primitive recursive] functions have the important property that, for each given set of values of the arguments, the value of the function can be computed by a finite procedure. Similarly, [primitive] recursive relations (classes) are decidable in the sense that, for each given n-tuple of natural numbers, it can be determined by a finite procedure whether the relation holds or does not hold (the number belongs to the class or not), since the representing function is computable.

While presumably motivated by his discoveries in the context of the system P, Gödel here detached the notions of computability and decidability from any definite formalisms. In contrast to other mathematical notions such as provability or representability, which can only be expressed relative to some formal system, the property of "being computable (decidable) by a finite procedure" stands on its own. In the same way that Hilbert's Entscheidungsproblem is indifferent to the particular form of its solution, as long as it decides each sentence of FOL in a finite number of steps, a function can be regarded "computable" in general whenever there exists *some* mechanism in which it can be fully represented. The problem was that at the time, this absolute understanding of computability only existed as a vague, intuitive notion, and had not been clearly demarcated by a mathematical definition. In a footnote to the above passage Gödel acknowledges this, but speculates that extending the class of primitive recursive functions by allowing more advanced forms of recursion might cause it to coincide with the class of computable functions.

[10] In his 1931 paper, Gödel calls these functions simply "rekursiv". Since Kleene (1936a), "primitive recursive" has become the most common term to denote this class, distinguishing it from what Gödel would later (1934) introduce as "general recursive" functions.

That not all forms of recursion are captured by primitive recursiveness had already been suggested and proved some years earlier by Hilbert (1926, p. 185) and Ackermann (1928), respectively. Central to this proof was a function, known today as the *Ackermann function*, whose definition made use of a recursion on multiple variables, which was not reproducible using primitive recursions. Motivated by the existence of such non-primitive forms of recursion and returning to his speculations, Gödel concludes his Princeton lectures with a section named "General recursive functions", where he poses "the question what one would mean by 'every recursive function'" (p. 26). Adapting a personal suggestion by the French mathematician Jacques Herbrand, he then goes on to present a new, more comprehensive definition of recursiveness that incorporates the aforementioned types of non-primitive recursive functions.

The main idea of Herbrand's suggestion was to consider a function f recursive if it can be defined by a system of equations E, whose equations express values of f in terms of its own values and those of other recursive functions, such that each value of f is deducible from E by a series of substitution steps. To guarantee the effectiveness of the definition, Gödel further specified the types of substitutions and derivations that were allowed, and required that for each n-tuple of natural numbers $(x_1, \ldots, x_n)$, there exists exactly one m such that $\overline{f}(\overline{x_1}, \ldots, \overline{x_n}) = \overline{m}$ can be derived from E. This criterion ensures that f is a *total* function. Thus, an n-ary function f is *general recursive* in the Herbrand-Gödel sense if:

$$(\exists E)\left[(\forall x_1, \ldots, x_n)(\exists! m)\left[E \vdash \overline{f}(\overline{x_1}, \ldots, \overline{x_n}) = \overline{m}\right]\right] \tag{4.2}$$

Expanding on Gödel's rather succinct introduction, Stephen C. Kleene (1936) proved that the general recursive functions can be constructed from the primitive recursive functions by the addition of a single operator. Complementing the existing operations of substitution and primitive recursion, this *μ-operator*[11] searches for the least natural number that satisfies a given relation. Given an $(n+1)$-ary general recursive relation R and the natural numbers $x_1, \ldots, x_n$, the expression $\mu y\big[R(x_1, \ldots, x_n, y)\big]$ denotes the least natural number y for which $R(x_1, \ldots, x_n, y)$ holds. Consider the function ϕ:

$$\phi(x) = \mu y\big[\rho(x, y) = 0\big], \tag{4.3}$$

where ρ is assumed to be a general recursive function. Using an auxiliary function σ, the following set of equations is a Herbrand-Gödel-style recursive definition of ϕ:[12]

$$\begin{aligned} \sigma(0, x, y) &= y, \\ \sigma\big(S(z), x, y\big) &= \sigma\big(\rho(x, S(y)), x, S(y)\big) \\ \phi(x) &= \sigma\big(\rho(x, 0), x, 0\big), \end{aligned} \tag{4.4}$$

[11] In Kleene (1936a), the author uses for this operator the symbol ε. Other commonly used names include "minimization operator", "least-number operator", "least search operator", "unbounded search operator".

[12] Example taken from Kleene (1936a) and Davis (1982).

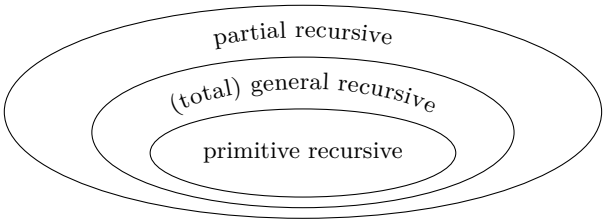

Figure 4.1: The recursion hierarchy

where S is the successor function. To guarantee that these equations obey the Herbrand-Gödel definition, one additional restriction is required. To meet the criterion expressed in (4.2), the function ρ must, beside being general recursive, satisfy the following condition:

$$(\forall x)(\exists y)\left[\rho(x,y)=0\right]. \tag{4.5}$$

What Kleene proved in his *normal form theorem* is that every general recursive function ϕ can be defined using only primitive recursion and a single instance of the μ-operator in the following form:

$$\phi(x_1,\ldots,x_n)=\psi(\mu y[R(x_1,\ldots,x_n,y)]), \tag{4.6}$$

where ψ is a primitive recursive function and R a primitive recursive relation and, analogous to eq. (4.5), $(\forall x_1,\ldots,x_n)(\exists y)\left[R(x_1,\ldots,x_n,y)\right]$ (Kleene 1936a, p. 736).

Later, Kleene (1938, 1943) explored the consequences of omitting the latter condition. For the function ϕ in eq. (4.3), this means that there might be natural numbers x for which no y exists such that $\rho(x,y)=0$. Clearly, for such numbers, ϕ has no value. Accordingly, attempts to recursively resolve the value of $\phi(x)$ using the equations of eq. (4.4) will never reach the base case and terminate, resulting in infinite iterations over the natural numbers. Permitting the unrestricted use of the μ-operator in function definitions thus violates Gödel's requirement that for every tuple of arguments a function should have *exactly* one value (eq. (4.2)). Kleene noted that the Herbrand-Gödel definition could be modified to include partial functions by relaxing this requirement so that it only requires the existence of *at most* one value for every tuple of arguments (Kleene 1938, p. 152). The resulting class of *partial recursive* functions "[includes] the general recursive functions as those which are defined for all sets of arguments." (Kleene 1943, p. 50, see also fig. 4.1) Due to its definability in terms of the μ-operator, the same class is often referred to by the term "μ-recursive functions".

Despite his conjecture that a more general notion of recursiveness might coincide with that of finite computability, Gödel was wary of asserting that his definition of general recursive functions was sufficiently exhaustive to realize this equivalence. Some 30 years after his 1934 lectures, he explained in a letter to Martin Davis: "However, I was, at the time of these lectures, not at all convinced that my concept of recursion comprises all possible recursions . . . ." (Davis 1982, p. 8) In fact, it would

take Gödel another two years and a fresh insight from a young British mathematician before he finally abandoned his reluctance and embraced the significance of his own work.

4.4 λ-definability and Church's thesis

While staying in Princeton for his lectures at the Institute of Advanced Study, Gödel had a discussion on computability with Alonzo Church, an American mathematician who taught at Princeton University. Not unlike Gödel's notion of finite computability, Church used the term "effectively calculable" to informally describe those functions whose values can be obtained by a finite (or *effective*) procedure.[13] Using modern terminology, Kleene (1981, p. 56) later characterized effective calculability as follows:

> For partial functions, the "effective calculability" of ϕ means that there is an algorithm that leads in a finite number of steps to the value of $\phi(n)$ for any n for which the value is defined, and that for any other n leads to no value (either by terminating in a situation that does not give a value or else by continuing ad infinitum).

In the spirit of Hilbert's program, Church had been developing a new system for mathematical logic since the late 1920s, which he published in two papers (Church 1932, 1933). Among other functions, the system defines an abstraction operator λ which binds variables in formulas. Church introduced this operator to avoid ambiguities in mathematical expressions arising from the use of free variables. For example, the expression $x + 1$ (where x is a free variable) can be interpreted either as a number (the successor of some definite number x) or as a definition of a function (the successor function; in this case x is no definite number but merely a placeholder). Using the λ-notation, we can distinguish between these cases by binding x and writing $\lambda x[x + 1]$ when we wish to denote the function.

The application of a function $\lambda x[M]$ to a term L is written $\{\lambda x[M]\}(L)$. When no syntactical ambiguities can arise, $\{F\}(L)$ will be abbreviated to $F(L)$ and $\lambda x_1[\ldots\lambda x_n[M]\ldots]$ will be written $\lambda x_1 \ldots x_n \cdot M$ to increase readability. For similar reasons, we write $F(L_1, \ldots, L_n)$ to denote the λ-expression $\{\ldots\{F\}(L_1)\ldots\}(L_n)$. Church defined several rules for converting one λ-expression into another without changing the meaning of the expression. The first is known today as α-conversion and simply enables the replacement of bound occurrences of a variable with another, previously unused variable. If M is *α-convertible* into L, we will write $M \rightarrow_\alpha L$. For example: $\lambda x \cdot x \rightarrow_\alpha \lambda y \cdot y$. The second conversion rule allows an expression of the form $\lambda x \cdot M(L)$ to be converted into $M[^x_L]$, which stands for the result of substituting L for all free occurrences of x in M. We will say that $\lambda x \cdot M(L)$ *β-reduces* to $M[^x_L]$, written $\lambda x \cdot M(L) \rightarrow_\beta M[^x_L]$. Thus, $\lambda x \cdot 3x + 5\,(2) \rightarrow_\beta 3 \cdot 2 + 5$. An expression is in *normal form* when no more β-reductions are possible. If B is a normal form of A,

[13] The intuitive meaning of effective calculability is discussed in greater detail in section 5.3.2.

written $A \to_{\beta *} B$, then B is unique in the sense that every other normal form of A is α-convertible into B (Church and Rosser 1936, p. 479):

$$(\forall A, B, C)\left[(A \to_{\beta *} B \wedge A \to_{\beta *} C) \to (C \to_{\alpha} B)\right] \tag{4.7}$$

Strictly speaking, the arithmetical symbols that I have been using in the previous examples ("+", "1", etc.) are not part of Church's formal system. In fact, his system used a very modest alphabet that beside symbols for variables and some punctuation characters only contained a handful of predefined function symbols. Church defined the positive integers as follows (Church 1933, p. 863):

$$\begin{aligned} 1 &\longrightarrow \lambda f x \cdot f(x). \\ S &\longrightarrow \lambda \rho f x \cdot f(\rho(f, x)), \end{aligned} \tag{4.8}$$

where S stands for the successor function. Now the expression that stands for the integer 2 is obtained by applying the successor function to the expression that stands for 1, followed by a series of β-reductions until a normal form is reached:[14]

$$\begin{aligned} 2 &\longrightarrow \lambda \rho f x \cdot f(\rho(f, x))\,(\lambda g y \cdot g(y)) \\ &\to_{\beta} \lambda f x \cdot f(\lambda g y \cdot g(y)\,(f, x)) \\ &\to_{\beta} \lambda f x \cdot f(f(x)). \end{aligned} \tag{4.9}$$

In the same fashion, the numeral for 3 is obtained by computing $S(2)$, yielding $\lambda f x \cdot f(f(f(x)))$. Generalizing, the *Church numeral* $\overline{n}$ of a natural number n is $\lambda f x \cdot f^n(x)$, where f^n stands for the n-fold composition of f. When a λ-expression F expresses a (partial) function $f\colon \mathbb{N}^n \longrightarrow \mathbb{N}$, such that

$$(\forall x_1, \ldots, x_n)\left[f(x_1, \ldots, x_n) = y \iff F(\overline{x_1}, \ldots, \overline{x_n}) \to_{\beta} \overline{y}\right], \tag{4.10}$$

we say that F *λ-defines* f, and that f is *λ-definable*.[15]

After defining lambda expressions for the operations of addition, multiplication, and subtraction, and translating Peano's axioms for arithmetic to formulas of his system, Church expressed his intentions to develop the full theory of positive integers in his system, before proceeding with those of the rational and real numbers. Regarding the implications of Gödel's incompleteness theorems for the possibility of a (finitary) consistency proof for his system, he notes (Church 1933, pp. 842–843):

> The impossibility of such a proof of freedom from contradiction for the system of *Principia Mathematica* has recently been established by Kurt Gödel. His argument, however, makes use of the relation of implication U between propositions in a way which would not be permissible under the system of this paper, and there is no obvious way of modifying the

[14] Note that for the sake of readability, I additionally applied the α-conversion $\lambda f x \cdot f(x) \to_{\alpha} \lambda g y \cdot g(y)$ to the argument.

[15] Kleene (1981, p. 55) notes that, while the concept had been around since 1931–32, the term "λ-definability" did not appear in publications until Church (1936a) and Kleene (1936b).

> argument so as to make it apply to the system of this paper. It therefore remains, at least for the present, conceivable that there should be found a proof of freedom from contradiction for our system.

Evidently, Church cherished the hope that the differences between his system and those considered in Gödel (1931) would suffice to escape the daunting spell of the second incompleteness theorem. He turned out to be right, be it in a way he had most likely not envisaged. In 1935, his students Stephen C. Kleene and J. Barkley Rosser, with whom he had been working on the system for years, discovered that the system was inconsistent, making all sentences, including the one asserting the consistency of the system, provable (Kleene and Rosser 1935). This discovery left Church with no other choice than to abandon his ambitious project. Yet, while the logical system as a whole had suffered a fatal blow, it appeared that his efforts had not been entirely futile. The sub-theory of λ-definability had remained unaffected and could be salvaged to live on as an independent theory, now known as the (untyped) λ-*calculus*.

Just how strong the notion of λ-definability was had slowly become clear in the preceding years. In their attempt to develop the theory of positive integers in Church's logic, the need arose for Church and Kleene to define several basic functions in the system. Kleene (1981, pp. 56–57) recalls that it took them considerable effort to come up with a definition for the predecessor function. In 1932, just when Church was about to succumb to despair, Kleene saw how to use the λ-notation to realize the intended definition (p. 57):

> When I brought this result to Church, he told me that he had just about convinced himself that there is no λ-definition of the predecessor function.
>
> The discovery that the predecessor function is after all λ-definable excited our interest in what functions are not just definable in the full system but actually λ-definable.

In the period that followed, Kleene confirmed the λ-definability of more and more functions. In a letter to Paul Bernays, Church wrote (Sieg 1997, pp. 155, 158):

> The results of Kleene are so general and the possibilities of extending them apparently so unlimited that one is led to conjecture that a [λ-]formula can be found to represent any particular constructively defined function of positive integers whatever. It is difficult to prove this conjecture, however, or even to state it accurately, because of the difficulty in saying precisely what is meant by "constructively defined". A vague description can be given by saying that a function is constructively defined if a method can be given by which its values could be actually calculated for any particular positive integer whatever.

Church been playing with this conjecture at least since late 1933, when Rosser confronted him with yet another function that had turned out to λ-definable (Sieg

1997, p. 159). By early 1934, around the time that Gödel started his series of lectures at the Institute of Advanced Study,[16] he had grown a strong confidence in the power of λ-definability and even thought of using it as a *definition* for constructive definability (or effective calculability). In a letter to Kleene, he reports how Gödel responded to this proposal (Davis 1982, p. 9):

> In regard to Gödel and the notions of recursiveness and effective calculability, the history is the following. In discussion [*sic*] with him the notion of lambda-definability, it developed that there was no good definition of effective calculability. My proposal that lambda-definability be taken as a definition of it he regarded as thoroughly unsatisfactory. I replied that if he would propose any definition of effective calculability which seemed even partially satisfactory I would undertake to prove that it was included in lambda-definability. His only idea at the time was that it might be possible, in terms of effective calculability as an undefined notion, to state a set of axioms which would embody the generally accepted properties of this notion, and to do something on that basis.

Shortly after this conversation, Gödel would propose his definition of general recursive functions. As we have seen in the previous section however, Gödel "was not at all convinced that [his] concept of recursion comprises all possible recursions", and only hinted at the equivalence between finite computation (or effective calculability) and recursiveness. This did not dissuade Church and his collaborators from carrying through the proposed proof that general recursiveness was included in λ-definability. Indeed, they found that λ-definability and general recursiveness are equivalent—that is, the class of λ-definable functions and the class of Herbrand-Gödel general recursive functions are exactly the same class (Kleene 1936b).

This astounding coincidence only strengthened Church's conjecture regarding the correspondence between λ-definability (or equivalently, as was now established, general recursiveness) and effective calculability. He first published his thesis in his 1936 paper "An Unsolvable Problem of Elementary Number Theory" (Church 1936a), where he proposed to "define the notion ... of an effectively calculable function of positive integers by identifying it with the notion of a [general] recursive function of positive integers (or of a λ-definable function of positive integers)." (Church 1936a, p. 356) In a footnote, he explained (p. 346, footnote 3):

> The fact ... that two such widely different and (in the opinion of the author) equally natural definitions of effective calculability turn out to be equivalent adds to the strength of the reasons ... for believing that they constitute as general a characterization of this notion as is consistent with the usual intuitive understanding of it.

We will model Church's thesis as follows:

[16] See Davis (1982, p. 8) for a more detailed account of the chronology of these events.

