Draft from November 13, 2023

Contents

Preamble

This book is an introduction to computability theory as well as to three of its main ramifications, namely, algorithmic randomness, reverse mathematics and higher computability theory. It is mainly intended for research master students and teachers in Computer Science and Mathematics, as well as for researchers wishing to acquire a solid knowledge of computability theory.

Reason for existence of the book

This book was initially written in French, aiming to make up for the lack of reference book on classical computability theory in the French literature. However, while writing this book, the authors realized this project was also for interest outside of the French community for the follows reasons:

There are many reference books in English on computability theory (*Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers* by Piergiorgio Odifreddi [177], *Computability Theory* by Barry Cooper [44] or *Turing computability : Theory and Applications* by Robert Soare [216]). Concerning algorithmic randomness, we will cite *Computability and Randomness* by André Nies [172], and *Algorithmic Randomness and Complexity* by Rodney Downey and Denis Hirschfeldt [53]. In reverse mathematics, the historical reference is *Subsystems of Second Order Arithmetic* by Stephen Simpson [211]. We will mention the more recent work by Denis Hirschfeldt, *Slicing the Truth* [91], and *Reverse Mathematics: Problems, Reductions, and Proofs* by Damir Dzhafarov and Carl Mummert. Finally, in higher computability theory, the two references are *Higher Recursion Theory* by Gerald Sacks [200] and *Recursion Theory: Computational Aspects of Definability* by Chi Tat Chong and Liang Yu [38]. Each of these works presents the state of the art of research for a specific sub-domain of computability theory, but there is no single book providing a consistent presentation of these different aspects.

Organization of the book

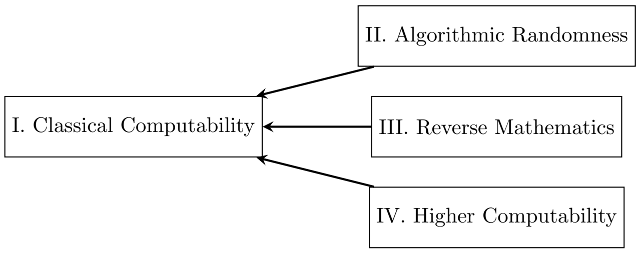
This book is structured in four main parts, namely classical computability theory, algorithmic randomness, reverse mathematics and higher computability theory.

General plan

- *Classical computability theory* is the study of Turing degrees, in other words, the computational power of sets of natural integers. It constitutes the historical heart of computability theory, and the epistemic base on which the following three parts are based.
- *Algorithmic randomness* uses classical computability theory to give an effective framework to measure theory, which makes it possible to study individually the sequences of bits known as "random". The hierarchies induced by classical computability theory make it possible to define various levels of randomness, in the light of which one can, for example, re-examine the meaning of such or such probabilistic theorem stating that a property is true almost everywhere.
- *Reverse mathematics* form a program on the foundations of mathematics, which aims to identify the axioms necessary to prove mathematical theorems of everyday life[1]. They are based on a basic theory, RCA_0, whose axioms capture "computable" mathematics, thanks to a correspondence between computability and definability by logical formulas.
- *Higher computability theory* extends the notion of computability to a more general framework joining set theory. Just as elementary arithmetic operations extend to ordinals, Turing machines can extend their computing time from integers to ordinals, and thus handle larger classes of reals. This is the approach by "models of computation" of hypercomputing, which corresponds, as for classical computability theory, to certain logical classes.

The last three parts all rely heavily on classical computability theory, but are relatively independent from each other, and can be mostly read in any order:

[1]From the everyday life of the mathematician.



Dependencies between the four main parts of the book

Let us note that the notion of "Borel class" introduced in Part II is fundamental for the understanding of Part IV.

Classical computability theory

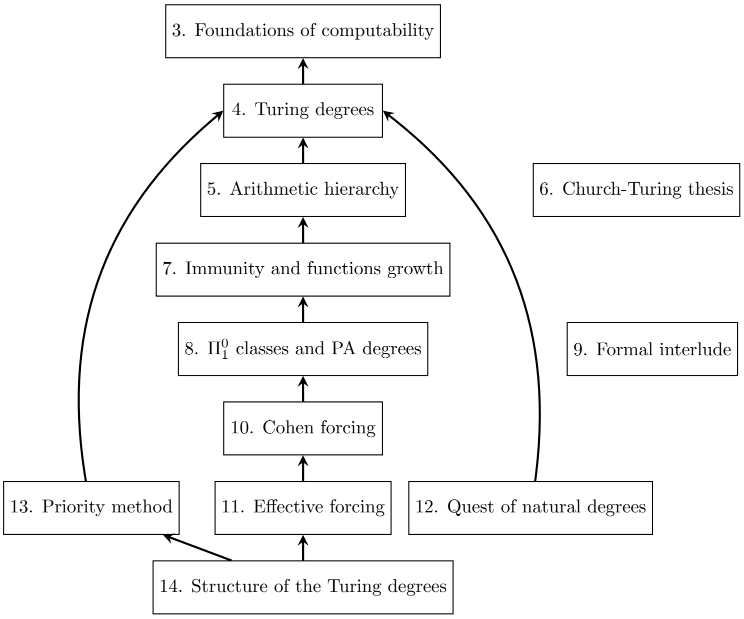
Classical computability theory has a preponderant role, in that it fixes a formal framework and a series of tools which will be used to develop the following parts. It is therefore advisable to linger on the first part and to detail the dependencies of its chapters. The fundamental chapters are mainly to be read linearly, with the following exceptions: chapters 6 and 9 can be read independently of the others, but will nevertheless be useful to serenely approach Part III of the book, on reverse mathematics. Chapters 12 and 14 will not be absolutely necessary for the understanding of the following parts, and aim to take a step back from our work. Chapter 12 — less technical— will do this through the examination of a specific question, on the borderline of philosophy; and Chapter 14 through a more abstract study of the structure of Turing degrees, and the presentation of some of the major open questions in the field.

How to read this book

For teachers

This book can be used as a support for an introductory course on computability theory at master's level, as well as for more advanced thematic courses, talking about algorithmic randomness, reverse mathematics and higher computability theory. The topics covered go well beyond the knowledge of computability theory that one would expect from a master's student. We are therefore going to propose a lesson plan containing the essential concepts.

The equivalence between the computational models forms a robust basis for the development of computability theory. However, the proofs can seem


3. Foundations of computability
4. Turing degrees
5. Arithmetic hierarchy
6. Church-Turing thesis
7. Immunity and functions growth
8. Π^0_1 classes and PA degrees
9. Formal interlude
10. Cohen forcing
13. Priority method
11. Effective forcing
12. Quest of natural degrees
14. Structure of the Turing degrees

Dependencies between the chapters of classical computability theory

quite long and tedious. Nowadays, with the popularization of computers, one can expect that students will have a certain intuition of what an algorithm is, and it seems better to start from this intuition to make the first developments in order to avoid a relatively heavy formalism. We recommend to approach the equivalence of computational models, in particular between general recursive functions and Turing machines, through a tutorial session, where students will have the opportunity to manipulate the formalisms by defining more and more complicated functions to convince oneself that these definitions allow to capture all algorithms.

We invite our readers to follow the developments of the various chapters 3, 4, 5, 7, 8, 10, 13 in that order. Chapter 3 establishes the first fundamental theorems of computability theory on the basis of the intuition that we have of algorithms. Chapter 4 defines the notions of oracle machine and of Turing degree. We find there the most central definitions of computability theory, such as the Turing jump, and the finite extension method, which is a very powerful technique to prove the existence of some Turing degrees. Chapter 5 on the arithmetic hierarchy establishes an essential link between the computational power of sets of integers and their definability by arithmetic formulas, through Post's theorem.

With Chapter 7, we begin the study of various fundamental computational properties, such as the notion of hyperimmune degree, and its links with the existence of fast-growing functions. This study is continued in Chapter 8 on Π^0_1 classes, where we define the notion of PA degree which is a central notion in computability theory. It is found throughout this book, particularly in algorithmic randomness and reverse mathematics.

Chapter 10 introduces a fundamental technique of computability theory, namely forcing, presented here as an elaboration of the finite extension method of Chapter 4. This chapter can also serve as the first step of an incremental understanding of the general technique of forcing in set theory.

Chapter 13 finally introduces the priority methods, another fundamental technique of computability theory, which makes it possible in particular to show the existence of some computably enumerable degrees.

For students

The skills required to understand the concepts presented in this work are those of a first year of a bachelor's degree in Computer Science or Mathematics. It is essential to understand the usual mathematical language (variables, quantifications, etc.), to have some elementary notions of logic (proof by contraposition, proof by the absurd, etc.), and to understand the elements of the basic mathematical corpus (understanding what a bijection is, what an intersection between two sets is, the power and logarithm functions, etc.). In addition to that, at least a basic experience in programming, or an understanding of what an algorithm is, is also necessary to serenely approach the reading of this book.

The mathematics that we will use and which are not taught in the first year of the license will be introduced and explained as and when required (basic notions of topology or measure theory for example). Having established this, let us note all the same that the degree of elaboration of the proofs, as well as the technicality of certain concepts, will undoubtedly make this work difficult to approach without a certain mathematical maturity.

The techniques developed in computability theory are quite different from those learned through a standard mathematical course. This particularity of computability theory is a strength and makes this discipline more accessible since it is not very sensitive to the gaps that one may have developed during his course (or his lack of course). On the other hand, this difference can also destabilize the student because it requires creating a conceptual universe. It goes without saying that in the absence of a teacher, it is all the more essential to do the exercises suggested in the book to properly integrate the concepts. The solutions are available at the end of the book.

The size of this book can be overwhelming for a student wanting to take his first steps in computability theory. We remind you that this book covers knowledge going far beyond what is expected of a master's student. We therefore recommend that autodidacts follow the lesson suggestion in the previous section, intended for teachers.

For researchers

This book is an introduction to computability theory and several of its main branches. However, we should not stop at the introductory aspect of this work, because most of the results presented correspond to the state of the art of research. This book is therefore aimed at researchers in related fields, wishing to acquire solid knowledge in computability theory, as well as Ph.D. students and researchers intending to do some research in computability theory. Indeed, the techniques and concepts of this book make the research articles in the field directly accessible.

Exercises

The chapters are interspersed with exercises of varying difficulty, the correction of which is given at the end of the book. We cannot stress enough the importance of doing exercises to properly assimilate the concepts presented in the chapters. The intuition of concepts is created by manipulating them in all their forms. The difficulty of the exercises is indicated using a star system ($\star$) ranging from 0 to 3: an exercise with no star is a direct application of the definitions, while a two-star exercise requires a deep mastery of concepts to be solved. There are also some three-star exercises, which are "research" level.

Through this work, we will present many computational properties on sets of integers or on other more complex structures. In addition to the given exercises, it is important to show an intellectual curiosity consisting in systematically seeking how these properties combine, knowing whether one can construct objects satisfying several of them simultaneously, and so on. Likewise, when the theorems have hypotheses, it is useful to look for counter-examples without these hypotheses, in order to better understand their necessity as well as their use in the proof.

Errata

You can't write a book of this size without letting a number of typos slip through. This book will probably not deviate from this rule. We will maintain a list of typos on the authors web page. You can report errors to one of the following addresses:

`benoit.monin@computability.fr` or
`ludovic.patey@computability.fr`

Acknowledgements

Our very first thanks go to the National Research Agency, which largely funded the collaboration between the two authors, especially during the writing of the book, within the framework of the project "Aspects Calculatoires des Théorèmes Combinatoires – ACTC"

`https://anr.fr/Projet-ANR-19-CE48-0012`

Several results presented in this book, in particular Theorem 25-3.23 on the pigeonhole principle or the simplified proof of Liu's theorem 25-3.24 come from articles funded by the same project.

We would also like to thank our teams and related organizations, which provided us with a breeding ground for intellectual emulation, as well as moral and financial support. During the writing of this work, Monin was lecturer in the Algorithmics, Complexity and Logic Laboratory of the University of Paris-Est Créteil and Patey researcher at the CNRS, in the team Algèbre, Géométrie, Logique at the Camille Jordan Institute in Villeurbanne.

Many colleagues have generously helped us to improve the quality of this work, by giving a critical look at its scientific and educational content, drawing on their respective expertize, or by pointing out the typographical errors that inevitably crept into the book. We would therefore like to thank Paul-Elliot Anglès d'Auriac, Sébastien Tavenas, Pascal Vanier, Mathieu Hoyrup, Benjamin Hellouin, Laurent Bienvenu, Denis Kuperberg, Julien Cervelle, Damir Dzhafarov, Loïc Gassmann, William Gaudelier, Denis Hirschfeldt, Quentin Le Houerou, Alexander Shen, Keita Yokoyama, Adrien Deloro, Pascal Monin and Shahin Amini.

The scientific content of the book is above all the work of the computability-theoretic community. The authors forged their intuitions by reading the works of their predecessors and adding their contribution to this magnificent intellectual edifice. We would like to thank our colleagues at the international level for the collaborations and mutual visits which have improved our understanding of the subject.

We would also like to thank Laurent Bienvenu, who through his work and through the direction of our respective theses, has been able to transmit his passion to us, and has largely contributed to the introduction of computability theory in France.

Writing a book of this magnitude takes a lot of time and energy, and could not have happened without the moral support of our families and friends.

Chapter 1

Introduction

The scientist does not study nature because it is useful to do so. He studies it because he takes pleasure in it, and he takes pleasure in it because it is beautiful. If nature were not beautiful it would not be worth knowing, and life would not be worth living. I am not speaking, of course, of the beauty which strikes the senses, of the beauty of qualities and appearances. I am far from despising this, but it has nothing to do with science. What I mean is that more intimate beauty which comes from the harmonious order of its parts, and which a pure intelligence can grasp.

Science and Method, Henri Poincaré

What is computability theory? Computability theory is classically considered to be one of the four pillars of logic, alongside set theory, model theory and proof theory. The field was initially forged on the question of what characterizes the functions $f : \mathbb{N} \to \mathbb{N}$ whose values can be obtained by a process purely *mecanizable* or *algorithmic*, in a finite time, although arbitrarily large. We will say that such functions are *effectively computable*. Long before the appearance of the first computers, computability theory based its theoretical basis on an observation —or rather a miracle — namely, the existence of a robust definition, consensual and independent of any formalism, of the epistemological notion of effectively computable function.

The initial question, that is, "What is a computable function?" having obtained a satisfactory answer, the study naturally turned to the question of knowing, among the natural functions, which are computable and which are not. Subsequently, the field has undergone considerable development thanks to the notion of relative computability, the question no longer being to determine whether a function is computable or not, but to identify the computational power intrinsic to this function, through questions like "If this function were computable, which other functions could we compute?"

More recently, the subject of study has extended to very many mathematical objects — for instance algebraic structures, or subsets of $\mathbb{R}$ — and has given many ramifications. We will see in this book in particular that computability theory serves as a robust foundation for algorithmic randomness , and for reverse mathematics, the objects of study of which are the mathematical theorems themselves.

Nowadays, the appellation "computability theory" for a field which studies arbitrary mathematical objects, most of which are not computable, may seem surprising, even a residue of its historical subject of study. In reality, this name is still valid, but its meaning has changed: the term *computability* no longer relates to the subject of the study, but to the angle from which the subject is approached. A modern one-sentence definition of computability theory might be: **Computability theory is the study of mathematics under the prism of their computational complexity**.

1. What is a computable function?

The main difficulty of this question lies in obtaining a class of functions sufficiently robust, not to depend on the computer model, the choice of programming language, technological progress, or the advance of knowledge in such a general way.

With the advent of computers, the notion of algorithm has gradually taken root in scientific culture. Anyone who has already had a first contact with programming will have formed a good idea of what an automatable task is. Based on our knowledge of computer engineering, the following definition would come naturally: "A function is effectively computable if it has an algorithm, in other words if it can be programmed in a sufficiently expressive programming language, and executed by a sufficiently powerful computer."

This definition, if it has the advantage of being in adequacy with our intuition, does not provide a sufficiently formal framework for reasoning about the class of computable functions. A second approach would consist in fixing a computer and a standard programming language, and defining a function as computable if it is programmable in this language, and executable by this computer in finite time, using sufficient memory. If one does not worry about the speed of execution, nor the necessary memory space, it quickly appears that this definition coincides with the preceding one.

In fact, the power and memory of computers increase with technological progress, and therefore allow programs to be executed more quickly, but do not necessarily increase the class of computable functions. Even computers based on new computing paradigms, like quantum or biological computers, are simulable —with the cost of an exponential time and space overhead— by classical computers, and therefore do not change the class of computable functions. As for programming languages, the existence of operating systems and interpreters make it easy to convince oneself that the main ones such as C ++, Java or Python, allow programming —more or less elegantly — the same mathematical functions. This therefore empirically shows a certain robustness in the definition of the class of programmable functions.

A problem remains: what is the guarantee that today's computers represent the limit of what is automatable, or computable by a human being? Who tells us that with the progress of science, we will not discover a new paradigm of computation or a new way of reasoning allowing to consider as computable a larger class of functions? This is the subject of a long foundational quest started in the 20th century, and culminating in the famous Church-Turing thesis in 1936, which we will present in Chapter 6.

2. What are the non-computable functions?

With regard to our previous definition, for the moment very informal, most —if not the entirety— of the mathematical functions used on a daily basis are computable: addition, multiplication, the function $(n, m) \mapsto n^m$, the function which at n associates the n-th prime number, or even the one which computes the greatest common divisor of two natural integers, are all computable. We can add to this list less trivial examples: the function which takes a computer program written in C ++ and determines if the program is syntactically correct —this is what does a C++ compiler, among others— or the one which returns the n-th decimal of π, $\sqrt{2}$ or the golden ratio — each of these numbers is the sum of a computable sequence of rationals with a sufficiently fast convergence— or to finish the one which to n associates the number of possible games that we can play in Go on a board —also known as *goban*— of size $n \times n$. This last example illustrates in particular the following fact: one does not deal in computability theory with the time that a computation takes. Only the existence of an algorithm matters to us. In the case of the number of games in Go, the algorithm in question is based on a simple idea; it "suffices" to list all the possible games and to count them. However, there are execution time of such an algorithm is so large that it makes it impossible to use in practice for $n > 2$ ([1]). For $n = 19$, which is the size of a standard goban, this number ranges

[1] There are already 386 356 909 593 possible games on a goban of size 2×2 [234]!

from $10^{10^{48}}$ to $10^{10^{171}}$ [234], which is clearly too many games to count, even if all the world's computers harnessed it for a billion years ...

Although the function of multiplying by 2 is, in a sense, much more accessible to us than the one that counts the number of games of go, there is an algorithm that computes each of them. These two functions are therefore not different from one another from the point of view of computability theory: they are both computable, and we will mainly be interested in the functions which *are not*, that is to say functions whose values *cannot* be obtained by a purely mechanizable or algorithmic process. The mere existence of such functions is not self-evident, and one of the first tasks we will tackle will be to demonstrate their existence. This will be done in the next chapter via Cantor's diagonal argument. We will then give many examples of such functions throughout the book, the best known of which is undoubtedly the *halting problem*, defined as the function which takes a program as input, and determines whether its execution will halt, necessarily in a finite amount of time. We will see that the halting problem cannot be computed; and it is important to understand that it is indeed a question here of a theoretical and fundamental impossibility, which does not depend on the power or speed of computation of the computers. The non-computability of the halting problem is not due to an ignorance of its algorithm which could one day be discovered, but indeed to an absolute impossibility, because the existence of such an algorithm would lead to a paradox.

3. Motivation

Computability theory relates mainly to the study of non-computable functions —or more general mathematical objects—. It is legitimate to wonder if such a study is really reasonable. If even some computable functions are inaccessible to us — like the number of possible plays at Go — then why bother to think about functions *even more inaccessible*?

A first motivation for our study is exploratory. There are inaccessible objects, so let's try to explore the universe, simply because it is there, and out of curiosity about the mysteries it contains. Our efforts will be rewarded with a series of theorems of great depth. Anyone who immerses himself seriously in the developments of this book, once perhaps past some difficulties of adaptation inherent in any scientific discipline, will see a world of astounding richness come to life in his mind, with its flora and fauna, its rules and mechanisms. Computability theory is characterized by the highly dynamic nature of its proofs, each of which provides insight into the detailed workings of a fragment of the titanic machinery that animates this universe.

A second motivation arises quite simply out of necessity. The Pythagoreans found themselves constrained and forced to admit the existence of irrational measures, such as the diagonal of a side square 1, which went against their understanding of the world, which they believed to be explicable only on the basis of the relationships between integers. But if we admit the existence of integers, and the existence of the square, we are forced to admit that of *incommensurable* quantities, which we call today irrational, like $\sqrt{2}$. In the same way, we will see that if we admit the existence of computable objects, we are also forced to admit the existence of objects which are not, and which nevertheless appear naturally in a whole series of situations.

A final motivation is of a practical nature. Computability theory, through its understanding of non-computable objects, has achieved major success in providing a formal framework for the study of questions at the borders between science and philosophy. We will see two of them: the search for the definition of random objects with Part II, and the understanding of what *strength* means of a theorem, in particular with respect to another, with Part III. Part IV of this book will bring computability theory to the frontier that it shares with set theory.

4. Overview of computability theory

Computability theory can be broken down into several subdomains, which are all based on the same robust notion of effectively computable function.

4.1. Areas covered by this book

This book is broken down into four parts, each of them covering an offshoot of computability theory: classical computability theory, algorithmic randomness, reverse mathematics, and higher computability theory.

Classical computability theory

As we have mentioned, computability theory relates above all to objects which cannot be computed. Classical computability theory focuses on functions $f : \mathbb{N} \to \mathbb{N}$ as well as on sets of integers $E \subseteq \mathbb{N}$. Note that such a set can also be represented by its characteristic function $\chi_E : \mathbb{N} \to \mathbb{N}$ defined by $\chi_E(x) = 1$ if $x \in E$, and $\chi_E(x) = 0$ otherwise.

The developments of classical computability theory revolve around a fundamental tool which will allow us to compare or even measure the *degree of non-computability* of a function, also called *degree of unsolvability* or *Turing degree*, with reference to the mathematician Alan Turing who introduced the notion. Let us fix a non-computable function $g : \mathbb{N} \to \mathbb{N}$. It is natural to ask "If I were able to compute g, which other functions could

I compute?" We say that a function $f : \mathbb{N} \to \mathbb{N}$ is *computable relative to g* (or g-computable) if there exists an algorithm allowing to compute f in an extended programming language, where we would have added the function g as primitive: a special instruction allows us to call the function g on a parameter n in our program, as if it really existed, and to retrieve the result. If f is g-computable, nothing tells us how to compute g, but if we had a "oracle" allowing us to compute the values of g, it would be possible to compute the values of f.

This notion of relative computability allows us to define a partial pre-order between the functions, noting $f \leqslant_T g$ if the function f is g-computable. This is the *Turing reduction.* Different functions can carry the same computational power, in the sense that they are mutually computable. We therefore define the *Turing degree* of a function $f : \mathbb{N} \to \mathbb{N}$ as the set $\deg_T f$ of all the functions g such that $f \leqslant_T g$ and $g \leqslant_T f$. The notion of Turing degree represents a computational power, in the sense that two functions of the same Turing degree are indistinguishable from the point of view of computability. The partial pre-order on the functions induces a partial order on the Turing degrees.

Classical computability theory relates mainly to the study of Turing degrees together with the partial order relation defined above. Are there an infinite number of computational powers? Are they linearly ordered? More generally, what are the properties of this partial order? It turns out that this structure is extremely rich and complex, as we will have the opportunity to see.

Algorithmic randomness

The classical probability theory studies probabilistic phenomena, successfully modelled via the notion of measure which is used to formally define the laws of probability. On the other hand, this theory does not have the necessary tools —and it is not its goal— to speak of random objects *individually.* It is this precise point that algorithmic randomness proposes to clarify, by relying on computability theory. Let's go through an example.

Let us represent a real number $R \in [0, 1)$ by its binary expansion, of the form $R = 0.b_0b_1b_2b_3\ldots$ where $(b_n)_{n\in\mathbb{N}}$ is a sequence of bits. Suppose that the real R is obtained by drawing its bits *randomly* by a tossing sequence. We assume of course that each draw is *equiprobable*: we have a 50% chance of getting heads and a 50% chance of getting heads. Intuition tells us that the real R thus obtained is *random.* What does this mean exactly? We do not expect, for example, to obtain only "heads" on the first hundred thousand throws: if each draw is equally likely, this cannot happen, or in any case with such a low probability that we can consider it negligible. We do not expect to obtain twice as much of "heads" as of "tails" either.

Again, the probability of this happening in 100,000 draws is so low that one will assume the draws to be biased rather than witnessing such an unlikely event. We can in fact identify a first property that we are entitled to expect from a sequence of equiprobable draws: the sequence obtained should respect the law of large numbers, that is to say that the number of draws "heads" and "tails" should roughly be the same.

However, is this sufficient? Suppose now for example that on each number n, if n is a prime number, we systematically obtain equally a "heads". In the hypothesis where a certain obsession with prime numbers would lead us to notice this curious phenomenon, we will again be faced with a — slightly absurd— enigma and we will be led to think that in one way or another, something abnormal is happening. But let's take a further step back. Basically, and regardless of the sequence of bits obtained, we can identify numbers $n_1 < n_2 < n_3 < \dots$ such that the draw numbers n_1, n_2, n_3, $\dots$ are all draws "heads". In the case where our sequence n_1, n_2, n_3, $\dots$ contains prime numbers, this seems to us to be a "probabilistic bug", but why should it be acceptable if n_1, n_2, n_3, $\dots$ are any integers? This is where computability theory comes into play, and will allow us to precisely formalize the properties that a sequence of random bits should have — according to our human intuition —.

Reverse mathematics

The notion of *theorem* relates to a system of axioms. When one omits to mention the system of reference, it is commonly accepted that one refers to the system of Zermelo Fraenkel (ZF), which represents a set of consensual axioms which serve as the foundation of all mathematics. The ZF system is, however, very powerful, and we have no guarantee that it will not be inconsistent.

Reverse mathematics aims to find the axioms necessary and sufficient to prove the theorems of everyday mathematics. It is therefore a question of studying existing theorems, to find more elementary proofs, or on the contrary to show the optimality of their proof. Better understanding the hypotheses of theorems allows to better control their "fragility" in the face of a potential contradiction of the proof system. It is therefore a meta-mathematical approach aimed at answering the question "Which confidence can we have in our mathematics?"

At first glance, this approach is not linked to computability theory. However, reverse mathematics relies to a basic theory, RCA_0, capturing the *computable mathematics*, and which represents a basis of confidence more in connection with the concrete world, because its objects being computable, they can be represented by an algorithm, therefore they have a finite description. Reverse mathematics therefore consists, given a theorem T, in searching for axioms A such that RCA_0 proves the equivalence between A and T. By the choice of the basic theory RCA_0, equivalences are computational processes calling on the tools of computability theory.

Higher computability theory

One of the reasons for the success of computability theory as a tool for analyzing mathematics lies in the existence of a strong intuition of the notion of computation, thus making it possible to guide the manipulation of concepts and to prove theorems without being embarrassed by a heavy formalism. higher computability theory aims to extend the reach of these tools to more powerful computational models, which can be seen as machines having the possibility of continuing their execution for an infinite computation time (formally in ordinal computation time). Just as notions of classical computability theory can be captured by logical formulas, so can higher computability theory. For example, where the sets of integers which can be enumerated (out of order) by a computer program are those which can be described by a Σ^0_1 formula of arithmetic, those which are enumerable by a hypercomputer programs are those which can be defined by a Π^1_1 formula of arithmetic.

We will see that this aspect of things brings higher computability theory closer to descriptive set theory, a branch of set theory which classifies sets according to the degree of difficulty in describing them. Higher computability theory can be seen as a bridge between descriptive set theory and classical computability theory.

4.2. Other branches of computability

In order to allow this work to keep a reasonable size, we have chosen to ignore two important branches of computability theory, namely computable structure theory and degrees of enumeration.

Computable structure theory

It is a branch of computability theory that studies the extent to which the algebraic properties of a mathematical structure affect their descriptive complexity. By structure, we mean sets equipped with operations, such as groups, rings and fields, but also any structure within the meaning of model theory. This branch borrows its techniques from both model theory and classical computability theory to answer this question.

Concretely, this theory studies countable structures and asks questions of the form “Given a computable structure $\mathcal{A}$, what are the possible Turing degrees of structures isomorphic to $\mathcal{A}$?” or “Given two isomorphic computable structures, what is the computational complexity of their isomorphism?” For example, some structures like the dense linear orders without endpoints are computably isomorphic to all their computable copies. They are called *computably categorical*.

Degrees of enumeration

Classical computability theory places “computable sets” as the reference computational power. But some problems are expressed naturally in the form of non-computable sets, the elements of which can however be enumerated out of order by a computable process. We call these sets *computably enumerable* (c.e.). In particular, if E is a set of integers c.e. and if $n \in E$, it is possible to realize it in finite time, by launching the enumeration procedure and waiting for n to appear. On the other hand, if $n \notin E$, then it will not be possible in general to know it in a finite time. For example, the set of Diophantine equations (equations with integer coefficients, of type $3x^3 - 2y^2 + x - 2 = 0$) which admit integer solutions is computably enumerable, because it suffices to search exhaustively for solutions, and to enumerate the equation if a such a solution exists.

Degrees of Enumeration place “computable enumeration” as the reference power. We can define an *enumeration reducibility* $A \leqslant_e B$ iff any enumeration of the elements of B computes an enumeration of the elements of A. This reduction is a partial pre-order, which induces a notion of *enumeration degree*: the degree of enumeration of A is the set $\deg_e(A)$ of all sets B such that $A \leqslant_e B$ and $B \leqslant_e A$. The study of the degrees of enumeration equipped with the partial order $\leqslant_e$ constitutes an active branch of research in computability theory.

Chapter 2

Cantor's infinity

If we had to give a date to the birth of modern logic, we would place it without hesitation in 1872, the date on which Georg Cantor exposes his first proof of Theorem 4.1 to come of the uncountability of real numbers. Cantor isolates later the quintessence of this first proof through is famous *diagonal argument*, which will have a central place in computability theory.

Georg Cantor, 1845–1918

Cantor's work then marked the beginning of a "complex" set theory which will play a big role in the fundational quest of mathematics in the early 20th century, which we will talk about in detail in Chapter 9. This crisis will lead to the development of mathematical logic as we know it today, with modern set theory, known as ZFC, but also with the development of the first theories of calculus, used by Gödel to show his famous incompleteness theorem, which can be seen as a sophisticated variation of Cantor's diagonal argument.

What is it exactly? Cantor shows that infinite sets do not all have the same "size". There are strictly more real numbers than integers, in a sense that we will define precisely in a few lines. Cantor will use this discovery to develop a mathematical study of infinity. In particular, he will create the transfinite numbers, which will constitute the backbone of mathematical definitions, and which we will discuss in Chapter 27.