ANALYSIS *Church's thesis*

$$
\begin{aligned}
D &:= \text{Functions of positive integers } f\colon \mathbb{N}^k \longrightarrow \mathbb{N} \\
U(f) &:= \text{“}f \text{ is effectively calculable.”} \\
M &:= \text{Herbrand-Gödel style systems of equations} \\
C(\mathcal{M}, f) &:= \text{“For every possible tuple of arguments } (n_1, \ldots, n_k) \in \mathbb{N}^k \\
&\quad \text{(where } k \text{ is the arity of } f\text{), it is possible to find the value of} \\
&\quad f(n_1, \ldots, n_k) \text{ within a finite amount of time using just } \mathcal{M}.\text{”}
\end{aligned}
$$

Since our definition of C is model-independent (see section 3.5.2), we can represent the "λ-definability-oriented" version of Church's thesis simply by redefining M as the set of λ-expressions.

In the final section of his 1936 paper, Church addressed the "unsolvable problem" that the paper's title refers to. Under his proposed definition of effective calculability as λ-definability, he proved that there exists no effectively calculable function that, given two λ-expressions A and B, decides whether A is convertible into B or not (by a series of α-conversions and/or β-reductions). Contrary to Gödel's earlier undecidability results, which only concerned decidability relative to a certain formal system, Church had now confirmed the existence of problems for which no solution exists in general. As a direct result of this theorem, Church showed that Hilbert's Entscheidungsproblem is unsolvable for any ω-consistent[17] system of symbolic logic in which the integers are represented. In such systems, Gödel numberings can be used to express propositions like "A is convertible into B". A solution to the Entscheidungsproblem requires that every such proposition is decidable, contradicting the finding that no effectively calculable function of this sort exists.

Church's proof did however not cover systems of pure first-order logic, for which Hilbert and Ackermann originally posed the Entscheidungsproblem. Since these systems contain no integers, the Gödel numbering method does not apply. In "A Note on the Entscheidungsproblem" (1936), Church strengthened his earlier results by showing that even for the "bare" logical calculus of Hilbert and Ackermann, the Entscheidungsproblem has no solution. After Gödel's disturbing discovery of the incompleteness theorems, this was the final blow for Hilbert's program.

4.5 TURING MACHINES

In the same year that Church's paper was published, the British mathematician Alan Turing independently arrived at a similar result. His findings were published in the 1936 volume of the *Proceedings of the London Mathematical Society* under the now famous title "On Computable Numbers, with an Application to the Entscheidungsproblem". Analogous to Church's notion of effective calculability, but initially

[17] See footnote 6.

applied to numbers instead of functions, Turing used the term "computable" to informally describe "the real numbers whose expressions as a decimal are calculable by finite means." To make this notion precise, he developed in detail a theoretical machine whose operations, he believed, "include all those which are used in the computation of a number."

It is important to note that in the 1930s, the terms "computer" and "computation" were naturally and almost exclusively associated with human beings and, respectively, the process by which they performed calculations using paper and pencil. The digital electronic computers that rule the world today were nowhere to be found yet. As such, the phrase "calculable by finite means" would in the first place be understood in this human sense. It is evident that Turing had this sense in mind when designing his *automatic machine* (or *a-machine*), which since Church (1937a) we know as the *Turing machine*. At the outset of his exposition, he explicitly grafted the idea of his machine onto that of "a man in the process of computing a real number" (Turing 1936–7, p. 231). Turing then proceeded to outline the constituent parts and processes of his machine—some of which are readily recognizable as mechanical abstractions of those involved in human computation—resulting in a device that could be likened to a simplified typewriter that operates automatically and deterministically once it is set in motion. Later in his paper, Turing in a "direct appeal to intuition" (p. 249) undertook to further justify his design choices, explaining each as an abstraction of some aspect of human computation.

As a one-dimensional abstraction of the gridded paper in an arithmetic book, Turing imagined an infinite tape running through a machine, divided into "squares" of which each could hold at most one symbol. The limited amount of information that a human can attend to at one moment he mimicked by allowing the machine to "scan", and thus be directly aware of, only a single square at a time. At each moment, the machine may write a new symbol on the scanned square if it is blank, or otherwise erase a scanned symbol, and subsequently shift its "attention" one square to left or right. The "state of mind" of a human calculator, which may include his or her memory of previously encountered information, is represented by a variable called the "m-configuration". At any moment, the combination of the m-configuration and the currently scanned symbol determine the next action(s) to be taken by the machine. The schema that unambiguously prescribes per pair of m-configurations and scanned symbols the action(s) that should be taken and the next m-configuration to be assumed is called the "program" of the machine. Since Turing required that the sets of symbols (the alphabet) and of possible m-configurations be finite, so will be the program of the machine.

A few minor variations aside, Turing's description so far is the essential definition of the Turing machine as it is still in use today. While Turing was less rigid in this respect, we will distinguish between on the one hand this "bare" definition, and on the other hand the remainder of his description as an implementational convention that one may as well choose to fill in differently. Following Post (1947), who maintains a similar distinction, we will speak of "Turing machines" and "Turing convention-machines", respectively. Turing chose to compute only the fractional

parts of real numbers, represented by infinite binary strings whose digits occupy alternate squares on the (one-way infinite) tape. Calling these digit-bearing squares "F-squares", he designated the remaining intermediate squares—the"E-squares"—as scrap paper, which could contain other symbols beside `0` and `1`. Printing and erasing symbols could be done freely on any E-square at any moment, whereas the digits on the F-squares should always form a continuous sequence starting at the leftmost square—Turing called this sequence "the sequence computed by the machine"—and could not be erased once printed.

Furthermore, if a machine's program causes it to endlessly keep printing digits on F-squares, it will be called a "circle-free" machine, while one that definitively stops printing digits after a finite amount of time is called "circular". Note that a circular machine does not necessarily halt altogether once it stops printing new digits; it may or may not continue to run, possibly still printing symbols on E-squares. The computable numbers then, according to Turing's definition, are those real numbers x for which there exists a circle-free Turing convention-machine $\mathcal{M}$ such that the sequence of digits computed by $\mathcal{M}$ is the binary notation of the fractional part of x. Based on Turing's description, we construct the following model:

ANALYSIS *Turing's thesis*

$$
\begin{aligned}
D &:= \mathbb{R} \\
U(x) &:= \text{"}x \text{ could naturally be regarded as computable (by an idealized human calculator)."} \\
M &:= \text{Circle-free Turing convention-machines (as specified in Turing 1936–7)} \\
C(\mathcal{M}, x) &:= \text{"For any } k \in \mathbb{N},\ \mathcal{M} \text{ can produce the first } k \text{ digits of } \langle x \rangle \text{ within a finite amount of time."}
\end{aligned}
$$

As a consequence of Turing's convention of only representing the fractional parts of real numbers, each binary sequence γ represents not a single number x, but a class X of (mostly) integer-spaced numbers, with for every $x \in X$:

$$x = s \cdot (n + \sum_{i=1}^{\infty} \gamma_i \cdot 2^{-i}) \tag{4.11}$$

for some $n \in \mathbb{N}^0$ and with $s = 1$ or $s = -1$.

Of course, we could easily alter Turing's convention as to include their integer part and make each sequence (and thus each machine) represent a unique number. One example[18] would be to divide a sequence γ into three parts, the first of which

[18] Adapted from Turing (1938).

consists of a single digit indicating the *sign* of the number that is represented. Then follows a sequence of digits `1` of length n that represents the (absolute value of the) integer part of the number. The end of this sequence of `1`s is marked with a single `0`. Finally, the remaining sequence c, of which the first digit c_1 is the $(n+3)$-th digit of γ, represents the fractional part of the number. Such a sequence γ then represents the following unique real number x:

$$x = (2\gamma_1 - 1) \cdot (n + \sum_{i=1}^{\infty} c_i \cdot 2^{-i}). \tag{4.12}$$

While this method preserves Turing's convention of using a strictly binary alphabet, we are even free to expand the alphabet to include all decimal digits, a minus sign, and a dot to construct Turing machines that compute real numbers in their more readable decimal representation.

As demonstrated in section 3.2, computation of real numbers, binary sequences and functions of natural numbers can be quite naturally expressed in terms of each other. In fact, Turing suggested the possibility of generalizing his results to "computable functions of an integral variable or a real or computable variable, computable predicates, and so forth." (Turing 1936–7, p. 230) In the same paper he developed a simple definition of computable unary integer functions by Turing convention-machines (p. 254). With every function $f\colon \mathbb{N} \longrightarrow \mathbb{N}$ he associated a binary sequence γ such that between every n-th and $(n+1)$-th digit `0` in γ, there are exactly $f(n)$ digits `1`. For every γ that is computable (i.e., there exists a Turing convention-machine that computes it) the corresponding function f is also computable. While of course requiring slightly more inventive association strategies, such definitions can be generalized to functions of multiple variables. Turing even expressed his intentions to develop "the theory of functions of a real variable expressed in terms of computable numbers" (p. 230), but this he never fully carried through.

Turing made the crucial observation that a *universal Turing machine* could be defined that mimics the behavior of any other machine whose program it receives encoded on its tape. Whereas the original Turing convention-machine computed a fixed number determined by a "hardcoded" program, the universal machine could be programmed to compute any computable number by allowing its program to be delivered by the user as a variable piece of "software". Essentially, Turing had here laid the foundations for the modern all-purpose digital computer. Furthermore, apart from strictly imitating the behavior of the machine whose program it is supplied with, a machine could also be made to analyze this behavior and answer questions about it. In the second part of his paper, Turing employed this property to prove a number of fundamental theorems about the nature of computation. Reminiscent of Church's results, yet entirely independently, he demonstrated the existence of generally undecidable problems of mathematics.

First, he proved that there exists no Turing machine that, when provided with the program of a Turing convention-machine, decides whether that machine is circle-free or not. Next, Turing proved that "there can be no machine $\mathcal{E}$ which, when

supplied with the [program] of an arbitrary machine $\mathcal{M}$, will determine whether $\mathcal{M}$ ever prints a given symbol (0 say)." (Turing 1936–7, p. 248) The former problem is closely related, but not identical to the now more well-known *halting problem*, while the latter problem foreshadowed *Rice's theorem*.

Finally, Turing arrived at his "application to the Entscheidungsproblem". Using his previously proved theorems, he showed that no Turing machine can decide for every formula of Hilbert and Ackermann's predicate calculus whether that formula is provable in the system or not. In other words, the Entscheidungsproblem is unsolvable for first-order logic and any system or theory based upon it. Working at King's College in Cambridge, England, Turing wrote his paper while unaware of the developments in Princeton. Unknowingly, he had arrived at the exact same result as his transoceanic colleague, while using a completely different method. Kleene (1981, p. 61) reports that Turing only learned about λ-definability and general recursiveness "just as he was ready to send off his manuscript, to which he then added an appendix outlining a proof of the equivalence of his computability to λ-definability."

As Church and Kleene had previously proved the equivalence of λ-definability and general recursiveness, Turing computability now became the third in a row of very different yet equivalent characterizations of computability. While Gödel remained reluctant to assert the equivalence of his notion of general recursiveness to that of effective calculability, Church and Turing posed their theses. The name "Church-Turing thesis" seems to have been proposed as late as 1967 by Kleene (1967, p. 232, emphasis in original):

> So Turing's and Church's theses are equivalent. We shall usually refer to them both as *Church's thesis*, or in connection with that one of its three versions which deals with "Turing machines" as *the Church-Turing thesis*.

In fact, a fourth equivalent model had been developed by the Polish-American mathematician Emil Post, also in 1936.[19] Under the name "formulation 1" he introduced a formalization of the process of computation that was remarkably similar to Turing's (Post 1936). Like Turing, Post considers a one-dimensional sequence of "spaces or boxes" that can be marked with symbols. Unlike Turing, computation in Post's scenario is not carried out by an automatic machine, but by a "problem solver or worker" operating according to a fixed set of rules. Post, being acquainted with the work of Church and Gödel but not with that of Turing, hypothesized that his formulation was logically equivalent to general recursiveness, although he did not proceed to prove this.

[19] Tragically, this was not the first time that Post had a brilliant idea of his own overshadowed by other publications. In the early 1920s he anticipated Gödel's first incompleteness theorem, based on an assumption which is equivalent to the Church-Turing thesis (Davis 1965, p. 338). Reluctant to publish his findings before having verified this assumption, he saw Gödel reap the glory in 1931. Post's account of this anticipation (Post 1941) was printed in Davis (1965, pp. 340–433).

Chapter 5

CRITICAL RECEPTION

From minor corrections and smart reformulations to fundamental objections and radical extensions, the Church-Turing thesis has since its inception stirred up a lively and fruitful debate across multiple fields of science. Moreover, it laid the foundations for the development of the digital electronic computer and inspired an entire new academic discipline known today as *computability theory* or *recursion theory*. In this chapter, we analyze and evaluate a selection of the challenges, criticisms, and alternatives that the revolutionary statement has received over the years.

5.1 Church and Gödel on Turing's work

Despite Turing's fundamentally different methods and his ignorance of the work of Church and Gödel, his approach to characterizing computability had yielded an equivalent definition. Should there have been any lingering doubts in Church's mind about the validity of his thesis, they must surely have been dispelled when Turing's work appeared. In fact, it appears that Church came to consider Turing's analysis superior to his own (Church 1937a, p. 43):

> As a matter of fact, there is involved here the equivalence of three different notions: computability by a Turing machine, general recursiveness in the sense of Herbrand-Gödel-Kleene, and λ-definability in the sense of Kleene and the present reviewer. Of these, the first has the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately—i.e. without the necessity of proving preliminary theorems. The second and third have the advantage of suitability for embodiment in a system of symbolic logic.

In a paper from 1938, Church furthermore described (an "adoption of") Turing's idea as "Perhaps the the most convincing form in which [the] definition of an

effective process can be put" (Church 1938, p. 227), listing it as the first candidate, followed by Herbrand-Gödel general recursiveness and finally his own λ-definability.

Even for the ever skeptic Gödel, who earlier dismissed Church's proposal to take λ-definability as a definition of effective calculability as "thoroughly unsatisfactory", it appears that Turing's work finally bridged the gap between the existing intuitive notions of computation and the formalizations by Church and himself. As opposed to the latter two, which were defined purely in abstract mathematical terms, Turing's definition of computability naturally developed from the starting point of "a man in the process of computing a real number". This may have appealed to Gödel's earlier intuitions that the notion of effective calculability might be defined by stating "a set of axioms which would embody the generally accepted properties of this notion, and to do something on that basis." In any event, the publication of Turing's pivotal paper had a dramatic impact on Gödel's attitude toward Church's thesis. Somewhere in the late 1930s—Shagrir (2006, p. 400) suspects the year to be 1938—he declared that it "was established beyond any doubt by Turing" that his own definition of general recursiveness "really is the correct definition of mechanical computability" (Gödel 193?, p. 168).

Later he declared (Gödel 1951, pp. 304–305):

> The greatest improvement was made possible through the precise definition of the concept of finite procedure, which plays a decisive role in these results. There are several different ways of arriving at such a definition, which, however, all lead to exactly the same concept. The most satisfactory way, in my opinion, is that of reducing the concept of finite procedure to that of a machine with a finite number of parts, as has been done by the British mathematician Turing.

Turing's contributions furthermore provided the basis for a generalization of Gödel's incompleteness theorems. In a 1964 postscript to his 1934 lectures in Princeton, Gödel wrote (Davis 1965, pp. 71-72, emphasis in original):

> In consequence of later advances, in particular of the fact that, due to A.M. Turing's work, a precise and unquestionably adequate definition of the general concept of formal system can now be given, the existence of undecidable arithmetical propositions and the non-demonstrability of the consistency of a system in the same system can now be proved rigorously for every consistent formal system containing a certain amount of finitary number theory.
>
> Turing's work gives an analysis of the concept of "mechanical procedure" (alias "algorithm" or "computation procedure" or "finite combinatorial procedure"). This concept is shown to be equivalent with that of a "Turing machine".

Gödel further expressed his inclination toward Turing's approach in a footnote to the above passage, where he referred to Church's definitions of effective calculability by general recursiveness and λ-definability as "previous equivalent definitions of computability, which, however, are much less suitable for our purpose. . . ."

5.2 Criticisms and modifications of the Turing machine

Despite being quickly recognized for its significance, the extensive exposure that Turing's paper has enjoyed through the years has also laid bare a sizable number of flaws and errors. In his 1965 anthology *The Undecidable*, Martin Davis introduced Turing's paper saying "This is a brilliant paper, but the reader should be warned that many of the technical details are incorrect as given. ... In any case, it may well be found most instructive to read this paper for its general sweep, ignoring the petty technical details." (p. 115) Paul Bernays called Turing's attention to some of the errors and Turing subsequently corrected them in a follow-up paper in 1938.

In the decades that followed, multiple authors proposed further corrections and modifications to the Turing machine, some of which have largely superseded Turing's original conventions. As a result, a modern reader might be surprised to find that Turing's description of his machine differs considerably from the form in which it is usually presented today. That said, the modifications mainly concern practical conventions and the errors, while great in number, are mostly technical and do not invalidate the fundamental ideas involved, which were truly groundbreaking and are relevant today as ever.

Emil Post, who with his strikingly similar formulation 1 must have been a perceptive reader of Turing's paper, dedicated an appendix to his 1947 paper to correcting and elucidating Turing's description of the universal machine. The body of that same paper concerned an unsolvability proof of the "problem of Thue" (also known as the word problem for semigroups). Post found that in order to carry out this proof with a Turing machine, he needed to modify Turing's formulation. Beside using a two-way rather than one-way infinite tape and simplifying the representation of a machine's program, Post disposed of Turing's convention of using alternating F and E-squares. Furthermore, Turing had required—probably due to his specific focus on computing binary sequences—that a digit, once printed, could not be erased. Post instead allowed the printing and erasing of symbols at arbitrary times on arbitrary squares of the tape. Similarly, Turing did not in the context of computing infinite sequences consider the option that a machine could be intentionally designed to halt after a finite number of steps. Post explicitly included the possibility of defining no actions for certain target configurations, such that machines will halt upon reaching them.

Jack Copeland's *The Essential Turing* (2004) contains a—possibly unsent—letter by Turing, addressed to Alonzo Church, in which Turing responds to Post's comments (Turing 1948?):

> Post observes that my initial description of a machine differs from the machines which I describe later in that the latter are subjected to a number of conventions (e.g. the use of E and F squares). These conventions are nowhere very clearly enumerated in my paper and cast a fog over the whole concept of a "Turing machine". Post has enumerated the conventions and embodied them in a definition of a "Turing convention machine".

> My intentions in this connection were clear in my mind at the time the paper was written; they were not expressed explicitly in the paper, but I think it is now necessary to do so. It was intended that the "Turing machine" should always be the machine without attached conventions, and that all general theorems about machines should apply to this definition. To the best of my belief this was adhered to. On the other hand when it was a question of describing particular machines a host of conventions became desirable.

In the same book, Copeland published a critique of Turing's paper by Donald Davies, who from the late 1940s worked with Turing on one of the first programmed computers (Davies 2004). In his extensive analysis, Davies proposed a number of corrections and improvements to Turing's universal machine. Clearly, in the process of designing his machine Turing was guided by proof-theoretical considerations rather than practical feasibility and efficiency. As a result, implementations of Turing's (corrected) original design would be rather slow and unmanageable. In the final section of his critique, Davies redesigned the universal Turing machine as to facilitate testing the correctness of his modifications. After running a successful simulation of a Turing machine, he concluded that "There can be reasonable confidence that there are no further significant errors in Turing's design ...." (p. 103).

5.3 Philosophical evaluations

Unlike the technical presentation of the Turing machine, which after a number of corrections and improvements has more or less stabilized into a standard form, the philosophical content of CTT has fueled an ongoing debate that even today is far from settled. In particular, the thesis generates a seemingly irreconcilable tension between intuition and mathematical rigor, making it an unusual but all the more interesting statement.

5.3.1 Epistemological disputes

What is striking in Church's writings regarding his thesis is that he not only hypothesized (the extension of) the notion of effective calculability to be included in that of general recursiveness (or Turing computability or λ-definability), but proposed to take the latter as a *definition* of the former. Post, on the other hand, took a more cautious approach with his formulation 1 and explicitly criticized Church for not doing so (Post 1936, p. 105, emphasis in original):

> The writer expects the present formulation to turn out to be logically equivalent to recursiveness in the sense of the Gödel-Church development. Its purpose, however, is not only to present a system of a certain logical potency but also, in its restricted field, of psychological fidelity. In the latter sense wider and wider formulations are contemplated. On the other hand, our aim will be to show that all such are logically reducible

> to formulation 1. We offer this conclusion at the present moment as a *working hypothesis*. And to our mind such is Church's identification of effective calculability with recursiveness.

Then, in a footnote:

> Actually the work already done by Church and others carries this identification considerably beyond the working hypothesis stage. But to mask this identification under a definition hides the fact that a fundamental discovery in the limitations of the mathematicizing power of Homo Sapiens has been made and blinds us to the need of its continual verification.

This continual verification, Post argued, would "change this hypothesis not so much to a definition or to an axiom but to a *natural law*." In line with Post's position, Kleene in 1943 was likely the first to characterize Church's statement as a "thesis", explaining that it was motivated by the "heuristic fact" that "such functions (predicates) as have been recognized as being effectively calculable (effectively decidable) ... have turned out always to be general recursive. ..." (Kleene 1943, p. 60) In a review of Post's 1936 paper, Church defended his view (Church 1937b):

> [Post] does not ... regard his formulation as certainly to be identified with effectiveness in the ordinary sense, but takes this identification as a "working hypothesis" in need of continual verification. To this the reviewer would object that effectiveness in the ordinary sense has not been given an exact definition, and hence the working hypothesis in question has not an exact meaning. To define effectiveness as computability by an arbitrary machine, subject to restrictions of finiteness, would seem to be an adequate representation of the ordinary notion, and if this is done the need for a working hypothesis disappears.

The characterizations by Post and Kleene further imply that CTT is not a mathematical statement in the sense that it can be subjected to absolute mathematical proof or refutation, but only corroborated by evidence. Kleene later emphasized this aspect, arguing that Church's statement "is a thesis rather than a theorem, in as much as it proposes to identify a somewhat vague intuitive concept with a concept phrased in exact mathematical terms, and thus is not susceptible of proof." (Kleene 1967, p. 232)

It appears that the disagreement between Church and Post stemmed from different views on the status of the notion of effective calculability. Implicit in Post's characterization of Church's thesis as a working hypothesis appears to be the assumption that humanity somehow shares a universal intuitive understanding of the concept of effective calculability such that, even in the absence of a precise definition, our intuitions will always agree on the extension of this concept. If this were the case, Post's approach of "continual verification" could be successful and eventually lead us, as Post expected, either to unanimously reject CTT or to gradually come to accept it as a natural law. Church, who might not have been

convinced that our understanding of effective calculability is really absolute and universal, deemed such a program meaningless as long as an exact definition for "effectiveness in the ordinary sense" was lacking.

Curiously, Church did not further investigate the intuitive qualities of effective calculability to see if a more precise definition could be constructed directly from these. In fact, he barely analyzed the intuitive meaning of the concept at all, confining himself to shallow observations such as "a function is constructively defined if a method is given by which its value could be actually calculated for any particular positive integer whatever." (Sieg 1997, p. 158) Instead, he simply proposed to take the existing notions of general recursiveness and λ-definability as a definition, seeking a posteriori justification in the systems' extensional correspondence with the intuitive notion. Such a definition could not satisfy Gödel, who suggested that an adequate definition should "embody the generally accepted properties" of effective calculability. Evidently, Gödel pursued a more intensionally motivated definition that directly accounted for the existing intuitions.

To the satisfaction of Gödel, Turing finally introduced a model that seemed to embody the generally accepted properties of effective calculability. Unlike Church, Turing quite convincingly substantiated his claim—that Turing machines machines could compute "all numbers which would naturally be regarded as computable"—by showing how the principles and operations that were usually involved in the computation of a number could be naturally reduced to mechanic counterparts in a Turing machine. Yet, he acknowledged that "[all] arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." (Turing 1936–7, p. 249) Thus, while agreeing with Church on the vague intuitive nature of the existing notion of computability, Turing nevertheless made an effort to translate these vague intuitions into more concrete principles, which then guided the development of his model.

Still, even though Church and Gödel embraced Turing's contribution as a definition of effective calculability, a satisfactory definition should in my opinion not be sought in some model that implements these principles, but rather in a coherent and model-independent formulation of these principles themselves. In a similar way, Sieg (2002, p. 290) speaks of "sharpening the informal notion, formulating its general features axiomatically, and investigating the axiomatic framework." Such a sufficiently precise intensional definition would potentially break Church's argument against Post. His identification of effective calculability with general recursiveness or λ-definability would indeed reduce to a working hypothesis, which would, in theory, be disproved as soon as a function is found that is verifiably effective according to this axiomatic definition, but not general recursive or λ-definable. The next section covers several attempts at capturing the intuitive notion of effective calculability directly in a more workable definition.

5.3.2 Sharpening informal notions

In the previous section, we have paid no attention to the origins and contents of the intuitive understandings of effective calculability or computability that existed

in the early 1930s.[1] We should not forget that the work of both Church and Turing was done in the direct context of investigations into the Entscheidungsproblem, of which Hilbert and Ackermann had stated that it "is solved if one knows a procedure that allows one to decide the validity ... of a given logical expression by a finite number of operations." (Hilbert and Ackermann 1928, p. 73, translation from Sieg 1994, p. 77) Unlike other mathematical problems of the day, a solution to the Entscheidungsproblem did not depend on any demarcated set of admissible methods, or on the inherent limits of some formal system. This especially complicated the construction of an unsolvability proof. In order to be able to prove that no procedure exists that satisfies the description given by Hilbert and Ackermann, a precise characterization of the extension of this description was required.

In the light of Hilbert's formalist mindset, which manifested itself most prominently in his ambitions to capture the whole of mathematics in one all-encompassing formal theory, it was most natural to assume that Hilbert had in mind a deterministic procedure whose execution would, for any given logical expression, require nothing more than the rote-like processing of an unambiguously specified list of instructions. His student Heinirch Behmann had stressed in 1921 that it was essential to the character of the Entscheidungsproblem[2] "that as method of proof only *entirely mechanical calculation* according to given instructions, without any activity of thinking in the narrower sense, is allowed." (Mancosu and Zach 2015, p. 176) Furthermore, it would be pointless to conceive of procedures whose operations exceed the capacities of the human mind and body, as, contrary to the present day, humans were the only relevant computing agents at the time. This human-centered conception of effective computation is supported by Robin Gandy in a comment on Church's thesis (Gandy 1980, pp. 123–124):

> Both Church and Turing had in mind calculation by an abstract human being using some mechanical aids (such as paper and pencil). The word "abstract" indicates that the argument makes no appeal to the existence of practical limits on time and space. The word "effective" in the thesis serves to emphasize that the process of calculation is deterministic—not dependent on guesswork—and that it must terminate after a finite time.

Turing was the first[3] to justify his thesis by a direct appeal to these intuitive, or "generally accepted" principles of effective computation. While Turing's analysis holds the key to his intuitive understanding of effective calculability, references to

[1] For a more thorough analysis of the developments that contributed to the emergence and relevance of the notion of effective calculability I refer the reader to Sieg (1994).

[2] See also footnote 3 of chapter 4.

[3] Although Turing's work was done more or less simultaneously with that of Post, we focus here on Turing's work for two reasons. First, Turing's development is more thorough and complete. Second, while Post's model is very similar to that of Turing, his justification is very different—in fact, opposite—as Sieg (1994, pp. 91–92) emphatically points out: "[Turing] did not try to extend a narrow notion reducibly and obtain in this way additional quasi-empirical support; rather, he analyzed the intended broad concept and reduced it to a narrow one—once and for all."

these principles are to be found dispersed across the text and are never aggregated into an explicit "set of axioms", as Gödel had suggested to Church in 1934.

The central question in Turing's analysis is "What are the possible processes which can be carried out in computing a number?" Three arguments are presented, followed by a demonstration of "examples of large classes of numbers which are computable". The most innovative and distinguishing part of Turing's argumentation is found in his argument I, where he most directly tries to answer the central question "by a direct appeal to intuition". In accordance with our foregoing discussion of the historical context, Gandy (1988, p. 77) observes that this argument "*makes no reference whatsoever to calculating machines.* Turing machines appear as a result, as a codification, of his analysis of calculation by humans." What are now, according to Turing, the fundamental axioms of human computation?

Sieg (1994, 1997, 2002) isolates five restrictive conditions[4] from Turing's analysis of human computation, comprising two *boundedness conditions* (**B**), two *locality conditions* (**L**), and one *determinacy condition* (**D**):

(**B.1**) *There is a fixed bound on the number of symbolic configurations a computor*[5] *can immediately recognize.*

(**B.2**) *There is a fixed bound on the number of internal states a computor can be in.*

(**L.1**) *A computor can change only elements of an observed symbolic configuration.*

(**L.2**) *A computor can shift attention from one symbolic configuration to another one, but the new observed configuration must be within a bounded distance of the immediately previously observed configuration.*

(**D**) *A computor's internal state together with the observed configuration fixes uniquely the next computation step and the next internal state.*

These axioms can be used to derive an explicit definition for Turing's underlying intuitive interpretation of computability:

DEFINITION 5 *Computability (Turing)*
A number can be naturally regarded as computable if it can be computed by an abstract human computor satisfying all axioms (**B**), (**L**), and (**D**).

Following Gandy, we take "abstract" to mean that no appeal is made to the existence of practical limits on time and space, and, additionally, that the computor is assumed to make no mistakes. The claim that the Turing-computable numbers include "all numbers which would naturally be regarded as computable" can then be reinterpreted using this definition to obtain a more precise statement.

[4] The formulation presented here is taken from Sieg (2002).

[5] Sieg, following a convention suggested by Robin Gandy, uses the word "computor" to refer to "a human computing agent who proceeds mechanically." (Sieg 2002, p. 395)

ANALYSIS *Turing's thesis (reinterpreted)*

$$
\begin{aligned}
D &:= \mathbb{R} \\
U(x) &:= \text{“}x \text{ can be computed by an abstract human computor satisfying the boundedness conditions (}\mathbf{B}\text{), the locality conditions (}\mathbf{L}\text{), and the determinacy condition (}\mathbf{D}\text{).”} \\
M &:= \text{Circle-free Turing convention-machines (as specified in Turing 1936–7)} \\
C(\mathcal{M}, x) &:= \text{“For any } k \in \mathbb{N}, \mathcal{M} \text{ can produce the first } k \text{ digits of } \langle x \rangle \text{ within a finite amount of time.”}
\end{aligned}
$$

Whereas Turing's axioms, as interpreted by Sieg, focus on the properties and limitations of the elementary operations that a human being carries out in the process of computation, Copeland (1997) takes a slightly different approach and characterizes effectiveness at the level of the method:

> A method, or procedure, M, for achieving some desired result is called "effective" (or "systematic" or "mechanical") just in case:
>
> 1. M is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);
> 2. M will, if carried out without error, produce the desired result in a finite number of steps;
> 3. M can (in practice or in principle) be carried out by a human being unaided by any machinery except paper and pencil;
> 4. M demands no insight, intuition, or ingenuity, on the part of the human being carrying out the method.

A function, then, is effectively calculable if there exists an effective procedure that computes it.

In both the Turing-Sieg and the Copeland accounts of effectiveness, the central themes are finiteness and determinism. Turing, in the opening statement of "On Computable Numbers", described the computable numbers as "the real numbers whose expressions as a decimal are calculable by finite means." Likewise, in the context of computability or effectiveness, Gödel often spoke of "finite" and "mechanical" procedures. In fact, even though in practice computing was still done almost exclusively by humans, the idea that computation was a task that could in principle be carried out by machines was by no means novel. As far back as the seventeenth century, Gottfried Leibniz had designed mechanical calculators and even described a framework that could be seen as anticipating Turing's universal

machine.[6]

In the more immediate context of Hilbert's program, Heinrich Behmann, when emphasizing the mechanical character of mathematical proof methods, had added that "One might, if one wanted to, speak of *mechanical* or *machine-like* [*maschinenmäßigem*] *thinking.* (Perhaps one can one day even let it be carried out by a machine.)" (Mancosu and Zach 2015, p. 176) In a lecture held in December 1933, Gödel underlined the purely formal character of rules of inference in formal systems and suggested that they "could be applied by someone who knew nothing about mathematics, or by a machine." (Gödel 1933, p. 45)

5.3.3 Mind versus mechanism

Thus it appears that the qualities of being finite and mechanical were generally considered essential to effective procedures. But what justified these assumptions? What withheld mathematicians from thinking of infinite algorithms or computation by non-mechanical procedures? Turing, having claimed that the computable numbers are calculable by finite means, sought provisional justification "in the fact that the human memory is necessarily limited." (Turing 1936–7, p. 231) The formulation of his boundedness conditions (labeled (**B.1**) and (**B.2**) by Sieg), which he presented later in his paper, could be interpreted as an elaboration on the former statement (p. 250):

> We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused.

Andrew Hodges, in his 1983 biography of Alan Turing, considers it "a bold act of imagination, a brave suggestion that 'states of mind' could be counted, on which to base his argument. It was especially noteworthy because in quantum mechanics, physical states *could* be 'arbitrarily close'." (Hodges 2012, p. 134) Gödel, despite his earlier laudatory comments on Turing's work, also expressed his misgivings about Turing's "states of mind" argument in a critical note[7] entitled "A philosophical error in Turing's work" (Gödel 1972, p. 306). In this short note, Gödel posits that Turing in his paper "gives an argument which is supposed to show that mental procedures cannot go beyond mechanical procedures." Gödel objects:

---

[6] An interesting account of the developments that started with Leibniz and culminated in Turing's universal machine is given by Martin Davis in his book *The Universal Computer: The Road from Leibniz to Turing.*

[7] An alternative formulation of Gödel's note is presented in Wang (1974, pp. 325–326), where Gödel does not speak explicitly about convergence to infinity in the course of a *procedure*, but in the course of the development of the mind in general.

> What Turing disregards completely is the fact that *mind, in its use, is not static, but constantly developing*, i.e., that we understand abstract terms more and more precisely as we go on using them, and that more and more abstract terms enter the sphere of our understanding. There may exist systematic methods of actualizing this development, which could form part of the procedure. Therefore, although at each stage the number and precision of the abstract terms at our disposal may be *finite*, both (and, therefore, also Turing's number of *distinguishable states of mind*) may *converge toward infinity* in the course of the application of the procedure.