Cantor, however, was not the first to notice that infinity does not obey the same rules as the finite. In particular, a surprising characteristic of infinite sets is that the whole is not necessarily greater than its parts. Galileo makes a luminous exhibition of it in his work "Dialogues Concerning Two New Sciences" [74] through a tasty dialogue between two characters, Salviati and Simplicio:

- Salviati. *This is one of the difficulties which arise when we attempt, with our finite minds, to discuss the infinite, assigning to it those properties which we give to the finite and limited.; but this I think is wrong, for we cannot speak of infinite quantities as being the one greater or less than or equal to another.[...] I take it for granted that you know which of the numbers are squares and which are not.*
- Simplicio. *I am quite aware that a squared number is one which results from the multiplication of another number by itself; thus 4, 9, etc., are squared numbers which come from multiplying 2, 3, etc., by themselves.*
- Salviati. *Very well; and you also know that just as the products are called squares so the factors are called sides or roots; while on the other hand those numbers which do not consist of two equal factors are not squares. Therefore if I assert that all numbers, including both squares and non-squares, are more than the squares alone, I shall speak the truth, shall I not?*
- Simplicio. *Most certainly.*
- Salviati. *If I should ask further how many squares there are one might reply truly that there are as many as the corresponding number of roots, since every square has its own root and every root its own square, while no square has more than one root and no root more than one square.*
- Simplicio. *Precisely so.*
- Salviati. *But if I inquire how many roots there are, it cannot be denied that there are as many as there are numbers because every number is a root of some square. This being granted we must say that there are as many squares as there are numbers because they are just as numerous as their roots, and all the numbers are roots.*

On reading Galileo's dialogue, we observe the paradox trap closing in on Simplicio. Galileo uses for that a concept which will be taken up by Cantor: two sets A and B "have as many elements" if one can make exactly correspond the elements of A and the elements of B, in other words if there is a bijection between the two sets.

1. Equipotence and subpotence

Recall that a function $f : E \to F$ is *injective* if

$$\forall x, y \quad x \neq y \Rightarrow f(x) \neq f(y),$$

surjective if its image is the whole set F, and *bijective* if it is both injective and surjective.

Definition 1.1. Two sets E and F are *equipotent* if there is a bijection between them. We will then write $|E| = |F|$. ♢

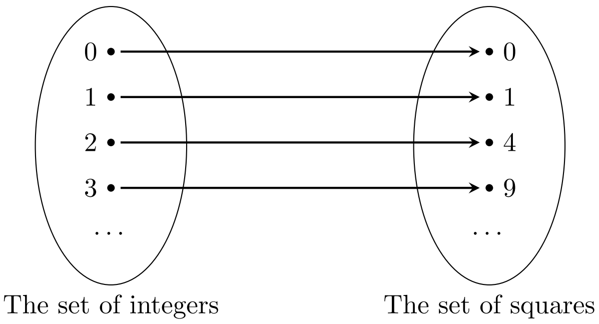


Figure 1.2: Galileo's argument to says there are "as many" integers as square numbers

According to our definition, the integers and the squares of integers therefore have the same cardinality: there are as many elements in the two sets. What seems paradoxical is that the squares of integers form a strict part of the set of integers. The paradox is solved in the simplest possible way: the intuition that we have on finite sets, which wants a strict subset of a set to contain fewer elements is simply no longer true for infinite sets.

Remark

The notation $|E| = |F|$ seems to suggest equality between two objects $|E|$ and $|F|$, which we call respectively *cardinality* of E and *cardinality* of F. It is possible to give a precise definition of $|E|$, as a mathematical object. For the moment, we will be satisfied with defining cardinality as the informal notion of the size of a set, and if the object $|E|$ is not clearly defined, the statement $|E| = |F|$ is, and is sufficient to our treatment of infinity.

Note that Galileo uses his idea to explain precisely why there is no sense in comparing the size of infinite sets. This is where all the genius of Cantor intervenes, who will discover that contrary to Galileo's intuition, it is possible to give a formal notion of size to infinite sets: there are infinites "larger" than others.

Intuitively, a set F is at least as big as a set E if we can match the elements of E to distinct elements of F, in other words if we can associate each element of E with its own "representative" in F.

Definition 1.3. A set E is *subpotent* to a set F if there is an injection of E into F. We then write $|E| \leqslant |F|$. If E is not subpotent to F, we write $|E| \nleqslant |F|$. ♢

It is easy to verify that this relation is *transitive*, ie, if $|E| \leqslant |F|$ and $|F| \leqslant |G|$, then $|E| \leqslant |G|$. Indeed, if the functions $f : E \to F$ and $g : F \to G$ are two injections, then their composition $g \circ f : E \to G$ is an injection witnessing the relation $|E| \leqslant |G|$. It is however much less clear that if $|E| \leqslant |F|$ and $|F| \leqslant |E|$, then $|E| = |F|$. Unrolling the definitions, the question comes back to knowing if, when there is an injection of E in F and another of F in E, there is a bijection between E and F. Let us see straight away that this is indeed the case: it is the Cantor–Bernstein theorem.

2. Cantor–Bernstein's theorem

The heart of the proof of the Cantor–Bernstein theorem lies in the following lemma.

Lemma 2.1. If $B \subseteq A$ are sets, and $f : A \to B$ is an injective function, then there is a bijection $h : A \to B$. ⋆

Proof. The reader can use Figure 2.2.

Let $C_0, C_1, C_2, \ldots$ be the sequence defined by induction as follows:

$$C_0 = A \setminus B \quad \text{and} \quad C_{n+1} = f(C_n).$$

Let $C = \bigcup_n C_n$. A simple reasoning by induction on n allows to show, using the injectivity of f, that the sets C_n for $n \in \mathbb{N}$ are pairwise disjoint, although this is not necessary in the proof. Let the function $h : A \to B$ be defined by:

$$h(x) = \begin{cases} f(x) & \text{if } x \in C \\ x & \text{if } x \notin C. \end{cases}$$

Let us show that h is injective. The function f being injective, the function h restricted to C is injective. The function h restricted to $A \setminus C$ is the identity function, and is therefore also injective. Finally, the image

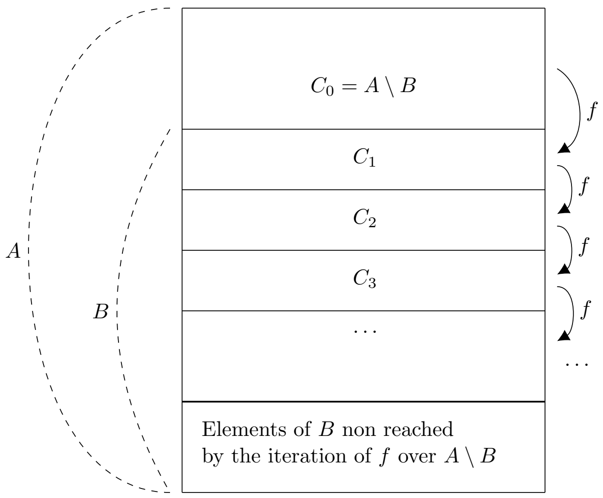

$C_0 = A \setminus B$
C_1
C_2
C_3
$\dots$
Elements of B non reached by the iteration of f over $A \setminus B$
A
B
f
f
f
f
$\dots$

Figure 2.2: Illustration of the proof of Lemma 2.1

of $C = \bigcup_n C_n$ by h is

$$\bigcup_n f(C_n) = \bigcup_n C_{n+1} \subseteq C,$$

and the image of $A \setminus C$ by h is $A \setminus C$. The function h is the union of two injective functions having disjoint images, and is therefore injective.

Finally, let us show that h is surjective. Let $y \in B$. If $y \notin C$, then $h(y) = y$ and therefore y has a predecessor by h. If $y \in C$, as $y \notin C_0$, there exists an $n \in \mathbb{N}$ such that $y \in C_{n+1}$. By definition of $C_{n+1} = f(C_n)$, there exists an $x \in C_n$ such that $h(x) = f(x) = y$. ■

We can now show the announced theorem.

Theorem 2.3 (Cantor-Bernstein)
If A and B are sets, and $f : A \to B$ and $g : B \to A$ are injective functions, then there is a bijection between A and B.

Proof. Let $A' = g(B)$. The application $g \circ f$ is an injection of A into A'. By Lemma 2.1, there is therefore a bijection $h : A \to A'$. The function g being a bijection between B and A', the function $g^{-1} \circ h : A \to B$ is a bijection. ■

Corollary 2.4
Two mutually subpotent sets are equipotent.

The question of knowing if there are infinites of distinct sizes, and in particular with one of them strictly larger than the other, therefore comes down to knowing if we can find two sets A, B for which $|A| \leqslant |B|$ but $|B| \nleqslant |A|$. We will see that this is indeed the case.

3. Countable sets

The infinite set par excellence is of course $\mathbb{N}$, the set of integers. In this case, a specific vocabulary is used.

Definition 3.1. A set A is said to be *countably infinite* if there is a bijection $f : \mathbb{N} \to A$, in other words if A and $\mathbb{N}$ are equipotent. A set A is *countable* if it is finite or countably infinite. Otherwise, A is said to be *uncountable*. ♢

Remark
Some authors use the term "denumerable" to denote countably infinite sets.

Intuitively, the infinity of natural numbers is the smallest infinity, in the sense that it is subpotent to any infinite set.

Proposition 3.2. The set $\mathbb{N}$ is subpotent to any infinite set. ⋆

Proof. Let A be an infinite set. We are going to define an injective function $f : \mathbb{N} \to A$ by induction on $\mathbb{N}$. Let $f(0)$ be any element of A. Suppose the values $f(0), \ldots, f(n)$ are defined. In particular, $B = \{f(0), \ldots, f(n)\} \subseteq A$ is a finite set while A is infinite. There is therefore necessarily an element in $A \setminus B$. Let $f(n+1)$ be this element. By construction, f is injective. ■

Note that in its full generality, the previous proof uses the *axiom of choice*, which we will talk about again in Section 9-4, and which is necessary in order to choose at each step an element of $A \setminus B$. In most cases, however (as in the following corollary), this axiom is not absolutely necessary.

Corollary 3.3
Any subset of a countable set is countable.

Proof. Let A be a countable set, and let $B \subseteq A$ be a subset. If B is finite, then it is countable. Suppose B is infinite. In particular, A is also infinite, so A is equipotent to $\mathbb{N}$. The set A being equipotent to $\mathbb{N}$, it is therefore subpotent to $\mathbb{N}$. As $B \subseteq A$, it is subpotent to A, therefore to $\mathbb{N}$. Furthermore, by Proposition 3.2, $\mathbb{N}$ is subpotent to B. By the Cantor-Bernstein theorem (Theorem 2.3), B and $\mathbb{N}$ are equipotent. ■

Exercise 3.4. (⋆) Let A be a countably infinite set, and let $f : \mathbb{N} \to A$ be a bijection. Finally, let $B \subseteq A$ be an infinite subset. Directly construct a bijection from $\mathbb{N}$ to B which is not based on the axiom of choice. By this, we mean that the definition of the function must be based on an explicit algorithm, and not on an abstract procedure making it possible to choose an element in a non-empty set, without knowing which element it is. ◇

Let us now introduce a bijection that we will use regularly in this book, which is the one commonly used to witness the countability of the product $\mathbb{N} \times \mathbb{N}$.

Proposition 3.5. The set $\mathbb{N} \times \mathbb{N}$ is countably infinite. ⋆

Proof. Let $\alpha : \mathbb{N} \to \mathbb{N}^2$ be the function such that the values

$$\alpha(0) = (0,0), \quad \alpha(1) = (1,0), \quad \alpha(2) = (0,1), \ldots$$

enumerate the pairs of integers by successive diagonals, as in Figure 3.6.

The function is injective by construction, and any pair will appear at one stage of the enumeration. Thus, α is a bijection from $\mathbb{N}$ to $\mathbb{N} \times \mathbb{N}$ witnessing the equipotence between the two sets. ■

It is possible to give an analytical definition of the reciprocal function of α defined in the previous proposition. It will therefore be a bijection from $\mathbb{N}^2$ to $\mathbb{N}$, which we will call α_2 and which the reader will be able to discover through the exercise below.

Exercise 3.7. (⋆) Let $\alpha_2 : \mathbb{N}^2 \to \mathbb{N}$ defined by:

$$\begin{aligned} \alpha_2(x,y) &= y + \textstyle\sum_{i=0}^{x+y} i \\ &= y + \frac{(x+y+1)(x+y)}{2}. \end{aligned}$$

1. Show that α_2 is bijective.
2. Show that $\alpha_2(a,b) \geqslant a$ and $\alpha_2(a,b) \geqslant b$. ◇

The bijection α_2 of the previous exercise will be very often used in the developments of this book, via the following notation.

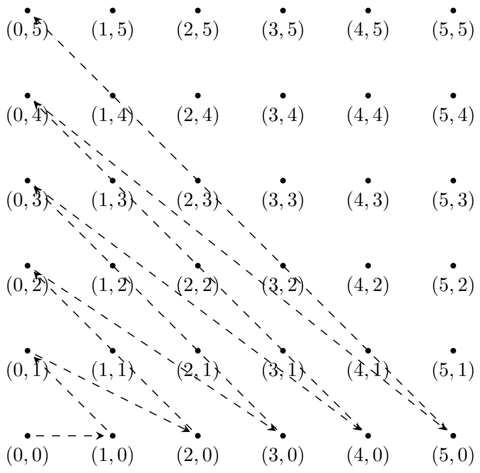


Figure 3.6: Illustration of the proof of Proposition 3.5

Notation

We denote by $\langle n, m\rangle$ the integer to which the pair (n, m) is sent via the bijection α_2.

Note that if we have a bijection $\alpha_2 : \mathbb{N}^2 \to \mathbb{N}$, we can define a bijection $\alpha_3 : \mathbb{N}^3 \to \mathbb{N}$ by simply taking $\alpha_3(x, y, z) = \alpha_2\big(x, \alpha_2(y, z)\big)$. We can continue in this way to define bijections from $\mathbb{N}^n$ to $\mathbb{N}$ for any $n \in \mathbb{N}^*$, which leads to the following notation.

Notation

We denote by $\langle x_1, \dots, x_k\rangle$ the integer on which the k-tuple $(x_1, \dots, x_k)$ is sent via the bijection from $\mathbb{N}^k$ to $\mathbb{N}$ described above.

Let us take advantage of our freshly introduced bijections to deduce the following corollary.

Corollary 3.8
The Cartesian product of two countably infinite sets is countably infinite.

Proof. Let A and B be countable sets and let $f : A \to \mathbb{N}$ and $g : B \to \mathbb{N}$ be bijections witnessing it. The function $h : A\times B \to \mathbb{N}$ defined by $h(x, y) = \langle f(x), g(y)\rangle$ is a bijection. ■

Let us end this section with three exercises, allowing to manipulate the concepts seen so far. In particular, we draw attention to the first of them, which will be used regularly in future developments.

Exercise 3.9. ($\star$) Show that $\mathbb{Z}$ is a countably infinite set. Deduce that $\mathbb{Z} \times \mathbb{Z}$ is a countably infinite set and finally that the set $\mathbb{Q}$ of rational numbers — which can be written under the form p/q with $p, q \in \mathbb{Z}$ with $q \neq 0$ — is also countably infinite. $\diamond$

Exercise 3.10. ($\star$) Let $f : \mathbb{N} \to A$ be a surjective function towards an infinite set A. Show that A is countably infinite. $\diamond$

Exercise 3.11. ($\star$) Let $(B_n)_{n\in\mathbb{N}}$ be a sequence of countable sets with their respective bijections f_n with $\mathbb{N}$. Show that $B = \bigcup_n B_n$ is a countable set.

Warning: if we do not have the bijections $(f_n)_{n\in\mathbb{N}}$, we can always show that B is countable, but we must use the *axiom of choice*, in order to choose uniformly for each B_n one of its bijections with $\mathbb{N}$. We will talk about it again in Section 9-4. $\diamond$

4. Cantor's diagonal argument

Let us now tackle the theorem announced at the beginning of this chapter: there exist infinities larger than others, and in particular an infinity larger than that of integers. The following theorem uses Cantor's famous diagonal argument, which will be taken up on numerous occasions and in various forms throughout this book.

Theorem 4.1 (Cantor)
The set of real numbers is uncountable.

Proof. The reader can use Figure 4.2, which illustrates the argument of the proof. We reason through the absurd. Let us suppose on the contrary that there exists a bijection $f : \mathbb{N} \to \mathbb{R}$. We are going to construct a real number $R \in \mathbb{R}$ which is not in the image of f. We simply define R as follows: the integer part of R is 0, and for any n, if the n-th decimal of $f(n)$ is different from 0, then the n-th decimal place of R is equal to 0. Conversely, if the n-th decimal of $f(n)$ is equal to 0, then the n-th decimal of R is equal to 1.

It is clear that for any integer n, our real number R cannot be equal to $f(n)$, because the n-th decimal of R is different from the n-th decimal place of $f(n)$. $\blacksquare$

R	$=$	$0,$	$9-x_{00}$	$9-x_{11}$	$9-x_{22}$	$9-x_{33}$	$9-x_{44}$	$9-x_{55}$	$9-x_{66}$
$f(0)$	$=$	$N_0,$	$\boldsymbol{x_{00}}$	x_{01}	x_{02}	x_{03}	x_{04}	x_{05}	x_{06}
$f(1)$	$=$	$N_1,$	x_{10}	$\boldsymbol{x_{11}}$	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}
$f(2)$	$=$	$N_2,$	x_{20}	x_{21}	$\boldsymbol{x_{22}}$	x_{23}	x_{24}	x_{25}	x_{16}
$f(3)$	$=$	$N_3,$	x_{30}	x_{31}	x_{32}	$\boldsymbol{x_{33}}$	x_{34}	x_{35}	x_{36}
$f(4)$	$=$	$N_4,$	x_{40}	x_{41}	x_{42}	x_{43}	$\boldsymbol{x_{44}}$	x_{45}	x_{46}
$f(5)$	$=$	$N_5,$	x_{50}	x_{51}	x_{52}	x_{53}	x_{54}	$\boldsymbol{x_{55}}$	x_{56}
$f(6)$	$=$	$N_6,$	x_{60}	x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	$\boldsymbol{x_{66}}$
...									

Figure 4.2: Illustration of Cantor's diagonal argument. We construct our real R using the diagonal of the table, the decimal $9 - x_{ii}$ being different from x_{ii}. Note that the real R described here is not exactly the one described by the proof, but the result is the same: the element R does not belong to the image of f.

What does the previous theorem tell us? That there are "more" real numbers than integers. These sets of numbers are both infinite, but the infinity of reals is "larger" than that of integers. Indeed, when we try to match an integer to each real number, we see that there are always "exceeding" reals, which do not correspond to any integer. We therefore have $|\mathbb{R}| \nleqslant |\mathbb{N}|$. Note that we have on the other hand $|\mathbb{N}| \leqslant |\mathbb{R}|$, via identity injection. In this case, we can write $|\mathbb{N}| < |\mathbb{R}|$, meaning that there is an injection of $\mathbb{N}$ in $\mathbb{R}$, but no injection of $\mathbb{R}$ in $\mathbb{N}$.

Is there an infinity greater than that of the reals? Cantor also answered this question, with a similar argument: for any set A, there exists a set B such that $|A| < |B|$.

Theorem 4.3 (Cantor)
For any set A, the set $\mathcal{P}(A)$ of the subsets of A is such that $|A| < |\mathcal{P}(A)|$.

Proof. The set A is subpotent to $\mathcal{P}(A)$, considering the injection which to any element $x \in A$ associates the singleton $\{x\}$. Thus, $|A| \leqslant |\mathcal{P}(A)|$. Let us now assume absurdly that $|\mathcal{P}(A)| = |A|$, i.e., suppose there is a bijection $f : A \to \mathcal{P}(A)$. Consider the set

$$B = \{x \in A : x \notin f(x)\},$$

and let y be such that $f(y) = B$. Then, $y \in B$ iff $y \notin f(y)$, in other words if $y \notin B$, which is a contradiction. ∎

One can legitimately wonder if the cardinality of the sets is always comparable: given two sets A, B, is there always an injection of one into the other? The answer to this question again depends on the *axiom of choice*. We will talk about it again in Section 9-4 as well as in Section 27-4.1.

5. Non-computable reals

Cantor's theorem will provide us our first argument for the existence of non-computable real numbers. We use for the following proposition and corollary the — for the moment informal — notions of "computer program" and "computable real". They will both be precisely defined later in this book; consider for the moment that a real is computable if there is a computer program which enumerates in order the infinite list of its decimals.

Proposition 5.1. The set of computer programs is countable. ⋆

Proof. A computer program (written in any programming language whatsoever) takes the form of a finite sequence of characters, where each character belongs to a finite alphabet (say the set of characters in the ASCII table). Let us show that there exists a bijection from $\mathbb{N}$ to the set of finite sequences of ASCII characters. For that, we define an infinite list of all the finite sequences of ASCII characters: we list first all the sequences comprising exactly one ASCII character, then all the sequences comprising two ASCII characters, then all those comprising three, etc. It is clear that any finite sequence of ASCII characters appears somewhere in our infinite list.

It is now sufficient to remove from our list the fnite sequences which do not correspond to a valid program. We then define the element $f(n)$ as being the n-th element of the list resulting from this operation. It is clear that f is a bijection. In particular, the set of computer programs can be counted.■

Corollary 5.2
There are real numbers that cannot be computed.

Proof. Let P be the set of computer programs. Suppose that every real is computed by an element of P. Let p_1 be the first computer program computing a real number. Here "first" means first according to the order obtained via the bijection between the computer programs and $\mathbb{N}$. Suppose that $p_1, \ldots, p_n$ are defined and all compute a different real. Let p_{n+1} be the first computer program computing a real number different from those computed by $p_1, \ldots, p_n$. We define by induction in this way the sequence $(p_n)_{n \in \mathbb{N}}$.

We then obtain a bijection between $\mathbb{R}$ and an infinite subset of P. As P is countable, this infinite subset is also countable, which gives a bijection between $\mathbb{N}$ and $\mathbb{R}$, and hence a contradiction. ∎

The proof of the corollary 5.2 is non-constructive: we show the existence of non-computable numbers without giving a precise example. This will obviously be done over and over again in the rest of this work, through a detailed study of different types of non-computable numbers. Before we get down to it, let's mention two or three important notions concerning Cantor's study of infinity following its discovery.

6. Cantor space

Cantor has shown that there is no greatest infinity. Among all the possible infinities, the smallest is "countably infinite". Another infinity is of great interest, namely, that of real numbers:

Definition 6.1. A set A is said to have *the power of the continuum* if $|A| = |\mathbb{R}|$. ◊

The power of the continuum therefore characterizes the infinity of reals. Cantor conjectured that there was no infinity strictly between $|\mathbb{N}|$ and $|\mathbb{R}|$, but without succeeding in proving it. His conjecture known as the *continuum hypothesis* will be considered for nearly a century as one of the most important mathematical questions. Gödel will show in 1938 [77] that it is not possible to demonstrate that the continuum hypothesis is false, and Cohen will put an end to the question in 1963 [40] by showing that it is not possible nor to show that the continuum hypothesis is true: it is a question independent of the rest of mathematics, which can be considered true or false without introducing any contradiction. We will discuss this in more detail in Section 9-4.

Among the sets having the power of the continuum, we will pay particular attention to the set of infinite sequences of 0 and 1, which will constitute the main part of our playing field throughout this work. By "infinite sequences of 0 and 1", we mean sequences indexed by integers, that is to say sequences of the form $x_0x_1x_2x_3\ldots$ where each $x_i \in \{0, 1\}$.

Definition 6.2. The *Cantor space* is the set of infinite sequences of 0 and 1. We denote it $2^{\mathbb{N}}$, and its elements will be denoted by uppercase letters, in general A, B, C, X, Y, Z. $\diamondsuit$

Cantor space has the power of the continuum.

Proposition 6.3. We have: $|2^{\mathbb{N}}| = |[0,1]| = |\mathbb{R}|$. $\star$

Proof. Let us first show $|[0,1]| = |\mathbb{R}|$. The identity function is an injection of $[0,1]$ into $\mathbb{R}$. We can easily verify that the function

$$f(x) = \begin{cases} 1/2 + 1/(x+2) & \text{if } x \geqslant 0 \\ 1/2 - 1/(|x|+2) & \text{if } x < 0 \end{cases}$$

is an injection of $\mathbb{R}$ into $[0,1]$. By the Cantor-Bernstein theorem (see Theorem 2.3), we therefore have $|[0,1]| = |\mathbb{R}|$.

Let us now show $|2^{\mathbb{N}}| = |[0,1]|$. To define an injection of $2^{\mathbb{N}}$ into $[0,1]$, one should be careful: some real numbers have two possible binary expansions, thus $1.00000\cdots = 0.11111\ldots$, where $0.11111\ldots$ is the decimal number $0.99999\ldots$ written in binary. It is the same for any real number whose binary expansion ends with an infinity of consecutive 0: this one then has an equivalent binary expansion which ends with an infinity of consecutive 1. We will therefore define the injection $f : 2^{\mathbb{N}} \to \mathbb{R}$ which to X associates the real number of $[0,1]$ whose *ternary* expansion, that is to say in base 3, consists of the bits of X. The use of the ternary representation makes it possible to circumvent these problems of equality and thus to make the function f injective.

The injection $g : [0,1] \to 2^{\mathbb{N}}$ is defined by associating to any real $r \in [0,1]$ its binary expansion R. When there are several representations of the same real number, we will choose by convention the one ending with an infinity of 0. By the Cantor-Bernstein theorem, we therefore have $|2^{\mathbb{N}}| = |[0,1]|$. ∎

Notation

Given $X \in 2^{\mathbb{N}}$ and $n \in \mathbb{N}$, we denote by $X(n)$ the n-th element of the sequence X which we will also call the n-th *bit* of X. Thus, if the sequence X is written $x_0x_1x_2\ldots$, then $X(0) = x_0$, $X(1) = x_1$, $X(2) = x_2$, $\ldots$

The fact that some real numbers have two possible binary representations make $|2^{\mathbb{N}}|$ and $|[0,1]|$ topologically different spaces. For example, for any $x, y \in \mathbb{R}$ with $x < y$, there is a real z strictly between the two. On the other hand, if we provide $2^{\mathbb{N}}$ with the lexicographic order $<_{lex}$ defined by

$$X <_{lex} Y \ \text{ if } \ X(n) < Y(n),$$

where n is the smallest integer such that $X(n) \neq Y(n)$, then the infinite sequences

$$X = 0111111\ldots \quad \text{and} \quad Y = 100000\ldots$$

have no element strictly between them for this order.

This difference is anecdotal from the point of view of the computational complexity of the elements, and we will sometimes speak of real or of infinite binary sequences indistinctly. The difference is however important for the development of computability theory in topological spaces other than $2^{\mathbb{N}}$. We will discuss this briefly in Section 22-4.2.

Let us see now that there exists, on the other hand, a very natural bijection between the set $\mathcal{P}(\mathbb{N})$ —the power set of $\mathbb{N}$— and Cantor space. Thus, $\mathcal{P}(\mathbb{N})$ has the power of continuum.

Proposition 6.4. We have: $|2^{\mathbb{N}}| = |\mathcal{P}(\mathbb{N})|$. ⋆

Proof. Let $f : 2^{\mathbb{N}} \to \mathcal{P}(\mathbb{N})$ be the function which to $X \in 2^{\mathbb{N}}$ associates the set $Y = \{n \in \mathbb{N} : X(n) = 1\}$. The function f is clearly a bijection. ∎

The bijection between the set $\mathcal{P}(\mathbb{N})$ of the parts of $\mathbb{N}$ and Cantor space $2^{\mathbb{N}}$ is so elementary that we can consider $\mathcal{P}(\mathbb{N})$ and $2^{\mathbb{N}}$ as two representations of the same mathematical concept. In the following, we will speak without distinction of a subset of $\mathbb{N}$ or of an infinite sequence of 0 and of 1, and we will use the same notation $2^{\mathbb{N}}$ to denote the set of these elements. Thus, given $X \in 2^{\mathbb{N}}$ seen as a sequence, we will have $X(n) = 1$ iff n belongs to X seen as a set.

Digression

The name "Cantor space" undoubtedly comes from the eponymous concept *Cantor's triadic set*. We define $A_0 = [0, 1]$, then A_1 is A_0 minus its central third:

$$A_1 = [0, 1/3] \cup [2/3, 1].$$

Then A_2 is A_1 minus the middle third of each of its intervals:

$$A_2 = [0, 1/9] \cup [2/9, 1/3] \cup [2/3, 7/9] \cup [8/9, 1].$$

In general, to go from A_n to A_{n+1}, we remove the middle third of each of the intervals of A_n. The triadic set of Cantor is ultimately the result of the application of this operation, that is to say the set $\bigcap_{n \in \mathbb{N}} A_n$.

Cantor space $2^{\mathbb{N}}$ defined above, is topologically equivalent to the triadic

Cantor set. In particular, each point of $\bigcap_{n\in\mathbb{N}} A_n$ can be described as a sequence of 0 and 1 as follows: the n-th bit of a point corresponds to determining whether it is to the right or to the left of the n-th third which is subtracted from the current interval. We can also see Cantor space as the set of reals of $[0, 1]$ whose ternary representation avoids the number 1.

Part I

Classical Computability Theory

Chapter 3

Foundations of computability

Our goal is to conduct a mathematical study of computability. To do this, it is customary to define mathematically what is meant by *computable fonction*. This quest for a formal definition capturing this epistemological concept was the genesis of computability theory, and resulted in what is nowadays called the *Church-Turing thesis*. This thesis states that any computable process can be executed with a *Turing machine*, a computational model imagined by Alan Turing in 1936 and which can be considered as a precursor of modern computers. Other approaches than that of Turing machines allow us to capture the notion of computable function, among which we shall cite the general recursive functions and λ-calculus. These different models will be presented in more detail in the interlude on the Church-Turing thesis (see Chapter 6), and we will adopt for this chapter a less formal approach.

1. Computable functions

In computability theory, the formalism of Turing machines tends to serve as a reference model, not only for historical reasons — Turing was the first to convince the community that his model captured all the computable processes — but also because this model highlights the notion of atomic stage of computation, which opens the door to complexity theory. It is customary to begin the study of computability theory by that of its computational models, and to prove their equivalence, in order to be convinced of the robustness of the notion of computable function and of the validity of the definitions. This is a fairly long and tedious development. So it is easy to be put off by this step at the end of which we believe too easily — wrongly — that computability theory comes down to complex and boring coding techniques.

We therefore made the choice to break with tradition, and defer the definition and the mathematical study of computational models to Chapter 6, in order to facilitate the first contact with computability theory, and to access its fundamental concepts more directly. The advent of computers and the democratization of programming education have firmly anchored the notion of algorithms in scientific culture. We will therefore adopt the following informal definition.

> **Definition 1.1.** A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if it can be defined by an algorithm, or in other words programmed in a modern programming language. ◊

It follows from our educational choice that the proofs of our first theorems will largely appeal to the intuition of the properties expected of a computable function. These are however theorems in their own right, in the sense that it is possible to prove them from the formal definitions of Chapter 6.

Integers in Computer Science

We are mainly interested in the functions from $\mathbb{N}$ to $\mathbb{N}$. In most standard programming languages, integers are bounded. For our theoretical definition, it is important to take into account *all the integers*.

Algorithms. In order to agree on what is meant by a modern programming language or algorithm, let us list its main aspects, which will be formally taken up in Section 6-3 for our definition of structured programs on the model of register machines.

(1) The language must be able to handle integers, using constants, integer variables, and the usual arithmetic operations, namely addition, subtraction, multiplication, and integer division. The reader will be able to add other types to it, such as strings or floating point numbers, without this changing the computational power.

(2) The language must be able to handle Boolean expressions, and perform comparison operations on integers.