Gödel thus rejected Turing's assumption that the effective number of states of mind involved in a human computation procedure is necessarily finite. Given the abundance of occasions where Gödel associated computation explicitly with "finite" and "mechanical" procedures, this seems a rather odd statement. In a 2006 article, the Israeli philosopher Oron Shagrir analyses Gödel's objections and tries to reconcile the ostensibly contradictory attitudes that he maintained toward Turing's paper. In our discussion of Gödel's objection, we will roughly follow the structure of Shagrir's analysis, supplementing it with additional material.

In the postscriptum to his 1934 Princeton lectures, added in 1964, Gödel also alluded to the existence of non-mechanical procedures ("i.e., such as involve the use of abstract terms on the basis of their meaning"), asserting that the question of whether they exist "has nothing whatsoever to do with the adequacy of the definition of 'formal system' and of 'mechanical procedure'." (Gödel 1934, p. 370) After stating some generalizations of his incompleteness theorems due to Turing's contributions, he warned that these generalizations may not apply to non-mechanical theories and procedures, and that they "do not establish any bounds for the powers of human reason, but rather for the potentialities of pure formalism in mathematics."[8] Reflecting on the question whether the classes of finite procedures and recursive procedures coincide, which he raised in footnote 3 of his 1934 lectures, Gödel was willing to "answer affirmatively" for his notion of general recursiveness only "if 'finite procedure' is understood to mean 'mechanical procedure'". Apparently, the potential power of finite *non*-mechanical procedures formed a significant obstacle for Gödel to fully and unconditionally embrace the momentous mathematical results of the 1930s.

Gödel further elucidated his perspective on the relationship between minds and machines in 1951, when he interpreted his incompleteness results as follows (Gödel 1951, p. 310):

> So the following disjunctive conclusion is inevitable: *Either . . . the human mind (even within the realm of pure mathematics) infinitely surpasses the powers of any finite machine, or else there exist absolutely*

[8] In a footnote to his 1972 note "A philosophical error in Turing's work", Gödel suggests that the entire note may be regarded as a footnote to be placed after the word "mathematics" in the just quoted part of his 1964 postscriptum.

> *unsolvable diophantine problems* ... (where the case that both terms of the disjunction are true is not excluded, so that there are, strictly speaking, three alternatives). It is this mathematically established fact which seems to me of great philosophical interest.

Hao Wang (1974, p. 324), who was in close touch with Gödel in the early 1970s, asserts that "Gödel thinks Hilbert was right in rejecting the second alternative", i.e., the one asserting that there exist absolutely unsolvable problems. With respect to Gödel's 1972 note on Turing's "states of mind" argument, Wang further reports (p. 326):

> In our discussions Gödel added the following. Turing's argument becomes valid under two additional assumptions, which today are generally accepted, namely: 1 There is no mind separate from matter. 2 The brain functions basically like a digital computer. (2 may be replaced by: 2′ The physical laws, in their observable consequences, have a finite limit of precision.) However, while Gödel thinks that 2 is very likely and 2′ practically certain, he believes that 1 is a prejudice of our time, which will be disproved scientifically (perhaps by the fact that there aren't enough nerve cells to perform the observable operations of the mind).

Gödel, who had a sustained interest in theology and considered himself a theist (Gödel 1986, p. 14; Wang 1987, p. 18), strongly believed in a world separate from and not bounded by the physical. In this light, it is no surprise that he considered Turing's argument, which, according to him, was "supposed to show that mental procedures cannot go beyond mechanical procedures", inconclusive. But was Gödel justified in ascribing these physicalist convictions to Turing? Sieg (1997, p. 171) thinks not:

> Let me emphasize that Turing's analysis is neither concerned with *machine computations* nor with general *human mental processes*. Rather, it is *human mechanical computability* that is being analyzed, and the special character of this intended notion motivates the restrictive conditions that are brought to bear by Turing.

On the one hand it is true that Turing, when formulating his boundedness conditions, did not explicitly make any assertions with regard to human mental processes in general. He simply supposed that the number of states *which need be taken into account* is finite. On the other hand, as Shagrir (2006, p. 410, footnote 40) remarks, Turing did state in section 1 of his paper that the justification for characterizing the computable numbers as "those whose decimals are calculable by finite means ... lies in the fact that the human memory is necessarily limited", and later introduced his argument I saying that it "is only an elaboration of the ideas of §1."

Let us for the moment assume that, as Sieg suggests, Turing intended his restrictions to apply only to the number of states that need to be taken account in the particular case of mechanical calculation. Shagrir (2006, p. 410) finds this proposal problematic:

> If the number of states of mind in general may be infinite, or at least unbounded, is there any reason to think that with respect to calculation the number of states of mind is bounded? It would seem that an implicit assumption is being made here, to the effect that that the process of (human) calculation is associated with a cognitive "module" with a finite number of states of mind. But this assumption is exceedingly contentious, and renders Turing's analysis, and Gödel's reasons for extolling it, vulnerable to obvious objections.

If we read Turing's analysis carefully, however, we must conclude that he did not explicitly make any assumptions of the kind brought up by Shagrir. Turing did not deny that the actual number of states of mind that a human being enters in the process of computation might be infinite; he expressed a suspicion that "If we admitted an infinity of states of mind, some of them will be 'arbitrarily close' and will be confused." (p. 250) The idea expressed in this terse remark was further elaborated by Wang (1974, p. 93):

> An alternative way of defending this application of the principle of finiteness is to remark that since the brain as a physical object is finite, to store infinitely many different states, some of the physical phenomena which represent them must be "arbitrarily" close to each other and similar to each other in structure. These items would require an infinite discerning power, contrary to the fundamental physical principles of today.

Gödel—who, as we saw earlier, ascribed "a finite limit of precision" to the physical laws—evidently appreciated this subtlety, admitting that at each stage of a procedure the "number of *distinguishable states of mind*" may be finite. The real error that Turing made, according to Gödel, was disregarding the dynamic nature of the mind, i.e., the idea that the number and complexity of abstract terms in our "sphere of understanding" increases over time, as a result of which the number of distinguishable states of mind "may *converge toward infinity* in the course of the application of the procedure." Gödel assumed here, as Webb (1990, p. 300) describes, "that by understanding [abstract terms] more precisely we become capable of states which are themselves more and more complicated." He did admit, though, that such processes were at the time "far from being sufficiently understood to form a well-defined procedure" and that "the construction of a well-defined [finite non-mechanical] procedure which could actually be carried out (and would yield a non-recursive number-theoretic function) would require a substantial advance in our understanding of the basic concepts of mathematics."

One could argue that, contrary to Gödel's allegations, Turing did anticipate the possibility of a dynamic mind with abstract concepts whose complexity evolves over time. Only, whereas Gödel believed that the the different stages of complexity should be represented by a sequence of distinct states, Turing regarded the complexity of a state at any moment as a property to be reflected in the contents of the tape rather than the state itself, as appears from his suggestion that "the use of more

complicated states of mind can be avoided by writing more symbols on the tape." (p. 250) As Webb (1990, p. 300) points out, this principle of relocating the source of complexity from program to tape found its ultimate application in the design of the universal Turing machine, which despite its fixed and finite set of states can simulate any machine of any complexity. By providing the universal machine with the description of a Turing machine on its tape, Turing (1947, p. 383) explained, "the complexity of the machine to be imitated is concentrated in the tape and does not appear in the universal machine proper in any way."

According to Webb, then, Gödel's objection finds its origins in the following discrepancy between Gödel's thinking and Turing's:

> Gödel was presumably not convinced, the universal machine notwithstanding, that all the states entered by a human computer using "finite *non*-mechanical procedures" could always be compensated for in Turing's purely symbolic manner, for in such states it just might exploit the meanings of ever more abstract concepts of proof and infinity to grasp *infinitely* complicated combinatorial relations. It is really this kind of possibility more than any convergence to an infinity of states that could undermine Turing's arguments, but, far from having disregarded it completely, it seems that Turing himself must have initially thought such an objection plausible; yet once he discovered the universal machine he saw that it could indeed compensate symbolically for a surprisingly wide class of increasingly complicated *machine* states.

The idea that the complexity of the dynamic human mind could be accounted for by symbolic representations was supported by Post, as his following remark concerning "the dualism of the physical world versus the mental world" bears witness (Post 1941, p. 431):

> Fundamental is the distinction between the static outer symbol-space with its assumed capacity for bearing symbol-complexes of unbounded complexity, and the dynamic mental world with, however, its obvious limitations. This has been fully emphasized by Turing in his finite number of mental states hypothesis ....

After all, it does appear to be the case that, contrary to Sieg's suggestion, Turing believed his restrictive conditions to be a direct consequence of the inherent limitations of the human mind (or, perhaps, more specifically, the human brain) rather than merely motivated by "the special character" of the notion of human *mechanical* computability. Turing however, possibly foreseeing the controversy that his "states of mind" argument would provoke, proposed in his argument III to "avoid introducing the 'state of mind' by considering a more physical and definite counterpart of it." This physical counterpart consisted of a "note of instructions" in which, after every step, the computer precisely specifies how the work is to be continued, allowing him at any moment "to break off from his work, to go away and forget all about it, and later to come back and go on with it." Where in

argument I the state of mind of the human computer is regarded as an integral part of the computational process, argument III dissociates the state of the computation from the internal state of its executor and stores it in an independent symbolic representation. Hodges (2012, p. 136) concurs that

> these arguments were quite different. Indeed, they were complementary. The first put the spotlight upon the range of thought within the individual – the number of "states of mind". The second conceived of the individual as a mindless executor of given directives. Both approached the contradiction of free will and determinism, but one from the side of internal will, the other from that of external constraints. These approaches were not explored in the paper, but left as seeds for future growth.

The juxtaposition that Hodges describes is what might explain Gödel's ambivalent sentiments regarding Turing's analysis. It is argued (e.g., Webb 1990, p. 297) that Gödel directed his misgivings only at Turing's argument I, but nonetheless accepted his analysis as a whole by virtue of his argument III. This seems plausible as argument III does not involve any contentious assumptions concerning the nature of the human mind. The view of the human computer as a "mindless executor" of a finite set of instructions furthermore seems to be in accordance with Gödel's recurring use of the terms "mechanical" and "finite" in the context of computational procedures. Gödel's perspective on Turing's work might be best summarized in the words of Webb (1990, p. 302), namely "that all Turing was really *analyzing* was the concept of 'mechanical procedure', but that in his *arguments* for the adequacy of his analysis he overstepped himself by dragging in the mental life of a human computer."

One question remains however. What, if not the nature of the human mind, underlay Gödel's conviction that computational procedures should be finite and mechanical? As opposed to argument I, argument III provides no such justification; it merely describes a mechanism that obeys these principles. Also, it should be clear from the foregoing discussion that Gödel seriously considered the existence of non-mechanical procedures that transcend their mechanical counterparts. Yet, somehow it seems that he did not ascribe to such procedures any appreciable role in the world of mathematics.

Shagrir (2006) argues that Gödel's fixation on finite and mechanical computation was not so much rooted in exclusionary convictions regarding the nature of human computation, but rather reflected the methods of the mathematical tradition in the context of which he did his work. As we saw earlier, Gödel began his career at the zenith of Hilbert's finitist program. In fact, with his doctoral dissertation he tackled one of the main problems that Hilbert's program was facing at the time. And although it is now generally accepted that Gödel's incompleteness theorems delivered a deathblow to Hilbert's program, Gödel in his 1931 paper still expressed the belief[9] that his second incompleteness theorem does not "contradict

[9] Only two years later, Gödel conceded that it was very unlikely that Hilbert's program could

Hilbert's formalistic viewpoint. For this viewpoint presupposes only the existence of a consistency proof in which nothing but finitary means of proof is used, and it is conceivable that there exist finitary proofs that *cannot* be expressed in the formalism of P ...." (Gödel 1931, p. 197, 1986, p. 195)

Shagrir contrasts Gödel's thorough awareness of the historical context with the relative nescience of Turing, whom he depicts as being "hardly aware of the fierce foundational debates." (Shagrir 2006, p. 412) Congruent with this characterization, Gödel had a tendency of evaluating Turing's work against the backdrop of the foundational developments, showing particular interest in its consequences for his own incompleteness results. Indeed, he found in the Turing machine a "precise and unquestionably adequate definition of the general concept of formal system", which enabled him to confirm that his incompleteness theorems apply not just to the very comprehensive class of systems described in his 1931 paper, but to "*every* consistent formal system containing a certain amount of finitary number theory." (Gödel 1986, p. 195, 1934, p. 369) That the notion of formal system is entirely captured by the definition of the Turing machine he explained by noting that for every formula-enumerating Turing machine, there exists a formal system that has the same provable formulas, and vice versa (computation thus becomes analogous to proof in a formal system).

The restriction to finite mechanical procedures, Gödel argued, "is required by the concept of formal system, whose essence it is that reasoning is completely replaced by mechanical operations on formulas." (Gödel 1934, p. 370) Why, then, should we not allow formal systems that are governed by infinite or non-mechanical procedures? Gödel did not explicitly answer this question; he merely invoked the constraints that were conventionally imposed on the definition of a formal system by formalists such as Hilbert. Shagrir (2006, pp. 410–411) conjectures that underlying these constraints was a principle of "public accessibility": the idea that a description of a proof or procedure must be complete and unambiguous as to ensure its perfect appreciation or execution by another human being. Thus, Sieg (2006, p. 200) argues, "it was the normative demand of radical intersubjectivity between humans that motivated the step from axiomatic to formal systems." This seems a very reasonable demand, but why should intersubjectivity between humans necessarily involve finiteness constraints?

In this context, Shagrir cites Sieg (2006) and Webb (1990) as holding that the finiteness constraints with respect to intersubjectivity, too, trace back to Turing's "limited human memory" argument, thereby cancelling out the essential difference between Turing's arguments I and III and effectively fusing them into a single argument. This analysis is, in my opinion, not entirely satisfactory. While granting the contrived character of the following scenario, it is in theory possible for (idealized) human beings to convey infinite procedures[10] in such a manner that

ever succeed, recognizing that "it seems that not even classical arithmetic can be proved to be non-contradictory by [intuitionistic methods], because this proof ... would be expressible in classical arithmetic itself, which [by the second incompleteness theorem] is impossible." (Gödel 1933, p. 52)

[10] The notion of infinity intended here pertains to the spatial size of the communication by

they can be perfectly interpreted and executed by other human beings. Imagine that one person starts writing an algorithm, line by line. Assuming that this first person continues this process infinitely, another person might at some point start reading and executing the algorithm. Should the second person sooner or later catch up with the first person, or be referenced by the algorithm to a line that has not yet been written down by the first person, the second person will pause the execution of the algorithm until the required line has become available, at which point the execution is resumed. Proceeding in this manner, every instruction will eventually be reached and executed. Furthermore, a process of this kind need not depend on more than a finite number of internal states—and this holds for the writer as well as for the executor of the algorithm.

An alternative, and, in the given context, more satisfactory reason for rejecting infinite procedures can be found in the epistemic role of the formal system (and, by analogy, of computation) in Hilbert's finitist ideology. Recall that Hilbert sought to secure mathematics—including its more controversial transfinite theories—in one complete axiomatic formal system, the consistency of which was to be proved in a metamathematical framework using only the most uncontroversial and intuitive methods. If, however, a complete specification of the axioms and rules governing a formal system cannot be given by a finite communication, it is impossible to reach a conclusive verdict about the consistency of the system by metamathematical considerations—let alone when restricting oneself to finitary methods. Note that I am not saying that, conversely, if such a specification *can* be given by a finite communication, it must be possible to prove or disprove its consistency; merely that to the pre-Gödelian formalist, the latter was at the very least a viable option—as Hilbert's fierce polemic against the pessimistic "ignorabimus" attests—while the possibility of a consistency proof on the basis of an infinite description could be ruled out beforehand.

To sum up: critics have argued that Gödel's finiteness and mechanicalness constraints on formal systems ultimately trace back—be it by a detour—to the same source as Turing's "states of mind" argument, thereby invalidating the distinction between Turing's arguments I and III on which Gödel allegedly based his 1972 critique of Turing's analysis. Our foregoing observations suggest a substantially different explanation though; namely that the constraints that Gödel appealed to were not rooted in the inherent limitations of the human mind, but in the requirement of metamathematical evaluability that was so crucial to Hilbert's foundational program. According to this reading, then, Turing was praised by Gödel for his accurate analysis of finite and mechanical computation by human beings, but at the same time rebuked for supposing that the origins of the finitude lay in the human condition. On Gödel's account, the restrictive factor was not

which a procedure is conveyed rather than the temporal aspects of its execution. A simple example of an infinite procedure in this sense is an algorithm that is given by a (countably) infinite list of instructions, each of which orders the addition of 1 to the value of a certain variable, as opposed to an alternative formulation of the algorithm where the same infinite repetition is instructed through a finite sequence of commands (e.g., in the form of a loop). Of course, not all infinite procedures can be conveyed by a finite number of instructions.

the human mind, which he believed "infinitely surpasses the powers of any finite machine", but, as Shagrir (2006, p. 412) expresses it, the "role [of computation] in the foundational project, which is defining a formal mathematical system."

5.4 Stronger versions

All disagreements on justification aside, Turing and Gödel did agree on the conclusion of Turing's analysis: that all human finite and mechanical computation could be simulated by Turing machines. Although difficult to prove or disprove, to this day, no serious objections to this interpretation of the Church-Turing thesis have emerged. But now that such a precise and powerful model of computation had become available, a new question arose: how powerful is it really? What more, beside human computation, could be accounted for by Turing machines? Does the class of Turing computable processes also include computations by modern electronic computers? In this section, we briefly discuss several previously published proposals to strengthen CTT by extending or reformulating the informal notion of computation (U) that it is supposed to capture.

5.4.1 Machine computation

After 1936, and not unrelated to the work of Turing himself, a shift occurred in the way we think about computation. With the advent of digital electronic computers, we started transferring more and more responsibilities to machines, as they outperformed humans in terms of both speed and reliability on many computational tasks. As a result, when CTT is introduced in contemporary textbooks, we are primed to understand words like "computation" and "computability" in terms of mostly electronic devices.[11] Particularly, the word "algorithm", although having existed for centuries, has evolved to be associated almost exclusively with computing machines, rather than humans. But does the Turing machine, which was modeled after a human being using only pencil and paper, stand up to the mind-blowing feats of modern computing machinery?

Turing analyzed the process of finite and mechanical computation *as carried out by a human being.* Consequently, some of the restrictive conditions described by Turing are relevant specifically to *human* computation, but do not necessarily apply to computation by machines. Robin Gandy, who had been a student and friend of Turing, published an influential paper in 1980 in which he analyzed the concept of machine computation. Although an extension of CTT toward machine computation may appear as trivial to some, Gandy argues that there are some important differences (1980, pp. 124–125):

> [There] are crucial steps in Turing's analysis where he appeals to the fact that the calculation is being carried out by a human being. One such appeal is used to justify the assumption that the calculation proceeds

[11] For concrete examples of this bias toward machine-based interpretations of CTT, see Shagrir (2002, p. 227).

> as a sequence of elementary steps. A human being can only write one symbol at a time. But, if we abstract from practical limitations, we can conceive of a machine which prints an arbitrary number of symbols simultaneously.

Gandy refers to John Conway's Game of Life (Gardner 1970) as an example of such a scenario, emphasizing that there "can be no guarantee that a further effort of imagination may not result in a device to which Turing's analysis is inapplicable." Then follows an analysis of "discrete deterministic mechanical devices" (referred to by some commentators as "Gandy machines"), resulting in the formulation of a set of four general principles. Anything computable by a machine satisfying these principles, Gandy then argues, is computable by a Turing machine.

Analysis *Gandy's thesis*

$$
\begin{aligned}
D &:= \textit{unspecified} \\
U(x) &:= \text{“}x \text{ is calculable by a discrete deterministic machine satisfying Gandy's principles I–IV.”} \\
M &:= \text{Turing machines} \\
C(\mathcal{M}, x) &:= \text{“}\mathcal{M} \text{ computes } x.\text{”}
\end{aligned}
$$

5.4.2 Physical and quantum computation

Another derivative of the Church-Turing thesis goes one step further by asserting that in fact all physical processes are Turing computable. Stephen Wolfram (1985), advancing a "physical form of the Church-Turing hypothesis",[12] suggests that "universal computers are as powerful in their computational capabilities as any physically realizable system can be, so that they can simulate any physical system." (p. 735) While several arguments against physical versions of CTT have emerged in literature, Copeland and Shagrir (2019) point out that most of these arguments are either "highly artificial" or rely on initial conditions whose physical possibility is disputed.

David Deutsch, in a famous paper published in the same year as Wolfram's, considers a similar extension of CTT (Deutsch 1985, p. 99):

> I propose to reinterpret Turing's "functions which would naturally be regarded as computable" as the functions which may in principle be computed by a real physical system. For it would surely be hard to regard a function "naturally" as computable if it could not be computed in Nature, and conversely.

[12] Copeland (1997) advises to abstain from using the names of Church and Turing when denoting "distant cousins" of CTT such as this, "since neither Church nor Turing endorsed, nor even formulated, any such proposition."

By replacing Turing's vague and intuitive notion of computation by this "more physical" and "unambiguous" formulation, Deutsch arrives at "the physical version of the Church-Turing principle": "Every finitely realizible physical system can be perfectly simulated by a universal model computing machine operating by finite means". Note that Deutsch uses the general description of a "universal model computing machine" and avoids stating his thesis specifically in terms of the universal Turing machine. In classical physics, he argues, the latter fails to satisfy the hypothesis since it cannot "perfectly simulate" classical systems, whose possible states[13] "necessarily form a continuum" and therefore outnumber those of any Turing machine (whose possible states are at most countably infinite in number).

Deutsch argues that unlike classical physics, which he discards as being "false", quantum theory is compatible with his physical version of CTT. He observes, however, that every existing model of computation is a classical system in the sense that it assumes the full measurability of all its states; an assumption that is incompatible with quantum theory. A model that can simulate all physical systems down to the quantum level would thus have to be a truly quantum model of computation. To this end, Deutsch describes a quantum generalization of the universal Turing machine. Beside "perfectly simulating every finite, realizible physical system", including any Turing machine, this universal quantum computer $\mathcal{Q}$ would be able to "simulate various physical systems, real and theoretical, which are beyond the scope of the universal Turing machine" (p. 107), including systems that generate true random numbers. Furthermore, Deutsch suggests that quantum parallelism may in some cases offer gains in terms of computation time that could not be achieved in classical models of computation. As remarkable as the physical realization of a quantum computer would be, however, Deutsch admits that the benefits of quantum computation "do not include the computation of non-recursive functions."

Analysis *Deutsch's thesis*

$$
\begin{aligned}
D &:= \text{Physical systems} \\
U(S) &:= \text{“}S \text{ is finitely realizable.”} \\
M &:= \text{The universal quantum computer } \mathcal{Q}^{14} \\
C(\mathcal{M}, S) &:= \text{“}\mathcal{M} \text{ ‘perfectly simulates’ } S\text{, i.e., } \mathcal{M} \text{ is functionally indistinguishable from } S\text{.”}
\end{aligned}
$$

---

[13] In this context, "state" refers not only to what Turing calls "state of mind" or "m-configuration", but to what he would call the "complete configuration", which is the m-configuration, the number of the scanned square, and the contents of the tape together.

[14] Strictly speaking, we should define M here as the set $\{\, \mathcal{Q}_\pi \mid \pi \in \Pi \,\}$, where Π is the set of all possible programs for $\mathcal{Q}$ and $\mathcal{Q}_\pi$ is the universal quantum computer $\mathcal{Q}$ prepared with program π.

Chapter 6

DEFYING THE TURING BARRIER

At the core of even the most sophisticated behavior of today's computers lies a small set of very basic arithmetic and logic operations, none of which transcend the limits of paper and pencil methods. Yet, when considering the previously inconceivable achievements that the digital revolution has made possible, it seems natural to expect that our understanding of the phrase "[that] which would naturally be regarded as computable" has evolved accordingly since 1936. Has our intuitive notion of computability changed fundamentally? Does this new notion entail types of computation that the good old Turing machine fails to capture?

6.1 Three new ingredients

At the turn of the twenty-first century, Wegner (1997) argues that this is indeed the case. He observes that computers at the time no longer follow the straightforward input-output pattern of the Turing machine. Rather, they maintain a constant exchange of information with their environment, receiving input and producing output throughout the entire process of computation. *Interactiveness*, the author claims, has become an essential property of the intuitive notion of computation and leads to behavior that is not reducible to that of a Turing machine.

While Van Leeuwen and Wiedermann (2001a) do consider interaction as a key ingredient to realizing super-Turing computing power, they do not share Wegner's belief that it does so on its own. In addition to interaction, they recognize two more ways in which contemporary computing systems diverge notably from the Turing machine paradigm. First, *infinity of operation* refers to the view according to which the entire lifespan of a computing system is considered a single computation. Even when the machine is switched off, the authors explain, its data can be preserved and used again in future sessions, thereby blurring the boundaries between distinct computations and resulting in one possibly infinite computation. Second, the fact that a system's program may change on the go through hardware or software updates and thus may treat equal inputs differently at different times, is called *non-*

uniformity of programs. Only in the interplay of these three properties, the authors argue, arises behavior that truly transcends the power of the Turing machine. In the following sections, each of the three properties will be discussed separately.

6.1.1 Interaction

Peter Wegner (1997) notices a paradigm shift in the world of computing technology that he characterizes as a shift from algorithmic computation to interactive computation. Traditional computation, as displayed by Turing machines, is a clearly delineated, deterministic process that uses an algorithm to calculate functions on its input. Once the input has been prepared, the machine is set in motion and "plays" the algorithm, not accepting any further communication with the outside world until an output has been produced. Modern computing systems, however, operate in a very different manner. Instead of computing one final output from predefined inputs, they are situated in a rich environment from which they receive a continuous stream of inputs, combining them with earlier results to produce a likewise never-ending stream of outputs. Not only has interaction between a system and its operator become much richer through peripherals such as microphones and cameras, but systems now also communicate intensively with each other through world wide networks such as the internet.

Due to their inability to simulate the passage of time or to process external input while computing, Wegner argues, Turing machines fail to capture interactiveness, which he considers an essential property of modern computation. Consequently, he abandons CTT and introduces the idea of *interaction machines*, which he describes as an extension to the Turing machine that supports dynamic interaction with the outside world. In Wegner (1998) he further explains that interaction machines operate on "interaction histories", which are described as time-sensitive streams that can be interactively extended. Since inputs to Turing machines are finite strings and since functions are too strong an abstraction to model time, Wegner contends that Turing machines are too weak capture interactive behavior. A formally defined example of an interaction machine is presented by Goldin et al. (2004) in the form of the Persistent Turing Machine (PTM), which is essentially a three-tape Turing machine that processes infinite input streams by continuously repeating a process of reading an input from its input tape, processing it on its work tape, and writing an output to its output tape.

We might derive from the theory of interaction machines a CTT version that is represented by the following instantiation of our model:

ANALYSIS *Interactive Church-Turing thesis*

D := The set of all computational processes

$U(x)$:= "x is an interactive computation, i.e., it may receive intermediate inputs and produce intermediate outputs."

M := *Interaction machines* that allow time-dependent dynamic interaction with their environments (e.g., PTMs)

$C(\mathcal{M}, x)$:= "$\mathcal{M}$ can simulate x, including the temporal aspects of its input and output operations."

Although interaction machines might overall provide a more elegant model of the *process* of interactive computation, I am convinced that they are of no added value with respect to the characterization of *computability*. It appears that Wegner's critique of CTT is rooted in a fundamental misconception regarding the very point of the thesis and of computability theory in general. This misconception expresses itself mainly in the C predicate of the above model. CTT was formulated as a statement about the theoretical concept of computability—i.e., *what* can be computed—rather than the practical aspects of computational processes—i.e., *how* it is computed. This may seem counterintuitive, as a major part of CTT consists of a model of computation that defines in detail some process of computation. However, ultimately it is not this specific process that matters to CTT, but the abstract function that it establishes between input and output. It is for this reason that CTT should not be understood as a definition of algorithms in general, as is sometimes wrongly stated, but rather as a characterization of *algorithmic computability*, i.e., the class of functions they can compute.

As is elaborated in Copeland and Shagrir (2019, p. 68), CTT concerns the input-output relation that an algorithmic process realizes, but is indifferent as to the process itself. A computational procedure is considered Turing-computable as long as its input-output function can be computed by a Turing machine. Whether the intermediate steps that the Turing machine uses are identical to those of the procedure is irrelevant. As such, the fact that a definition of the Turing machine does not allow for certain elementary operations such as parallel cell updates in John Conway's Game of Life, is by no means problematic as long as the same final result can be achieved using other (serial) mechanisms. Wegner, however, does not appear to acknowledge this principle (Wegner 1998, p. 317):

> Interaction cannot be expressed by or reduced to transformations (functions). Time is a nonfunctional property since the effect of functions (algorithms) does not depend on their computation time or on the time at which the effect occurs. Interaction extends computing to *computable nonfunctions* over histories rather than *noncomputable functions* over strings. Airline reservation systems and other reactive systems provide

interactive services over time that cannot be specified by functions.

However, while computation time may have an influence on the course of a computation, this does not say anything about the computability of the process.

To illustrate this point, let us consider the following model of interactive computation by Van Leeuwen and Wiedermann (2006), of which preliminary versions appeared in Van Leeuwen and Wiedermann (2000) and Van Leeuwen and Wiedermann (2001c). A component C and its environment E communicate with each other by exchanging symbols. The sequence $e = (e_0, e_1, e_2, \ldots)$ represents the stream of information that C receives from E, where e_t is the symbol that is sent at time step t. Likewise, C's output to E is represented by the sequence $c = (c_0, c_1, c_2, \ldots)$. The *interaction pair* (e, c) then captures the interactive computation of C in response to E. At some moments, C or E may have nothing to send, which is represented by the silent symbol τ. It is assumed that for any t, C first becomes aware of e_t at the next time step $t+1$, and vice versa. Furthermore, communication is always initiated by E, meaning that c_0 has a fixed value of τ. Finally, another important thing to note is that for any t, c_t is fully determined by the combination of C's program and $(e_0, e_1, \ldots, e_{t-1})$, while e_t is indeterministic with respect to the interaction history and can take on any possible value.

The following informal theorem and proof are loosely adapted from Theorem 1 and its proof in Van Leeuwen and Wiedermann (2006):

THEOREM 2

Given a component C and its environment E, it is possible to define a Turing machine $\mathcal{M}_C$ (or, equivalently, recursively define a function) that for any interactive computation (e, c) between C and E and any $t \in \mathbb{N}$, when given $(e_0, e_1, \ldots, e_t)$ as input, produces $(c_0, c_1, \ldots, c_t)$ as output.

Proof. For the sake of convenience, we design $\mathcal{M}_C$ as a three-tape Turing machine with separate input, work, and output tapes. Note however, that it is possible to define an equivalent one-tape Turing machine, as is discussed in chapter 3. Given any sequence $x = (x_0, x_1, \ldots, x_t) \in \{0, 1, \tau\}^*$ on its input tape, we let $\mathcal{M}_C$ simulate C's program, reading the next input symbol whenever C would. Likewise, whenever C would output a symbol, we let $\mathcal{M}_C$ write the same symbol to its output tape, including τ's. $\mathcal{M}_C$ halts after it has processed x_{t-1} (i.e., x_t is ignored). It follows that whenever x equals $(e_0, e_1, \ldots, e_t)$ for some interactive computation (e, c) between C and E, $\mathcal{M}_C$ will output $(c_0, c_1, \ldots, c_t)$. □

Despite the fact that $\mathcal{M}_C$ cannot simulate C's behavior in "real time", accepting new external input as it computes, it does transform the same input into the same output, making it functionally—and thus in terms of computability—equivalent to C. We should conclude that time is an irrelevant factor with regard to questions of computability. The fact that Turing machines cannot perfectly simulate the temporal aspects of interactive processes does not imply anything about the computability of those processes. Wegner's objection that a Turing machine cannot

handle the passage of external time is therefore not a valid argument against the validity of CTT.

6.1.2 Infinity of operation

We have so far paid little attention to the fact that both Wegner (1997) and Van Leeuwen and Wiedermann (2006) mainly consider not finite, but infinite computations. The "infinity" here originates from the fact that modern computers can store the results of their computations, possibly later combining these results with new inputs, thereby forming a process that can be regarded as a single computation that lasts for the entire (theoretically infinite) lifetime of the device.[1] To Wegner's argument the aspect of infinity seems to be no more than an incidental property, while Van Leeuwen and Wiedermann consider it essential for achieving super-Turing computing power. Contrary to earlier papers by the same authors (e.g., Van Leeuwen and Wiedermann, 2001; 2001), where "infinity of operation" is treated as a distinct property, it is here interwoven with the definition of interactive computation, as implied by the so-called *interactiveness condition* (Van Leeuwen and Wiedermann 2006, p. 123):

> For all times t, when E sends a non-τ signal to C at time t, then C sends a non-τ signal to E at some time t' with $t' > t$ (and vice versa).

Any interactive computation obeying this condition, once initiated by E, never ends. Interaction pairs will thus always consist of two infinite sequences. This infinite behavior, the authors claim, leads to super-Turing computation power. I will now present several arguments that contradict this claim.

First, note that the supposed super-Turing power of interactive computations critically depends on these computations being infinite—if this is not the case, ordinary Turing machine suffice to simulate them, as we demonstrated in theorem 2. Yet, I would like to point out that the practice of promoting infinity as an intrinsic property of interactive computation is disputable to say the least. Since computations are processes that necessarily involve a temporal component, their infinity is only a potential one. At no stage in the process, a computation can be said to have reached a state of actual or "completed" infinity. With every passing moment, one of two events occurs: either the process terminates, or it is extended by a finite amount of time. Whether the newly defined end point is definitive or temporary, the length of a computation is always naturally bounded to a finite amount by the "current moment".

This means that, by theorem 2, at any moment in time t, it is possible for any interactive computation (e, c) of some component C and its environment E that started at some time $t' \leq t$ in the past, to define a Turing machine or classically computable function that, given $(e_{t'}, e_{t'+1}, \ldots, e_t)$, reproduces C's entire output history $(c_{t'}, c_{t'+1}, \ldots, c_t)$. How the computation evolves in the future, is

[1] Note that we are here speaking of infinity as a property of an object from the domain (viz., an interactive computation), irrespective of the process that is used to simulate this computation.

computable by neither an interaction machine nor a Turing machine, as a result of the assumption that E's behavior is indeterministic and unpredictable. Before c_{t+k} can be determined for any k, both C itself and any Turing machine simulating C will have to await moment $t+k$, since this is the moment at which they become "aware" of the final symbol that is required for the computation of c_{t+k}. Evidently, in reality there are no interactive input-output relations that a classical Turing machine cannot realize. It is only the choice of representing interactive computations by infinite sequences that creates the false impression of completed infinity, and thus of uncomputability by Turing machines.