(3) The language must contain the usual control structures, namely conditional instructions of type "`if` ... `then` ... `else` ..." and loops "`for` " and "`while`", which repeat as long as a certain condition is true.

(4) We will assume that the machine's memory is unbounded, and that it can be use as much as necessary (especially to read its input, which can be an arbitrarily large integer).

Data types

Point (1) emphasizes that adding data types other than the integer type is unnecessary. This is true on the condition of course of considering that our integers can be arbitrarily large (for example to encode large strings of characters), which will always be the case by convention. The reader will be able to find an example of encoding of arrays by integers in Proposition 6-3.26.

Unbounded memory

One might be surprised by point (4) above, which allows unbounded memory. Let us insist on the fact that one does not allow oneself an infinite memory for all that: a computation which ends only carried out a finite number of operations and therefore could only use a finite amount of memory. Simply, we do not deal in computability theory with the spatial complexity of the algorithms.

Examples. Before starting the formal study of computable functions and their properties, let's list some examples of computable functions, in order to begin to form an intuition.

(1) The usual arithmetic operations can be computed. In particular, addition, multiplication, subtraction and integer division are computable.

(2) The function which associates the n-th prime number with n is computable, as is the decomposition of a number into its prime factors.

(3) The function which, taking as parameter a list of city positions, returns one of the shortest paths passing through all these cities only once, is computable[1].

Conversely, there are, as we will see, many non-computable functions. However, while it suffices to give an algorithm to show that a function is computable, showing that a function is not computable often requires more elaborate reasoning, because it is not enough that the function does not have any known algorithm; it must be shown that it is theoretically impossible to program it. Each of the following statements is therefore a theorem in its own right.

[1] This is the famous *travelling salesman problem*.

(1) The function which takes as input a computer program (coded by an integer or a string), and decides if its execution will one day halt, is not computable (see Theorem 7.8).

(2) The function which takes as input a formula in the language of arithmetics (for example, "$\forall x\ \forall y\ \forall z\ x^3 + y^3 \neq z^3$"), and returns 1 if there is a mathematical proof of this formula in the axiomatic system of arithmetic, and 0 otherwise, is not computable (see Theorem 9-3.9).

(3) The function which takes as input a Diophantine equation, ie, an equation with integer coefficients —par exemple $3x^2 + 2xy + 4y^2 = 0$— and decides if this equation admits an integer solution, is not computable (see Theorem 12-1.2).

Partial functions. As explained in the frame "Integers in Computer Sciences" above, we will generally restrict ourselves to programs taking an integer as a parameter, and returning another integer if the computation ends. It is essential to note that the functions generated by the computer programs are *partial*, in the sense that the computation can never halt on some of its inputs. This bias is due to control structures of type "`while`" whose termination condition may never be satisfied, as shown in the following example, which computes — in the worse possible manners — the square root of an integer:

```
function Root (n){
    r = 0;
    while (r * r ≠n){
      r = r + 1;
    }
    return r;
}
```

If n is not the square of an integer, the loop `while` will execute ad infinitum, and the program will never return a value. When the program does not halt on an input n, it is considered that the function from $\mathbb{N}$ to $\mathbb{N}$ which is associated with it is not defined on n. The domain of definition of the partial function associated with a program is therefore the set of inputs on which it halts. Functions defined by programs are called *partial computable functions*. When the program halts on all its inputs, the generated function is then called *total computable function*, or quite simply *computable fonction*.

Notation

If f and g are two partial functions, we will denote by $f(x) = g(x)$ to signify that either f and g are both defined and return the same value on x, or neither f nor g are defined on x.

Codes. In our mathematical study, we will represent computer programs by integers, assuming a fixed coding function. By *computer programs*, we must understand here a finite sequence of characters — supposed to have meaning in a programming language chosen beforehand. Concretely, we will say that an integer e *codes* for a program P if e is the integer encoded by the binary representation of the string corresponding to the program P. In particular, this coding is injective and intuitively computable. Note that one could imagine many other possible encodings. We will see another in detail in the proof of Theorem 6-3.27. In the meantime, this one will be suitable.

Notation

We denote by $\Phi_e : \mathbb{N} \to \mathbb{N}$ the partial function defined by the computer program of code e.

We will assume that any integer e codes for a valid program. In practice, in all programming languages, there are strings corresponding to illformed programs. Some integers e encode unintelligible strings of characters. If this is the case, we can then realize it (this corresponds to having a syntax error when we try to compile a program) and consider that e then corresponds to a program which never halts. Then Φ_e is the nowhere-defined function.

Notation

Let $\Phi_e : \mathbb{N} \to \mathbb{N}$ be a partial computable function and $x \in \mathbb{N}$ an integer. We will write $\Phi_e(x)\downarrow$ if the program encoded by e halts on the input x (necessarily after a finite number of computation steps) and returns an integer result. In the opposite case, we will write $\Phi_e(x)\uparrow$.

The domain of definition of the partial computable function $\Phi_e : \mathbb{N} \to \mathbb{N}$ is therefore $\{x \in \mathbb{N} : \Phi_e(x)\downarrow\}$. If $\Phi_e(x)\downarrow$, we will sometimes write $\Phi_e(x)\downarrow= y$ to mean that the code program e halts on the input x and returns the integer y. Conversely, we will sometimes write $\Phi_e(x) \uparrow\neq y$ to mean the opposite, ie $\Phi_e(x)\uparrow \vee \Phi_e(x)\downarrow\neq y$.

Computation time. Any programming language comes with a notion of execution step and computation time. An execution step is an atomic operation of the language, indecomposable into sub-steps. It corresponds to an elementary instruction of the language. The notion of execution step induces that of computation time, which is defined by the number of execution steps carried out since the launching of computation. When a program halts on an input, its computation time is finite.

Notation

Let $\Phi_e : \mathbb{N} \to \mathbb{N}$ be a partial computable function and $x, t \in \mathbb{N}$ two integers. We will write $\Phi_e(x)[t]\downarrow$ if the program encoded by e halts *before t computation steps.* In the opposite case, we will write $\Phi_e(x)[t]\uparrow$.

Note that $\Phi_e(x)\downarrow$ iff there exists a computation time t such that $\Phi_e(x)[t]\downarrow$. Furthermore, if $\Phi_e(x)[t]\downarrow$, then $\Phi_e(x)[s]\downarrow$ for all $s \geqslant t$.

Remark

We will have to handle computable functions with several parameters. For a fixed integer $n \in \mathbb{N}^*$, we will denote as above by $\Phi_e : \mathbb{N}^n \to \mathbb{N}$ the partial function with n integer parameters encoded by e and we will write $\Phi_e(x_1, \ldots, x_n)\downarrow= y$ if the code program e halts on the inputs $x_1, \ldots, x_n$ and returns the integer y.

2. Computable sets

The adjective "computable" naturally applies to functions; indeed as described above, any integer e codes for a program to which a partial computable function corresponds. Most often, however, when we speak of computable functions, we will consider that they are total functions.

Definition 2.1. Let $n \in \mathbb{N}^*$. A function $f : \mathbb{N}^n \to \mathbb{N}$ is *computable* if there exists a program code e such that, for all $x_1, \ldots, x_n$, we have

$$\Phi_e(x_1, \ldots, x_n)\downarrow= f(x_1, \ldots, x_n)$$

Computability can also be used to measure the descriptive complexity of countable mathematical objects, and in particular that of sets of integers. Intuitively, a set of integers $E \subseteq \mathbb{N}$ is computable if it can be described by a computable process. A set being totally specified by its elements, it is computable if its characteristic function is computable.

Definition 2.2. Let $n \in \mathbb{N}^*$. A set $A \subseteq \mathbb{N}^n$ is *computable* if there exists a program code e such that, for any $x_1, \ldots, x_n$, we have

- $\Phi_e(x_1, \ldots, x_n)\downarrow= 1$ iff $(x_1, \ldots, x_n) \in A$.

- $\Phi_e(x_1, \ldots, x_n){\downarrow}= 0$ iff $(x_1, \ldots, x_n) \notin A$. ◇

We will sometimes call *predicates* the subsets of $\mathbb{N}^n$, then using the notation $A(x_1, \ldots, x_n)$ to mean $(x_1, \ldots, x_n) \in A$. The term predicate comes from the view of $A \subseteq \mathbb{N}^n$ not as a set, but as a property of n-tuples of integers. Thus, $A(x_1, \ldots, x_n)$ means that the n-tuple $(x_1, \ldots, x_n)$ has the property A. Conversely, $\neg A(x_1, \ldots, x_n)$ means that the n-tuple $(x_1, \ldots, x_n)$ does not have the A property.

Exercise 2.3. Show that the pairing bijection defined in Proposition 2-3.5 and Exercise 2-3.7, which to $(a, b) \in \mathbb{N}^2$ associates $\langle a, b \rangle \in \mathbb{N}$, is computable. Show that the inverse functions π_0, π_1 such that $a = \pi_0(\langle a, b \rangle)$ and $b = \pi_1(\langle a, b \rangle)$ are also computable. ◇

Exercise 2.4. Let $A \subseteq \mathbb{N}^2$ be a computable predicate. Show that the sets:

(1) $\{(x, y) \in \mathbb{N}^2 : \forall z < y \ (x, z) \in A\}$

(2) $\{(x, y) \in \mathbb{N}^2 : \exists z < y \ (x, z) \in A\}$

are also computable. ◇

Recursive set

Historically, computable sets were called *recursive* due to the computation paradigm of general recursive functions in which definitions by induction play a dominant role (see Chapter 6). Gradually, with the improvement of our understanding of the concept of computation, the terminology of the field has evolved. One can however regularly see the terms of *recursion theory* and of *recursive set* to speak about computability theory and computable set.

3. Universal program

The computational model devised by Turing in 1936 consists in defining a machine for each computable function. If we compare a Turing machine to a physical device, it consists, for each task that we want to accomplish, to create a robot performing that specific task.

We are now going to state a first fundamental theorem of computability theory and proved by Turing in his original article: the existence of a *universal* Turing machine, capable of simulating all other Turing machines.

This work is sometimes considered a precursor of von Neumann's architecture[2], one aspect of which is the storage of programs in memory. In modern Computer Science, this theorem can be seen as stating the existence of a *universal* computer program, allowing all other computer programs to be simulated.

Theorem 3.1
Let $n \in \mathbb{N}^*$. There exists a computer program code e for which $\Phi_e : \mathbb{N}^{n+1} \to \mathbb{N}$ is such, that for all $a, x_1, \dots, x_n$, we have

- $\Phi_e(a, x_1, \dots, x_n)\uparrow$ *iff* $\Phi_a(x_1, \dots, x_n)\uparrow$;
- $\Phi_e(a, x_1, \dots, x_n)\downarrow = y$ *iff* $\Phi_a(x_1, \dots, x_n)\downarrow = y$.

In practice, a universal program exists very concretely in many languages. For example, in Java, it is simply the virtual machine that compiles and runs any program written in Java. For languages that do not use a virtual machine, a universal program will simply be an interpreter which decomposes the program which it receives into a sequence of instructions, and executes it step by step. While such a program is of course complex to design, there should be no doubt for the programmer that this is something very possible: it is simply a virtual machine. The reader can consult the proof of Theorem 6-3.27, which demonstrates the existence of a universal program for the specific computational model of register machines.

The existence of such a program allows us to define functions which perform manipulations on codes before executing them, or dynamically execute codes passed as parameters. For example,

$$f(x, y) = \Phi_{x+1}(y + 2)$$

is a valid definition, because it is equivalent to $f(x, y) = \Phi_e(x + 1, y + 2)$, where Φ_e is the universal program.

4. SMN theorem

The SMN theorem is our first theorem on the manipulation of computer program codes. It is not conceptually difficult. Let $\Phi_e : \mathbb{N}^{m+n} \to \mathbb{N}$ be the function of code e. Then, given $x_1, \dots, x_m$, we can computably transform our code e into a code a such that the computation $\Phi_a(y_1, \dots, y_n)$ gives the same result as the computation

$$\Phi_e(x_1, \dots, x_m, y_1, \dots, y_n)$$

[2]Which is almost always the one in use today.

The important point is that the transformation of e into a is computable as a function of e and $x_1, \ldots, x_m$.

Theorem 4.1 (SMN theorem)
For all $n, m \in \mathbb{N}^*$, there exists a total computable function

$$S_n^m : \mathbb{N}^{m+1} \to \mathbb{N},$$

such that for all $e, x_1, \ldots, x_m, y_1, \ldots, y_n$,

$$\Phi_{S_n^m(e,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \Phi_e(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

Proof. Let us describe the function S_n^m. Given e, $x_1, \ldots, x_m$, decode e to get the program P_e. Computably modify the program P_e to add "hard-coded" instructions assigning the values $x_1, \ldots, x_m$ to the corresponding variables, then compute the code of the new program. For example, if the program $\Phi_e : \mathbb{N}^4 \to \mathbb{N}$ is

```
function MyProgram (x1, x2, x3, x4){
    // CODE
}
```

The program $\Phi_{S_2^2(e,5,3)}$ matches the string

```
function MyProgram2 (x3, x4){
    x1 = 5;
    x2 = 3;
    // CODE
}
```

■

The SMN theorem will be used from now on without calling on it explicitly, with sentences like "for every x, let e_x be the code of the program which takes y as input, and does ...[something which depends on x and y]", it being understood then that the process to obtain e from x is computable. We also say in this case that the process is *uniform* in x.

Acceptable coding

The universal program existence theorem and the SMN theorem are not valid for all coding functions. Let us recall the one we use here: each program P is encoded by the integer corresponding to the binary representation of the string which contains P.
Rogers [191] proved that any coding function satisfying the Universal Program Theorem and the SMN Theorem was the result of a computable

permutation of this canonical coding. However, there are other codings of partial computable functions which do not satisfy these theorems. In particular, Friedberg [67] defined a computable encoding of all partial functions computable without repetition. This is of course not the case for our coding, for which the same partial function has an infinity of different codes. This is what we are about to do with Lemma 5.1.

5. Padding lemma

The padding lemma is useful from time to time and indicates that for any computer program e, there exists an infinite number of equivalent programs, the code of which can also be computed from e: it suffices to add instructions that are useless.

Lemma 5.1 (Padding lemma). There exists a total computable function $h : \mathbb{N}^2 \to \mathbb{N}$ such that for all $e, n \in \mathbb{N}$, we have $h(e,n) \geqslant n$, and $\Phi_{h(e,n)} = \Phi_e$. ⋆

Proof. Given the code e and an integer n, decode e to obtain the program P_e. Add n unnecessary instructions to P_e, then encode the new program to an integer i.

For example, if the language has an instruction `skip` which does nothing, then it suffices to add to the program a sequence of instructions `skip` as follows:

```
function MyProgram (x){
    // CODE
    skip;
    skip;
    ...
}
```

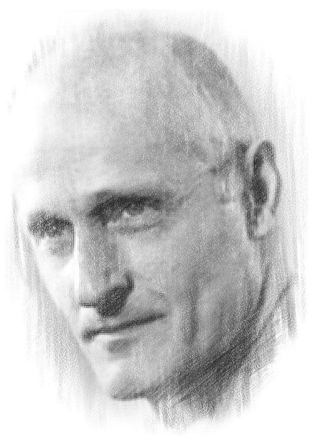
■

Note that $\Phi_{h(e,n)}$ and Φ_e are equal as mathematical functions, but have different computer codes.

6. Kleene's fixed point theorem

Kleene's fixed point theorem, also known as *recursion theorem*, is much more subtle than the SMN theorem. Stephen Cole Kleene is considered with Kurt Gödel, Alan Turing, Emil Post and Alonzo Church, his teacher, as one of the founders of computability theory. He formalizes with Post the notion of *degree of unsolvability*, which we will call later *Turing degree* and which will be precisely defined in Chapter 4.

Among his most remarkable works, figure the definition and the study of hyperarithmetic sets and computable ordinals [118], which we will see in Part IV and for which we will need to create computer programs which can *access their own code*. The notion may seem doubtful: how can we use in the definition of an object A the object A itself? Self-references often lead to paradoxes. However, we will see that in the case of computer programs, access to its own code is quite valid, and even very useful in some cases. We will see a remarkable example of the use of the fixed point theorem in the proof of Theorem 19-1.7. Let's see more precisely what it is. The theorem states that for any function that modifies programs, there exists a program whose behavior is not modified by the function.


Stephen Cole Kleene, 1909–1994

Theorem 6.2
For any total computable function $f : \mathbb{N} \to \mathbb{N}$, there exists $e \in \mathbb{N}$ such that for any n,
$$\Phi_{f(e)}(n) = \Phi_e(n).$$

Before moving on to the proof, let's see how this mysterious claim allows you to create programs with access to their own code. Suppose that a program M uses a variable `var` that was initialized to a certain value at the start of its execution. We can then easily define a total computable function f which takes an integer n as a parameter, and returns the code of the program M, which begins its execution with `var` initialized to n. According to the fixed point theorem, there is a value e such that the programs of code e and $f(e)$ have the same behavior. So e is a program code equivalent to that of the M program running with the variable `var` initialized to e: there is a version of M that can access its own code. Here is, more formally, what we have just stated.

Corollary 6.3
For any partial computable function $g : \mathbb{N}^2 \to \mathbb{N}$, there exists $e \in \mathbb{N}$ such that for any n, we have

$$\Phi_e(n) = g(e, n)$$

Proof. Let i be such that $\Phi_i(x, n) = g(x, n)$ for all x, n. By the SMN theorem (see Theorem 4.1), for all x, n, we have $\Phi_{S_2^1(i,x)}(n) = \Phi_i(x, n)$. Let f be the function defined by $f(x) = S_2^1(i, x)$. By the fixed point theorem (see Theorem 6.2), there exists e such that, for all n, $\Phi_{f(e)}(n) = \Phi_e(n)$. In particular, $\Phi_e(n) = \Phi_{f(e)}(n) = \Phi_i(e, n) = g(e, n)$. ■

The proof of Theorem 6.2, although concise, is somewhat obscure; this is why we provide beforehand a piece of code whose objective is to give the programmer an intuition of how to write a program with access to its own code. The example is given here in the JavaScript language.

In the following, the two ellipses must each contain the same code, no matter which one. In JavaScript, "backquotes" delimit a string over several lines. Function `replace` will replace the first occurrence of '# 'with the content of variable `v`.

```
function fct (){
  let v = `
function fct (){
  let v = '#'
  v = v.replace ('#', v)
  ... // my code
} `
  v = v.replace ('#', v)
  ... // my code
}
```

The execution of this program will be done with the variable `v` having for content the program itself, except that the "backquotes" are then transformed into simple "quotes". It is of course possible to make the variable `v` contain *exactly* the program, but that would make the example much less understandable. Aiming to give an intuition and not a proof, we have kept it that way. Let us now proceed to the formal proof of our theorem, within the framework of the notations and principles that we have defined so far.

Proof of Theorem 6.2. Let a be the code of a one-parameter machine, which on input n returns the code of a one-parameter machine m, which performs the following operations.

(1) It starts the computation of $f\big(\Phi_n(n)\big)$.

(2) If we have $f\big(\Phi_n(n)\big)\downarrow$, then it returns the result of the computation of the code machine $f\big(\Phi_n(n)\big)$ on the input m (and otherwise does not halt).

Formally, a is such that, for all $n, m \in \mathbb{N}$,

$$\Phi_{\Phi_a(n)}(m) = \Phi_{f(\Phi_n(n))}(m).$$

There is a slight abuse of notation here: if $\Phi_n(n)\uparrow$, then $m \mapsto \Phi_{f(\Phi_n(n))}(m)$ denotes the nowhere defined function. Note that Φ_a is a total function: for any n, in the computation of $\Phi_a(n)$, we do not try to do the steps (1) and (2), but only to compute the code of a machine that makes them

The proof that such a code a exists is given by the SMN theorem, here is how. The function $(n, m) \mapsto \Phi_{f(\Phi_n(n))}(m)$ is computable (possibly partial), and there is therefore a code b such that

$$\Phi_b(n, m) = \Phi_{f(\Phi_n(n))}(m).$$

According to the SMN theorem, there is a total computable function s such that $\Phi_{s(b,n)}(m) = \Phi_{f(\Phi_n(n))}(m)$. As s is total computable, there is a code a such that $\Phi_a(n) = s(b, n)$.

The fixed point will then be $\Phi_a(a)$. Indeed, we have

$$\forall m,\ \ \Phi_{\Phi_a(a)}(m) = \Phi_{f(\Phi_a(a))}(m).$$

This concludes the proof. ∎

Fixed point theorem and paradox

As explained above, Kleene's fixed point theorem makes it possible to design *self-referential* programs, i.e., programs which can read their code during execution, and adapt their behavior accordingly. The ability to self-refer is often a source of paradoxes.

The Barber's Paradox, for example, tells the story of a philanthropic barber who decided to shave all the people who did not shave themselves. Should he shave himself? Paradox ... In mathematics, Russel's paradox follows the same pattern: let E be the set of all sets which do not belong to themselves. For example, $\mathbb{N}$ is not an integer, so $\mathbb{N} \notin \mathbb{N}$, therefore $\mathbb{N} \in E$. The problematic question is then "Does $E \in E$?"

Why does Kleene's fixed point theorem not generate a paradox? If we try to follow the same scheme as the two previous paradoxes, we will define a function Φ_e which knows its code e, and therefore can decide, for any input n, to execute $\Phi_e(n)$ and return a different value than the one returned by $\Phi_e(n)$. We would therefore have $\Phi_e(n) \neq \Phi_e(n)$.

The solution comes from the partiality of the functions: indeed, Φ_e will simply not be defined in n and will run indefinitely.

The reader wishing to explore the possibilities of the fixed point theorem can tackle the following exercise, the object of which is to prove Rice's theorem, which will be discussed in more detail in Section 5-6.

Exercise 6.4. **($\star$)** Let $A \subseteq \mathbb{N}$ be such that $A \neq \mathbb{N}$, $A \neq \emptyset$ and such that A is computable.

1. Show that there exists a computable function f such that we have $x \in A$ iff $f(x) \notin A$.
2. Using the fixed point theorem, deduce that there are i, j such that $\Phi_i = \Phi_j$, with $i \in A$ and $j \notin A$.
3. Deduce that there are no computable predicates A such that $e \in A$ iff e is the code of a program which multiplies by 2.
4. Generalize the previous question to show *Rice's theorem*: for all "non-trivial behavior", it is not possible to compute the set of program codes having this behavior. Formally: let a predicate $P \subseteq \mathbb{N}$ with $P \neq \mathbb{N}$ and $P \neq \emptyset$ such that, for all e_1, e_2 for which $\Phi_{e_1} = \Phi_{e_2}$, we have $e_1 \in P \leftrightarrow e_2 \in P$. Then, P is not computable. $\diamond$

7. Computably enumerable sets

Computability places "computable" as the reference computational power. This is the weakest computational notion that this paradigm allows to identify. As we have seen, a set E is computable if there is a procedure which, given an element, indicates whether this element belongs to E or not. The procedure should always halt and give a correct answer.

There are, however, a certain number of mathematical problems which can be expressed naturally in the form of sets whose elements are enumerable by a computable procedure, but out of order. These sets are called *computably enumerable.*

For example, let E be the set of mathematical theorems. There is no procedure which, given a mathematical formula, returns true or false depending on whether this formula is provable or not. However, it is possible to enumerate the theorems, listing all possible strings, testing whether this is a valid proof, and if so, listing the conclusion of the proof (we will discuss this in more detail in Chapter 9).

We are now going to formally define the notion of computably enumerable set in a form which may seem far from the informal definition that we have just given. We will see through Proposition 7.2 that these definitions coincide.

> **Definition 7.1.** A set $A \subseteq \mathbb{N}$ is *computably enumerable* (c.e.) If there exists a program code e such that $n \in A \leftrightarrow \Phi_e(n)\downarrow$, for all $n \in \mathbb{N}$. $\diamondsuit$

Note that computably enumerable sets were historically called *recursively enumerable.* It is still common to see articles using the old terminology, and in particular the abbreviation r. e. instead of c.e.

We can see the machine code e as a process which enumerates A: for each integer n, we look for an integer t such that $\Phi_e(n)$ halts in t computation steps. If such a t is found, then we enumerate n in our set. Knowing that there is an infinity of integers, we need to fix an order of execution to carry out only a finite number of computations at each step. The idea is to break it down step by step as follows.

1. Test if $\Phi(0)[0]\downarrow$.
2. Test whether $\Phi(0)[1]\downarrow$ or $\Phi(1)[1]\downarrow$.
3. Test whether $\Phi(0)[2]\downarrow$ or $\Phi(1)[2]\downarrow$ or $\Phi(2)[2]\downarrow$.
4. ...

The first time that we find a step t such that $\Phi(n)[t]\downarrow$ for a certain integer n, we enumerate the latter. Such an enumeration can be seen as a computable function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the n-th element listed in A. This idea is repeated in the proof of the following proposition.

Proposition 7.2. An infinite set A is computably enumerable iff there exists an injective total computable function $f : \mathbb{N} \to \mathbb{N}$ such that

$$f(\mathbb{N}) = A.$$

Proof. Let A be an infinite computably enumerable set, and let e be such that $\Phi_e(n)\downarrow$ iff $n \in A$. We define the computable function $f : \mathbb{N} \to \mathbb{N}$ as follows.

- To compute $f(0)$, we look for the smallest t such that $\Phi_e(s)[t]\downarrow$ for some $s \leqslant t$, and we then return the smallest such integer $s \leqslant t$.
- To compute $f(n+1)$, we first run the computation of $f(i)$ for all $i \leqslant n$, then we look for the smallest t such that $\Phi_e(s)[t]\downarrow$ for some $s \leqslant t$ such that s is different from each $f(i)$ for $i \leqslant n$, and we then return the smallest such integer $s \leqslant t$.

The process for determining $f(n)$ is indeed computable, and since our computably enumerable set is infinite, the function f will halt on all its values. It is then clear that $f(\mathbb{N}) = A$.

Suppose now that $f(\mathbb{N}) = A$ for a total computable function f. We define the function g which on n searches for the smallest t such that $f(t) = n$, and then halts (and otherwise searches indefinitely without ever halting). We have $g(n)\downarrow$ iff $\exists t\ f(t) = n$. ■

It should be clear to the reader that a computable set is computably enumerable.

Exercise 7.3. Show that any computable set is computably enumerable. ◇

We will see that the reverse is not necessarily true: there are computably enumerable sets which are not computable. The following proposition gives more precisely the connection between these two concepts.

Proposition 7.4. A set A is computable iff A and $\mathbb{N} \setminus A$ are both computable enumerable. ⋆

Proof. Let A be a computable set. According to Exercise 7.3, it is computably enumerable. Moreover, the set $\mathbb{N} \setminus A$ is also computable, and therefore computable enumerable.

Suppose now that we have two codes e_1, e_2 such that $\Phi_{e_1}(n)\downarrow$ iff $n \in A$ and $\Phi_{e_2}(n)\downarrow$ iff $n \in \mathbb{N} \setminus A$. We define the computable function $f(n)$ which searches for the smallest t such that $\Phi_{e_1}(n)[t]\downarrow$ or such that $\Phi_{e_2}(n)[t]\downarrow$, then returns 1 in the first case and 0 in the second. It is clear that the function f computes the set A. ■

If a computably enumerable set is not in general computable, it can be approximated by an increasing sequence of uniformly computable sets, in the following sense. We say that a sequence of sets $A_0, A_1, \dots$ is *uniformly computable* if there exists a computable function $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ such that, for all x, n, we have $f(x, n) = 1$ iff $x \in A_n$.

Definition 7.5. A *c.e. approximation* of a set A is a uniformly computable sequence of sets $A_0, A_1, \dots$ such that $A_n \subseteq A_{n+1}$ for all n, and $\bigcup_n A_n = A$. ◇

Proposition 7.6. A set A is computably enumerable if, and only if, it has a c.e. approximation ⋆

Proof. Suppose that A is c.e. By definition, there is a program code e such that $x \in A \leftrightarrow \Phi_e(x)\downarrow$, for all $x \in \mathbb{N}$. Let $A_0, A_1, \dots$ be the uniformly computable sequence of sets defined by $A_n = \{x : \Phi_e(x)[n]\downarrow\}$. It is clear that $A_n \subseteq A_{n+1}$, because if the machine Φ_e halts on an input x before n steps, it will halt before $n+1$ steps.

Suppose now that the set A has a c.e. approximation, namely, $A_0, A_1, \dots$ Let Φ_e be the program which, for an input x, computes $A_0(x)$, then $A_1(x)$, then $A_2(x)$, and so on, until $A_n(x) = 1$ for some n and halts there. If $A_n(x) = 0$ for all n, then $\Phi_e(x)\uparrow$, otherwise $\Phi_e(x)\downarrow$. By construction, $\{x : \Phi_e(x)\downarrow\} = \bigcup_n A_n = A$. ■

Notation

Given a c.e. set A, we denote by $A[0], A[1], \dots$ a fixed c.e. approximation of A. In particular, $A[s]$ is the *approximation of A at step s*. By convention, $A[s]$ is a finite set with $\max A[s] < s$ if $A[s] \neq \emptyset$. We will sometimes also use the notation A_s instead of $A[s]$.

As we have said, there are computably enumerable sets which are not computable. The canonical example is known as the "halting problem".

Definition 7.7. The *halting problem* $\emptyset'$ is the set

$$\emptyset' = \{n \in \mathbb{N} : \Phi_n(n)\downarrow\}. \qquad \diamondsuit$$

Alan Turing demonstrates in 1936 that the halting problem is not computable, using a diagonal argument, like the one introduced by Cantor to demonstrate the non-countability of real numbers. Before giving a mathematical proof, we give an intuition via a little code using Java syntax.

Let us absurdly assume that we have at our disposal a function `boolean Halt(String p, String v)` which returns `true` if the function in the string `p` halts when it is executed with the parameter `v`, and returns `false` otherwise. As usual, if `p` contains a string that does not match a valid function or does not match a function taking a parameter of type `String`, then `Halt` returns `false`. Consider the following program.

```
function Diagonal (x){
   if (Halt (x, x)){
      while (true); //infinite loop
   } else{
      return; //end
   }
}
```

What does the execution of program `Diagonal` return with as parameter a string `x` corresponding to the program `Diagonal` itself? We see that we come up with paradox:

- if `Halt(x,x)` returns `true`, then `Diagonal` does not halt on `Diagonal` as a parameter;
- in the opposite case, `Diagonal` halts on `Diagonal` as parameter.

Thus, the function `Halt` does not keep its promises.

Theorem 7.8
The halting set is a computably enumerable set which is not computable.

Proof. The partial function $f : n \mapsto \Phi_n(n)$ is clearly computable, and we have $f(n)\downarrow$ iff $\Phi_n(n)\downarrow$. By definition, $\emptyset' = \{n : f(n)\downarrow\}$. Therefore, $\emptyset'$ is computably enumerable.

Let us now assume absurdly that $\emptyset'$ is a computable set. In particular, according to Proposition 7.4, the set $\mathbb{N} \setminus \emptyset'$ is computably enumerable, and there exists a code e such that $n \in \mathbb{N} \setminus \emptyset'$ iff $\Phi_e(n)\downarrow$. Then, for all n,

$$\Phi_e(n)\downarrow \;\leftrightarrow\; n \in \mathbb{N} \setminus \emptyset' \;\leftrightarrow\; \Phi_n(n)\uparrow.$$

In particular, for $n = e$,

$$\Phi_e(e)\downarrow \;\leftrightarrow\; e \in \mathbb{N} \setminus \emptyset' \;\leftrightarrow\; \Phi_e(e)\uparrow,$$

which is a contradiction. The sentence is therefore not computable, and in particular the negation of the sentence is not computably enumerable. ■

We invite the reader to consider the following two exercises, which should not pose any difficulties and which allow us to reflect a little on computably enumerable sets.

Exercise 7.9. ($\star$) An infinite set is *computably enumerable in the order* if there exists a total function $f : \mathbb{N} \to \mathbb{N}$ such that $f(\mathbb{N}) = A$ and such that $f(n) < f(n+1)$. Show that if an infinite set A is computably enumerable in order, then A is computable. ◇

Exercise 7.10. ($\star$) Show that any infinite computable enumerable set contains an infinite computable subset. ◇

Exercise 7.11. ($\star\star$) Two sets A and B are *computably inseparable* if $A \cap B = \emptyset$ and if no computable set C allows to separate A and B; in other words, no computable set C is such that $A \subseteq C$ and $C \cap B = \emptyset$. Show that there are two computably inseparable c.e. sets. ◇

Exercise 7.12. ($\star\star$) Show that given $A, B \subseteq \mathbb{N}$ two computable sets, the set $D_{A,B} = \{x - y : x \geqslant y \text{ and } x \in A \text{ and } y \in B\}$ is not necessarily computable.

Indication .– Show that for any c.e. set $C \subseteq \mathbb{N}$, there exist two computable sets $A, B \subseteq \mathbb{N}$ such that $x \in C$ iff $2^x \in D_{A,B}$. $\diamond$

Some people —a small minority— will perhaps have integrated with great ease everything that has been seen so far, to the point no doubt of getting bored a little. This is what the following exercise is aimed at, which should occupy them for a little while . . .

Exercise 7.13. ($\star\star\star$) **(***Friedberg [67]***).** A c.e. set X is *maximal* if $\mathbb{N} \setminus X$ is infinite and if any c.e. set $Y \supseteq X$ is such that $Y \setminus X$ is finite or such that $\mathbb{N} \setminus Y$ is finite. Show that there exists a maximal c.e. set.

Indication .– We denote by W_e the c.e. set given by $\{n \in \mathbb{N} : \Phi_e(n) \downarrow\}$. We can start by finding a uniform process which, on each $e \in \mathbb{N}$, associates a code d such that $\mathbb{N} \setminus W_d$ is infinite, and such that if $W_d \subseteq W_e$, then either W_e is finite, or $\mathbb{N} \setminus W_e$ is finite. $\diamond$

Chapter 4

Turing degrees

A non-computable set can be seen as an insoluble problem: there is no algorithm making it possible to decide whether or not an integer belongs to this set. One of the goals of computability theory is to study and understand the universe of insoluble problems, through different comparisons and classifications. Various tools are introduced for this. The most important of them is the subject of this chapter and finds its genesis in “Systems of logic based on ordinals” [235], the famous article by Alan Turing presenting his thesis work, and will be formalized and studied later by Post [189] then Post and Kleene [122]: given a non-computable set X, we imagine being able to use it as “oracle” in order to increase the computational power of our machines. We will then say that two sets are in the same *degree of insolubility* or in the same *Turing degree*, if each can be computed with an algorithm using the other as “oracle”.

1. Finite strings

Before getting to the heart of the matter, we need to introduce some vocabulary and notation about binary strings. Formally, a binary string is a partial function from $\mathbb{N}$ to $\{0, 1\}$ whose domain of definition is an initial segment of $\mathbb{N}$. More informally, it is a finite sequence of 0 and 1.

Definition 1.1. We denote by $2^{<\mathbb{N}}$ the set of finite sequences of 0 and 1. The variables σ, τ, ρ will normally be used to denote elements of $2^{<\mathbb{N}}$, which will generally be called *strings*. These sequences will be handled via the following symbols:

- ϵ: the empty string, of length 0
- i^n for $i \in \{0,1\}$: a sequence of n bit repetitions i
- $\sigma\tau$ or $\sigma\hat{}\tau$: the concatenation of σ and τ
- $\sigma \preceq \tau$: the string σ is a prefix of τ, that is $\exists\rho$ tel que $\sigma\rho = \tau$
- $\sigma \prec \tau$: the string σ is a strict prefix of τ, ie $\exists\rho \neq \epsilon$ such that $\sigma\rho = \tau$
- $|\sigma|$: the length of σ
- $\sigma(n)$ for $n < |\sigma|$: the value of the n-th bit of σ, starting at 0
- We will say that two strings σ, τ are *incompatible* if we have neither $\sigma \preceq \tau$ nor $\tau \preceq \sigma$ (we will also write $\sigma \npreceq \tau$ and $\tau \npreceq \sigma$). If, on the other hand, $\sigma \preceq \tau$ or $\tau \preceq \sigma$, the two strings σ and τ are compatible. ◊

We will sometimes identify a bit $i \in \{0,1\}$ with the string of length 1 whose only bit is i. Thus, we will denote by σi or $\sigma\hat{}i$ the concatenation of a string σ and a bit i. According to the previous definitions, $|\epsilon| = 0$, and for any non-empty string σ, the first and last bit are respectively $\sigma(0)$ and $\sigma(|\sigma|-1)$. Chains and infinite sequences can be combined.

Definition 1.2. We adopt the following notations for $\sigma \in 2^{<\mathbb{N}}$ and $X \in 2^{\mathbb{N}}$:

- σX: the concatenation of σ and X
- $\sigma \prec X$: the string σ is a prefix of X, that is $\exists Y \in 2^{\mathbb{N}}\ \sigma Y = X$
- $X\restriction_n$ for $n \geqslant 0$: the prefix of X of size n. ◊

2. Computation with oracle

Intuitively, a computation with an oracle $X \subseteq \mathbb{N}$ is easy to understand: it is a computation which can at any time use an instruction of the form "does n belong to X?". This instruction can be compared to a function call, which always returns the correct answer.

If the oracle used is not computable, it is therefore possible to write computer programs which computes, using this oracle, objects which would not be computable otherwise. In particular, X itself with computable with oracle X. Our goal now is to study sets of integers in terms of the computational power they provide when used as oracles.

Definition 2.1. A *Turing functional* or simply a *functional* is a function computable by an algorithm in a programming language enriched with instructions of the form "does n belong to the oracle?". ♢

A functional therefore has two kinds of parameters: the usual integer parameters, as for computable functions, and one oracle parameter, which is an infinite binary sequence, or equivalently a set of integers, representing the oracle. We will also call *first-order parameters* the integer parameters and *second-order parameter* the oracle parameter. A functional can be seen as a scheme of partial functions, parameterized by the oracle: the same functional "fed in" with a different oracle will yield a different function.

Notation

We note $\Phi_e(X,n)$ or $\Phi_e^X(n)$ for the result of the computation of the functional Φ_e with the oracle X and on the input n. We will write in the same way $\Phi_e(X,n)\downarrow$, $\Phi_e(X,n)\uparrow$, $\Phi_e(X,n)[t]\downarrow$, $\Phi_e(X,n)[t]\uparrow$ to signify that the functional Φ_e respectively halts, does not halt, halts in time lower than t, does not halt in time lower than t, with the oracle X and on the input n.

In order to have a uniform vision, we now suppose that we only work with functionals. It is easy to see that the computable functions are exactly those which are computable by functions using the empty set as oracle (or any other computable set).

Definition 2.2. A set A is said to be X-*computable* or computable relative to X if it is computable by a Turing functional using X as an oracle. A set A is said to be *computably enumerable relative to* X if it is the domain of definition of a Turing functional using X as an oracle. ♢

The notion of oracle is generalized to functions. With an oracle $f:\mathbb{N}\to\mathbb{N}$, a computation consists in adding the function f to the primitives of the programming language. Thus, the program can at any time query f on inputs to know the results. We can therefore speak of an f-*computable* set if it is computable by a Turing functional using f as an oracle. Equivalently, a set is f-computable if it is G_f-computable, where $G_f \in 2^{\mathbb{N}}$ is an encoding of the graph of the function f by an element of $2^{\mathbb{N}}$, for example with $\langle n,m\rangle \in G_f$ iff $f(n)=m$.

Notation

We will also write X-c.e. to mean computably enumerable relative to X.

Example 2.3. Suppose for example that we have an oracle $X \subseteq \mathbb{N}$ such that the function f which to n associates the n-th element of X increases fast enough to dominate the halting time of computer programs. Formally: $\Phi_e(e)\downarrow$ implies $\Phi_e(e)[f(e)]\downarrow$ for all $e \in \mathbb{N}$. It is then easy to create a Turing functional allowing to compute $\emptyset'$ from X: to know if $e \in \emptyset'$, it suffices to browse X until you find its e-th element $f(e)$, then to compute $\Phi_e(e)$ during $f(e)$ stages of computation. If $\Phi_e(e)[f(e)]\downarrow$, then $e \in \emptyset'$. Otherwise, $e \notin \emptyset'$.

3. Relativization of proofs

Most of the computability-theoretic arguments apply to oracle machines by replacing "machine" by "machine with an oracle X". This purely syntactic operation, which we call *relativization* (to an oracle), therefore gives for free a scheme of similar results, parameterized by an oracle X. Let us take the example of the undecidability of the halting problem, by replacing the notion of computation by that of computation with oracle X.

Theorem 3.1
For any oracle $\mathbf{X}$*, the set* $Y = \{n : \Phi_n^{\mathbf{X}}(n)\downarrow\}$ *is not* $\mathbf{X}$*-computable.*

Proof. The partial function $f : n \mapsto \Phi_n^{\mathbf{X}}(n)$ is clearly $\mathbf{X}$-computable, and we have $f(n)\downarrow$ iff $\Phi_n^{\mathbf{X}}(n)\downarrow$. By definition, $Y = \{n \in \mathbb{N} : f(n)\downarrow\}$. Then, Y is $\mathbf{X}$-computably enumerable.

Let us now assume absurdly that the set Y is $\mathbf{X}$-computable. In particular, according to Proposition 3-7.4 **relativized to X**, the set $\mathbb{N} \setminus Y$ is $\mathbf{X}$-computably enumerable, and there exists a code e such that $n \in \mathbb{N} \setminus Y$ iff $\Phi_e^{\mathbf{X}}(n)\downarrow$. We then have for all n

$$\Phi_e^{\mathbf{X}}(n)\downarrow \leftrightarrow\ n \in \mathbb{N} \setminus Y\ \leftrightarrow\ \Phi_n^{\mathbf{X}}(n)\uparrow .$$

In particular, for $n = e$, we have

$$\Phi_e^{\mathbf{X}}(e)\downarrow \leftrightarrow\ e \in \mathbb{N} \setminus Y\ \leftrightarrow\ \Phi_e^{\mathbf{X}}(e)\uparrow,$$

which is a contradiction. Therefore, Y is not $\mathbf{X}$-computable, and in particular the complement of Y is not $\mathbf{X}$-computably enumerable. ■

However, one must be a little cautious when relativizing an argument. Indeed, some definitions mask calls to machines, and their definition must therefore also be relativized. For example, if we define the *halting problem* as the set $\{n : \Phi_n(n)\downarrow\}$, the relativization of the statement "The halting problem is not computable" is not "For all X, the halting problem is not X-computable", but "For all X, the halting problem relativized to X is not X-computable", where the "halting problem relativized to X" is in fact defined as the set $\{n : \Phi_n^X(n)\downarrow\}$.

In general, the relativization of a theorem is obtained by adding an oracle X to each machine, and by replacing *computable* by *X-computable*. This is for example the case for a naive relativization of the SMN theorem.

Theorem 3.2 (Relativized SMN theorem - version 1)
For any oracle* X**, *and for all non-zero integers n and m, there exists a total* ***X-computable *function $S_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ such that for all e,*

$$\Phi^{\mathbf{X}}_{S_n^m(e,x_1,\ldots,x_m)}(y_1,\ldots,y_n) = \Phi^{\mathbf{X}}_e(x_1,\ldots,x_m,y_1,\ldots,y_n).$$

An analysis of the proof of the SMN theorem reveals however that the function S_n^m does not depend on the oracle X, because it performs purely syntactic manipulations on the machine. It is therefore possible to formulate a stronger relativized version of the SMN Theorem, where the function S_n^m is computable, although the previous version is also valid.

Theorem 3.3 (Relativized SMN theorem - version 2)
For any oracle* X**, *for any $n, m \in \mathbb{N}^*$ there exists a total* ***computable *function $S_n^m : \mathbb{N}^{m+1} \to \mathbb{N}$ such that for all e,*

$$\Phi^{\mathbf{X}}_{S_n^m(e,x_1,\ldots,x_m)}(y_1,\ldots,y_n) = \Phi^{\mathbf{X}}_e(x_1,\ldots,x_m,y_1,\ldots,y_n).$$

Digression

The relativization of arguments is an empirical phenomenon which in a way reflects our partial understanding of the notion of calculus. However, and especially in complexity theory, proofs not necessarily relativize. The emblematic example is the question of the separation of the complexity classes P and NP. Each of these classes can be relativized to an oracle. Baker, Gill, and Solovay [9] have shown that there are oracles X for which $\mathrm{P}^{\mathrm{X}} = \mathrm{NP}^{\mathrm{X}}$ and others for which $\mathrm{P}^{\mathrm{X}} \neq \mathrm{NP}^{\mathrm{X}}$. In order to solve the question P vs NP, it is then necessary to use an argument which cannot be relativized. This is in computability theory something quite unusual, but not necessarily in complexity theory, where we have other examples of solved problems that do not relativize. We know for example that the IP and PSPACE classes coincide [204], while being able to produce oracles X for which $\mathrm{IP}^{\mathrm{X}} \neq \mathrm{PSPACE}^{\mathrm{X}}$ [63].

4. Use property

Given a computation $\Phi_e(A, n)$ on an oracle A, when $\Phi_e(A, n)\downarrow$, then the functional Φ_e has only used a finite part of the oracle A to return the result: this follows simply from the fact that a computation is always carried out in a finite number of steps. There is therefore an initial segment $\sigma \prec A$ such that for any other oracle B having the same initial segment, $\Phi_e(A, n)$ and $\Phi_e(B, n)$ have exactly the same behavior. This leads us to define the notion of computation with finite oracles.

Notation

Given a finite sequence $\sigma \in 2^{<\mathbb{N}}$, we write $\Phi_e(\sigma, n)\downarrow$ or $\Phi_e^\sigma(n)\downarrow$ to signify that the computation halts on the input n and with σ as a partial oracle, where the oracle has not been accessed outside of its domain.

In the previous notation, if the computation needs to access oracle values that exceed the size of σ, then we write $\Phi_e(\sigma, n)\uparrow$ or $\Phi_e^\sigma(n)\uparrow$. The finite part of the oracle A queried until the computation $\Phi_e(A, n)$ finishes is called the *use* of the computation.

Proposition 4.1 (Use property). Let Φ_e be a Turing functional, X an oracle and $n \in \mathbb{N}$ an input. Then, $\Phi_e(X, n)\downarrow$ if, and only if, there exists a finite prefix $\sigma \prec X$ such that $\Phi_e(\sigma, n)\downarrow$. ⋆

We will see in Section 8-2 that the use property corresponds to the concept of continuity on Cantor space. Despite its conceptual simplicity, the use property plays a primordial role in computability theory. For example, we will make a strong use of it (no pun intended) in Section 8 on the finite extension method.

Definition 4.2. Given a functional Φ and an oracle X, we denote by $\mathrm{use}_\Phi^X : \mathbb{N} \to \mathbb{N}$ the partial X-computable function which on n returns the minimum length of the prefix of X used in the computation of $\Phi(X, n)$. ♢

Remark

According to the use property, any Turing functional Γ can be represented by the c.e. set W of triples (σ, x, y) such that if $(\sigma, x, y) \in W$, then $\Gamma^\sigma(x) \downarrow= y$. Such an enumeration satisfies the following consistency property: for all $(\sigma, x, y) \in W$ and $(\tau, x, y') \in W$ such that $\sigma \prec \tau$, we have $y = y'$.

Exercise 4.3. Show that if Y is X-computable and Z is Y-computable, then Z is X-computable. ⋄

5. Turing degrees

Oracle machines induce a notion of relative computability. Informally, a set Y is X-computable if X is at least as powerful as Y, in the sense that everything that is computable by Y is also computable by X. This gives rise to the *Turing reduction.*

Definition 5.1 (Turing reduction). Given two sets $X, Y \subseteq \mathbb{N}$, we say that X is *Turing reducible* to Y — written $X \leqslant_T Y$ — if X is Y-computable. We write $X <_T Y$ if $X \leqslant_T Y$ but $Y \not\leqslant_T X$. $\diamondsuit$

The Turing reduction forms an *pre-order* on the sets of integers, i.e., this relation is reflexive and transitive. Indeed, if Y is X-computable and Z is Y-computable, Z is X-computable. However, it is not a partial order on the sets, because the Turing reduction is not anti-symmetric. For example, the sets of even numbers and odd numbers are trivially mutually computable since they are both computable, but they are not equal as sets of integers.

It is however possible to transform this pre-order into a partial order by identifying all the mutually computable sets. This gives the notion of *Turing degree* which represents a more robust computational power than the notion of set for the Turing reduction.

Definition 5.2. We write $X \equiv_T Y$ if $X \leqslant_T Y$ and $Y \leqslant_T X$. We will then say that X and Y are *Turing-equivalent.* We call *Turing degrees* the equivalence classes of the relation $\equiv_T$. The Turing degree of a set X is the collection $\deg_T(X) = \{Y : Y \equiv_T X\}$. $\diamondsuit$

By construction, if $X \leqslant_T Y$, then any element in the Turing degree of Y computes any element in the Turing degree of X. The Turing reduction therefore induces a partial order on the Turing degrees, which we will simply note $\leqslant$.

Notation

We denote by $(\boldsymbol{\mathcal{D}}, \leqslant)$ the set of Turing degrees $\boldsymbol{\mathcal{D}}$ partially ordered by $\leqslant$. In general, the letters $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \dots$ will be used to designate its elements. We will sometimes write $X \leqslant_T \mathbf{d}$ for $\deg_T(X) \leqslant \mathbf{d}$.

Exercise 5.3. Show that if $X \equiv_T Y$ and $A \equiv_T B$, then we have $X \leqslant_T A$ iff $Y \leqslant_T B$. $\diamond$

Two sets are therefore Turing-equivalent if they have the same computational power, and the relation $\leqslant$ makes it possible to compare no longer sets, but degrees of computational power. In particular, the Turing degrees are stable under finite variations, in the sense that if $X \in \mathbf{d}$ and $Y =^* X$, then $Y \in \mathbf{d}$. Here, $Y =^* X$ means that X and Y differ only over a finite number of bits.

Exercise 5.4. Show that Turing degrees are stable under finite variation. $\diamond$

A large part of classical computability theory consists in understanding the structure $(\boldsymbol{\mathcal{D}}, \leqslant)$. Is the order $\leqslant$ total over $\boldsymbol{\mathcal{D}}$? Is it well founded? If it is a partial order, what is the maximum size of an anti-chain? We will see that the structure of the Turing degrees is of great richness, but also of great complexity. Let's start with some immediate observations.

First of all, Turing degrees have a minimal element, namely the degree of computable sets. We will note it $\mathbf{0}$. We then have the following proposition.

Proposition 5.5. Any Turing degree is countably infinite. $\star$

Proof. Let $\mathbf{d}$ be a Turing degree and let $X \in \mathbf{d}$. In particular,

$$\mathbf{d} = \deg_T(X) = \{Y : X \equiv_T Y\} \subseteq \big\{\{n : \Phi_e^X(n)\downarrow = 1\} : e \in \mathbb{N}\big\},$$

therefore $\mathbf{d}$ is countable. Moreover, $\mathbf{d}$ is stable under finite variations, therefore is infinite. Thus, $\mathbf{d}$ is countably infinite. $\blacksquare$

The collection of subsets of $\mathbb{N}$ being uncountable, and each belonging to a Turing degree, it follows from the previous proposition that there are uncountably many Turing degrees. Suppose indeed the opposite. Then, according to Exercise 2-3.11, the union of all Turing degrees would be a countable set, as a countable union of countable sets is countable. As this union is equal to $2^{\mathbb{N}}$, there we obtain a contradiction. Note that Exercise 2 -3.11 uses the axiom of choice (which we will discuss in detail in Section 9 -4). The use of the axiom of choice is however not necessary: we will see several effective constructions of functions $f : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ (see Exercise 8-5.4 or Exercise 8-5.3) such that $f(X)$ and $f(Y)$ are in degrees Turing different for all $X \neq Y$, which makes f an injection of $2^{\mathbb{N}}$ into Turing degrees and in particular shows $|2^{\mathbb{N}}| \leqslant |\boldsymbol{\mathcal{D}}|$[1].

Definition 5.6. The *effective join* of two sets A and B is the set $A \oplus B = \{2n : n \in A\} \cup \{2n+1 : n \in B\}$ (note that the intersection between the two sets is disjoint). $\diamondsuit$

The effective join of two sets is a way of encoding the information of each set so as to be able to decode it computably. There are of course many

[1]The inequality $|\boldsymbol{\mathcal{D}}| \leqslant |2^{\mathbb{N}}|$ seems obvious, but it uses the axiom of choice, in order to uniformly select an element in each Turing degree. We will discuss this briefly in Section 12-2.3.

ways to encode two sets into one, the effective join being the most direct and efficient, in the following sense.

Proposition 5.7. Let A and B be two sets. Then, $\deg_T(A \oplus B)$ is the least upper bound of the degrees $\deg_T(A)$ and $\deg_T(B)$, i.e., any degree above $\deg_T(A)$ and $\deg_T(B)$ is also above $\deg_T(A \oplus B)$. Thus, any pair of Turing degrees $\mathbf{c}$ and $\mathbf{d}$ admits an upper bound which we will denote by $\mathbf{c} \cup \mathbf{d}$. ⋆

Proof. It is clear that the join $A \oplus B$ allows to compute A and B. Thus, $\deg_T(A \oplus B)$ is an upper bound of $\deg_T(A)$ and $\deg_T(B)$. Let $\deg_T(C)$ be an upper bound of $\deg_T(A)$ and $\deg_T(B)$. In particular, $C \geqslant_T A$ and $C \geqslant_T B$. Let i and j be codes such that $\Phi_i(C, n) = A(n)$ and $\Phi_j(C, n) = B(n)$ for all n.

The function $f : \mathbb{N} \to \{0, 1\}$ defined by

$$f(n) = \begin{cases} \Phi_i(C, n/2) & \text{if n is even} \\ \Phi_j(C, (n-1)/2) & \text{otherwise} \end{cases}$$

is C-computable, and we have $f(n) = 1$ iff $n \in A \oplus B$ for any integer n. Then, $C \geqslant_T A \oplus B$. ■

Notation

In the proof of the previous proposition, we made use of the predicate "$\Phi_i(C, n) \downarrow= A(n)$" for all n. Sometimes we will use the shorter notation $\Phi_i(C) = A$.

Beware, the least upper bound $\mathbf{c} \cup \mathbf{d}$ of $\mathbf{c}$ and $\mathbf{d}$ has of course nothing to do with the set-theoretic union of the degrees $\mathbf{c}$ and $\mathbf{d}$. In general, the letters in bold type $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \dots$ will be used to talk about degrees as abstract objects within a partial order, the detail of the sets of integers constituting each degree then not being relevant.

We will now be juggling between sets of integers and Turing degrees. Since each Turing degree can be represented by a set of integers (one of its members), an operation on the Turing degrees will normally be done via an operation on one of its representatives, so that the expected result is independent of the choice of such representative. The effective join constitutes a first example illustrating our point.

Exercise 5.8. Show that if we assume that $X \leqslant_T Y$ and $A \leqslant_T B$, then

$$X \oplus A \leqslant_T Y \oplus B.$$

◇

The previous exercise shows that the effective join induces an operation on Turing degrees. We will study in the next section a new operation on degrees playing an essential role in computability theory: the *Turing jump*.

6. Turing jump

The Turing jump is a fundamental operation in computability theory, and is defined as the relativization of the halting problem.

Definition 6.1. Given a set X, we define

$$X' = \{n : \Phi_n^X(n)\downarrow\}.$$

The *Turing jump* is the $X \mapsto X'$ operator. ♢

For example, we can now define the halting problem relative to the halting problem: the set of computer program codes that halt on their own input, but using the halting problem as an oracle. We denote it $\emptyset''$. It is not very difficult to show that the Turing jump induces an operation on Turing degrees, and we leave the proof as an exercise.

Exercise 6.2. **($\star$)** Show that if $X \leqslant_T Y$, then $X' \leqslant_T Y'$. ⋄

We will see a reinforcement of the result of the previous exercise with Exercise 5-5.7. We will therefore denote by $\mathbf{d}'$ the Turing jump of a Turing degree $\mathbf{d}$. In particular, $\mathbf{0}'$ is the Turing degree of the halting problem.

Proposition 6.3. We have $X <_T X'$ for all $X \in 2^{\mathbb{N}}$. Thus we have $\mathbf{d} < \mathbf{d}'$ for any Turing degree $\mathbf{d}$. ⋆

Proof. Let us first show that X' computes X. Let $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ be the partial X-computable function defined by

$$f(e, n) = \begin{cases} 1 & \text{if } e \in X \\ \uparrow & \text{otherwise.} \end{cases}$$

By the SMN theorem relativized to X (see Theorem 3.3), there exists a total computable function $g : \mathbb{N} \to \mathbb{N}$ such that for all integers e and n, we have $\Phi_{g(e)}^X(n) = f(e, n)$. Thus, for all e:

- if $e \in X$, then $\Phi_{g(e)}^X$ is the constant function 1, and therefore $g(e) \in X'$;
- if $e \notin X$, then $\Phi_{g(e)}^X$ is the nowhere defined function, and $g(e) \notin X'$.

In particular, X' can compute X: to know if $n \in X$, it suffices to look at whether $g(n) \in X'$.

The proof that $X \not\geqslant_T X'$ is a relativization of the fact that $\emptyset'$ is not computable, which was previously demonstrated with Theorem 3.1. ■

There is therefore a strictly increasing hierarchy of Turing degrees:

$$\mathbf{0} < \mathbf{0}' < \mathbf{0}'' < \dots$$

In the previous proof, note that the Turing functional used to compute X from X' is *the same* for any oracle X. This is something that will be used from time to time, for example in Chapter 26. We give the unconvinced reader the opportunity to reflect on this in the following exercise.

Exercise 6.4. Show that there exists a functional Φ_e such that:

- $\Phi_e(X', n) = X(n)$ for all $X \in 2^{\mathbb{N}}$ and for all $n \in \mathbb{N}$.
- $\Phi_e(Y, n)\downarrow$ for all $Y \in 2^{\mathbb{N}}$ and for all $n \in \mathbb{N}$. $\diamond$

Finally, note that the Turing jump is not an injective operator, as we will see later through the low and high sets. We will occasionally use the following notion of Turing-completeness.

Definition 6.5. A set A is said to be *Turing-complete* or (simply) *complete* if $A \geqslant_T \emptyset'$. A *complete* Turing degree is a degree $\mathbf{d} \geqslant \mathbf{0}'$. A set or degree which is not complete is *incomplete*. $\diamondsuit$

7. Limit computability

We are now going to study certain properties of sets computable by the halting problem. These sets admit in particular a very natural characterization in terms of approximations.

Definition 7.1. A function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is *stable* if for all $x \in \mathbb{N}$, $\lim_y f(x, y)$ exists, that is, for every $x \in \mathbb{N}$, there is some threshold $t \in \mathbb{N}$ such that for every $y > t$, $f(x, y) = f(x, t)$. A set A is *limit-computable* if there exists a computable stable function $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ such that for all x

$$\lim_y f(x, y) = 1 \text{ iff } x \in A$$

$\diamondsuit$

We obtain our first characterization of the $\emptyset'$-computable sets.

Lemma 7.2 (Shoenfield limit lemma). A set $A \subseteq \mathbb{N}$ is $\emptyset'$-computable if, and only if, it is limit-computable. $\star$

Proof. We can refer to figures 7.4 and 7.3 to help us in understanding the proof.

$\Rightarrow$. Suppose that $A \leqslant_T \emptyset'$ via a functional Φ_e. Let $\emptyset'_0 \subseteq \emptyset'_1 \subseteq \dots$ be a c.e. approximation of $\emptyset'$. Let $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ be the function which, for an input (x, s), checks if $\Phi_e(\emptyset'_s, x)[s]\downarrow$. If so, $f(x, s) = \Phi_e(\emptyset'_s, x)[s]$. Otherwise, $f(x, s)$ is assigned an arbitrary value.

Let us show that f is stable and that its limit is A. Let $x \in \mathbb{N}$. By the use property, as $\Phi_e(\emptyset', x)\downarrow$, this computation is done using the n first bits of the oracle for some integer n. Let then s be large enough so that $\emptyset'_s \upharpoonright_n = \emptyset' \upharpoonright_n$ and so that $\Phi_e(\emptyset', x)\downarrow$ halts before s stages of computation (any sufficiently large s works). Then, for all $t \geqslant s$,

$$\Phi_e(\emptyset'_t, x)[t]\downarrow = \Phi_e(\emptyset', x),$$

in which case $\lim_t f(x, t) = \Phi_e(\emptyset', x) = A(x)$.

$\Leftarrow$. Let us now suppose that A is limit-computable, by a computable stable function $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$. Then,

$$A = \{x : \exists y \forall z \geqslant y\ f(x, z) = 1\} \quad \text{and} \quad \overline{A} = \{x : \exists y \forall z \geqslant y\ f(x, z) = 0\}.$$

We will define a $\emptyset'$-computable procedure to determine if $x \in A$ or $x \in \overline{A}$. Let $u, v : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be two total computable functions such that for all x, y, n,

$$\Phi_{u(x,y)}(n) = \begin{cases} 1 & \text{if } \exists z \geqslant y\ f(x, z) \neq 1 \\ \uparrow & \text{otherwise,} \end{cases}$$

$$\Phi_{v(x,y)}(n) = \begin{cases} 1 & \text{if } \exists z \geqslant y\ f(x, z) \neq 0 \\ \uparrow & \text{otherwise.} \end{cases}$$

Specifically,

$$u(x, y) \notin \emptyset' \text{ iff } \forall z \geqslant y\, f(x, z) = 1 \quad \text{et} \quad v(x, y) \notin \emptyset' \text{ iff } \forall z \geqslant y\, f(x, z) = 0.$$

Thereby,

$$A = \{x : \exists y\ u(x, y) \notin \emptyset'\} \quad \text{and} \quad \overline{A} = \{x : \exists y\ v(x, y) \notin \emptyset'\}.$$

Ǵiven an integer x, to know if $x \in A$ or $x \in \overline{A}$, it suffices to look for the smallest y such that $u(x, y) \notin \emptyset'$ or $v(x, y) \notin \emptyset'$. We will necessarily end up finding such an integer y. If $u(x, y) \notin \emptyset'$, then $x \in A$. If $v(x, y) \notin \emptyset'$, then $x \notin A$. The procedure is $\emptyset'$-computable, and we thus have $A \leqslant_T \emptyset'$. ■

Exercise 7.5. Show that if Y is X-c.e. and Z is Y-c.e., then Z is not necessarily X-c.e. ◇

Any $\emptyset'$-computable set A is therefore associated with a stable function f whose limit is A. This function is often presented in the form of a succession of *uniformly computable* sets $A_0, A_1, \ldots$ defined by $A_y = \{x : f(x, y) = 1\}$ for all y.

$$\begin{tikzcd}
\text{Approximation of the bit } x & \text{Machine codes} \\
0 \arrow[r, dotted] & u(x,0) \text{ and } v(x,0) \\
1 \arrow[r, dotted] & u(x,1) \text{ and } v(x,1) \\
0 \arrow[r, dotted] & u(x,2) \text{ and } v(x,2) \\
0 \arrow[r, dotted] & u(x,3) \text{ and } v(x,3) \\
1 \arrow[r, dotted] & u(x,4) \text{ and } v(x,4) \\
0 \arrow[r, dotted] & u(x,5) \text{ and } v(x,5) \\
0 \arrow[r, dotted] & u(x,6) \text{ and } v(x,6)
\end{tikzcd}$$

$\dots$

Figure 7.3: Illustration of the reciprocal ($\Leftarrow$) of the proof of Shoenfield's lemma : to approximate the bit x at stage y, one create the code $u(x, y)$ of the program which halts everywhere if a value different from 1 occurs for x at a stage later than y, and the code $v(x, y)$ of the program which halts everywhere if a value different from 0 occurs for x at a stage later than y.