For theoretical purposes, however, we might prefer to think of time as a more abstract variable, independently of the way and order in which we humans experience it. In this context, it is certainly interesting to consider infinite time sequences as an exercise to reason about mathematical limits. Here I would like to point out that the concept of infinity itself is nothing new to CTT. While Turing machines are usually defined to compute functions from finite input strings to finite output strings, Turing (1936–7) originally designed a machine that computes the (fractional part of the) decimal expansion of a real number. Since this consists of an infinite series of digits, the process of printing these digits is necessarily infinite.

Suppose that we have a machine $\mathcal{M}$ that computes some computable real number x according to Turing's original conventions. When $\mathcal{M}$ is set to work on an empty tape, its behavior consists of writing down the fractional digits of x one by one. While $\mathcal{M}$'s output string develops progressively and is finite at each moment $t \in \mathbb{N}$, we can imagine its total output as a unique infinite string $\langle x \rangle$ that is the result of taking t to the limit. Recall from section 3.2 that we can equivalently represent x by the set $S_x \subseteq \mathbb{N}$ that contains all indexes i of the binary representation of x that contain a `1`:

$$S_x = \{\, i \mid \langle x \rangle_i = \mathtt{1} \,\} \tag{6.1}$$

The machine $\mathcal{N}$ that decides S_x is then equivalent to $\mathcal{M}$, as a result of the equivalence between x and S_x. Alternatively, we could define $\mathcal{N}$ such that on input $\langle i \rangle$ it not only outputs the single digit $\langle x \rangle_i$, but the entire substring of $\langle x \rangle$ up to and including $\langle x \rangle_i$, i.e., $\langle x \rangle_1^i$. We then have a machine that for any number $i \in \mathbb{N}$ computes the first i decimals[2] of x. Traditionally, those real numbers whose finite approximations can be computed to an arbitrary degree of precision by Turing machines such as $\mathcal{M}$ and $\mathcal{N}$ are considered the "computable" real numbers. Apparently, the fact that these machines are unable to explicitly reproduce the full decimal expansion of those numbers within a finite amount of time has never been considered a serious problem in computability theory. The fact that they can reproduce x up to *any* desired number of decimals, and thus implicitly represent x in its entirety, suffices.

From our observations in this particular example we might infer the more general principle that an infinite sequence may be considered computable if there exists a Turing machine that can compute finite approximations of the sequence up to any degree of accuracy. If we were to extend this principle to the context

[2] Note that while the term "decimal" is somewhat confusing in this context, we refer here to the fractional digits of the *binary* representation of x.

of interactive computations—which I believe is a natural thing to do—this would mean that an interactive computation (e, c) should be considered computable if there is a Turing machine that can compute $(c_0, c_1, \ldots, c_t)$ for any $t \in \mathbb{N}$. From theorem 2 it follows that such a Turing machine can be defined for any interactive computation. Whenever such a machine is provided with a finite prefix of e of any length, it will output the corresponding prefix of c of the same length, much like $\mathcal{N}$ produces the finite prefix $\langle x \rangle_i$ when given i as input. It would therefore be rather inconsistent to consider x computable, while not treating (e, c) as such.

6.1.3 Non-uniformity of programs

In their 2006 paper, Van Leeuwen and Wiedermann aimed to improve Wegner's theory of interactive computation toward super-Turing computing power by supplementing it with their interactiveness condition, which ensures infinite behavior. In Van Leeuwen and Wiedermann (2001a), the authors take their quest for non-effectivity a step further. In addition to accommodating interaction and infinity of operation, they include a mechanism in their model that, separately from the regular input channels, lets external information enter into the system while computing. This mechanism serves to simulate hardware or software updates that might be performed by an external operator during the lifetime of a system. Since interactive computation is viewed as an ongoing process spanning the entire lifetime of a system, intermediate changes in its constitution have an immediate effect on the course of computation. This quality of computing systems of not having a fixed, but evolving program, is referred to by the authors as "non-uniformity of programs".

Interactive Turing machines with advice

Van Leeuwen and Wiedermann ground their theory of interactive, non-uniform computation in the classical Turing machine model. They extend the definition of the Turing machine in two stages. First, they implement interaction and infinity of operation by equipping a Turing machine with finitely many input and output ports, through which it continuously receives and outputs symbols. At each time step, one symbol is delivered to every input port, and one symbol is sent to every output port. Silent moments (both at input ports and output ports) are represented by the "empty symbol" τ. Such an *interactive Turing machine* (ITM) is similar in functionality and computational power to the models found in Van Leeuwen and Wiedermann (2000, 2001c, 2006) and Goldin (2000). In the second stage, non-uniformity of programs is introduced in the model to mimic the modification of hardware or software by an external operator. In choosing an appropriate mechanism to model such changes, the authors take account of two considerations (Van Leeuwen and Wiedermann 2001a, p. 1145):

> We want this change to be quite independent of the current input read by the machine up to the moment of change. If this wouldn't be the case, one could in principle enter ready-made answers to any pending questions or problems that are being solved or computed into

> the machine, which is not reasonable. By a similar argument we do not want the change to be too large, as one could otherwise smuggle information into the machine related to all possible present and future problems.

Van Leeuwen and Wiedermann find a suitable candidate in *advice functions*, as used in the study of non-uniform complexity theory by Karp and Lipton (1980). The general idea behind advice functions was conceived as early as 1939 by Alan Turing, who in his Ph.D. thesis introduced so-called "oracles", which serve as black boxes that provide Turing machines with external, possibly non-computable information (Turing 1939). An advice function is any function $h\colon \mathbb{N} \longrightarrow \{0,1\}^*$ and can be either recursive or not. In order to meet the two "realism conditions" stipulated in the above citation, the authors impose two constraints on the use of advice functions. First, when a (non-interactive) *Turing machine with advice* (TMA) receives an input string x, it may consult its advice function using only the *length* $|x|$ of this input string. This ensures that the advice function does not provide problem-specific information, as required by the first condition. The second condition—that the change should not be too large—is implemented by setting an upper bound for the length of advice strings. Only advice functions h are allowed that, for any $n \in \mathbb{N}$, produce an advice string $h(n)$ of which the length $|h(n)|$ is bounded by a polynomial in n. Hereafter, we will refer to such functions as "polynomially bounded advice functions".

Traditionally used in the context of classical Turing machines, Van Leeuwen and Wiedermann slightly adapt the theory of advice functions in order to use them with interactive Turing machines. Recall that ITMs process one symbol at a time at each input port. Therefore, the principle of calling advice functions with the length of an input string does not directly translate to this context. Instead, the authors choose to let the advice string depend on the time t at which the advice function is called. More specifically, they allow ITMs to call their advice function at any time $t > 0$, but only for values of *at most* t. Note that, since an ITM receives exactly one symbol per input port at each time step, all total input strings received up to moment t are of length t. Again, only advice functions are allowed that are polynomially bounded (this time in t). The resulting composite model, which is an adaptation of the Turing machine that features both interactivity and advice functions, is naturally called an *interactive Turing machine with advice* (ITMA).

The adequacy of advice functions in modeling non-uniformity

Regarding the bare aim of introducing non-uniformity of programs into the Turing machine, advice functions do qualify as effective candidates. When considering them as an integral part of the machine, they can cause the machine to treat one and the same input value differently at different times. On a more practical level however, I believe that as a means of realistically modeling the evolution of a computer program, the use of advice functions leads to undesirable side effects and is moreover superfluous. Since an advice function can be any recursive or non-recursive function,

the abstract mathematical model of Turing machines with advice has super-Turing computing power and is in theory perfectly able to solve all kinds of classically undecidable problems, such as the halting problem. The mechanism that advice functions are supposed to model, however, namely the execution of system updates in a real-world scenario, obviously does not give rise to such super-Turing qualities. This uncomfortable discrepancy is rooted, I think, in two—in the present context somewhat unfortunate—properties of advice functions and can easily be resolved within the bounds of the classical Turing machine paradigm. In this section I will first sketch the undesirable situation, followed by descriptions of the two responsible properties and a proposed solution.

Van Leeuwen and Wiedermann demonstrate the theoretical power of Turing machines with advice by defining a (non-interactive) Turing machine with advice that is able to solve a variant of the halting problem. This variant asks for an algorithm that decides the set K of "encodings of those Turing machines that accept their own encoding" (Van Leeuwen and Wiedermann 2001a, p. 1147). In classical computability theory, K is recursively enumerable. In other words, there exists a classical Turing machine that accepts all members of K, but does not necessarily reject every Turing machine description that does not belong to K within a finite amount of time. Next, the authors propose that there exists a Turing machine with linearly bounded advice that decides K, and therefore does halt on every input (Proposition 5 in Van Leeuwen and Wiedermann 2001a). To prove this proposition, the TMA in question is defined as follows (Van Leeuwen and Wiedermann 2001a, pp. 1147–1148):

> Define an advice function f as follows. For each n it returns the encoding $\langle N \rangle$ of the machine N for which the following holds: $\langle N \rangle$ is of length n, N accepts $\langle N \rangle$, and N halts on input $\langle N \rangle$ after performing the maximum number of steps, where the maximum is taken over all machines with an encoding of length n that accept their own encoding. If no such machine exists for the given n, f returns a fixed default value corresponding to a machine that halts in one step. It is easily seen that f exists and is linearly bounded. On an arbitrary input w, machine A works as follows. First it checks whether w is the encoding of some Turing machine. If not then A rejects w. Otherwise, if w is the encoding $\langle M \rangle$ of some Turing machine M, then A asks its advice for the value $f(n)$, with $n = |w|$. Let $f(n) = \langle N \rangle$, for some machine N. A now simulates both M and N, on inputs $\langle M \rangle$ and $\langle N \rangle$, respectively, by alternating moves, one after the other. Now two possibilities can occur:
>
> 1. N will stop sooner than M. That is, M has not accepted its input by the time N stops. Since the time of accepting $\langle N \rangle$ by N was maximum among all accepting machines of the given size, we may conclude that M does not accept w. In this case A does not accept w either.
> 2. M will stop not later than N. Then A accepts w if and only if M accepts w.

Clearly A stops on every input and accepts K.

Formally, the definition of f perfectly obeys all criteria and constraints set by the authors. It receives only the lengths of A's input values, and is not just polynomially, but even linearly bounded. Yet, it violates both of the previously communicated realism conditions. Due to the deceptively short length of f's output values, one may naively assume that its contribution to the computational process of A is of minor significance. However, I will argue that on the contrary, the solution to the problem lies entirely in these little advice values, while A itself only performs an auxiliary role.

This will become evident when we consider the meaning of f's output value. For any $n \in \mathbb{N}$, $f(n)$ is a Turing machine description that, among all length n descriptions of Turing machines that accept their own encoding, describes a Turing machine that does so after a maximum number of computation steps. Though we cannot specify a full algorithm for f, we can infer from its description the following global definition:

$$f(n) = \begin{cases} \langle \mathcal{N} \rangle & \text{if } K_n = \varnothing \\ \arg\max\limits_{s \in K_n} g(s) & \text{otherwise,} \end{cases} \tag{6.2}$$

where $\mathcal{N}$ is some Turing machine that halts in one step, K_n is the (finite) subset of K that consists of all encodings of length n, and $g\colon K \longrightarrow \mathbb{N}$ is a function such that $g(\langle \mathcal{M} \rangle)$ is the number of steps that a Turing machine $\mathcal{M}$ takes to terminate on its own description.

From this definition, it becomes clear that for each n, the value of $f(n)$ depends on a full knowledge of the set K_n. For example, the arg max function ranges over K_n, which requires some method of deciding which strings belong to K_n and which do not. Since the domain of f is $\mathbb{N}$, and $\bigcup_{n \in \mathbb{N}} K_n = K$, full knowledge of the values of f inevitably implies full knowledge of K. The very functionality that A is designed to achieve is thus already implicitly present in its advice function f. In other words, the actual work of deciding K is done entirely "behind the scenes" by f. The only thing that prevents f from directly providing A with the ready-made answer is the constraint that A may not consult f for a particular input string, but only for its length. However, this has been easily bypassed by letting f "encode" its answer in a generalized form. What remains for the Turing machine itself is a routine procedure that simply recovers an input-specific answer from this generalized answer—note that a single advice value implicitly contains the answers to *all* inputs of a certain length. The fact that this is possible shows that the system fails to meet both realism conditions, which demand that no "ready-made answers to any pending questions" or information "related to all possible present and future problems" can be entered into the machine.

While I am aware that Proposition 5 was presented by Van Leeuwen and Wiedermann as a purely theoretical example with the intention of demonstrating the computational power of advice functions, as opposed to a realistic simulation of the evolution of an actual computing system, it nevertheless shows that the present model fails to safeguard the prerequisite realism conditions set by the authors.

The problem I am trying to point out is not that advice functions are unable to model the non-uniformity that is found in modern computing systems as a result of software or hardware updates; on the contrary, my objection is that they are *too* powerful and that a more natural model should be sought that fits the real-world situation more tightly as to not create false impressions in terms of computing power. I will now discuss the two properties of the present model that I think are mainly responsible for these false impressions and propose a simple solution.

First, the authors' choice to model system updates as a function is, though not incorrect technically, in my opinion unnatural and somewhat misleading. In the ITMA model, the input from the unpredictable environment is assumed to appear spontaneously at the input ports during computation, without specification in advance. The execution of system updates however—while likewise described by the authors as "unpredictable"—is modelled by a definite function, creating the impression that their contents and execution times are established beforehand for the course of the entire computation. Furthermore, functions are often used in mathematics to describe some meaningful pattern, be it definable in a recursive way or not. Especially in the case of advice functions, they are expected to be *prescriptive*: their values are expected to be related to their arguments according to some mathematical principle. In the ITMA model of Van Leeuwen and Wiedermann, on the other hand, advice functions are used in a purely *descriptive* way: there is no meaningful or mathematically predictable relation between arguments (i.e., time) and values (i.e., contents of system updates); it is more or less random. This property, in combination with the infinite duration of computations, does make the advice functions in the ITMA model technically non-recursive, but in a way that is fundamentally different from the non-recursiveness of a procedure that solves the halting problem. Van Leeuwen and Wiedermann recognize this difference (van Leeuwen and Wiedermann 2001a, pp. 1152–1153):

> Do the results in this paper mean that now we are able to solve some concrete, a priori given undecidable problems with them? The answer is negative. What we have shown is that some computational evolutionary processes which by their very definition are of an interactive and unpredictable nature, can be modeled *a posteriori* by interactive Turing machines with advice. In principle, observing such a running process in sufficient detail we can infer only a posteriori, after we noted them, all its computational and evolutionary characteristics (such as the time and the nature of their changes).

In other words, in reality system updates "present" themselves to a computing machine in irregular and unpredictable ways very similar to those in which ordinary input values appear at input ports. I therefore believe that advice functions, which are mainly associated with prescriptive patterns and even *a priori* undecidability, are a rather unnatural choice for modeling system updates.

A second reason for rejecting advice functions as a model of system updates is that advice functions are easily mistaken as being an integral part of the Turing machine, while operators effectuating system changes clearly are not part of the

computer. In fact, in the tradition of oracle machines and Turing machines with advice, machines consulting non-recursive oracles are commonly regarded as having super-Turing computing power. In more abstract complexity-theoretic settings it can be useful to consider the advice function as a component within the machine, but in our concrete scenario it should be clear that the source of changes in system architecture (be it a human operator or another system to which it is connected via, for example, the internet) is strictly external to the machine. Therefore, we should treat the operator's contributions to the computational process as such and avoid the spurious impression that the machine as a whole acquires super-Turing computing powers as a result of the execution of system updates.

In order to better reflect the real-world situation, we could choose not to implement a distinct mechanism within the machine for simulating hardware or software updates, but rather process the contents of these updates via the "regular" input channels. Like input values, we should not assume system updates to follow any predictable pattern. In accordance with Van Leeuwen and Wiedermann's realism conditions, we want their contents to be in principle independent from any ongoing computational processes. It is therefore unnecessary and, as we have seen in the case of advice functions, even rather undesirable to extend existing definitions by equipping Turing machines with dedicated quasi-integrated modules for simulating system updates. A system update should be presented to the machine as an independent input value. This way of modeling system updates does not require the development of any new models, but can be easily implemented within existing frameworks such as the classical Turing machine or Van Leeuwen and Wiedermann's interactive Turing machine.

In interactive Turing machines, for example, one input port may be reserved for receiving system updates. At each time step, there either appears a new update at this port or nothing happens at all. An update, unlike regular input values, may consist of more than one symbol. The machine will at each step of a computation first check for available updates. If any are available, these will be executed before the values at the regular input ports are read. As we have seen in section 6.1.2, infinite interactive computations can be simulated up to an arbitrary finite number of time steps by classical Turing machines. In this scenario, too, can system updates be simulated as regular input values. Similarly to the interactive scenario, one tape of the Turing machine is reserved for system updates. Since only a finite portion of the process is simulated, the number of system updates that could have been executed in the relevant time frame must be finite, too. This means that before the simulation is started, the "update tape" can be prepared with the entire update history, where each update is marked with the time step at which it should be offered to the machine. During simulation, the Turing machine will at each simulated time step check the update tape for updates and, if any are specified for the concerned time step, execute them before processing regular input values.