Uniform computability

We have introduced the concept of *uniformly computable* sequence $A_0, A_1, \ldots$: each element A_n of the sequence is computable by *the same function*, parameterized by an additional parameter n which indicates that the n-th element of the sequence is computed.
Let us insist on the fact that a sequence of computable sets $(X_i)_{i\in\mathbb{N}}$ is not necessarily uniformly computable: there does not necessarily exist an algorithm allowing to compute X_i as a function of i. As a trivial example, each X_i can simply be an infinite sequence of 0, except for the bit in position i which is equal to the i-th bit of $\emptyset'$. Each X_i is a finite set and is therefore computable. On the other hand, an algorithm allowing to compute *uniformly* X_i as a function of i would make it possible to compute the halting set, and therefore cannot exist.

Definition 7.6. Let $A \leqslant_T \emptyset'$ be a set. A Δ^0_2 *approximation* of A is a uniformly computable sequence of sets $A_0, A_1, \ldots$ such that for all x, $\lim_y A_y(x)$ exists and is equal to $A(x)$. $\diamondsuit$

The Δ^0_2 approximations are not canonical. It is always possible for example to "accelerate" a Δ^0_2 approximation $A_0, A_1, \ldots$ by considering the sequence $A_{g(0)}, A_{g(1)}, \ldots$ for a computable and strictly increasing function $g : \mathbb{N} \to \mathbb{N}$.

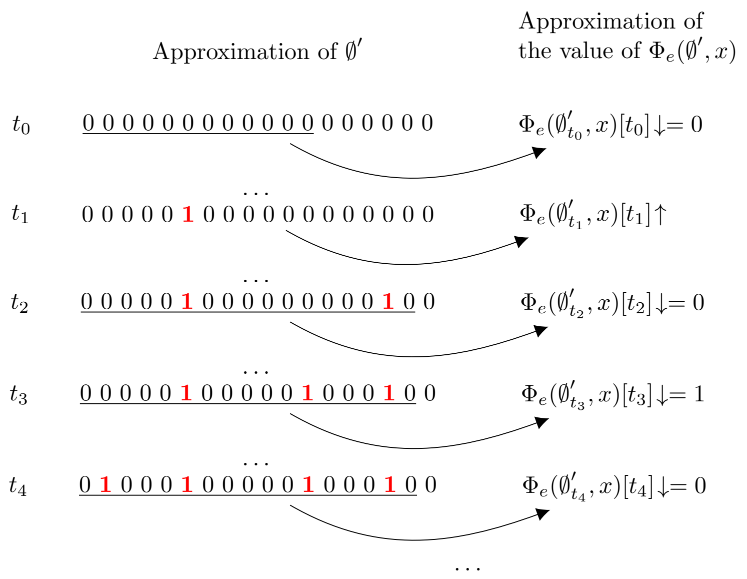

Approximation of $\emptyset'$
Approximation of the value of $\Phi_e(\emptyset', x)$
t_0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$\Phi_e(\emptyset'_{t_0}, x)[t_0]\downarrow= 0$
t_1
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
$\Phi_e(\emptyset'_{t_1}, x)[t_1]\uparrow$
t_2
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
$\Phi_e(\emptyset'_{t_2}, x)[t_2]\downarrow= 0$
t_3
0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0
$\Phi_e(\emptyset'_{t_3}, x)[t_3]\downarrow= 1$
t_4
0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0
$\Phi_e(\emptyset'_{t_4}, x)[t_4]\downarrow= 0$

Figure 7.4: Illustration of the forward direction ($\Rightarrow$) of the proof of Shoenfiel's lemma. The first column represents successive approximations of $\emptyset'$. The underlined part represents the use of the computation $\Phi_e(\emptyset', x)[t_n]$ when it halts. While we enumerate the ith element in the halting set at stage t_i, one launches the computation of Φ_e with the current approximation of the halting set as oracle, for t_i steps of computation. By the use property, the process converges necessarily.

Remark

The name "Δ^0_2 approximation" will be justified in Chapter 5, where we will give a new characterization of $\emptyset'$-computable sets as those definable by a Δ^0_2 predicate.

The Δ^0_2 approximations make it possible to define two important functions, namely the modulus and the computation function. These functions express the computational complexity of the set A in the form of growth rate: any function increasing faster than the modulus of A or than its computation function allows to recompute A.

Definition 7.7. Let $A_0, A_1, A_2, \dots$ be a Δ^0_2 approximation of a set A.

1. The *modulus* of the Δ^0_2 approximation is the function $\mu_A : \mathbb{N} \to \mathbb{N}$ which to x associates the smallest integer n such that the se-

quence $A_n\restriction_x, A_{n+1}\restriction_x, \ldots$ is constant.

2. The *computation function* of the Δ^0_2 approximation is the function $c_A : \mathbb{N} \to \mathbb{N}$ which to x associates the smallest integer $n \geqslant x$ such that $A_n\restriction_x = A\restriction_x$. ♢

Note that any c.e. approximation is a degenerate Δ^0_2 approximation. In particular, every c.e. set is $\emptyset'$-computable. Unlike c.e. approximations which, when they cause an element to appear in A, never remove it again, a Δ^0_2 approximation of A has the right to “change its mind” an arbitrarily large (but finite) number of times, whether x belongs to A or not. In particular, the computation function generally grows slower than the modulus, because a set A_n can coincide with the set A on a long initial segment without having reached its threshold of stability on this segment.. The computation function, unlike modulus in general, is computable by the set A.

It is easy to verify that any function dominating the modulus of a Δ^0_2 approximation of a set computes this set.

Exercise 7.8. Let $A_0, A_1, \ldots$ be a Δ^0_2 approximation of a set A. Let μ_A be its modulus. Show that any function dominating μ_A computes A. A function f *dominates* a function g if $f(n) \geqslant g(n)$ for all $n \in \mathbb{N}$. ◇

However, it is not clear that this is also the case with the computation function. This is however what the following proposition shows.

Proposition 7.9 (Martin and Miller [162]). Let $A_0, A_1, \ldots$ be a Δ^0_2 approximation of a set A. Let c_A be its computation function. Any function dominating c_A computes A. ⋆

Proof. Let f be a function dominating c_A. Let $M(x)$ be the largest $y \leqslant x$ such that for all $x \leqslant t \leqslant f(x)$, $A_t\restriction_y = A_{f(x)}\restriction_y$. The function M is total f-computable. Moreover, M tends towards $+\infty$, because the approximation of A being Δ^0_2, it will stabilize on increasingly larger initial segments. Finally, as $x \leqslant c_A(x) \leqslant f(x)$, then if $M(x) = y$, $A_x\restriction_y = A_{c_A(x)}\restriction_y = A\restriction_y$. Then, to decide if $n \in A$, it suffices to find an integer x such that $M(x) > n$, then test if $n \in A_x$. This procedure is f-computable. ■

Note that it is important to ask that $c_A(x) \geqslant x$ in the definition of the computation function. The preceding proposition becomes false when this precision is omitted.

Remark

The notions of modulus and computation function are not characteristic of a $\emptyset'$-computable set, but of a Δ^0_2 approximation of a set.

The same $\emptyset'$-computable set has an infinity of Δ^0_2 approximations, each having its modulus and its computation function.

8. Finite extensions method

We are now going to present a relatively simple and yet very powerful method, allowing to create sets satisfying "custom" computational properties. This is the *finite extension method.*

In computability theory, we call *weakness property* a property on sets, which is downward-closed under the Turing reduction, i.e., if X has a weakness property and if $Y \leqslant_T X$, then Y also has this weakness property. Conversely, a *strength property* is a property on sets, which upward-closed under the Turing reduction, that is, if X has a strength property and if $X \leqslant_T Y$, then Y also has this strength property.

The finite extension method is particularly suitable when we ask ourselves the question of the existence of sets simultaneously satisfying some strength and some weakness properties. It is then necessary to create a made-to-measure set, neither too strong nor too weak from a computational point of view. We will illustrate this method by proving two propositions.

Proposition 8.1 (Kleene and Post, 1954). There are two sets A and B which are incomparable by the Turing reduction. ⋆

Proof. We are going to build simultaneously two sets A and B, seen as infinite binary sequences — remember the correspondence between the two.

The set A must satisfy a strength property (A is not computable by B) and a weakness property (A does not compute B). The set B must for its part satisfy the dual properties of strength and weakness.

Requirements. Computational properties, whether of strength or weakness, are in general schemes of properties, in the sense that they are declined in an infinity of more elementary properties and easier to satisfy independently. For example, the property "$A \not\geqslant_T B$" corresponds to the collection of properties "$\Phi_e^A \neq B$" for any functional code e, where "$\Phi_e^A \neq B$" means that either Φ_e^A is a partial function, or $\Phi_e^A(x) \downarrow \neq B(x)$ for an $x \in \mathbb{N}$. These elementary properties are called *requirements.* The first step in a construction using the finite extension method consists in identifying the requirements. We therefore have two kinds $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$:

$$\begin{aligned} \mathcal{R}_e \quad &: \quad \exists x\ \Phi_e^A(x)\uparrow\ \vee \exists x\ \Phi_e^A(x)\downarrow \neq B(x) \\ \mathcal{S}_e \quad &: \quad \exists x\ \Phi_e^B(x)\uparrow\ \vee \exists x\ \Phi_e^B(x)\downarrow \neq A(x). \end{aligned}$$

If all the $\mathcal{R}$-requirements are satisfied, then $A \not\geq_T B$, while if all the $\mathcal{S}$-requirements are satisfied, $B \not\geq_T A$. Since we are only interested in functionals computing elements of $2^{\mathbb{N}}$, we will consider —as often in this kind of case— that $\Phi_e(X,n)\uparrow$ if ever $\Phi_e(X,n)\downarrow\notin \{0,1\}$.

Satisfaction of a requirement. Suppose we want to satisfy only one requirement, say $\mathcal{R}_e$. Two cases arise.

- Case 1: there exists a set X such that $\Phi_e^X(0)\downarrow= i$ for a given $i \in \{0,1\}$. We can then fix $A = X$ and B can be any set such that $B(0) \neq i$. We will then have satisfied the requirement $\mathcal{R}_e$ by ensuring that $\Phi_e^A(0)\downarrow\neq B(0)$.
- Case 2: whatever the set X, we have $\Phi_e^X(0)\uparrow$. This case is even simpler, A and B can be any set.

The previous argument fully specified the sets A and B in order to satisfy a requirement $\mathcal{R}_e$, without leaving freedom for the other requirements to be satisfied. We will therefore try to be more economical to leave room for the satisfaction of other requirements. For that, we will make sure to specify only a finite initial segment of A and B to satisfy a given requirement.

Construction. The sets A and B will be built by finite approximations, in the form of two sequences of finite binary strings, representing increasingly long prefixes of A and B.

$$\sigma_0 \preceq \sigma_1 \preceq \dots \quad \text{and} \quad \tau_0 \preceq \tau_1 \preceq \dots$$

We do not necessarily ask that $\sigma_n \prec \sigma_{n+1}$: the sequence of strings may stagnate for a certain time. On the other hand, for all n, we ask that for $m > n$ sufficiently large we have $\sigma_n \prec \sigma_m$. In this way, the strings $\sigma_0 \preceq \sigma_1 \preceq \dots$ gradually converge towards a single infinite sequence A and the strings $\tau_0 \preceq \tau_1 \preceq \dots$ gradually converge towards a single infinite sequence B. Formally, we define A and B via new notations: given a binary string σ, we will denote by $[\sigma]$ the set of infinite binary sequences having σ as a prefix. We will therefore have $A \in [\sigma_n]$ and $B \in [\tau_n]$ for all $n \in \mathbb{N}$. By making sure that there are arbitrarily long strings, we make $\bigcap_n[\sigma_n]$ and $\bigcap_n[\tau_n]$ each contain exactly one element: A and B respectively.

The finite approximations of A and B will be defined in stages, so as to successively satisfy each requirement. As at a given step, only finite prefixes of A and B are known, it will therefore be necessary to satisfy a requirement whatever comes after these prefixes in the rest of the construction. A pair of strings σ_n and τ_n *force* a requirement $\mathcal{R}_e$ (or a requirement $\mathcal{S}_e$) if the requirement property is satisfied for all $A \succeq \sigma_n$ and $B \succeq \tau_n$. We must therefore ensure that the requirement is satisfied for all the elements of $[\sigma_n]$

and $[\tau_n]$. Note in passing that if σ_n and τ_n force a requirement, then for all $\sigma \succeq \sigma_n$ and $\tau \succeq \tau_n$ we have $[\sigma] \subseteq [\sigma_n]$ and $[\tau] \subseteq [\tau_n]$, and therefore σ and τ still force the requirement.

The different requirements are going to be intertwined so that each receives attention at a stage of the construction. During an even step $n = 2e$, we will define σ_{n+1} and τ_{n+1} so as to force $\mathcal{R}_e$, while during an odd step, $n = 2e+1$, we will force $\mathcal{S}_e$. The requirements will therefore be satisfied in the order

$$\mathcal{R}_0, \mathcal{S}_0, \mathcal{R}_1, \mathcal{S}_1, \mathcal{R}_2, \mathcal{S}_2, \ldots$$

Satisfaction of a requirement. We are now going to satisfy the requirement $\mathcal{R}_e$ again, this time specifying only a finite prefix of the oracles A and B. The case of $\mathcal{S}_e$ requirements is symmetrical. Suppose that initial segments σ_n and τ_n have already been specified for A and B (we start the construction with $\sigma_0 = \tau_0 = \epsilon$ where ϵ is the empty word). In other words, the final infinite binary sequences A and B must respect $\sigma_n \preceq A$ and $\tau_n \preceq B$. Let $x = |\tau_n|$. In particular, x is the first position where B is not yet specified. All the values of B in the positions preceding x are already set by τ_n. The following two cases arise.

- Case 1: there exists a set $X \succeq \sigma_n$, such that $\Phi_e^X(x)\downarrow = i$ for a given $i \in \{0, 1\}$. In this case, by the use property, this computation calls upon a finite number of bits of the oracle, and there thus exists an initial segment $\sigma_{n+1} \preceq X$ such that $\Phi_e^{\sigma_{n+1}}(x)\downarrow = i$, or in other words such that $\Phi_e^Y(x)\downarrow = i$ for any set $Y \succeq \sigma_{n+1}$. We can choose the initial segment σ_{n+1} so that it is at least as long as σ_n, which ensures $\sigma_{n+1} \succeq \sigma_n$. Knowing that the set A will have σ_{n+1} for initial segment, we ensured that $\Phi_e^A(x)\downarrow = i$. Let τ_{n+1} be the string obtained from τ_n by adding the bit $1 - i$ to it. In other words, $|\tau_{n+1}| = |\tau_n| + 1$, $\tau_{n+1} \succeq \tau_n$ and $\tau_{n+1}(x) = 1 - i$. We then ensured that for all $B \succeq \tau_{n+1}$, $B(x) = 1 - i$. Thus, the strings σ_{n+1} and τ_{n+1} extend σ_n and τ_n respectively, and force the requirement $\mathcal{R}_e$ by making sure that $\Phi_e^A(x)\downarrow \neq B(x)$ for all $A \succeq \sigma_n$ and $B \succeq \tau_n$.
- Case 2: for any set $X \succeq \sigma_n$, we have $\Phi_e^X(x)\uparrow$. In this case, σ_n and τ_n already force the requirement $\mathcal{R}_e$ by making sure $\Phi_e^A(x)\uparrow$ for a certain x. It is therefore sufficient to take $\sigma_{n+1} = \sigma_n$ and $\tau_{n+1} = \tau_n$.

We will ensure that the lengths of the elements of the sequences $(\sigma_n)_{n\in\mathbb{N}}$ and $(\tau_n)_{n\in\mathbb{N}}$ are increasingly large, by adding an arbitrary bit at the end of each string at the end of each step, before moving on to next step. As explained previously, forcing a requirement is closed under strings extension, so this will not alter the validity of the construction. The proof of Proposition 8.1 is complete. ■

A new notation has been introduced in the previous proof; we oficialize its use below.

Notation

Given $\sigma \in 2^{<\mathbb{N}}$, we denote by $[\sigma]$ the set of $X \in 2^{\mathbb{N}}$ such that $\sigma \prec X$.

We will give another illustration of the finite extension method by proving a stronger proposition, implying Proposition 8.1. This time, we fix an arbitrary non-computable set A. We then construct a set B that A does not compute, and which does not compute A. This is a priori more difficult construction, because only one of the two sets is controlled. Note also that it will be necessary for this proof to use the fact that A is non-computable: indeed, if it were computable, it would be in particular computable from any set B.

Proposition 8.2. For any non-computable set A, there exists a set B such that $B \not\leq_T A$ and $A \not\leq_T B$. ⋆

Proof. Unlike Proposition 8.1, the set A is already fixed. We are going to build only the set B by the finite approximation method. The set B must still satisfy a strength property ($B \not\leq_T A$) and a weakness property ($A \not\leq_T B$). The requirements are therefore identical to those of Proposition 8.1, namely

$$\begin{aligned} \mathcal{R}_e &: \quad \exists x \Phi_e^A(x)\uparrow \vee \exists x \Phi_e^A(x)\downarrow \neq B(x) \\ \mathcal{S}_e &: \quad \exists x \Phi_e^B(x)\uparrow \vee \exists x \Phi_e^B(x)\downarrow \neq A(x). \end{aligned}$$

However, the sets A and B no longer playing a symmetrical role, the requirements $\mathcal{R}_e$ and $\mathcal{S}_e$ will each be satisfied in their own way.

Construction. The set B will be built by successive approximations

$$\tau_0 \preceq \tau_1 \preceq \tau_2 \preceq \dots$$

to define B as the only element of the set $\bigcap_n [\tau_n]$. A string τ_n *forces* a requirement $\mathcal{R}_e$ (or a requirement $\mathcal{S}_e$) if the property is satisfied for all $X \in [\tau_n]$. At each stage of the construction, a requirement will be forced, by interlacing them as before:

$$\mathcal{R}_0, \mathcal{S}_0, \mathcal{R}_1, \mathcal{S}_1, \mathcal{R}_2, \mathcal{S}_2, \dots$$

Satisfaction of a requirement $\mathcal{R}_e$. At step n, assume that the string τ_n is defined. We want to find an extension $\tau_{n+1} \succeq \tau_n$ forcing the requirement $\mathcal{R}_e$. The satisfaction of this type of requirement is very similar to that of Proposition 8.1. Let $x = |\tau_n|$. Two cases arise.

- Case 1: $\Phi_e^A(x)\downarrow= i$ for an $i \in \{0,1\}$. It is then enough to define τ_{n+1} as the unique string of length $|\tau_n| + 1$ extending τ_n such that
$$\tau_{n+1}(x) = 1 - i.$$
As $B \in [\tau_{n+1}]$, $B(x) = \tau_{n+1}(x) = 1 - i$, so $\Phi_e^A(x)\downarrow\neq B(x)$.
- Case 2: $\Phi_e^A(x)\uparrow$. In this case, the requirement $\mathcal{R}_e$ is trivially satisfied, because Φ_e^A is a partial function. It is therefore sufficient to take $\tau_{n+1} = \tau_n$.

Note that no assumptions were made on the set A to satisfy the requirements $\mathcal{R}_e$. The assumption according to which A is not computable will be exploited to satisfy the requirements $\mathcal{S}_e$.

Satisfaction of a requirement $\mathcal{S}_e$. The difficulty in satisfying a requirement such as $\mathcal{S}_e$ comes from the fact that we have no control over the set A, which is fully specified. More precisely, during the satisfaction of a requirement $\mathcal{R}_e$, we fix an input x which is not yet specified for B, so as to choose its value in case 1 to make it different from the value of $\Phi_e^A(x)$. This is not possible in the case of the $\mathcal{S}_e$ requirement, all the values of A being fixed. It will therefore be necessary to exploit the fact that the set A is not computable. Note on the other hand that the satisfaction of the requirements $\mathcal{R}_e$ left a certain freedom, in particular in the choice of x. Indeed, only a finite number of entries is specified at a given step for B, and therefore almost all entries can be chosen for x. We will exploit this freedom of choice to satisfy the requirements $\mathcal{S}_e$. Three cases arise.

- Case 1: there is an input x and a set $X \succeq \tau_n$ such that
$$\Phi_e^X(x)\downarrow\neq A(x).$$
By the use property, then there exists a finite string $\tau_{n+1} \succeq \tau_n$ such that $\Phi_e^X(x)\downarrow\neq A(x)$ for all $X \in [\tau_n]$. The string τ_{n+1} therefore forces the requirement $\mathcal{S}_e$.
- Case 2: there is an input x such that for all sets $X \succeq \tau_n$, we have $\Phi_e^X(x)\uparrow$. In this case, the string τ_n already forces the requirement $\mathcal{S}_e$ ensuring that $\Phi_e^B(x)\uparrow$.
- Case 3: neither of the two previous cases occurs. We will then show that it is possible to compute the set A, and therefore to deduce a contradiction from it.
Here is the procedure to compute the value of $A(x)$: find a finite string $\tau \succeq \tau_n$ such that $\Phi_e^\tau(x)\downarrow$ and return the result of this computation. We claim the following two facts:
    (1) there is such a string, so the search will end,

(2) whatever τ such that $\Phi_e^\tau(x)\downarrow$, then $\Phi_e^\tau(x)\downarrow = A(x)$.

To show (1), notice that the negation of case 2 means that for any x (and in particular for this x considered), there exists a set $X \succeq \tau_n$ such that $\Phi_e^X(x)\downarrow$. By the use property, there then exists an initial segment $\tau \succeq \tau_n$ of X such that $\Phi_e^\tau(x)\downarrow$.

Let us show (2). If $\Phi_e^\tau(x)\downarrow \neq A(x)$, then case 1 would be true taking any $X \succeq \tau$. It follows that $\Phi_e^\tau(x)\downarrow$ implies $\Phi_e^\tau(x) = A(x)$. We have therefore described a computable procedure to determine the value of $A(x)$ whatever x, contradicting the hypothesis according to which A is not computable.

This concludes the proof of Proposition 8.2. ■

Before concluding this section dedicated to the finite extension method, let us mention that in general, this method is not *effective*, in the sense that no computability constraint is imposed on the finite strings under construction. It is however possible to do a fine analysis of the argument to determine the computational power necessary to find a τ_{n+1}, given τ_n. We then obtain upper bounds on the complexity of the constructed set.

9. Low degrees

As we have seen, the Turing jump is invariant by Turing degree. Thus, for any computable set X, $X' \equiv_T \emptyset'$. It is natural to wonder if only computable sets have a Turing jump equivalent to $\emptyset'$, and more generally if the Turing jump is an injective function on Turing degrees. The following proposition shows that this is not the case.

Proposition 9.1. There exists a non-computable set A such that

$$A' \equiv_T \emptyset'.$$ ⋆

Proof. The set A will be built using an effective version of the finite extension method (see Section 8), making sure that the entire construction is computable in $\emptyset'$.

Requirements. The set A must satisfy a strength property (A non-computable) and a weakness property ($A' \leqslant_T \emptyset'$).

The strength property comes in an infinity of requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$:

$$\mathcal{R}_e : \exists x \Phi_e(x)\uparrow \ \vee \ \exists x \Phi_e(x)\downarrow \neq A(x).$$

The weakness property is of a new type. To satisfy it, we will make sure to "control" the Turing jump of A as the construction progresses, while making sure that the whole construction itself is computable in $\emptyset'$. With the help of this fact, we will have a $\emptyset'$-computable function f for which we will have to satisfy an infinity of requirements $(\mathcal{S}_e)_{e\in\mathbb{N}}$:

$$\mathcal{S}_e : \quad \Phi_e^A(e)\downarrow \to f(e) = 1 \text{ and } \Phi_e^A(e)\uparrow \to f(e) = 0.$$

Informally, the requirement $\mathcal{S}_e$ is satisfied if, at a finite moment of the construction, we know whether $\Phi_e^A(e)\downarrow$ or $\Phi_e^A(e)\uparrow$, whatever the continuation of the construction.

Construction. The set A will be built by successive approximations

$$\sigma_0 \preceq \sigma_1 \preceq \sigma_2 \preceq \dots$$

to define A as the only element of $\bigcap_n [\sigma_n]$. A string σ_n *forces* a requirement $\mathcal{R}_e$ or $\mathcal{S}_e$ if the property is satisfied for all $B \in [\sigma_n]$. At each stage of the construction, a requirement will be forced, by interlacing them as before:

$$\mathcal{R}_0, \mathcal{S}_0, \mathcal{R}_1, \mathcal{S}_1, \mathcal{R}_2, \mathcal{S}_2, \dots$$

We must also make sure that the construction is computable in $\emptyset'$. Thus, to know the value of $A'(e)$, it will suffice to execute using $\emptyset'$ the construction up to step $2e$ satisfying $\mathcal{S}_e$, and return the result. This $\emptyset'$-computable procedure ensures that $A' \leqslant_T \emptyset'$. We will therefore show how to satisfy each type of requirement independently, while analyzing the computational complexity of each step to ensure that σ_{n+1} can be obtained from σ_n using the oracle $\emptyset'$.

Satisfaction of a requirement $\mathcal{R}_e$. Suppose σ_n is already defined. We want to find $\sigma_{n+1} \succeq \sigma_n$ forcing the requirement $\mathcal{R}_e$. Let $x = |\sigma_n|$. In other words, x is the first value of A that is not yet specified. Two cases arise.

- Case 1: $\Phi_e(x) \uparrow$ in which case $\mathcal{R}_e$ is trivially satisfied, and we can define $\sigma_{n+1} = \sigma_n$
- Case 2: $\Phi_e(x) \downarrow$, in which case the string σ_{n+1} obtained from σ_n by adding the bit $1 - \Phi_e(x)$ force $\mathcal{R}_e$.

Depending on the case, we will therefore define a different σ_{n+1} extension. It must be ensured that this extension can be obtained computably using the oracle $\emptyset'$. The distinction between the two cases is not computable in itself, because it asks to decide if $\Phi_e(x)$ halts or not. However, we can use $\emptyset'$ to answer this question as follows: let Φ_i be the partial computable function defined for all n by $\Phi_i(n) = \Phi_e(x)$. The code i can be computably constructed, because it is a simple machine manipulation. It follows that we are in the first case iff $i \notin \emptyset'$. In the first case, $\sigma_{n+1} = \sigma_n$ is trivially computable, while in the second case, it suffices to run $\Phi_e(x)$ to retrieve the returned value, and thus obtain σ_{n+1}. We can therefore find an extension σ_{n+1} forcing the requirement $\mathcal{R}_e$ using the oracle $\emptyset'$.

Satisfaction of a requirement $\mathcal{S}_e$. Suppose σ_n is already defined. We want to find $\sigma_{n+1} \succeq \sigma_n$ such that the behavior of $\Phi_e^A(e)$ is already defined by σ_{n+1}, in other words $\Phi_e^X(e)\downarrow$ for all $X \in [\sigma_{n+1}]$ or $\Phi_e^X(e)\uparrow$ for all $X \in [\sigma_{n+1}]$. Two cases still arise.

- Case 1: there is a string $\tau \succeq \sigma_n$ such that $\Phi_e^\tau(e)\downarrow$. In this case, taking $\sigma_{n+1} = \tau$, we ensure that $\Phi_e^A(e)\downarrow$, because $A \in [\sigma_{n+1}]$.
- Case 2: for any string $\tau \succeq \sigma_n$, $\Phi_e^\tau(e)\uparrow$. In this case, by the use property, whatever the oracle $A \in [\sigma_n]$, $\Phi_e^A(e)\uparrow$. By defining $\sigma_{n+1} = \sigma_n$, we therefore force $\Phi_e^A(e)\uparrow$.

Then, in each case, we have forced the behavior of $\Phi_e^A(e)$ with a finite prefix of the oracle.

Here again, we have to find σ_{n+1} from σ_n computably using the oracle $\emptyset'$. As for the requirement $\mathcal{R}_e$, we define a partial computable function Φ_i which, for each of its inputs, searches for a string $\tau \succeq \sigma_n$ such that $\Phi_e^\tau(e)[|\tau|]\downarrow$, and halts if it finds any. Thus, $i \in \emptyset'$ if, and only if, we are in the first case. Once the case has been determined, the string σ_{n+1} can be found computably. This concludes the proof of Proposition 9.1. ■

Among the first notions of weakness introduced and studied in *Computability Theory* by Cooper and Soare independently, is the hierarchy of low$_n$ sets, of which we give here the first level.

Definition 9.2. A set $A \subseteq \mathbb{N}$ is *low* if $A' \leqslant_T \emptyset'$. ◇

Informally, a set is low if it is indistinguishable from a computable set from the point of view of the Turing jump. We have seen that if $X \leqslant_T Y$, then $X' \leqslant_T Y'$.
Thus, as $\emptyset \leqslant_T A$ for any set A, $\emptyset' \leqslant_T A'$. It follows that a set is low iff $A' \equiv_T \emptyset'$. We will see further examples of non-computable low sets, in particular computably enumerable sets (see Chapter 13).

10. High degrees

If we consider a set $A \leqslant_T \emptyset'$, what are the extreme powers that can be taken by its Turing jump A'?

Recall that if $X \leqslant_T Y$, then $X' \leqslant_T Y'$. In particular, $A' \geqslant_T \emptyset'$. On the other hand, $A' \leqslant_T \emptyset''$ because $A \leqslant_T \emptyset'$. So we have

$$\emptyset' \leqslant_T A' \leqslant_T \emptyset'' \text{ if } A \leqslant_T \emptyset'.$$

Note that in the case where $A \not\leqslant_T \emptyset'$, the Turing jump of A can be arbitrarily complex.

The sets whose Turing jump is equal to the minimum bound, namely $\emptyset'$, are the low sets. We have seen that there are non-computable low sets. We are now going to look at the sets whose Turing jump computes the maximum bound $\emptyset''$.

Definition 10.1. A set $A \subseteq \mathbb{N}$ is *high* if $\emptyset'' \leqslant_T A'$. ◊

Remark

The notion of high set has historically been defined only for sets $A \leqslant_T \emptyset'$. It has since been extended to all sets, but it's important to keep this historical difference in mind when reading the founding articles on computability theory.