The approaches presented here ensure that any external information entering the system is explicitly registered as input, thereby preventing the false impression of super-Turing computing power as a result of "hidden" inputs. In the purely hypothetical case then that system updates *are* used to compute, say, non-recursive

functions, it will always be fully transparent that this super-Turing computing power did not arise from within the machine itself but was injected from an external source. In the end we must conclude that non-uniformity of programs as caused by system changes does not pose a significant threat to the classical Turing machine paradigm. The use of advice functions to model this phenomenon, while strictly speaking not "incorrect", turns out to be a superfluous brute force solution to a problem that can be elegantly solved without appeal to any additional machinery.

6.2 The Extended Church-Turing thesis

According to Van Leeuwen and Wiedermann, the three new ingredients described in the previous section put such a strain on the Turing machine paradigm, that in its original form it does not suffice to describe modern computation anymore. They therefore propose to extend the Church-Turing thesis accordingly (Van Leeuwen and Wiedermann 2001a, p. 1143):

Extended Church-Turing thesis *Any (non-uniform interactive) computation can be described in terms of interactive Turing machines with advice.*

Analysis *Extended Church-Turing thesis (ECTT)*

$D :=$ The set of all computational processes

$U(x) :=$ "x is non-uniform and interactive" (as defined in Van Leeuwen and Wiedermann 2001a)

$M :=$ The set of Interactive Turing Machines with Advice (as defined in Van Leeuwen and Wiedermann 2001a)

$C(\mathcal{M}, x) :=$ "Over the course of one (possibly infinite) run, $\mathcal{M}$ can ingest x's full input history and transform it into x's full output history."

On the basis of the arguments presented in the previous section, I believe that there is no necessity for an extension of CTT such as the one proposed by Van Leeuwen and Wiedermann. We have seen that the classical Turing machine is perfectly capable of simulating all three of the new ingredients, showing that they have brought about no fundamental changes in computing power.

When breaking down ECTT as we did in our model above, we may notice another oddity. Most CTT versions compare some model of computation to a class of mathematical or mechanical objects that, depending on the input values that they receive, produce different output values. In other words, these objects, such as functions or physical machines, represent a certain input-output *pattern*. The extended Church-Turing thesis, on the other hand, considers as its domain

not general input-output patterns, but isolates individual processes that convert a distinct input stream into a distinct output stream, without regard for the mechanisms from which they originated. In effect, a model instance $\mathcal{M}$ aiming to simulate some computation need only successfully transform the input to that particular computation into the correct output, in principle allowing arbitrary behavior on any other input. It seems more natural to consider not individual computations, but the components performing them as the domain of this thesis. Using the terminology from Van Leeuwen and Wiedermann (2006), a model instance $\mathcal{M}$ aiming to simulate a component C then needs to produce the correct output to *any* input history originating from the environment E.

Why did Van Leeuwen and Wiedermann not take this approach with their extended Church-Turing thesis? The answer might be found in another disadvantage of the use of advice functions. In order to see this, let us extend the notational conventions of Van Leeuwen and Wiedermann (2006) to include non-uniformity and describe the computations of a non-uniform interactive component C with k input ports and l output ports as an interaction tuple $(e_1, \ldots, e_k, c_1, \ldots, c_l, u)$, abbreviated[3] as $(\mathbf{e}, \mathbf{c}, u)$, where $e_i = ((e_i)_1, (e_i)_2, \ldots)$ is the i-th input stream, $c_j = ((c_j)_1, (c_j)_2, \ldots)$ is the j-th output stream, and $u = (u_1, u_2, \ldots)$ is the update stream. For each $t \in \mathbb{N}^0$, u_n is a string that is either empty or represents the contents of a system update. At each time step t, C first checks whether u_t contains a system update or not. If yes, the system update is processed. Next, C reads and processes the (possibly empty) symbols $\mathbf{e}_t$ that have appeared at its input ports and writes the (possibly empty) symbols $\mathbf{c}_t$ to its output ports.

As with the input streams, the contents of the updates that will be offered to a component are completely unpredictable and unknown at the start of computation. Since the component itself operates deterministically, the course of the computational process is determined entirely by the contents of the input and update streams, which will each evolve along one of infinitely many possible paths. The disadvantage of modeling system updates with advice functions that emerges here is that a given advice function can model only one specific course of the computational process. As a result, each ITMA does not model a non-uniform interactive component as a whole, but only a single possible course of its computational process. If we wish to simulate an alternative course for the same component, we need to replace our advice function, resulting in a different ITMA.

This situation can be easily resolved by adopting the approach outlined at the end of section 6.1.3 and disposing of advice functions altogether, instead reserving a regular input channel for receiving system updates. This means that in our model of ECTT, we can not only change our domain D such that it becomes a thesis about computing devices rather than individual computations, but also define M to be simply the set of Interactive Turing Machines (i.e., without advice), without sacrificing computational power. Furthermore, we established that the interactive computations of ITMs and even ITMAs can be satisfactorily simulated in a finite

[3] We will also write $\mathbf{e}_t$ to denote the t-th symbols of all k input streams, i.e., $(e_1)_t, \ldots, (e_k)_t$. Also, following van Leeuwen and Wiedermann (2006), we will write $pref_t(\mathbf{e})$ to denote the length-t prefixes of all input streams.

way on a regular (i.e., non-interactive and without advice) Turing machine. If we carry through this change in our model, we obtain the following model:

ANALYSIS *Extended Church-Turing thesis (modified)*

$D :=$ The set of discrete modern computing devices

$U(x) :=$ "Each of x's computations can be described by an interaction tuple $(\mathbf{e}, \mathbf{c}, u)$."

$M :=$ The set of classical Turing machines

$C(\mathcal{M}, x) :=$ "For any $t \in \mathbb{N}^0$, when provided with $pref_t(\mathbf{e})$ and $pref_t(u)$ of some computation $(\mathbf{e}, \mathbf{c}, u)$ of x, $\mathcal{M}$ produces $pref_t(\mathbf{c})$ within a finite amount of time."

Chapter 7

DISCUSSION

In this master's thesis we have discussed the historical origins, legacy, and epistemological status of the Church-Turing thesis, as well as its role and relevance in today's computational paradigm. Embedded in the context of Hilbert's formalist program and inspired by Gödel's shocking incompleteness results, the 1930s saw the development of several equivalent models of computation. The quasi-mathematical hypothesis that these models, or specifically that of the Turing machine, adequately capture the intuitive notion of computation that existed at the time, has become known as the Church-Turing thesis. Since the 1930s, the Church-Turing thesis has been the subject of heated philosophical debates, inspired numerous adaptations and extensions, initiated entire new branches of mathematics, and was arguably the spark that ignited the digital revolution which has been radically changing the world to this day. In this final chapter, we will reflect on the contents of this thesis, draw a final conclusion, evaluate our methods and explore further research directions.

7.1 Summary

We designed an analytical framework that uses the common logical structure of CTT and its derivatives to enable their structured breakdown into components and facilitate effective comparison. We used this framework to model several historical and modern versions of CTT, which exposed a great variety in approaches and underlying assumptions about the nature of computation. First, we reviewed the historical background from which CTT emerged and analyzed the original papers by Church, Turing, and others in which it was first communicated. Next, we discussed a number of criticisms and alternative versions that appeared in response to CTT. One particularly interesting philosophical remark came from Kurt Gödel, who on the one hand praised Turing for his correct analysis of human mechanical computation, but on the other hand criticized him for asserting that the finite and mechanical nature of computation was a result of limitations on the human

mind. Gödel believed that instead, the human mind is immaterial and "infinitely surpasses" any finite machine. We argued that from Gödel's perspective, the finite and mechanical nature of computation was more likely to be rooted in the role of formal systems in Hilbert's finitist ideology.

Other CTT versions that we discussed include Gandy's thesis about machine computation, and Deutsch's thesis about physical computation in general. In particular, we analyzed a paper by Van Leeuwen and Wiedermann from 2001, in which the authors argued that due to three new ingredients of modern computing, the Turing machine did no longer suffice in describing the present day computational paradigm. Instead, they proposed a new model called Interactive Turing Machines with Advice (ITMA), which extends the classical Turing machine in two ways. First, *interactivity* and *infinity of operation* are accommodated by allowing the machine to receive a continuous stream of input symbols throughout the process of computation, as opposed to a finite input string that is prepared before the start of computation. Second, *non-uniformity of programs* is accommodated by equipping the machine with an advice function whose values are used to simulate software or hardware updates that are executed in the course of computation.

While the model by Van Leeuwen and Wiedermann is not faulty, we argued that the three ingredients that were put forward by the authors do not give rise to properties that require a more powerful model of computation than the classical Turing machine. For each of the three ingredients, we showed how they can be satisfactorily modeled using a classical Turing machine. While a Turing machine cannot model the temporal aspects of interactive computation, in the sense that it can process input values that are presented spontaneously throughout the process of computation, it can nevertheless in an "offline" fashion transform any finite prefix of one or multiple input streams of an interactive computation into the corresponding output streams. This ability to realize a certain input-output relation, regardless of the temporal or algorithmic particularities, is what constitutes the essence of computing power.

Concerning the alleged infinity of modern computations and the inability of classical Turing machines to simulate this infinity, we argued that, first, in reality computations never actually reach a state of "true" infinity, and second, that the ability to approximate infinite processes up to an arbitrary finite degree is sufficient for the type of simulation considered here. We furthermore showed that non-uniformity of programs, as caused by the execution of system updates, can be perfectly processed via a regular input tape. This solution not only obviates the need for additional machinery, but also makes the computational process more transparent than is the case with advice functions, which obscure part of the transformation process from input to output.

Finally, we used the considerations above to reformulate Van Leeuwen and Wiedermann's Extended Church-Turing thesis in terms of regular Turing machines. This reformulation had the additional advantage that each Turing machine can be used to simulate every possible computational process of a non-uniform interactive computing device, whereas the specificity of advice functions prohibited ITMAs from being able to simulate more than one possible course of a computational

process.

7.2 Conclusion

The world in which the Church-Turing thesis was first posed was in many respects a different world from the one we live in today. In just eighty years, CTT has seen the deceptively simple idea of an automatic computing machine evolve into an omnipresent physical reality that has reshaped our daily lives in ways we could not have imagined. In this light, it is no wonder that the relevance of CTT to today's computational paradigm has been called into question by critics. It is difficult to imagine that a primitive device such as the Turing machine could account for the immensely complex behavior displayed by modern artificial intelligence. Yet, it seems that the Turing machine has stood the test of time surprisingly well. Although modern computers have substantially benefited from an exponential growth of processing power and through peripheral devices and massive interconnectivity can achieve things that a bare Turing machine cannot physically achieve in a reasonable amount of time, "under the hood" the principles governing the computational process have not fundamentally changed.

That said, if we wish to accurately evaluate the relevance and adequateness of CTT in today's computational paradigm, we should first make precise what we mean by "Church-Turing thesis". As we have seen, a plethora of related but essentially different versions of CTT have entered the scene that tend to obscure the original intention of CTT: to characterize the process of *human* computation. Even Gödel, who believed that human mental processes in general could never be simulated by finite machines, adhered to the idea that the process of computation, as carried out by humans, was nevertheless of a finite and mechanical nature. We may safely assert that in its original human interpretation, CTT is as credible today as it was in the 1930s.

Whether the Turing machine is an adequate model of modern computation is, thus, strictly speaking a different question that has nothing to do with the validity of CTT. On the other hand, human computation was virtually the only type of computation that existed in the 1930s, which means that at the time, human computation *was* computation in general. It is therefore still worth asking if our modern understanding of the concept of computation can still be adequately modeled by the Turing machine. As regards the three properties of interaction, infinity of operation, and non-uniformity of programs as described by Van Leeuwen and Wiedermann (2001a), I contend that the Turing machine still suffices and does not need to be replaced with a more extensive or more powerful model.

There are several other possible directions in which CTT can be "stretched". Deutsch (1985) proposed to not only consider computations in the classical sense, as performed by either humans or human-made machines, but physical processes in general. While considering all classical models of computation incompatible with quantum theory and promoting the theoretical quantum computer instead for several alleged benefits, Deutsch admits that even the quantum computer will

not be able to solve classically undecidable problems, such as the halting problem. Another direction that could be explored is inspired by Gödel's argument against the finiteness of the human mind. Note that if all physical processes turned out to be Turing-computable, this would not necessarily invalidate Gödel's argument. As Wang (1974) pointed out, Gödel admitted that the human brain "functions basically like a digital computer". On the other hand, he strongly rejected the idea that there exists no mind separate from matter. A CTT-version that claimed the Turing-computability of all physical processes of the human brain would therefore probably have met with Gödel's approval, whereas he believed that the human mind as a whole, which he might have conceived as an entity consisting of a physical brain receiving input from an immaterial source, does transcend the Turing barrier.

In this thesis we have pointed out that no serious arguments exist against the validity of the original, human version of CTT. Additionally, we have encountered no convincing counterexamples to the adequateness of the Turing machine in modeling today's computational paradigm, which led us to propose an extension of CTT that not only applies to human computation, but to the interactive, infinite, and non-uniform computations as performed by modern computers as well. As several authors such as Emil Post and Stephen Kleene have argued in regard to the epistemological status of CTT in its original form, I believe it is equally difficult if not impossible to provide a conclusive proof for the stronger assertion that modern computation can be simulated by Turing machines. Yet, we can state that its credibility is at least reinforced by the fact that each of the three potential counterexamples discussed in this thesis proved unconvincing after all.

7.3 Further research directions

Of course, by no means do I wish to assert that the survey presented here is exhaustive in the sense that it has considered all available criticisms and alternatives to CTT. A decisive answer about the validity of CTT can therefore never be given. Many interesting developments in modern computing were left unexplored: while we scratched the surface of quantum computing, it would be worth investigating the subject in greater depth. In recent years, quantum computing has attracted attention due to speculations, and finally an official announcement by Google in 2019 that so-called quantum supremacy had been achieved (Arute et al. 2019). While still a major complexity-theoretic breakthrough, quantum supremacy does not mean that functions have become computable that are not (in theory) computable by a Turing machine.

BIBLIOGRAPHY

Ackermann, Wilhelm F.

1928 Zum Hilbertschen Aufbau der reellen Zahlen, *Mathematische Annalen*, vol. 99, pp. 118–133, DOI: 10.1007/BF01459088. English translation in Van Heijenoort (1967, pp. 493–507).

Arute, Frank, Kunal Arya, Ryan Babbush, Dave Bacon, et al.

2019 Quantum supremacy using a programmable superconducting processor, *Nature*, vol. 574, pp. 505–510, DOI: 10.1038/s41586-019-1666-5.

Bernstein, Ethan and Umesh Vazirani

1997 Quantum Complexity Theory, *SIAM Journal on Computing*, vol. 26, pp. 1411–1473, DOI: 10.1137/S0097539796300921.

Cabessa, Jérémie and Alessandro E. P. Villa

2014 On Interactively Computable Functions, http://jcabessa.byethost32.com/papers/CabessaVillaCiE14.pdf (visited on May 15, 2020).

Church, Alonzo

1932 A Set of Postulates for the Foundation of Logic, *Annals of Mathematics*, vol. 33, pp. 346–366, DOI: 10.2307/1968337.

1933 A Set of Postulates For the Foundation of Logic (second paper), *Annals of Mathematics*, vol. 34, pp. 839–864, DOI: 10.2307/1968702.

1936a An Unsolvable Problem of Elementary Number Theory, *American Journal of Mathematics*, vol. 58, pp. 345–363, DOI: 10.2307/2371045.

1936b A Note on the Entscheidungsproblem, *The Journal of Symbolic Logic*, vol. 1, pp. 40–41, DOI: 10.2307/2269326.

1937a [Review of "On Computable Numbers, with an Application to the Entscheidungsproblem" by A. M. Turing], *The Journal of Symbolic Logic*, vol. 2, pp. 42–43, DOI: 10.2307/2268810.

1937b [Review of "Finite combinatory processes—formulation 1" by E. L. Post], *The Journal of Symbolic Logic*, vol. 2, p. 43, DOI: 10.2307/2268811.

1938 The constructive second number class, *Bulletin of the American Mathematical Society*, vol. 44, pp. 224–232, DOI: 10.1090/S0002-9904-1938-06720-1.

Church, Alonzo and J. Barkley Rosser

1936 Some Properties of Conversion, *Transactions of the American Mathematical Society*, vol. 39, pp. 472–482, DOI: 10.2307/1989762.

Cook, Stephen A. and Robert A. Reckhow

1973 Time bounded random access machines, *Journal of Computer and System Sciences*, vol. 7, pp. 354–375, DOI: 10.1016/S0022-0000(73)80029-7.

Copeland, B. Jack

1997 The Church-Turing Thesis, in *Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, https://plato.stanford.edu/entries/church-turing/ (visited on May 29, 2020).

2004 (ed.) *The Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life: Plus The Secrets of Enigma* (Oxford: Clarendon Press).

Copeland, B. Jack and Oron Shagrir

2019 The Church-Turing thesis: logical limit or breachable barrier?, *Communications of the ACM*, vol. 62, no. 1, pp. 66–74, DOI: 10.1145/3198448.

Curry, Haskell B.

1930 Grundlagen der Kombinatorischen Logik, *American Journal of Mathematics*, vol. 52, pp. 509–536, DOI: 10.2307/2370619.

Davies, Donald W.

2004 Corrections to Turing's Universal Computing Machine, in Copeland (2004), pp. 103–124.

Davis, Martin D.