Now let's think about the high sets. Unlike the low, their Turing jump has more computational power than expected: it allows the double jump to be computed. The halting problem itself is obviously a trivial example of a high set. As for low sets, the notion of high set is useful for non-trivial examples: high sets which do not compute $\emptyset'$. It is in fact possible to show that for any non-computable set C there exists a high set which does not compute C.

In the following proof, we use for the first time oracles which are not sets of integers, but functions $f : \mathbb{N} \to \mathbb{N}$. Such a function can be represented by the infinite binary sequence G_f such that $G_f(\langle n, m \rangle) = 1$ iff $f(n) = m$. It is clear that G_f allows to compute f, and that any set Y allowing to compute f can also compute G_f: the set G_f is a minimum representation of the function f in Turing degrees. Other equivalent representations in terms of Turing degree are possible, for example with $G_f = 1^{f(0)}01^{f(1)}01^{f(2)}0\dots$ (We start with the $f(0)$ first bits at 1, followed by a 0, then we continue with the next $f(1)$ bits to 1, followed by a 0, etc.).

Proposition 10.2. For any non-computable set C, there exists a high set A such that $A \not\geqslant_T C$. ⋆

Proof. The proof is quite similar to that of Proposition 8.2 with the finite extension method.

Requirements. The set A must satisfy a strength property ($A' \geqslant_T \emptyset''$) and a weakness property ($C \not\leqslant_T A$). Requirements for the weakness property are standard, namely

$$\mathcal{S}_e : \exists x \Phi_e^A(x)\uparrow \ \vee \exists x \Phi_e^A(x)\downarrow \neq C(x).$$

The case of the strength property is different. First of all, it is not a question of controlling what the set A computes, but of controlling what its Turing jump computes. Then, this strength property is not expressed in a negative form (not to be computed by another set) but in a positive form (to compute a complicated object). Negative formulations are often proven by diagonalization, while positive formulations are more constructive. We are therefore not going to express the strength property in the form of requirements, but to impose it structurally in the nature of the object that we construct.

In previous uses of the finite extension method, we constructed a set by creating an infinite sequence of binary strings forming finite approximations of the set.

This time, we are going to build a stable function $f : \mathbb{N} \times \mathbb{N} \to \{0,1\}$ whose limit is $\emptyset''$. As explained in the paragraph preceding the proof, let us recall that this can be reduced to the construction of an element of $2^{\mathbb{N}}$ by considering the set G_f defined by $\langle n, m\rangle \in G_f$ iff $f(n) = m$. By the relativization of the Shoenfield limit lemma to f (see Lemma 7.2), a set B is limit-f-computable iff $B \leqslant_T f'$. In particular, for $B = \emptyset''$, if $\emptyset''$ is limit-f-computable, then $\emptyset'' \leqslant_T f'$, in other words f is high. The set A is therefore any set in the Turing degree of f.

Construction. The function f will be built from successive finite approximations which are more and more precise. These approximations, instead of being binary strings, will be pairs (g, m), where

- $g \subseteq \mathbb{N} \times \mathbb{N} \to \{0,1\}$ is a partial function with two parameters whose domain is finite, representing a piece of the function f that we are building.
- m is an integer meaning that henceforth, when we extend the domain of g with a new input (x, y), if $x < m$ then $g(x, y) = \emptyset''(x)$.

In other words, the limit of the m first "columns" of the function f has already been reached and has the correct value. We will call these couples *conditions*, because they condition part of the behavior of the function f.

In the same way that the suffix relation $\sigma \preceq \tau$ for binary strings means that τ is a more precise approximation of the sequence that we construct, we will define an extension relation on the conditions $(g, m) \preceq (h, n)$ to mean that the condition (h, n) is more precise, or more restrictive, than the condition (g, m).

				$m = 3$			
y	$f(0,y)$	$f(1,y)$	$f(2,y)$	$f(3,y)$	$f(4,y)$	$f(5,y)$	$f(6,y)$
0	0	1	0	0	0	0	0
1	0	0	0	0	1	0	0
2	1	1	1	0	1	1	0
3	0	0	1	0	0	1	1
4	1	0	0	1	1	0	1
5	1	0	0	0	0	1	1
6	1	0	0	0	1	1	1
7	1	0	0	1	0	0	0
	1	0	0	0	1	1	0
	=	=	=	=	=	=	=
	$\emptyset''(0)$	$\emptyset''(1)$	$\emptyset''(2)$	$\emptyset''(3)$	$\emptyset''(4)$	$\emptyset''(5)$	$\emptyset''(6)$

Figure 10.3: The dark grey part represents a condition with $m = 3$. The light grey part represents an extension of this condition: each ith column for $i < m$ is fixed for every extension to a value which has to correspond to the ith bit of $\emptyset''$.

Given a partial function $h \subseteq \mathbb{N}\times\mathbb{N} \to \{0,1\}$, we denote by $\operatorname{dom} h$ its domain of definition. We therefore say that the condition (h,n) *extends* (g,m) (denoted $(h,n) \succeq (g,m)$) if $n \geqslant m$ and if, in addition,

(P1) $g \subseteq h$, i.e. $\operatorname{dom} g \subseteq \operatorname{dom} h$ and for all $(x,y) \in \operatorname{dom} g$,

$$g(x,y) = h(x,y)\,;$$

(P2) for all $(x,y) \in \operatorname{dom} h \setminus \operatorname{dom} g$, if $x < m$, then $h(x,y) = \emptyset''(x)$.

The property (P1) means that the finite functions must be compatible, and more precisely that the function h must extend the function g, while the property (P2) formalizes the idea according to which the second parameter of a condition fixes the columns of the function by stabilizing them. Figure 10.3 illustrates what has just been explained.

Let's stop for a moment to familiarize ourselves with this new mathematical object and learn how to manipulate it. First, for any condition (g,m) and any n, (g,n) is also a condition. If in addition $n \geqslant m$, then (g,n) is an extension. The integer m does not impose any constraint in itself on the finite function g to form a condition (g,m). On the other hand, m imposes restrictions on the extensions of (g,m). More precisely, if $n \geqslant m$, then the set of extensions of (g,n) is a subset of the extensions of (g,m). Finally, $(\emptyset,0)$ is a valid condition, where $\emptyset$ is the function defined nowhere.

From the point of view of computability theory, the set of conditions is a computable set. On the other hand, the extension relation between two conditions cannot be computed because of the property (P2) which involves $\emptyset''$. However, if we fix m, then the extension relation between conditions having m for second component is computable, because this only involves a finite segment $\emptyset'' \upharpoonright_m$ of the set $\emptyset''$. It suffices to "hardcode" this initial segment in the program. This observation will be exploited to satisfy the requirements $\mathcal{S}_e$.

In the same way that we denote by $[\sigma]$ the set of infinite binary sequences having for initial segment σ, we will denote by $[g, m]$ the set of total functions $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ such that $g \subseteq f$ and such that for all $(x, y) \notin \operatorname{dom} g$ with $x < m$ we have $f(x, y) = \emptyset''(x)$. Thus, $[g, m]$ is the collection of the set of candidate functions that can be obtained by completing the partial approximation (g, m). Note that $[g, m]$ does not only contain stable functions.

We are therefore going to build f by successive approximations in the form of conditions

$$(g_0, m_0) \preceq (g_1, m_1) \preceq (g_2, m_2) \preceq \dots$$

to define $f = \bigcup_t g_t$. In other words, $\operatorname{dom} f = \bigcup_t \operatorname{dom} g_t$, and for any pair $(x, y) \in \operatorname{dom} f$, $f(x, y) = g_t(x, y)$ for a t such that $(x, y) \in \operatorname{dom} g_t$. The function f is well defined thanks to the compatibility property (P1). If we make sure that the integers m_t become arbitrarily large, it is easy to verify by the property (P2) that the resulting function f is stable, and has a limit of $\emptyset''$. Note in particular that $f \in \bigcap_t [g_t, m_t]$.

A condition (g, m) *forces* a requirement $\mathcal{S}_e$ if the property is satisfied for all $f \in [g, m]$. Here, we have replaced the occurrences of A by f in the requirement $\mathcal{S}_e$. At each stage of construction, a requirement is going to be forced.

Satisfaction of a requirement $\mathcal{S}_e$. The argument is similar to that of Proposition 8.2, but handling conditions and not binary strings. Let (g_t, m_t) be a condition. The following three cases arise.

- Case 1: there is an input x and a function f in $[g_t, m_t]$ such that

$$\Phi_e^f(x)\downarrow \neq C(x).$$

  In this case, by the use property, there exists an extension (g_{t+1}, m_t) of (g_t, m_t) such that $\Phi_e^f(x) \downarrow \neq C(x)$ for all f in $[g_{t+1}, m_t]$. Note that $m_{t+1} = m_t$. The condition (g_{t+1}, m_t) therefore forces the requirement $\mathcal{S}_e$.

- Case 2: there is an input x such that for all the functions f in $[g_t, m_t]$, we have $\Phi_e^f(x)\uparrow$. In this case, the condition (g_t, m_t) already forces the requirement $\mathcal{S}_e$ ensuring that $\Phi_e^f(x)\uparrow$.

- Case 3: neither of the two previous cases occurs. We will then show that it is possible to compute the set C, and therefore to deduce a contradiction from it. Here is the procedure to compute the value of $C(x)$: look for a condition (h, m_t) extending (g_t, m_t) with the same second component m_t, such that $\Phi_e^h(x)\downarrow$. We claim the following two facts:

  (1) there is such an extension, and therefore the search will end,
  (2) whatever $(h, m_t) \succeq (g_t, m_t)$, $\Phi_e^h(x)\downarrow \rightarrow \Phi_e^h(x) = C(x)$.

  To show (1), notice that the negation of Case 2 means that for any x (and in particular for this x considered), there exists a function $f \in [g_t, m_t]$ such that $\Phi_e^f(x)\downarrow$. By the use property, there then exists a condition $(h, m_t) \succeq (g_t, m_t)$ such that $\Phi_e^h(x)\downarrow$.
  Let us show (2). If $\Phi_e^h(x)\downarrow \neq C(x)$, then Case 1 would be true taking any $f \in [h, m_t] \subseteq [g_t, m_t]$. It follows that $\Phi_e^h(x)\downarrow = C(x)$. Finally, notice that we only considered conditions with the same m_t. As explained above, for m_t fixed, the extension relation is computable.

  We have therefore described a computable procedure to determine the value of $C(x)$ whatever x, contradicting the hypothesis according to which C is not computable.

It is then enough to satisfy each requirement $\mathcal{S}_e$ by gradually extending our conditions, while making "artificially" their second components increase towards $+\infty$ in order to ensure that the final solution is indeed a stable function whose limit is $\emptyset''$. This concludes the proof of Proposition 10.2. ■

Corollary 10.4
There is a set A both high and Turing-incomplete. In other words, $A' \geqslant_T \emptyset''$ and $A \not\geqslant_T \emptyset'$.

Proof. Immediate by Proposition 10.2, taking $C = \emptyset'$. ■

Remark

The name "condition" is a generic name borrowed from the theory of forcing, and which will be called upon to designate very different mathematical objects throughout this work, although all corresponding to the idea of an approximation of an object that we construct. This notion will be developed in Chapter 11.

Note that the argument of the previous proof does not depend specifically on $\emptyset''$ and we could have constructed, for any B, a set A such that $A' \geqslant_T B$ and $A \ngeqslant_T C$. We will see in Chapter 13 that there are non-computable low c.e. sets. Sacks [195] has also shown the existence of high c.e. sets of incomplete degrees.

There is a big difference between the low and high sets: the former are all computable in $\emptyset'$, and they are therefore in countable quantity. The second, on the other hand, can be arbitrarily complex, and it is easily shown that they are in uncountable quantity. However, we will see with Corollary 19-3.9 and Proposition 10-3.38 that there are "few" high sets, from the point of view of measure theory and the theory of categories of Baire.

Let's end this chapter with a few exercises.

Exercise 10.5. ($\star$) Adapt the proof of Proposition 10.2 to show that for any set B and any non-computable set C, there exists a set A such that $A' \geqslant_T B$ and $A \ngeqslant_T C$. $\diamond$

Exercise 10.6. ($\star\star$) Adapt the proof of Proposition 10.2 to show that there exists an incomplete and $\emptyset'$-computable high set. $\diamond$

Exercise 10.7. ($\star\star$) Show by the finite extension method that there exists a sequence of sets $(A_n)_{n\in\mathbb{N}}$ pairwise incomparable for the Turing reduction, i.e., such that

$$A_n \nleqslant_T A_m \quad \text{et} \quad A_m \nleqslant_T A_n,$$

for all $n \neq m \in \mathbb{N}$. $\diamond$

Chapter 5

Arithmetic hierarchy

We introduce a complexity hierarchy on the sets of integers. The computably enumerable sets are called Σ_1^0 and their complements are called Π_1^0. The effective countable intersections of Σ_1^0 sets are said to be Π_2^0 and the effective countable union of Π_1^0 sets are said to be Σ_2^0, and so on.

We call this construction *arithmetic hierarchy*, because the sets it contains are exactly those which are definable by a first-order formula in the language of Peano arithmetic. We will talk about this equivalence again in Section 9-3. There is a direct correspondence between the definition of a set by meetings/intersections, and the fact of being able to define it by a formula comprising quantifiers $\exists/\forall$. Thus, a union corresponds to an existential quantifier and an intersection to a universal quantifier.

Example 1. The set of codes e for the total computable functions can be written

$$\{e \in \mathbb{N} : \forall n \; \exists t \; \Phi_e(n)[t]\downarrow\} = \bigcap_n \bigcup_t \{e \in \mathbb{N} : \Phi_e(n)[t]\downarrow\}.$$

It is a Π_2^0 set, that is to say a countable intersection of Σ_1^0 sets, each of them being moreover a countable union of computable sets.

1. Elementary properties

Let's start with a formal definition of the arithmetic hierarchy.

Definition 1.1. Let $m, n \geqslant 1$.

1. A set $A \subseteq \mathbb{N}^m$ is said to be Σ_n^0 if there exists a computable set R included in $\mathbb{N}^{n+m}$ such that
$$A = \{(y_1, \ldots, y_m) : \overbrace{\exists x_1 \forall x_2 \ldots Q x_n}^{n \text{ quantifiers}} (x_1, \ldots, x_n, y_1, \ldots, y_m) \in R\},$$
where Q is $\exists$ if n is odd, and $\forall$ if n is even.
2. A set $A \subseteq \mathbb{N}^m$ is said to be Π_n^0 if there exists a computable set R included in $\mathbb{N}^{n+m}$ such that
$$A = \{(y_1, \ldots, y_m) : \overbrace{\forall x_1 \exists x_2 \ldots Q x_n}^{n \text{ quantifiers}} (x_1, \ldots, x_n, y_1, \ldots, y_m) \in R\},$$
where Q is $\forall$ if n is odd, and $\exists$ if n is even. $\diamondsuit$

Beware, in the preceding definition, it is the alternation between the quantifiers $\exists$ and $\forall$ which counts.

Then, for example, the set
$$\{y : \exists x_1 \forall x_2 \; R(y, x_1, x_2)\}$$
for R computable is a Σ_2^0 set, but the set
$$\{y : \exists x_1 \exists x_2 \exists x_3 \; R(y, x_1, x_2, x_3)\}$$
is, despite its three quantifiers, a Σ_1^0 set: one can easily eliminate repetitions of quantifiers of the same type. Consider for example a formula of the form
$$\exists x_1 \; \exists x_2 \; \forall y_1 \; \forall y_2 \; R(x_1, x_2, y_1, y_2).$$
The latter can simply be rewritten as $\exists x \; \forall y \; R'(x, y)$ where R' will be a modified version of R, which will consider x and y respectively as pairs $\langle x_1, x_2 \rangle$ and $\langle y_1, y_2 \rangle$. The predicate R' will then use the projections π_1 and π_2 realizing the inverse functions of the computable coupling bijection $\langle \; , \; \rangle : \mathbb{N}^2 \to \mathbb{N}$, in order to recover x_1, x_2, y_1, y_2. The reader can go back to Exercise 3-2.3 to convince himself that the bijection $\langle \rangle$ and that its two inverse functions are computable. An abuse of notation will consist for example of writing $\exists \langle x_1, x_2 \rangle \; \forall \langle y_1, y_2 \rangle \; R(x_1, x_2, y_1, y_2)$, meaning that the predicate R takes care of recovering x_1, x_2 from $\langle x_1, x_2 \rangle$, and the same for y_1, y_2.

Predicates

We will sometimes speak of Σ_n^0 predicates to denote a formula of the form $\exists x_1 \; \forall x_2 \; \ldots Q x_n \; R(x_1, x_2, \ldots, x_n)$, where R is a computable predicate.

Note that by using the fact that the complement of a computable set is computable, we can easily see that the Σ_n^0 sets are the complements of the Π_n^0 sets. It is also quite possible for a set to be both Σ_n^0 and Π_n^0. For this, the following notion is introduced.

Definition 1.2. Let $m \geqslant 1$. A set $A \subseteq \mathbb{N}^m$ is said to be Δ_n^0 for $n > 0$ if it is both Σ_n^0 and Π_n^0. $\diamondsuit$

The arithmetic hierarchy establishes a first level of distinction between arithmetically definable sets. Intuitively, a Σ_{n+1}^0 set is strictly more complex than a Σ_n^0 or even Π_n^0 set. Indeed, each of the last two can be written in a Σ_{n+1}^0 form by simply adding unused quantifiers.

Example 1.3. The Σ_2^0 set given by $\{y : \exists x_1 \forall x_2\ R(y, x_1, x_2)\}$ can also be written in a Π_3^0 form: $\{y : \forall z \exists x_1 \forall x_2\ R(y, x_1, x_2)\}$ or in a Σ_3^0 form: $\{y : \exists x_1 \forall x_2 \exists z\ R(y, x_1, x_2)\}$.

However, it is not always possible to use fewer quantifiers. We will show with Corollary 5.6 that for all $n > 0$, there exist Δ_{n+1}^0 sets which cannot be described in a Σ_n^0 or Π_n^0 way: the arithmetic hierarchy is strict.

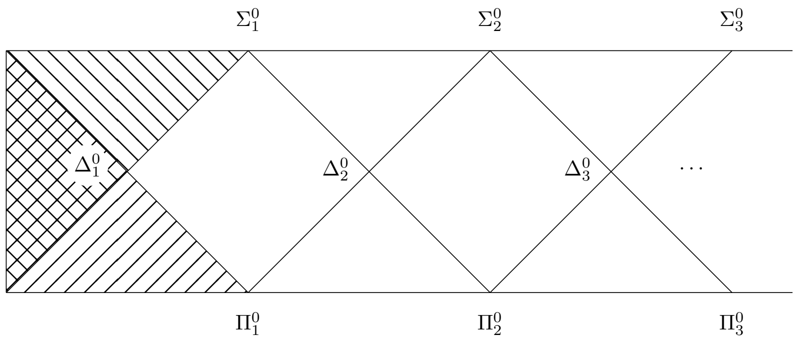


Figure 1.4: Representation of the arithmetic hierarchy. The upper left triangle represent Σ_1^0 sets. The lower left hatched triangle represents Π_1^0 sets. The intersectuion between the two — the double hatched triangle — represents Δ_1^0 sets. The remainder of the diagram follows similarly.

Before continuing, let us give some examples of sets of the arithmetic hierarchy.

Example 1.5.

- Any computable set $A \subseteq \mathbb{N}$ is Δ_1^0 (we will see a proof of this with

Proposition 3.4). It suffices to add an unnecessary existential or universal quantification to the computable predicate A to be able to express it respectively as a Σ^0_1 or Π^0_1 set.

- Any computably enumerable set $A \subseteq \mathbb{N}$ is Σ^0_1 (we will see this precisely with Proposition 3.3). Indeed, such A is described as $\{x : \exists t\ \Phi_e(x)[t]\downarrow\}$ for some e.
- We have seen with Example 1 that the set of total function codes is Π^0_2. Its complement, namely the set of partial function codes, is therefore Σ^0_2: $\{e : \exists n\ \forall t\ \Phi_e(n)[t]\uparrow\}$
- The set of function codes which are total, except for a finite number of elements, is Σ^0_3:

$$\{e : \exists n\ \forall m\ \exists t\ \text{ such that } m < n \text{ or } \Phi_e(m)[t]\downarrow\}.$$

We will see a little later that the previous examples are optimal. For example, the set of function codes which are total except for a finite number of elements cannot be expressed in a Π^0_3 way: it is a strict Σ^0_3 set.

We now show a series of three propositions on the stability of Σ^0_m and Π^0_m sets under different operations. For example, the stability under finite union means that a finite union of Σ^0_m sets is still a Σ^0_m set. The propositions will be proved only by considering subsets of $\mathbb{N}$, but generalize without problem to subsets of $\mathbb{N}^n$ for arbitrary n.

Proposition 1.6. The Σ^0_m (resp. Π^0_m) sets are stable under finite unions and finite intersections. ⋆

Proof. Let $m > 0$. Be

$$\begin{aligned} A_0 &= \{n : \exists x_1\ \forall x_2\ \dots\ Qx_m\ R_0(n, x_1, \dots, x_m)\} \\ A_1 &= \{n : \exists y_1\ \forall y_2\ \dots\ Qy_m\ R_1(n, y_1, \dots, y_m)\} \end{aligned}$$

two Σ^0_m sets, where Q is the symbol $\exists$ if m is odd and $\forall$ if m is even. We leave it to the reader to show, by inclusion in one direction and then in the other, that:

$$A_0 \cap A_1 = \left\{ n : \begin{array}{c} \exists\langle x_1, y_1\rangle\ \forall\langle x_2, y_2\rangle\ \dots\ Q\langle x_m, y_m\rangle \\ \text{such that } R_0(n, x_1, \dots, x_m) \wedge R_1(n, y_1, \dots, y_m) \end{array} \right\}.$$

The Σ^0_m sets are therefore stable under finite intersections. We also have in the same way:

$$A_0 \cup A_1 = \left\{ n : \begin{array}{c} \exists\langle x_1, y_1\rangle\ \forall\langle x_2, y_2\rangle\ \dots\ Q\langle x_m, y_m\rangle \\ \text{such that } R_0(n, x_1, \dots, x_m) \vee R_1(n, y_1, \dots, y_m) \end{array} \right\}.$$

The Σ^0_m sets are therefore stable under finite union. By passing to the complement, the Π^0_m sets are also stable under finite unions and intersections.■

Let us remember the concept of uniformly computable sequence, introduced on the occasion of the developments which follow Lemma 4-7.2. The concept of uniformity passes to the arithmetic hierarchy as follows: in the next proposition, a countable union $(A_i)_{i\in\mathbb{N}}$ of sets *uniformly* Σ^0_m is therefore a union such that each set A_i admits the same Σ^0_m description, but with i as a parameter. Formally, if

$$A_i = \{n : \exists x_1\ \forall x_2\ \dots\ Qx_m\ R_i(n, x_1, \dots, x_m)\},$$

there must be a computable process allowing from i to return a code for the predicate R_i. Equivalently, we can consider that A_i is described as

$$A_i = \{n : \exists x_1\ \forall x_2\ \dots\ Qx_m\ R(i, n, x_1, \dots, x_m)\},$$

where R is a computable predicate taking i as an additional parameter.

Proposition 1.7. The Σ^0_m (resp. Π^0_m) sets are stable under uniform countable unions (resp. uniform countable intersections). ⋆

Proof. Let $m > 0$ and let $(A_i)_{i\in\mathbb{N}}$ be a uniform sequence of Σ^0_m sets:

$$A_i = \{n : \exists x_1\ \forall x_2\ \dots\ Qx_m\ R(i, n, x_1, \dots, x_m)\},$$

where Q is the symbol $\exists$ if m is odd and $\forall$ if m is even. It is easily shown by mutual inclusion that

$$\bigcup_i A_i = \{n : \exists\langle i, x_1\rangle\ \forall x_2\ \dots\ Qx_m\ R(i, n, x_1, \dots, x_m).$$

The Σ^0_m sets are therefore stable under uniform countable unions. By passing to the complement, the Π^0_m sets are themselves also stable under countable uniform intersections. ■

Note that the stability of Σ^0_n sets by uniform countable union is equivalent to stability under existential quantification (resp. universal quantification for Π^0_n sets).

Exercise 1.8. Show that the Σ^0_n sets are not stable under non-uniform countable unions. ◇

We now show the stability under uniform bounded quantification. A bounded quantification of the form $\forall x < m$ can be seen as a finite union of m sets parameterized by x. The following proposition is however not identical to Proposition 1.6: here we want uniformity as a function of the bound which can itself be a variable, or depend on other variables on which we quantify.

Roughly speaking, what the following proposition says is that the set

$$\{n : \exists x\ \forall y < x\ \exists t\ R(n, x, y, t)\},$$

where R is a computable predicate, can be rewritten as

$$\{n : \exists x\ \exists t\ \forall y < x\ \exists s < t\ R(n, x, y, s)\}.$$

The predicate $\forall y < x\ \exists s < t\ R(n, x, y, s)$ being computable, the set is Σ^0_1.

Proposition 1.9. The Σ^0_m and Π^0_m sets are stable under uniform bounded quantifications. ⋆

Proof. We show that we can always move the bounded quantification to the right, until it is found next to the computable predicate.

Let $A = \{(n, k) : \forall y < k\ \exists x\ R(n, y, x)\}$, where R is a computable or Π^0_m set for $m > 0$. So we also have

$$A = \{(n, k) : \exists x\ \forall y < k\ \exists z < x\ R(n, y, z)\}.$$

The equality comes from the fact that if for all $y < k$ a certain x_k witnesses that a formula is true, since the number of witnesses is finite, these are bounded in $\mathbb{N}$. The variable x which previously served as a witness is now used as a bound on all possible witnesses. We have in the same way by passing to the complement

$$\{(n, k) : \exists y < k\ \forall x\ R(n, y, x)\} = \{(n, k) : \forall x\ \exists y < k\ \forall z < x\ R(n, y, z)\},$$

where R is a computable or Σ^0_m set.

If we now have $A = \{(n, k) : \exists y < k\ \exists x\ R(n, y, x)\}$, where R is a computable or Π^0_m set, then we also have $A = \{(n, k) : \exists x\ \ \exists y < kR(n, y, x)\}$ immediately. We have in the same way by passing to the complement $\{(n, k) : \forall y < k\ \forall x\ \ R(n, y, x)\} = \{(n, k) : \forall x\ \ \forall y < k\ \ R(n, y, x)\}$, where R is a computable or Σ^0_m set.

By induction, we thus move the bounded quantifications to the right until they are all stuck to the computable predicate. Finally, we use the fact that the computable predicates are stable under bounded quantifications (see Exercise 3-2.4) to conclude. ■

2. Arithmetic hierarchy and computability

Now let's see what the first levels of the arithmetic hierarchy correspond to.

Proposition 2.1. A set $A \subseteq \mathbb{N}$ is Σ^0_1 iff it is computably enumerable. ⋆

Proof. Suppose that the set A is computably enumerable. Then, $A = \{n : \exists t\ \Phi_e(n)[t]\downarrow\}$ for some e. The predicate $\Phi_e(n)[t]\downarrow$ being computable, A is Σ^0_1. Suppose now that A is Σ^0_1. Let $A = \{n : \exists t\ R(n,t)\}$, where R is a computable predicate. Then, we easily define the code machine e which on the input n searches for the smallest t such that $R(n,t)$ and halts, or continues its search indefinitely otherwise. We then have $\Phi_e(n)\downarrow$ iff $\exists t\ R(n,t)$. ∎

Proposition 2.2. A set $A \subseteq \mathbb{N}$ is Δ^0_1 iff it is computable. ⋆

Proof. A set A is Δ^0_1 iff A and $\mathbb{N} \setminus A$ are Σ^0_1 iff A and $\mathbb{N} \setminus A$ are computably enumerable (according to Proposition 2.1), or equivalently if A is computable (according to Proposition 3-7.4). ∎

We have seen that there are computably enumerable sets which are not computable, and in particular whose complement is not computably enumerable. This implies that some sets are Σ^0_1 but not Δ^0_1, and in particular not Π^0_1. We will see in the next sections that the hierarchy is strict everywhere: for all n, there are Σ^0_n sets which are not Π^0_n, and there are Δ^0_{n+1} sets which are not neither Σ^0_n, nor Π^0_n.

Intuitively, the number of quantifiers corresponds to the "number of times it would take to infinity" to determine the membership of an element in the set. Thus, for a Σ^0_1 set written as $\{n : \exists t\ R(t,n)\}$, where R is a computable predicate, it would be necessary to test the truth value of $R(t,n)$ for all integers t, so to find one which witnesses the belonging of n to the set, or else in order to be sure that n does not belong to it.

For a Π^0_2 set of the form

$$\{n : \forall t_1\ \exists t_2\ R(t_1,t_2,n)\},$$

we would need a first procedure which tests all the integers t_1, and which for each of these tests, examines the truth value of $R(t_1,t_2,n)$ for all the integers t_2. This would sort of correspond to two nested loops each ranging over the set of all integers.

3. Relativization to an oracle

The arithmetic hierarchy makes it possible to define more and more complex sets, and in a sense less and less computable. However, the class of arithmetically definable sets remains countable. So there remain "many" sets, so to speak the majority, which cannot be computed, nor even defined by an arithmetic formula. It is obviously difficult to speak about them, and any attempt to define them more precisely would make them definable in a certain language, widening only a little more the inevitably countable class of the sets of which one manages to say something, leaving aside the majority of the other sets, hidden, inaccessible.

We'll work around this problem throughout the next few chapters, primarily by looking at "groups of sets" rather than each set individually. We will see in particular in the chapters to come many computational properties shared by certain sets, in general an uncountable quantity of them. We will then carry out in Part II a study of the typical sets from the point of view of measure theory, that is to say of the sets which one obtains with probability 1 if one selects their bits at random.

For the moment, we are content to relativize the arithmetic hierarchy to an oracle: given any set X, we consider the Σ^0_n, Π^0_n and Δ^0_n sets that we can define relative to the knowledge of X. We base ourselves for this on the oracle computations of Definition 4-2.2, and we iterate as in Definition 1.1.

Definition 3.1. Let $m, n \geqslant 1$. Let $X \subseteq \mathbb{N}$.

1. A set $A \subseteq \mathbb{N}^m$ is said to be $\Sigma^0_n(X)$ if there exists an X-computable set $R \subseteq \mathbb{N}^{n+m}$ such that

$$A = \{(y_1, \ldots, y_m) : \overbrace{\exists x_1 \forall x_2 \ldots Q x_n}^{n \text{ quantifiers}} (x_1, \ldots, x_n, y_1, \ldots, y_m) \in R\},$$

where Q is $\exists$ if n is odd, and $\forall$ if n is even.

2. A set $A \subseteq \mathbb{N}^m$ is said to be $\Pi^0_n(X)$ if there exists an X-computable set $R \subseteq \mathbb{N}^{n+m}$ such that

$$A = \{(y_1, \ldots, y_m) : \overbrace{\forall x_1 \exists x_2 \ldots Q x_n}^{n \text{ quantifiers}} (x_1, \ldots, x_n, y_1, \ldots, y_m) \in R\},$$

where Q is $\forall$ if n is odd, and $\exists$ if n is even. ♢

Definition 3.2. Let $m \geqslant 1$. Let $X \subseteq \mathbb{N}$. A set $A \subseteq \mathbb{N}^m$ is said to be $\Delta^0_n(X)$ for $n > 0$ if it is both $\Sigma^0_n(X)$ and $\Pi^0_n(X)$. ♢

The following propositions are proved in the same way as their respective equivalents of the previous section.