1965 (ed.) *The Undecidable: Basic Papers On Undecidable Propositions, Unsolvable Problems And Computable Functions* (Hewlett, N.Y.: Raven Press).

1982 Why Gödel didn't have Church's thesis, *Information and Control*, vol. 54, pp. 3–24, DOI: 10.1016/S0019-9958(82)91226-8.

2018 *The Universal Computer: The Road from Leibniz to Turing*, 3rd ed. (Boca Raton, Fla.: CRC Press), DOI: 10.1201/9781315144726.

Davis, Martin D., Ron Sigal, and Elaine J. Weyuker

1994 *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd ed. (San Francisco: Morgan Kaufmann), DOI: 10.1016/B978-0-08-050246-5.50012-X.

Deutsch, David E.

1985 Quantum theory, the Church-Turing principle and the universal quantum computer, *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, vol. 400, pp. 97–117, DOI: 10.1098/rspa.1985.0070.

Ewald, William B.

1996 (ed.) *From Kant to Hilbert: A Source Book in the Foundations of Mathematics*, vol. 2 (Oxford: Oxford University Press).

Gandy, Robin O.

1980 Church's Thesis and Principles for Mechanisms, in *Studies in Logic and the Foundations of Mathematics*, vol. 101: *The Kleene Symposium*, ed. by Jon Barwise, H. Jerome Keisler, and Kenneth Kunen (Elsevier), pp. 123–148, DOI: 10.1016/S0049-237X(08)71257-6.

1988 The Confluence of Ideas in 1936, in *The Universal Turing Machine: A Half-Century Survey*, ed. by Rolf Herken, 2nd ed. (Oxford: Oxford University Press), pp. 51–102.

Gardner, Martin

1970 Mathematical Games. The fantastic combinations of John Conway's new solitaire game "life", *Scientific American*, vol. 223, no. 4, pp. 120–123, DOI: 10.1038/scientificamerican1070-120.

Gödel, Kurt F.

1929 Über die Vollständigkeit des Logikkalküls, Doctoral dissertation, University of Vienna. Reprinted and translated in Gödel (1986, pp. 60–101), rewritten in Gödel (1930).

1930 Die Vollständigkeit der Axiome des logischen Funktionenkalküls, *Monatshefte für Mathematik und Physik*, vol. 37, pp. 349–360, DOI: 10.1007/BF01696781. English translation in Van Heijenoort (1967, pp. 582–591) and Gödel (1986, pp. 102–123).

1931 Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatshefte für Mathematik und Physik*, vol. 38, pp. 173–198, DOI: 10.1007/BF01700692. English translations in Davis (1965, pp. 4–38), Van Heijenoort (1967, pp. 596–616), and Gödel (1986, pp. 144–195).

1933 The present situation in the foundations of mathematics, in Gödel (1995), pp. 45–53.

1934 On undecidable propositions of formal mathematical systems, in Gödel (1986), pp. 346–371. Notes taken by S. C. Kleene and J. B. Rosser of a series of lectures given by Gödel at the Institute for Advanced Study in Princeton; first printed in Davis (1965, pp. 41–74).

193? ⟦Undecidable diophantine propositions⟧, in Gödel (1995), pp. 164–175. Manuscript taken from handwritten notes in English.

1951 Some basic theorems on the foundations of mathematics and their implications, in Gödel (1995), pp. 304–323. Manuscript of Gödel's "Gibbs Lecture", given at Brown University.

1972 Some Remarks on the Undecidability Results, in Gödel (1990), pp. 305–306.

1986 *Collected Works*, vol. I: *Publications 1929–1936*, ed. by Solomon Feferman, John W. Dawson Jr., Stephen C. Kleene, Gregory H. Moore, et al. (New York: Oxford University Press).

Gödel, Kurt F.

1990 *Collected Works*, vol. II: *Publications 1938–1974*, ed. by Solomon Feferman, John W. Dawson Jr., Stephen C. Kleene, Gregory H. Moore, et al. (New York: Oxford University Press).

1995 *Collected Works*, vol. III: *Unpublished essays and lectures*, ed. by Solomon Feferman, John W. Dawson Jr., Warren Goldfarb, Charles Parsons, et al. (New York: Oxford University Press).

Goldin, Dina Q.

2000 Persistent Turing machines as a model of interactive computation, in *Lecture Notes in Computer Science*, vol. 1762: *Foundations of Information and Knowledge Systems*, First International Symposium, FoIKS 2000, ed. by Klaus-Dieter Schewe and Bernhard Thalheim (Berlin Heidelberg: Springer-Verlag), pp. 116–135, DOI: 10.1007/3-540-46564-2_8.

Goldin, Dina Q., Scott A. Smolka, Paul C. Attie, and Elaine L. Sonderegger

2004 Turing machines, transition systems, and interaction, *Information and Computation*, vol. 194, pp. 101–128.

Graves, Alex, Greg Wayne, and Ivo Danihelka

2014 *Neural Turing Machines*, arXiv: 1410.5401.

Hartmanis, Juris and Richard E. Stearns

1965 On the Computational Complexity of Algorithms, *Transactions of the American Mathematical Society*, vol. 117, pp. 285–306, DOI: 10.2307/1994208.

Hilbert, David

1900 Mathematische Probleme, *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-physikalische Klasse*, pp. 253–297. Lecture given at the *International Congress of Mathematicians* in Paris, held 6–12 August 1900. Partial English translation in Ewald (1996, pp. 1096–1105).

1917 Axiomatisches Denken, *Mathematische Annalen*, vol. 78, pp. 405–415. Lecture delivered to the *Schweizerische Mathematische Gesellschaft*, 11 September 1917. English translation in Ewald (1996, pp. 1105–1115).

1922 Neubegründung der Mathematik. Erste Mitteilung, *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, vol. 1, pp. 157–177, DOI: 10.1007/BF02940589. Series of lectures given at the University of Hamburg, 25–27 July 1921. English translation in Ewald (1996, pp. 1115–1134) and Mancosu (1998, pp. 198–214).

1923 Die logischen Grundlagen der Mathematik, *Mathematische Annalen*, vol. 88, pp. 151–165, DOI: 10.1007/BF01448445. Lecture delivered to the *Deutsche Naturforscher Gesellschaft* in Leipzig, September 1922. English translation in Ewald (1996, pp. 1134–1148).

1926 Über das Unendliche, *Mathematische Annalen*, vol. 95, pp. 161–190, DOI: 10.1007/BF01206605. Lecture delivered at the Weierstraß memorial meeting of the *Westfälische Mathematische Gesellschaft* in Münster, 4 June 1925. English translation in Van Heijenoort (1967, pp. 367–392).

1929 Probleme der Grundlegung der Mathematik, in *Atti del Congresso Internazionale dei Matematici*, Bologna 3–10 settembre 1928 (VI) (Bologna: Nicola Zanichelli), vol. 1, pp. 135–141. Reprinted in Hilbert (1930b), of which an English translation appeared in Mancosu (1998, pp. 227–233).

1930a Naturerkennen und Logik, *Naturwissenschaften*, vol. 18, pp. 959–963, DOI: 10.1007/BF01492194. Lecture given at the *Kongress der Gesellschaft Deutscher Naturforscher und Ärzte* in Königsberg, September 1930. English translation in Ewald (1996, pp. 1157–1165).

1930b Probleme der Grundlegung der Mathematik, *Mathematische Annalen*, vol. 102, pp. 1–9, DOI: 10.1007/BF01782335. English translation in Mancosu (1998, pp. 227–233).

Hilbert, David and Wilhelm F. Ackermann

1928 *Grundzüge der theoretischen Logik* (Berlin: Julius Springer). English translation of the second edition (1938) in Hilbert and Ackermann (1950).

1950 *Principles of Mathematical Logic*, ed. by Robert E. Luce, trans. by Lewis M. Hammond, George G. Leckie, and F. Steinhardt (New York: Chelsea Publishing Company).

Hodges, Andrew

2012 *Alan Turing: The Enigma* (London: Vintage). Originally published in 1983 (UK: Burnett Books/Hutchinson; USA: Simon & Schuster).

Hofstadter, Douglas R.

1979 *Gödel, Escher, Bach: an Eternal Golden Braid* (New York: Basic Books).

Immerman, Neil

2004 Computability and Complexity, in *Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, https://plato.stanford.edu/entries/computability/ (visited on Jan. 10, 2020).

Karp, Richard M. and Richard J. Lipton

1980 Some Connections Between Nonuniform and Uniform Complexity Classes, in *STOC '80: Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, ed. by Raymond E. Miller, Walter A. Burkhard, Richard J. Lipton, and Seymour Ginsburg (New York: Association for Computing Machinery), pp. 302–309, DOI: 10.1145/800141.804678.

Kleene, Stephen C.

1936a General recursive functions of natural numbers, *Mathematische Annalen*, vol. 112, pp. 727–742, DOI: 10.1007/BF01565439.

1936b λ-definability and recursiveness, *Duke Mathematical Journal*, vol. 2, pp. 340–353, DOI: 10.1215/S0012-7094-36-00227-2.

1938 On Notation for Ordinal Numbers, *The Journal of Symbolic Logic*, vol. 3, pp. 150–155, DOI: 10.2307/2267778.

Kleene, Stephen C.

1943 Recursive Predicates and Quantifiers, *Transactions of the American Mathematical Society*, vol. 53, pp. 41–73, DOI: 10.2307/1990131.

1967 *Mathematical Logic* (New York: Wiley).

1981 Origins of Recursive Function Theory, *Annals of the History of Computing*, vol. 3, pp. 52–67, DOI: 10.1109/MAHC.1981.10004.

Kleene, Stephen C. and J. Barkley Rosser

1935 The Inconsistency of Certain Formal Logics, *Annals of Mathematics*, vol. 36, pp. 630–636, DOI: 10.2307/1968646.

Lewis, Harry R. and Christos H. Papadimitriou

1998 *Elements of the Theory of Computation*, 2nd ed. (Upper Saddle River, N.J.: Prentice-Hall).

Mancosu, Paolo

1998 (ed.) *From Brouwer to Hilbert: The debate on the foundations of mathematics in the 1920s* (New York: Oxford University Press).

Mancosu, Paolo and Richard Zach

2015 Heinrich Behmann's 1921 lecture on the decision problem and the algebra of logic, *The Bulletin of Symbolic Logic*, vol. 21, pp. 164–187, DOI: 10.1017/bsl.2015.10.

Marr, David C.

1982 *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information* (San Francisco: W.H. Freeman and Company).

Post, Emil L.

1936 Finite combinatory processes—formulation 1, *The Journal of Symbolic Logic*, vol. 1, pp. 103–105, DOI: 10.2307/2269031.

1941 Absolutely unsolvable problems and relatively undecidable propositions: Account of an anticipation, in Davis (1965), pp. 340–433.

1944 Recursively enumerable sets of positive integers and their decision problems, *Bulletin of the American Mathematical Society*, vol. 50, pp. 284–316, DOI: 10.1090/S0002-9904-1944-08111-1.

1947 Recursive unsolvability of a problem of Thue, *The Journal of Symbolic Logic*, vol. 12, pp. 1–11, DOI: 10.2307/2267170.

Rosser, J. Barkley

1936 Extensions of Some Theorems of Gödel and Church, *The Journal of Symbolic Logic*, vol. 1, pp. 87–91, DOI: 10.2307/2269028.

Russell, Bertrand A. W.

1902 Letter to Frege, in van Heijenoort (1967), pp. 124–125.

Schönfinkel, Moses I.

1924 Über die Bausteine der mathematischen Logik, *Mathematische Annalen*, vol. 92, pp. 305–316, DOI: 10.1007/BF01448013.

Shagrir, Oron

2002 Effective Computation by Humans and Machines, *Minds and Machines*, vol. 12, pp. 221–240, DOI: 10.1023/A:1015694932257.

2006 Gödel on Turing on Computability, in *Ontos Mathematical Logic*, vol. 1: *Church's Thesis After 70 Years*, ed. by Adam Olszewski, Jan Woleński, and Robert Janusz (Frankfurt: De Gruyter, Ontos Verlag), pp. 393–419.

Shannon, Claude E.

1956 A universal Turing machine with two internal states, in *Annals of Mathematics Studies*, vol. 34: *Automata Studies*, ed. by Claude E. Shannon and John McCarthy (Princeton: Princeton University Press), pp. 157–165, DOI: 10.1515/9781400882618-007.

Sieg, Wilfried

1994 Mechanical Procedures and Mathematical Experience, in *Mathematics and Mind*, ed. by Alexander George (New York: Oxford University Press), pp. 71–117.

1997 Step by Recursive Step: Church's Analysis of Effective Calculability, *The Bulletin of Symbolic Logic*, vol. 3, pp. 154–180, DOI: 10.2307/421012.

2002 Calculations by man and machine: conceptual analysis, in *Lecture Notes in Logic*, vol. 15: *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, ed. by Wilfried Sieg, Richard Sommer, and Carolyn Talcott (Ithaca, N.Y.: Association for Symbolic Logic, Cambridge University Press), pp. 390–409, DOI: 10.1017/9781316755983.019.

2006 Gödel on computability, *Philosophia Mathematica*, vol. 14, pp. 189–207, DOI: 10.1093/philmat/nkj005.

Siegelmann, Hava T. and Eduardo D. Sontag

1992 On the computational power of neural nets, in *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, ed. by David Haussler (New York: Association for Computing Machinery), pp. 440–449, DOI: 10.1145/130385.130432.

Sipser, Michael F.

2013 *Introduction to the Theory of Computation*, 3rd ed. (Boston, Mass.: Cengage Learning).

Turing, Alan M.

1936–7 On Computable Numbers, with an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society*, 2nd ser., vol. 42, pp. 230–265, DOI: 10.1112/plms/s2-42.1.230. Corrected in Turing (1938).

Turing, Alan M.

1938 On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction, *Proceedings of the London Mathematical Society*, 2nd ser., vol. 43, pp. 544–546, DOI: 10.1112/plms/s2-43.6.544.

1939 Systems of Logic Based on Ordinals, *Proceedings of the London Mathematical Society*, 2nd ser., vol. 45, pp. 161–228, DOI: 10.1112/plms/s2-45.1.161.

1947 Lecture on the Automatic Computing Engine, in Copeland (2004).

1948? Draft of a Letter from Turing to Alonzo Church Concerning the Post Critique, in Copeland (2004), p. 102.

Van Heijenoort, Jean L. M.

1967 (ed.) *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931* (Cambridge, Mass.: Harvard University Press).

Van Leeuwen, Jan and Jiří Wiedermann

2000 On Algorithms and Interaction, in *Lecture Notes in Computer Science*, vol. 1893: *Mathematical Foundations of Computer Science 2000*, 25th International Symposium, MFCS 2000, ed. by Mogens Nielsen and Branislav Rovan (Berlin Heidelberg: Springer-Verlag), pp. 99–113, DOI: 10.1007/3-540-44612-5_7.

2001a The Turing Machine Paradigm in Contemporary Computing, in *Mathematics Unlimited — 2001 and Beyond*, ed. by Björn Engquist and Wilfried Schmid (Berlin Heidelberg: Springer-Verlag), pp. 1139–1155, DOI: 10.1007/978-3-642-56478-9_59.

2001b Beyond the Turing limit: Evolving interactive systems, in *Lecture Notes in Computer Science*, vol. 2234: *SOFSEM 2001: Theory and Practice of Informatics*, 28th Conference on Current Trends in Theory and Practice of Informatics, ed. by Leszek Pacholski and Peter Ružička (Berlin Heidelberg: Springer-Verlag), pp. 90–109, DOI: 10.1007/3-540-45627-9_8.

2001c A computational model of interaction in embedded systems. Technical Report UU-CS-2001-02, Department of Computer Science, Utrecht University.

2006 A Theory of Interactive Computation, in *Interactive Computation: The New Paradigm*, ed. by Dina Q. Goldin, Scott A. Smolka, and Peter Wegner (Berlin Heidelberg: Springer-Verlag), pp. 119–142, DOI: 10.1007/3-540-34874-3_6.

Wang, Hao

1974 *From Mathematics to Philosophy* (London: Routledge & Kegan Paul), DOI: 10.4324/9781315542164.

1987 *Reflections on Kurt Gödel* (Cambridge, Mass.: The MIT Press).

Webb, Judson C.

1980 *Synthese library*, vol. 137: *Mechanism, Mentalism, and Metamathematics: An Essay on Finitism* (Dordrecht: Springer Science+Business Media), DOI: 10.1007/978-94-015-7653-6.

1990 Introductory note to [Gödel (1972)], in Gödel (1990), pp. 281–304.

Wegner, Peter

1997 Why Interaction is More Powerful Than Algorithms, *Communications of the ACM*, vol. 40, no. 5, pp. 80–91, DOI: 10.1145/253769.253801.

1998 Interactive foundations of computing, *Theoretical computer science*, vol. 192, pp. 315–351, DOI: 10.1016/S0304-3975(97)00154-0.

Weiss, Gail, Yoav Goldberg, and Eran Yahav

2018 *On the Practical Computational Power of Finite Precision RNNs for Language Recognition*, arXiv: 1805.04908.

Wolfram, Stephen

1985 Undecidability and Intractability in Theoretical Physics, *Physical Review Letters*, vol. 54, pp. 735–738, DOI: 10.1103/PhysRevLett.54.735.

Zach, Richard

2003 Hilbert's Program, in *Stanford Encyclopedia of Philosophy*, ed. by Edward N. Zalta, https://plato.stanford.edu/entries/hilbert-program/ (visited on Dec. 17, 2019).