Proposition 3.3. A set $A \subseteq \mathbb{N}$ is $\Sigma^0_1(X)$ iff it is X-c.e. ⋆

Proposition 3.4. A set $A \subseteq \mathbb{N}$ is $\Delta^0_1(X)$ iff it is X-computable. ⋆

Note that an oracle will be able to provide additional computational power, allowing certain sets that are normally not Σ^0_n, to become $\Sigma^0_n(X)$. We can even for example define an oracle X such that all arithmetic sets, that is to say Σ^0_n for a certain n, become $\Delta^0_1(X)$. However, whatever the computational power of X, the arithmetic sets in X remain in countable quantity.

4. Many-one degrees

The classification of the arithmetic hierarchy in terms of Σ^0_n and Π^0_n sets is not a notion on Turing degrees, because a Σ^0_n set can be Turing equivalent to a set which is not. In fact, any Σ^0_n set A is Turing equivalent to its Π^0_n complement $\overline{A}$. In this sense, the Turing reduction is "coarse", because it does not distinguish a set from its complement.

We are going to introduce a concept finer than the Turing reduction, in the sense that it implies the latter. This is the many-one reduction, which as we will see, preserves the arithmetic hierarchy. In fact, a lot of proofs of Turing reduction that we have seen are actually many-one reductions.

Definition 4.1. Let A, B be two sets. We say that A is *many-one reducible* to B, and we write $A \leqslant_m B$ if there exists a total computable function $f : \mathbb{N} \to \mathbb{N}$ such that $n \in A \leftrightarrow f(n) \in B$. If $A \leqslant_m B$ and $B \leqslant_m A$, we write $A \equiv_m B$. We write $A <_m B$ if $A \leqslant_m B$ but $B \nleqslant_m A$. We call *degrees many-one* the equivalence classes of the relation $\equiv_m$. ♢

When $A \leqslant_m B$ holds, the knowledge of B is sufficient to compute A, and we have in particular $A \leqslant_T B$: let f be the total computable function such that $n \in A \leftrightarrow f(n) \in B$. Then, to know if $n \in A$, we use f for "ask a question" to the set B: is that $f(n) \in B$? If the answer is yes, then $n \in A$. Otherwise, $n \notin A$. The many-one reduction is very restrictive: if $A \leqslant_m B$, then to know a bit of A we only have the right to ask the oracle B only one question. As if that were not enough, the answer to the question directly determines whether the bit belongs to A without it being possible to reverse this decision. The importance of this very restrictive reduction comes from the fact that it preserves the arithmetic hierarchy.

Proposition 4.2. Let $A \subseteq \mathbb{N}$ be a $\Sigma^0_m(X)$ set (resp. $\Pi^0_m(X)$) for a certain $X \subseteq \mathbb{N}$, and let $B \leqslant_m A$. Then, B is $\Sigma^0_m(X)$ (resp. $\Pi^0_m(X)$). ⋆

Proof. Let A be a $\Sigma^0_m(X)$ set. Then,

$$A = \{n : \exists x_1 \ \forall x_2 \ \dots \ Qx_m \ R(n, x_1, \dots, x_m)\},$$

where R is an X-computable set. Let f be the total computable function such that $n \in B$ iff $f(n) \in A$. So we have

$$B = \{n : \exists x_1 \ \forall x_2 \ \ldots \ Qx_m \ R(f(n), x_1, \ldots, x_m)\},$$

which is a $\Sigma^0_m(X)$ description of B.

We show in the same way that if A is $\Pi^0_m(X)$ and $B \leqslant_m A$, then B is $\Pi^0_m(X)$. ■

Among the Σ^0_1 sets, the halting problem plays a particular role: it is the most "powerful" of the Σ^0_1 sets, in the sense that any Σ^0_1 set is many-one reducible to the halting problem.

Proposition 4.3. A set A is Σ^0_1 iff $A \leqslant_m \emptyset'$. ⋆

Proof. Suppose A is Σ^0_1. Then, there exists e such that $n \in A$ iff $\Phi_e(n)\downarrow$. Using the SMN theorem, we define a computable function $s : \mathbb{N} \to \mathbb{N}$ such that for all m we have $\Phi_e(n) = \Phi_{s(n)}(m)$. We will have in particular $n \in A$ iff $\Phi_e(n)\downarrow$, or equivalently if $\Phi_{s(n)}(s(n))\downarrow$, or even if $s(n) \in \emptyset'$. Then, $A \leqslant_m \emptyset'$.

Suppose now $A \leqslant_m \emptyset'$. Then, as $\emptyset'$ is Σ^0_1, the set A is also Σ^0_1, according to Proposition 4.2. ■

We also say that $\emptyset'$ is a Σ^0_1-complete set, as we will see with Definition 5.2.

5. Post's theorem

We will see that there is a precise correspondence between the iterations of the Turing jump and the arithmetic hierarchy.

Definition 5.1. Given a set $X \subseteq \mathbb{N}$, we define recursively on $n \geqslant 0$:

1. $X^{(0)} = X$
2. $X^{(n+1)} = X^{(n)'}$. ◇

Thus, $\emptyset^{(1)} = \emptyset'$, $\emptyset^{(2)} = \emptyset''$, etc. We are now going to show that, for all n, the set $\emptyset^{(n)}$ is Σ^0_n-complete: it is a Σ^0_n set, which also has a maximum computational power among the Σ^0_n sets, in the sense that any Σ^0_n set is many-one reducible to $\emptyset^{(n)}$.

Definition 5.2. A set $A \subseteq \mathbb{N}$ is $\Sigma^0_n(X)$-*complete* (resp. $\Pi^0_n(X)$-*complete*) if it is $\Sigma^0_n(X)$ (resp. $\Pi^0_n(X)$) and if, for any $\Sigma^0_n(X)$ set (resp. $\Pi^0_n(X)$) B, we have $B \leqslant_m A$. ♢

Proposition 5.3. Let $n > 0$. The set $\emptyset^{(n)}$ is Σ^0_n-complete. Likewise, for any set $X \subseteq \mathbb{N}$, the set $X^{(n)}$ is $\Sigma^0_n(X)$-complete. ⋆

Proof. Let us show by induction on n that for all $n > 0$ the set $\emptyset^{(n)}$ is Σ^0_n. By definition, the set $\emptyset^{(1)}$ is Σ^0_1. Suppose the set $\emptyset^{(n)}$ is Σ^0_n. Then, the set $\emptyset^{(n+1)}$ is defined as:

$$\left\{ m : \exists t \in \mathbb{N}\ \exists \sigma \in 2^{<\mathbb{N}} \quad \begin{array}{l} \left(\forall s < |\sigma| \quad \begin{array}{c} (\sigma(s) = 1 \text{ and } s \in \emptyset^{(n)}) \\ \text{or} \\ (\sigma(s) = 0 \text{ and } s \notin \emptyset^{(n)}) \end{array}\right) \\ \text{and} \quad \Phi_m(\sigma, m)[t]\downarrow \end{array} \right\}.$$

The description of $\emptyset^{(n+1)}$ is therefore done with existential quantifiers, followed by a predicate using $s \in \emptyset^{(n)}$, which is Σ^0_n by induction, and using $s \notin \emptyset^{(n)}$, which is Π^0_n by induction, and therefore Σ^0_{n+1}. Using the stability properties of propositions 1.9 and 1.6, the predicate which follows existential quantifiers $\exists t \in \mathbb{N}\ \exists \sigma \in 2^{<\mathbb{N}}$ is in particular Σ^0_{n+1} uniformly in m, σ and t. Now using the uniform countable union stability of Proposition 1.7, the set $\emptyset^{(n+1)}$ is therefore Σ^0_{n+1}.

Let us now show by induction on n that any Σ^0_n set is many-one reducible to $\emptyset^{(n)}$. This is the case with Proposition 4.3 for $n = 1$. Suppose this is the case for some n. Let $A = \{x : \exists y\ R(x, y)\}$, where R is a Π^0_n set.

By induction, there exists a total computable function $g : \mathbb{N} \to \mathbb{N}$ such that $(x, y) \in R$ iff $g(\langle x, y\rangle) \notin \emptyset^{(n)}$. We define the total computable function f such that $\Phi^{\emptyset^{(n)}}_{f(x)}$ halts on any input if $\exists y\ g(\langle x, y\rangle) \notin \emptyset^{(n)}$, and does not halt on any input otherwise. We therefore have $f(x) \in \emptyset^{(n+1)}$ iff $\Phi_{f(x)}(\emptyset^{(n)}, f(x))\downarrow$,in other words if $\exists y\ g(\langle x, y\rangle) \notin \emptyset^{(n)}$, or even if $\exists y\ R(x, y)$, or finally if $x \in A$. The set A is therefore many-one reducible to $\emptyset^{(n+1)}$.

The relativization to an oracle X is similar and does not present any particular difficulty. ■

Corollary 5.4
A set A is $\Sigma^0_n(X)$ iff $A \leqslant_m X^{(n)}$.

Proof. By the previous proposition and by the definition of $\Sigma^0_n(X)$-completeness.■

We finally come to Post's theorem.

Theorem 5.5 (Post's theorem)
Let A be a set and $n \geqslant 0$.

(1) A is Σ^0_{n+1} iff A is $\Sigma^0_1(\emptyset^{(n)})$, iff A is $\emptyset^{(n)}$-c.e.

(2) A is Δ^0_{n+1} iff A is $\Delta^0_1(\emptyset^{(n)})$, iff $A \leqslant_T \emptyset^{(n)}$

Proof. Let us show (1). By Corollary 5.4, A is Σ^0_{n+1} iff $A \leqslant_m \emptyset^{(n+1)}$. By relativizing Proposition 4.3 to $\emptyset^{(n)}$, we obtain $A \leqslant_m \emptyset^{(n)'}$ (which is equal to $\emptyset^{(n+1)}$) if A is $\Sigma^0_1(\emptyset^{(n)})$. Finally, according to Proposition 3.3, A is $\Sigma^0_1(\emptyset^{(n)})$ iff A is $\emptyset^{(n)}$-c.e.

Let us show (2). By definition, A is Δ^0_{n+1} iff A and $\overline{A}$ are both Σ^0_{n+1}. By the previous point, this is equivalent to saying that A and $\overline{A}$ are $\emptyset^{(n)}$-c.e. By relativizing Proposition 3-7.4 to $\emptyset^{(n)}$, A and $\overline{A}$ are $\emptyset^{(n)}$-c.e. iff $A \leqslant_T \emptyset^{(n)}$. Finally, according to Proposition 3.4, we have $A \leqslant_T \emptyset^{(n)}$ iff A is $\Delta^0_1(\emptyset^{(n)})$.■

Corollary 5.6
The arithmetic hierarchy is strict. In other words,

- *for all $n > 0$, there exists a Σ^0_n set which is not Π^0_n and a Π^0_n set which is not Σ^0_n;*
- *for all $n > 0$, there exists a Δ^0_{n+1} set which is neither Σ^0_n nor Π^0_n.*

Proof. According to Proposition 5.3, the set $\emptyset^{(n)}$ is Σ^0_n. It cannot be Π^0_n in which case it would be Δ^0_n, and therefore computable in $\emptyset^{(n-1)}$ according to Theorem 5.5, contradicting the fact that X never computes its Turing jump. We show the same way that $\mathbb{N} \setminus \emptyset^{(n)}$ is Π^0_n, but not Σ^0_n.

Finally, we can construct for all n the following Δ^0_{n+1} set: the set X such that $X(2m) = \emptyset^{(n)}(m)$ and such that $X(2m+1) = (\mathbb{N} \setminus \emptyset^{(n)})(m)$. It is clear that X is $\emptyset^{(n)}$-computable, and therefore Δ^0_{n+1} according to Theorem 5.5. Finally, if we assume by the absurd that X is Σ^0_n (resp. Π^0_n), this allows to give a Σ^0_n description of $\mathbb{N} \setminus \emptyset^{(n)}$ by keeping only the odd bits of X (resp. a Π^0_n description of $\emptyset^{(n)}$ keeping only the even bits of X), which is in contradiction with the fact that $\emptyset^{(n)}$ is not Π^0_n (resp. with the fact that $\mathbb{N} \setminus \emptyset^{(n)}$ is not Σ^0_n). ■

Exercise 5.7. ($\star$) Show that, for all $X, Y \in 2^{\mathbb{N}}$, we have $X \equiv_T Y$ iff $X' \equiv_m Y'$. ◇

6. Rice's theorem

Rice's theorem basically says that no semantic property of programs is decidable. For example, as seen in Exercise 3-6.4, it is impossible to decide whether a computer program performs a multiplication by 2 or not. We mean *semantic properties* as opposed to *syntactic properties*. The latter are sensitive to code variations, for example "this program has three distinct variables" or "this program contains two loops `for`". Semantic properties do not speak directly about programs, but about the functions they represent. For example, property "this function is defined on input 42" or property "this function only returns even values" do not depend on their code. Semantic properties do not depend on the implementation details of the function.

Definition 6.1. An *index set* is a set A included in $\mathbb{N}$, such that for all $x, y \in \mathbb{N}$,

$$\text{if} \quad x \in A \quad \text{and} \quad \Phi_x = \Phi_y, \quad \text{then} \quad y \in A.$$

Among the index sets, we will note the empty set $\emptyset$ which corresponds to a property never satisfied, and the set $\mathbb{N}$ representing a property that is always true. An index set is *non trivial* if $A \neq \emptyset$ and $A \neq \mathbb{N}$.

Theorem 6.2
If A is a non-trivial index set, then either $\emptyset' \leqslant_m A$ or $\emptyset' \leqslant_m \overline{A}$.

Proof. Let Φ_{e_0} be the nowhere-defined function. Suppose that $e_0 \in \overline{A}$, the other case being treated by symmetry. Since A is non-trivial, A is non-empty. Let's fix a code $e_1 \in A$. In particular, $\Phi_{e_0} \neq \Phi_{e_1}$. By the SMN theorem (see Theorem 3-4.1), there exists a total and computable function $f : \mathbb{N} \to \mathbb{N}$ such that

$$\Phi_{f(x)}(y) = \begin{cases} \Phi_{e_1}(y) & \text{if } x \in \emptyset' \\ \uparrow & \text{if } x \notin \emptyset'. \end{cases}$$

Let us show that the function f is a many-one reduction from $\emptyset'$ to A. If $x \in \emptyset'$, then $\Phi_{f(x)} = \Phi_{e_1}$, and so $f(x) \in A$. Conversely, if $x \notin \emptyset'$, $\Phi_{f(x)} = \Phi_{e_0}$, then $f(x) \in \overline{A}$. ∎

Rice's theorem asserts that no non-trivial property on partial computable functions is decidable. Although the applications of this theorem are relatively limited in computability theory, Rice's theorem is of great importance for the understanding it brings about the nature of computability. In particular, the undecidability of the halting problem is not an isolated phenomenon, because it is shared by all the properties on the partial computable functions.

Corollary 6.3 (Rice's theorem)
Let $\mathcal{C}$ be a class of partial computable functions $\mathbb{N} \to \mathbb{N}$. Then, the set $A = \{x : \Phi_x \in \mathcal{C}\}$ is non-computable unless $\mathcal{C} = \emptyset$ or $\mathcal{C}$ is the class of all partial computable functions.

Proof. The set A is an index set. If $\mathcal{C} = \emptyset$ or $\mathcal{C}$ is the class of all partial computable functions, then $A = \emptyset$ or $A = \mathbb{N}$ and, in both cases, A is computable. If $\mathcal{C}$ is neither empty nor the class of all partial computable functions, then A is non-trivial, and by Theorem 6.2 either $\emptyset' \leqslant_m A$ or $\emptyset' \leqslant_m \overline{A}$. In both cases, $\emptyset' \leqslant_T A$, and A is not computable. ■

7. Arithmetic codes

Sets of integers are generally infinite objects, and therefore cannot be represented by natural integers without some of them being omitted from this representation. This is the object of Cantor's diagonal argument (see Section 2-4). Certain sets of integers can however be described in a finite way, starting with the computable sets.

Definition 7.1. A Δ^0_1 *code* of a computable set A is an integer e such that $\Phi_e = A$ (ie $\forall n\ \Phi_e(n) \downarrow= A(n)$). ♢

Note that a set is computable iff it has a Δ^0_1 code. By the Padding lemma 3-5.1, any computable set is represented by an infinity of Δ^0_1 codes. According to Rice's theorem, the set of Δ^0_1 codes of a fixed computable set is not decidable. There is not even a procedure for deciding whether an integer e is a Δ^0_1 code of a set. Indeed, that would amount to being able to enumerate in a computable way all the computable sets, and would leave place to Cantor's diagonal argument.

Exercise 7.2. ($\star$) Let TOT be the set of Δ^0_1 codes, in other words

$$\text{TOT} = \{e : \Phi_e \text{ is total}\}$$

(assuming without loss of generality that $\Phi_e(n)$ returns 0 or 1 for all n). Prove that TOT is Π^0_2-complete, ie. that $\mathbb{N} \setminus \emptyset'' \leqslant_m \text{TOT}$. ⋄

The Δ^0_1 codes allow to give a new definition of a uniformly computable sequence of sets. Recall that a sequence $X_0, X_1, \dots$ of sets is uniformly computable if the function $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ defined by $f(x, s) = 1$ iff $x \in X_s$ is total computable.

Proposition 7.3. A sequence $X_0, X_1, \ldots$ of sets is uniformly computable if, and only if, there exists a computable sequence $e_0, e_1, \ldots$ such that e_s is a Δ^0_1 code of X_s for all s. ⋆

Proof. $\Rightarrow$. Suppose that $X_0, X_1, \ldots$ is uniformly computable through the function $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$. By the SMN theorem, there exists a total computable function $h : \mathbb{N} \to \mathbb{N}$ such that for all e, x, we have the equality $\Phi_{h(e)}(x) = f(x, e)$. It follows that the sequence $h(0), h(1), \ldots$ is a computable sequence such that $h(s)$ is a Δ^0_1 code of X_s.

$\Leftarrow$. Suppose now that there exists a computable sequence $e_0, e_1, \ldots$ of Δ^0_1 codes. Then, given the existence of a universal machine, the function f defined on $\mathbb{N} \times \mathbb{N}$ by $f(x, s) = \Phi_{e_s}(x) \in \mathbb{N}$ is total computable, and shows that the sequence of sets $X_0, X_1, \ldots$ is uniformly computable. ■

Computably enumerable sets are also representable by a code system. The following notation will often be used for this.

Notation

We denote by W_e the set $\{n \in \mathbb{N} : \Phi_e(n)\downarrow\}$, which is computably enumerable. The notation is relativized to an oracle X, and W_e^X or $W_e(X)$ will thus denote the set $\{n \in \mathbb{N} : \Phi_e(X, n)\downarrow\}$.

Definition 7.4. A Σ^0_1 *code* of a c.e. set A is an integer e such that $W_e = A$. ♢

Still by Rice's theorem, the set of Σ^0_1 codes of a fixed c.e. set is not computable. However, unlike Δ^0_1 codes, *any integer* is a Σ^0_1 code. The computable sets being *a fortiori* computable enumerable, they have both Δ^0_1 and Σ^0_1 codes. The representation by Δ^0_1 codes is however computationally more informative, in the sense that it gives access to more information on the set that it represents. In particular, the partial function $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ such that if e is a Δ^0_1 code of a set A, is defined for $x \in \mathbb{N}$ by $f(e, x)\downarrow= 1$ if $x \in A$ and $f(e, x)\downarrow= 0$ if $x \notin A$, is computable[1], while the equivalent function for the Σ^0_1 codes is not.

Exercise 7.5. Show that there is no function $f : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ which is total, computable and such that $f(e, x) = 1$ if $x \in W_e$ and $f(e, x) = 0$ if $x \notin W_e$. ⋄

In general, we can represent any set of the arithmetic hierarchy by integers using Post's theorem.

[1] If e is not a Δ^0_1 code, the function f still acts as if it were the case, and we will eventually have $f(e, x)\uparrow$.

Definition 7.6. A Σ^0_{n+1} *code* (resp. Δ^0_{n+1}) of a set A is an integer e such that $W_e(\emptyset^{(n)}) = A$ (resp. $\Phi_e(\emptyset^{(n)}) = A$). ♢

Remark

Given a set A, we will tend to favor the type of code which provides the most computational information on A. Thus, if A is computable, it is often better to manipulate a Δ^0_1 code rather than a Σ^0_1 code.

This recommendation also applies to more elaborate computability-theoretic concepts, such as low sets. As a reminder, a set A is low if $A' \leqslant_T \emptyset'$. As $A \leqslant_T A' \leqslant_T \emptyset'$, the low sets are in particular Δ^0_2, and can therefore be represented by a Δ^0_2 code, that is to say an integer e such that $\Phi_e(\emptyset') = A$. However, this representation loses information specific to low sets (see Exercise 7.11), such as the ability of $\emptyset'$ to decide A'. It is therefore preferable to represent the set A by a code e such that $\Phi_e(\emptyset') = A'$.

Definition 7.7. A *lowness code* of a set A is an integer e such that $\Phi_e(\emptyset') = A'$. ♢

Finally, in the case of finite sets, it is possible to store more information than the simple fact of being computable. Indeed, given a sequence $e_0, e_1, \dots$ of Δ^0_1 codes of finite sets, it is not possible to compute uniformly the cardinality of these sets (see Exercise 7.10). We will therefore prefer the notion of canonical code which notably contains information on the size of the set.

Definition 7.8. The *canonical code* of a finite set F is the natural integer $\sum_{i \in F} 2^i$. ♢

We can easily verify via the following exercise that a canonical code contains all the information of a finite set, including the possibility of knowing its last element.

Exercise 7.9. Let $D_0, D_1, \dots$ be the sequence of finite sets such that D_n has canonical code n.

1. Show that the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n) = |D_n|$ is computable.
2. Show that the function $g : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ defined by $g(n, x) = 1$ iff $x \in D_n$ is computable.

◇

The following two exercises allow us to show that the information given to us by canonical codes of finite sets or by lowness codes cannot be obtained via computable or Δ^0_2 codes.

Exercise 7.10. ($\star\star$) Suppose absurdly that there exists a partial computable function $f : \mathbb{N} \to \mathbb{N}$ such that if e is a Δ^0_1 code of a finite set, then $f(e)$ returns the size of this set. Using the fixed point theorem, show that there exists a Δ^0_1 code a of a finite set such that the size of this set is different from $f(a)$. $\diamond$

Exercise 7.11. ($\star\star$) Suppose absurdly that there exists a partial computable function $f : \mathbb{N} \to \mathbb{N}$ such that if e is a Δ^0_2 code of a low set X, then $f(e)$ returns a Δ^0_2 code for X'. Using the fixed point theorem, show that there exists a Δ^0_2 code a of a computable set X such that $f(a)$ is the Δ^0_2 code of a different set of X'. $\diamond$

Chapter 6

Church-Turing thesis

1. The Entscheidungsproblem and the quest for the Grail

In the year 1928, in a period troubled by deep questioning of the foundations of mathematics, David Hilbert and Wilhelm Ackermann asked the question of the existence of an algorithm allowing to decide the validity of any mathematical statement. Here, the word *algorithm* is to be taken in a broad sense, to designate a set of elementary computation steps that a mathematician can perform. This challenge raised to logicians, and passed down to posterity under the name of *Entscheidungsproblem* — decision problem —, marks the beginning of a long foundational quest on the formalization of the notion of algorithm.

The incompleteness theorems of Gödel[1] proved in 1931, stating the existence of fundamentally undecidable statements in any reasonable theory allowing formalization of arithmetic, were particularly unwelcome in a a period which sought to initiate a new wave of optimism and confidence in mathematics. They also tipped the scales towards the existence of a negative solution to the Entscheidungsproblem.

A positive response to the Entscheidungsproblem would have been an algorithm or a series of deterministic steps, allowing to demonstrate any mathematical statement or its negation. A negative answer consists in the proof that such a systematic method cannot exist. This direction poses a completely different problem, namely to formally define the concept of

[1]See Chapter 9

algorithm, or in a roughly equivalent way, to find a robust and consensual formalization of the concept of computable function.

It should be noted that the first computer, the ENIAC, was built in 1940, a decade after the formulation of the Entscheidungsproblem. The notion of effectively computable function therefore does not refer to what can be computed by a computer, but by the human mind. It was a question of finding a systematic process, or algorithm in the informal sense of the term, allowing a mathematician to answer any mathematical question.

Several definitions were proposed in the years that followed, each conjectured more or less convincingly as exactly capturing the epistemological notion of effectively computable function. These definitions were fairly quickly proved to be equivalent, but struggled to convince the scientific community of their capacity to capture all the functions that could be effectively computed. It was not until 1936 that Alan Turing came to a consensus by presenting his computational model, the *Turing machine*, with a striking demonstration of its equivalence with other models, in particular with the one used by Gödel to show his famous incompleteness theorem, thus providing a definitive answer to the Entscheidungsproblem. The reader interested in the history of computability theory will find an excellent chapter dedicated to the subject in the work *Turing computability: Theory and Applications* by Robert Soare.

Although the different computational models have since been proven to be equivalent, we will detail the main among those which marked this history, each one presenting its own interest, by emphasizing a different aspect of the notion of computable function. In what follows, we will only consider partial functions, from $\mathbb{N}^n$ to $\mathbb{N}$ for $n \geqslant 1$. We will write $g(\overline{x}) \downarrow = y$ to mean that g is defined on $\overline{x} \in \mathbb{N}^n$ and is equal to y; the notation $\overline{x}$ being a shorthand for $x_1, \ldots, x_n$.

General recursive functions. Recursive functions were introduced in a restricted form by Gödel as part of his incompleteness theorems in 1931. This class of functions was generalized by Gödel and Herbrand in 1934 to obtain the general recursive functions, capturing according to the thesis of Church-Turing — thesis detailed in the following section — the entirety of the computable functions.

The idea underlying the definition of general recursive functions is very simple: start from a few elementary functions, the computability of which leaves no room for doubt, then combine them to obtain new, more complex functions, which are still computable. What are the valid combinations for creating new functions? If two functions are computable, their composition should naturally be computable: it suffices to execute in series the steps of computing each of the functions. In a slightly less obvious way, functions defined by recursion from computable functions are still computable. Indeed, if we define a function $f : \mathbb{N} \to \mathbb{N}$ by $f(0) = v$ for an integer v, and $f(n+1) = g(n, f(n))$ for a computable function $g : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$, to obtain the value of $f(n)$, it suffices to successively compute $f(1), f(2), \ldots, f(n)$ using the preceding values and by following the steps of computing the function g. Finally, if a function $g : \mathbb{N} \to \mathbb{N}$ is computable, the search for the smallest input n such that $g(n) = 0$ is also computable, via a loop which increments n until having $g(n) = 0$ (this research may never be successful).

Definition 1.1. The class $\mathcal{C}$ of *general recursive functions* is the smallest class of partial functions containing the following basic functions:

(a) The successor function: $\mathrm{succ}(x) = x + 1$

(b) The constant functions: $c^n_m(x_1, \ldots, x_n) = m$ for all $n, m \geqslant 0$

(c) The projections: $p^n_i(x_1, \ldots, x_n) = x_i$ for all n and all $i \in 1, \ldots, n$

and which is closed under the following operations:

(i) Composition: if $g_1, \ldots, g_m \in \mathcal{C}$ are functions of n variables and $h \in \mathcal{C}$ is a function of m variables, then the function

$$f(\overline{x}) = h(g_1(\overline{x}), \ldots, g_m(\overline{x}))$$

is in $\mathcal{C}$.

(ii) Primitive recursion: if $g, h \in \mathcal{C}$ are partial functions of respectively n and $n+2$ variables, the function f defined by $f(\overline{x}, 0) = g(\overline{x})$ and $f(\overline{x}, m+1) = h(\overline{x}, m, f(\overline{x}, m))$ is a partial function of $n + 1$ variables in $\mathcal{C}$.

(iii) Minimization: if $g \in \mathcal{C}$ is a partial function of $n + 1$ variables, then the partial function f which to $\overline{x}$ associates the smallest integer m, if it exists, such that $g(\overline{x}, i) \downarrow$ for all $i \leqslant m$ and such that $g(\overline{x}, m) = 0$, is in $\mathcal{C}$. If m does not exist, then f is not defined on $\overline{x}$.

The class of *primitive recursive functions* is the smallest class of total functions containing the functions of (a), (b), (c) and closed under the schemes (i) and (ii) of composition and primitive recursion. ♢

General recursive functions have the advantage of highlighting the closure properties of the notion of computable function, starting with closure under composition. Note that the primitive recursive functions are total, unlike the general recursive functions, for which the minimization scheme introduces a possibility of partiality. Intuitively, its programming implementation would consist of a loop `while` exhaustively searching for an integer m

satisfying the property. If this integer does not exist, program execution will never exit the loop, and the program will not halt. We will study this model in detail in Section 3 by trying to show that it corresponds well to the notion of effectively computable function. Let us mention before that two other computation models.

λ-calculus. Defined by Church in the 1930s, λ-calculus is a minimalist formalism serving as a theoretical foundation for programming languages. Unlike general recursive functions, which handle two types of objects, namely integers and functions on integers, λ-calculus has only one primitive object: the *λ-functions*. These do not take integers as parameters, but λ-functions. It will therefore be necessary to resort to the coding of integers by λ-functions to define a notion of computable function on integers.

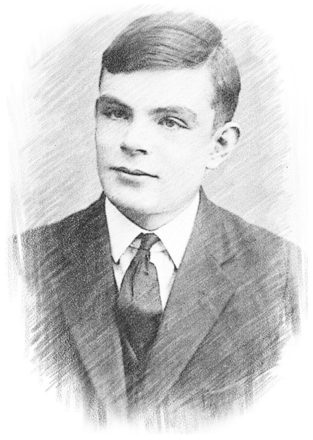
λ-functions are defined by an expression language, as it is often the case in mathematics. For example, $x, y \mapsto x + y$ is an expression defining the sum of two integers. However, in λ-calculus, the parameters themselves being λ-functions, the only valid operations are those for manipulating functions, namely, adding a parameter to a function, and the application of a function to its parameters. For example, $f \mapsto (g \mapsto f(g))$ defines the function which takes as parameter a function f, and returns the function which takes as parameter a function g, and returns the result of applying f to g.

The minimalist aspect of λ-calculus facilitates reasoning on the formalism itself, but requires much more work to define non-trivial functions on integers. In particular, it is more difficult to convince oneself that all the computable functions can be represented by λ-functions. As expressed previously, one needs to fix a convention to represent the integers. It seems quite natural to identify the integer n with the λ-function taking as input a λ-function f and returning its n-th iteration. For example, the integer 0 is represented by the function which takes as input a function f, and returns its 0th iteration, in other words returns the function identity. In the formalism of λ-calculus, it is written $f \mapsto (x \mapsto x)$. Likewise, the integer 2 corresponds to the function $f \mapsto (x \mapsto f(f(x))$. Let us call $\overline{n}$ the λ-function representing the integer n. A function $g : \mathbb{N} \to \mathbb{N}$ is *λ-definable* if there exists a λ-function h which to $\overline{n}$ associates $\overline{g(n)}$.

The syntax of λ-calculus is rather abstruse at first glance, and requires a little manipulation to become familiar with the concepts. Nowadays, many variations and enrichments are studied in order to provide a theoretical basis for functional programming languages. This is a very active area of research.

Turing machines. If λ-calculus provides a theoretical basis for programming languages, Turing machines can be seen as precursors of the modern computer.

Designed in 1936 by Alan Turing, this machine was inspired by his father's typewriter. It has a *tape*, which can be seen as a succession of *cells* indexed by integers. We can make the analogy with bits which are accessed in the memory of a computer via an addressing system. The machine also has a read/write head which can move from one cell to another, then read or modify its content. The machine will run based on an input n. At the start of the computation, the read head is at the start of the tape, the first n cells of which are initialized to 1, while the remaining cells are equal to 0.


Alan Turing, 1912–1954

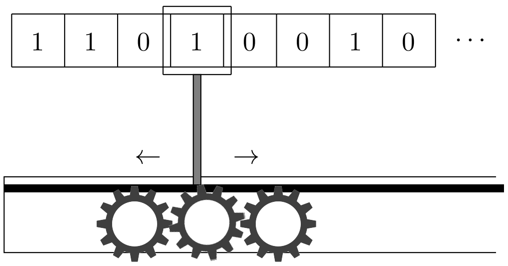


Figure 1.3: Representation of a Turing machine, with its head, moving along the cells of the tape

The machine also has a finite number of *states*, including a start state in which it is at the start of the computation, as well as a final state which indicates the end of the computation when the machine is in it. The movements of the read head, the replacement of the value of the current cell, as well as the changes of states, are subject to a set of instructions represented as follows: given the current state of the machine and the value from the cell where the head is located, a rule will decide the next state of the machine, possibly change the value of the cell, and move the head one cell to the right or to the left. A Turing machine is therefore an elaborate automaton designed to perform a specific task. In his founding article, Turing demonstrated the existence of a *universal* Turing machine: a machine able to simulate the computation of any other Turing machine.

Unlike general recursive functions or λ-calculus, Turing's model constitutes a very concrete mechanical process. We can in fact actually build a universal Turing machine. This model has in particular the advantage of making explicit the notion of atomic step of computation as well as of quantifying the memory space used. For these reasons, Turing machines are taken as a reference model to define the theory of Algorithmic Complexity.

Getting away from models. Like a quote from Michael Fellows[2], computability theory is no more the study of computational models than astronomy is the science of telescopes. Computability is very quickly freed from the details of implementation of computational models, to manipulate programs in an abstract way. It is nevertheless instructive to see in detail at least one model of computation, in particular to forge an intuition when it is lacking, and this is what we will do very soon in Section 3.

2. Church-Turing thesis

In 1934, facing the success of λ-calculus, Church submitted the idea to Gödel that λ-definable functions would capture the notion of an effectively computable function, leaving Gödel doubtful. With the development of the general recursive functions of Herbrand and Gödel the same year and the proof of their equivalence with the λ-calculus, Church publicly formulated his thesis, known as *Church thesis*, asserting that general recursive functions coincided with effectively computable functions. His argument did not convince Gödel, although he was one of the instigators of general recursive functions. With his eponymous machine model, Alan Turing finally reached consensus in 1936, by demonstrating that Turing machines were equivalent to λ-calculus model and to general recursive functions, which led to what is called today the Church-Turing thesis.

Thesis 2.1 (Church-Turing).
The effectively computable functions are those computable by a Turing machine, or equivalently the general recursive functions or the λ-definable functions. ■

If Church was the first to formulate the thesis according to which the λ-definable functions corresponded to the effectively computable functions, we generally attribute the fathership of computability theory to Turing. The Church-Turing thesis not being a mathematical statement, it is not possible to prove it formally. It can nevertheless be validated by what comes closest to a proof in the social sense of the term, that is to say by an argument generating a consensus in the scientific community. This is what Turing achieved by the following proof.

[2] "Computing is no more the study of computers than astronomy is that of telescopes." [62]

Turing's proof. To justify his thesis, Turing resorted to three types of arguments: (1) a description of the process by which a mathematician performs a computation and its formalization by a Turing machine, (2) the proof of the equivalence of Turing machines with existing computational models, (3) the explicit development of large classes of functions computable by Turing machines. Here is the outline of Turing's first argument:

Consider a mathematician, Mr. Smith, performing a computation. Mr. Smith has a pencil, and a potentially unlimited amount of paper. Given the finite precision of his pencil, each sheet can only contain a finite number of symbols. For simplicity, and without loss of generality, we can consider that each sheet is a cell of an infinite tape, containing only one symbol belonging to a sufficiently large finite alphabet. The computation process is as follows: while he is in a mental state e_0, Mr. Smith is located in front of the current sheet, in his field of vision. He reads the notes, before correcting them, erasing and changing the symbol written on the sheet. This reading will change his thoughts, and he will find himself in a mental state e_1. He will potentially turn the page, or go back to reread previous notes, until he reaches the end of his computation. Mr. Smith will then consider his computation finished, and will find himself in the corresponding mental state which we will call *final state*.

Proper use of the Church-Turing thesis. It is important to get a clear idea of what the Church-Turing thesis says, its limits and its use. Church-Turing's thesis is neither a theorem nor a conjecture. It cannot be formally proven or refuted, by the simple fact that it is not a mathematical statement, but rather a bridge between a mathematical concept and an epistemological concept. This thesis is not, however, an arbitrary assertion, for it is supported by reasoning which can be subject to criticism.

If Church-Turing's thesis can be called into question, and even one day mostly rejected, the development of computability theory nonetheless rests on solid foundations, independent of this correspondence. The equivalence between functions computable by Turing machines, general recursive functions, λ-definable functions, and functions programmable in C, Java or Python, is indeed a theorem which does not depend on the Church-Turing thesis. While it is common in computability theory to informally describe an algorithm and then deduce the existence of a Turing machine implementing it, this process is not strictly speaking a call to the Church-Turing thesis. Rather, it is an informal proof to convince the interlocutor that, if necessary, it would be easy to program this algorithm in any programming language.

Moreover, if Church-Turing's thesis were to be invalidated, the conceptual edifice built around computability theory would remain, and would likely continue to be studied. There is a hierarchy of formal languages and computational models, called *Chomsky hierarchy*. We find there for example the *rational languages*, corresponding to the languages recognized by a class of machines called *finite automata*. Although these models are for the most part less expressive than the Turing machines, they are nonetheless a very active subject of research today. If one day new computational paradigms were found, the existing notion of computable function would nonetheless remain a very robust class of functions, and would likely continue to be studied in the same way as rational languages or any other level of Chomsky hierarchy.

Some natural phenomena are studied in the hope of solving non-computable problems. These notions of computability are united under the name of *hypercomputing* (we will see a formal approach of it in Part IV). To date, there is no prospect of making such computations. The discovery of new computational phenomena in nature would however probably not invalidate the Church-Turing thesis because they would have little chance of satisfying the definition of an effectively computable function according to Rosser [192], i.e., say "a method in which each step is precisely predetermined and which will reliably produce an answer in a finite number of steps."

3. Detailed study of recursive functions

The objective of this section is to convince the reader that general recursive functions coincide with the effectively computable functions as defined informally in sections 3-2 and 3-1, that is, "functions that can be programmed ". The interests of such a study are manifold. In the first place, it makes it possible to have a precise mathematical definition of what a computable function is: without loss of generality, it is a general recursive function. Then the study which we give will provide at the same time a mathematical proof of the existence of a universal machine such as defined in Theorem 3-3.1, concept used throughout this work. Finally, our study will isolate the notion of primitive recursive function as a strict subclass of computable functions; besides a certain epistemological importance (see Theorem 3.22), this subclass presents an undeniable interest in mathematical logic. We will see an example of its use in the study of reverse mathematics, in Section 23-7.

3.1. Register machines and programming diagrams

In order to convince ourselves that the general recursive functions coincide with the computable functions, we start from a computational model close

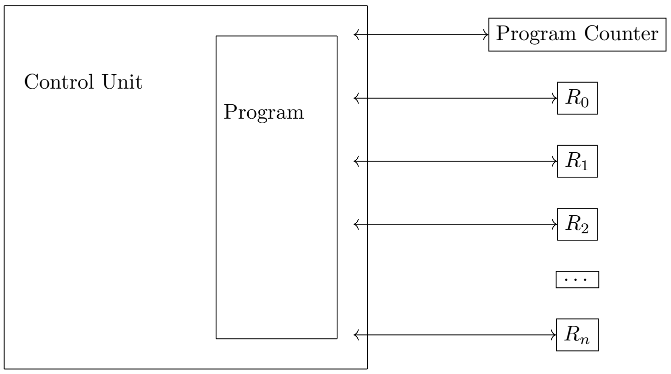


Figure 3.1: Model of register machines

to modern programming languages: *structured programs*, executed by *register machines*. The developments in this section comes in broad outline from the computability theory course of Arnaud Durand and Paul Rozière [54] of the Master of mathematical logic of the University of Paris Diderot.

Register machines. The scientific literature has declined several versions, under the name of "random access machine" (Melzak [160], Lambek [139], Shepherdson and Sturgis [205], Peter [182], Elgot and Robinson [58]). These are machines known as "random access memory" or RAM, a generic name today to designate the random access memory of computers. The term "random access" should be understood as opposed to "sequential access", and refers to the fact that each memory slot can be accessed directly from its address, unlike, for example, the model of Turing machines, for which a read head must move cell after cell in memory to arrive at the desired location.

The memory of a register machine will however be more elementary than modern RAM: it is simply a finite number of *registers* $R_0, R_1, \ldots, R_k$ for $k \in \mathbb{N}$ arbitrary. Each register can contain any positive or zero integer. Note that the integers can be arbitrarily large, and therefore that each register represents an "unbounded" memory space.

Structured programs. A register machine will execute a program which consists of a finite list of instructions for performing computations. There are many possible variations on the instruction set that one allows oneself. We present one deliberately close to that of a modern imperative programming language.

Definition 3.2. A *structured program* can contain the following instructions:

1. - Incrementation of a register: "$R_i := R_i + 1$".
   - Decrementation of a register "$R_i := R_i - 1$".
   - Assignment of a register: "$R_i := x$" for $x \in \mathbb{N}$ or "$R_i := R_j$"

   These instructions respectively increment the value of R_i by 1, decrement it by 1 (unless the value is 0 in which case nothing happens), set the value of R_i to x or set it to that of R_j).

2. The conditional statement: "if $R_i = 0$ then S else S'", where $S = (S_0, \dots, S_n)$ and $S' = (S'_0, \dots, S'_m)$ are finite sequences of structured instructions.

   Each instruction in S is executed sequentially if R_i is equal to 0. Otherwise each instruction in S' is executed sequentially.

3. The for loop statement: "for $i = 1$ to R_i do S", where $S = (S_0, \dots, S_n)$ is a finite sequence of structured instructions.

   Let N be the number present in register R_i when the program starts this instruction. Each S instruction is executed sequentially, all N times. Note that if the value of R_i changes while the instructions of S are being executed, this does not change the number of times the loop occurs.

4. The while loop statement: "while $R_i \neq 0$ do S" where $S = (S_0, \dots, S_n)$ is a finite sequence of structured instructions.

   Each instruction in S is executed sequentially as long as the register R_i is different from 0.

The execution of a structured program halts after the last instruction has been executed. ♢

Note that a structured program has only a finite number of instructions and can therefore only use a finite number of registers.

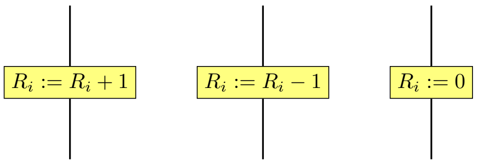
`for` loop vs `while` loop

The reader may notice that the `for` loop statement is redundant in that it can always be replaced by a `while` loop statement. We will see that the reverse is not true. In particular the number of times that a `for` loop is executed is necessarily finite, which is not the case for `while` loops, within which a computation can get stuck in what is classically called in programming *an infinite loop*. We will see that the `while` loop instruction is essential, and that some computable (and total) functions cannot be programmed using only `for` loops.

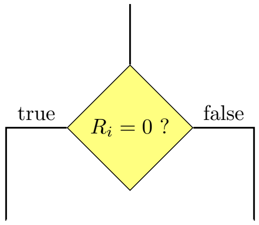
Programming diagrams. Structured programming finds its genesis in the work of Goldstine and von Neumann [78] who, from 1946, show a concern no longer to capture mathematically the notion of computation, but that of developing a programming system. They develop a way of presenting algorithms based on *programming diagrams*. We give here the simplified presentation that one finds in the reference work of Piergiorgio Odifreddi [177].

Definition 3.3. A programming diagram is obtained by connecting between them basic bricks of two types:

- Assignment instructions:



- A conditional instruction:



A programming diagram has one input and one or more outputs. The computation is carried out linearly by executing each block from the input to one of its outputs. $\diamondsuit$

By way of example, Figure 3.4 is a programming diagram corresponding to the addition function.

Programming diagrams can be programmed on register machines with an instruction set including conditional and unconditional jumps (the latter simply called "jumps"), that is, instructions of type *goto*, which allow to determine which is the next instruction of the program which will be executed. It is still today the mechanism at work in the different assembly languages of micro-processors.

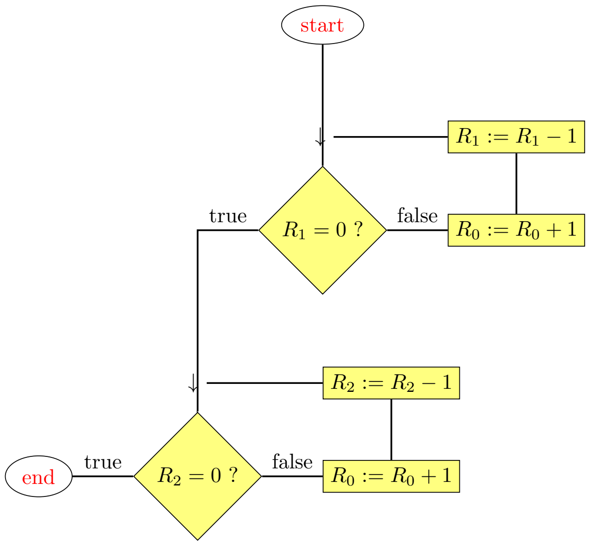


Figure 3.4: A programming diagram for the addition function of two integers. One supposes that the computation starts with $R_0 = 0$, $R_1 = n_1$, $R_2 = n_2$. At the end of the execution, one will have R_0 equal to $n_1 + n_2$.

Definition 3.5. A *goto* program is a numbered sequence of instructions $I_0, I_1, \ldots, I_n$ where each instruction is of one of the following types:

1. - Incrementation of a register: "$R_i := R_i + 1$".
   - Decrementation of a register "$R_i := R_i - 1$".
   - Assignment of a register: "$R_i := 0$".
2. The conditional jump instruction: "if $R_i = 0$ goto n_1 else goto n_2".
   If R_i is equal to 0, instruction number n_1 is the next to be executed, otherwise it is instruction number n_2.
3. The unconditional jump: "goto m".
   Instruction number m is the next to be executed.

The execution of a goto program halts when there is no more next instruction to execute (which in particular can happen after an instruction of type "goto m" in a program with less than m instructions.) ♢

Remark

Note that the unconditional jump "goto m" can be replaced by a conditional jump "if $R_i = 0$ goto n else goto n". A *goto* program can therefore be expressed without instruction 3.

It is clear that programming diagrams are interchangeable with goto type programs, and we will use one formalism or the other depending on the situation.

Computable functions. Notations $\Phi_e(x_1, \dots, x_n) \downarrow = y$ and $\Phi_e(x_1, \dots, x_n) \uparrow$ used throughout the book naturally apply to programs executed by register machines, once the conventions for passing parameters and retrieving the result have been fixed:

Definition 3.6. Given a structured or goto type program P, we write $P(x_1, \dots, x_k)$ to denote the execution of P with the registers $R_1, \dots, R_k$ initialized to $x_1, \dots, x_k$, and all the other registers initialized to 0. We write $P(x_1, \dots, x_k) \downarrow = x$ to signify that such an execution halts with the value x in the R_0 register, and $P(x_1, \dots, x_k) \uparrow$ to signify that the execution does not halt. ♢

We can now use our computational model to give a precise mathematical definition of a computable function. Let us formalize beforehand the notations $\operatorname{dom} f$ and $\operatorname{Im} f$ which denote respectively the domain and the image of a function f.

Notation

Given a (possibly partial) function $f : A \to B$, we denote by $\operatorname{dom} f$ the domain of definition of f, and $\operatorname{Im} f = \{Y \in B : \exists X \in \operatorname{dom} f \ f(X) = Y\}$ its image.

Definition 3.7. A (possibly partial) function $f : \mathbb{N}^n \to \mathbb{N}$ is *computable by structured program* (resp. *by goto program*) if there is a structured program (resp. a goto program) P_f, such that $P_f(x_1, \dots, x_n) \downarrow = f(x_1, \dots, x_n)$ for all $x_1, \dots, x_n \in \operatorname{dom} f$ and such that $P_f(x_1, \dots, x_n) \uparrow$ for all $x_1, \dots, x_n \notin \operatorname{dom} f$. ♢

Justification of the model of computation. The programmer will perhaps not be convinced by the fact that the model of register machines with structured programs (or goto) indeed allows to program all the functions which he could write in his favorite language.

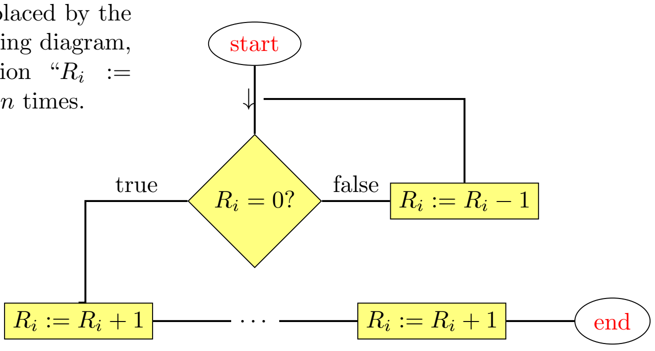
One aspect in particular may cause concern: A modern programming language allows the use of arrays, and even dynamic arrays, which can increase in size as needed. We will see for example in Definition 3.24 that the function known as Ackermann is computed naturally using a stack structure, and that it is not clear at all that we can do without it. Fortunately, we will show with Proposition 3.26 that it is perfectly possible to simulate the manipulations of dynamic arrays within register machines, by coding the latter in registers, which, let us remember, can contain arbitrarily large integers.

Simulation of structured programs by goto programs. We start by showing that goto programs, despite their simplicity, are sufficient to compute everything that structured programs can compute.

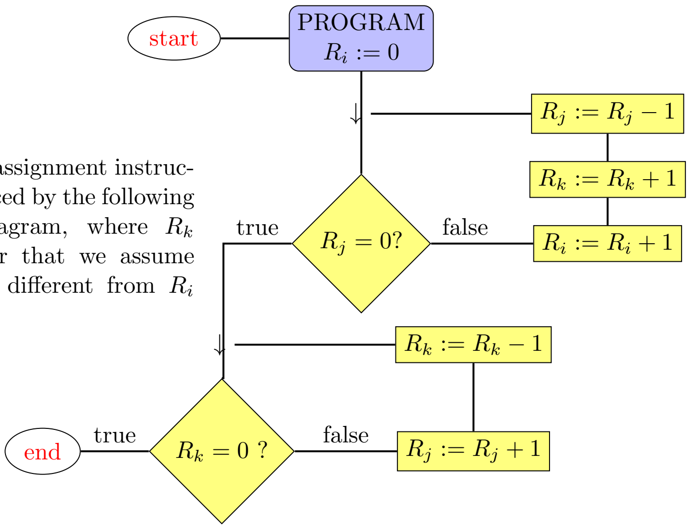
Proposition 3.8. Let $n \in \mathbb{N}^*$ and $f : \mathbb{N}^n \to \mathbb{N}$ be a function computable by a structured program. Then, f is computable by a goto program. ⋆

Proof. It suffices to show that each instruction of a structured program can be replaced by a programming diagram.

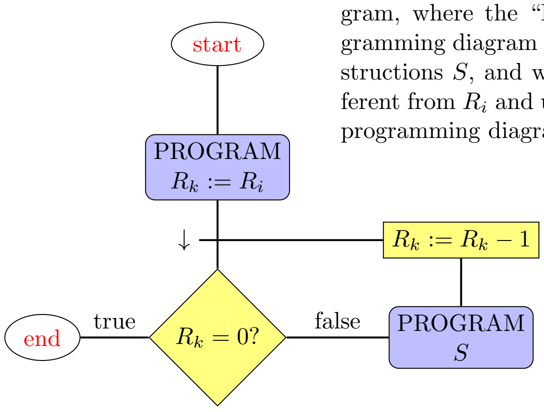
The "$R_i := n$" assignment instruction can be replaced by the following programming diagram, where the instruction "$R_i := R_i + 1$" is repeated n times.



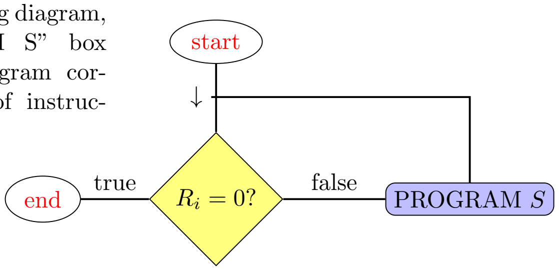
The "$R_i := R_j$" assignment instruction can be replaced by the following programming diagram, where R_k is a new register that we assume to equal 0, and different from R_i and R_j.

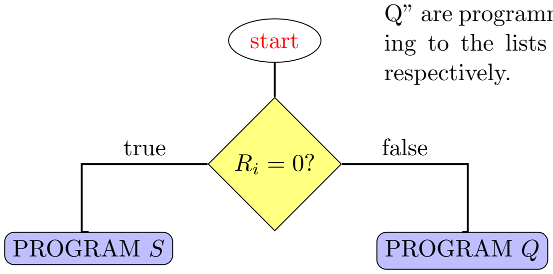


The "for $i = 1$ to R_i do S" loop instruction can be replaced by the following programming diagram, where the "PROGRAM S" box is a programming diagram corresponding to the list of instructions S, and where R_k is a new register different from R_i and unused in the "PROGRAM S" programming diagram.



The "while $R_i \neq 0$ do S" loop instruction can be replaced by the following programming diagram, where the "PROGRAM S" box is the programming diagram corresponding to the list of instructions S.





The "if $R_i = 0$ then S else Q" conditional instruction can be replaced by the following programming diagram, where both boxes "PROGRAM S" and "PROGRAM Q" are programming diagrams corresponding to the lists of instructions S and Q, respectively.

■

3.2. Recursive functions are computable

Structured programs follow the concepts of so-called *imperative* programming: instructions modifying the state of the machine are executed one after the other. The general recursive functions follow the paradigm of so-called *functional* programming: a program is a composition of mathematical functions and a computation is the evaluation of these functions. An advantage of functional programming often highlighted is the absence of *side effects*: the result of a function depends on its parameters and only on its parameters (the state of the machine on which the function is executed has no effect). For example, you can connect the output of one function to the input of another without expecting any unpleasant surprises. By way of comparison, the combination of structured programs P_f, P_g computing functions $f, g : \mathbb{N} \to \mathbb{N}$ in a program computing the function $x \mapsto f(g(x))$ requires a little work, in order to avoid side effects.

Definition 3.9. A structured or goto type program is *clean* if it finishes its computation with all its registers — except R_0 — in the same state as at the start of the computation. ♢

Exercise 3.10. Show that for any structured program M, there exists a clean structured program N such that $M(\overline{x}) \downarrow= y \leftrightarrow N(\overline{x}) \downarrow= y$. ⋄

Theorem 3.11 (Wang [238], Peter [183], Ershov [60])
Any (possibly partial) general recursive function can be computed by a

structured program. Any primitive recursive function can be computed by a structured program with no `while` *loops.*

PROOF. We leave it to the reader to show that the constant functions, the projection functions and the successor function are all computable by a structured program. The following cases remain to be dealt with:

Composition scheme. Let

$$f(x_1, \ldots, x_m) = g(h_1(x_1, \ldots, x_m), \ldots, h_k(x_1, \ldots, x_m))$$

for functions $g : \mathbb{N}^k \to \mathbb{N}$ and $h_1, \ldots, h_k : \mathbb{N}^m \to \mathbb{N}$ computable by structured programs $G, H_1, \ldots, H_m$. By Exercise 3.10 we can assume that $G, H_1, \ldots, H_m$ are proper. Suppose each of these programs uses at most the registers $R_0, \ldots, R_z$ for $z > m$. The program F for computing f is given by the following sequence of instructions.

```
Program $F$
$\langle$Instructions of $H_1\rangle$
$R_{z+1} := R_0$
$R_0 := 0$
$\langle$Instructions of $H_2\rangle$
$R_{z+2} := R_0$
$R_0 := 0$
...
$\langle$Instructions of $H_k\rangle$
$R_{z+k} := R_0$
$R_0 := 0$
$R_1 := R_{z+1}$
...
$R_k := R_{z+k}$
$\langle$Instructions of $G\rangle$
```

Note that if $G, H_1, \ldots H_m$ does not use a `while` loop, then the program to compute F does not use one either.

Primitive recursion scheme. Let

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y+1) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)) \end{aligned}$$

for g and h computable respectively by clean structured programs G and H, using at most the registers $R_0, \ldots, R_z$ for $z > n+2$. The following program F allows to compute f:

```
Program $F$
$R_{z+1} := R_{n+1}$
⟨Instructions of G⟩
$R_{n+1} := 0$
$R_{n+2} := R_0$
for $i = 1$ to $R_{z+1}$ do
    $R_0 := 0$
    ⟨Instructions of H⟩
    $R_{n+1} := R_{n+1} + 1$
    $R_{n+2} := R_0$
end
```

Note once again that if G, H does not use `while` loops, then the program to compute F does not use any either.

Minimization scheme. Let

$$f(x_1, \dots, x_n) = \min\{x \in \mathbb{N} : \forall i \leqslant x\ (g(x_1, \dots, x_n, i)\downarrow \wedge g(x_1, \dots, x_n, x) = 0)\}$$

for g computable by a clean program G, using at most the registers $R_0, \dots, R_z$ for $z > n+1$. The following program computes f:

```
Program $F$
$R_{z+1} := 1$
$R_{n+1} := 0$
while $R_{z+1} \neq 0$ do
    $R_0 := 0$
    ⟨Instructions of G⟩
    $R_{z+1} := R_0$
    $R_{n+1} := R_{n+1} + 1$
end
$R_{n+1} := R_{n+1} - 1$
$R_0 := R_{n+1}$
```

This concludes the proof. ■

3.3. Study of primitive recursive functions

We now move on to the more difficult part of this chapter. In order to show that the functions computable by structured programs are general recursive functions, we need a certain number of tools, in particular on the primitive recursive functions. We alternate different propositions and exercises allowing to see that it is a large class of functions. Let us recall the notations of Definition 1.1 for the basic primitive recursive functions:

Notation

We denote by $p_i^n : \mathbb{N}^n \to \mathbb{N}$ the projection such that $p_i^n(x_1, \dots, x_n) = x_i$. We denote by $c_i^n : \mathbb{N}^n \to \mathbb{N}$ the constant function such that $c_i^n(x_1, \dots, x_n) = i$. We denote by $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$ the successor function, defined by $\mathrm{succ}(n) = n + 1$.

Exercise 3.12. ($\star$) Show that addition, multiplication and the exponential function are primitive recursive. $\diamond$

Exercise 3.13. ($\star$) Show that the predecessor and subtraction functions are primitive recursive (as our functions are valued in $\mathbb{N}$, we will use 0 instead of the result if it is negative). $\diamond$

Exercise 3.14. ($\star$) Show that the function $\mathrm{sg} : \mathbb{N} \to \mathbb{N}$ which associates 0 with 0 and which associates 1 with all other integers, is primitive recursive. Show that the function $\overline{\mathrm{sg}} : \mathbb{N} \to \mathbb{N}$ which associates 0 with 1 and which associates 0 with all other integers, is primitive recursive. $\diamond$

Definition 3.15. A predicate $P \subseteq \mathbb{N}^n$ is primitive recursive (resp. general recursive) if there exists a primitive recursive (resp. general recursive) function $f : \mathbb{N}^n \to \{0, 1\}$ such that $(x_1, \dots, x_n) \in P \leftrightarrow f(x_1, \dots, x_n) = 1$ for all $x_1, \dots, x_n \in \mathbb{N}$. $\diamondsuit$

Example 3.16. The comparison predicates $\leqslant, <, \geqslant, >, =, \neq$ are primitive recursive via the following functions:

$$\begin{array}{rcl@{\qquad}rcl} a \leqslant b & = & \mathrm{sg}(\mathrm{succ}(b) - a) & a > b & = & b < a \\ a < b & = & \mathrm{sg}(b - a) & a = b & = & (a \leqslant b) \times (b \leqslant a) \\ a \geqslant b & = & b \leqslant a & a \neq b & = & \overline{\mathrm{sg}}(a = b) \end{array}$$

Formally, the projections are used if necessary, for example to reverse the order of the parameters in the definition of $\geqslant$ from that of $\leqslant$.

Proposition 3.17. The primitive recursive functions are closed under definition by case on a primitive recursive predicate: if g and h are two primitive recursive functions from $\mathbb{N}^p$ to $\mathbb{N}$, and if P is a primitive recursive predicate on $\mathbb{N}^p$, then the function

$$\begin{array}{rcll} f(x_1, \dots, x_p) & = & g(x_1, \dots, x_p) & \text{if } P(x_1, \dots, x_p) \\ & = & h(x_1, \dots, x_p) & \text{otherwise} \end{array}$$

is primitive recursive. ⋆

Proof. Since P is a primitive recursive predicate, there exists a function $d : \mathbb{N}^n \to \mathbb{N}$ such that

$$\begin{array}{rcll} d(x_1, \dots, x_p) & = & 1 & \text{if } P(x_1, \dots, x_p) \\ & = & 0 & \text{otherwise.} \end{array}$$

We therefore define:

$$\begin{array}{rcl} f(x_1, \dots, x_p) & = & g(x_1, \dots, x_p) \times \text{sg}(d(x_1, \dots, x_p)) + \\ & & h(x_1, \dots, x_p) \times \overline{\text{sg}}(d(x_1, \dots, x_p)) \end{array}$$ ■

The proof of the following proposition provides an example of application of the definition by case.

Proposition 3.18. The integer division is primitive recursive. ⋆

Proof. We use the primitive recursion scheme coupled with the case definition scheme. We define $\text{div}(a, b) = \text{div}(a, b, a)$ where:

$$\begin{array}{rcll} \text{div}(a, b, 0) & = & 0 & \\ \text{div}(a, b, n+1) & = & \text{succ}(n) & \text{if } \text{succ}(n) \times b = a \\ & = & \text{div}(a, b, n) & \text{otherwise.} \end{array}$$ ■

Exercise 3.19. (⋆) Show that the primitive recursive predicates are closed under conjunction, disjunction, negation, bounded existential quantification and bounded universal quantification. ◇

Exercise 3.20. (⋆) Show that the primitive recursive functions are closed under bounded minimization: if $f : \mathbb{N}^{p+1} \to \mathbb{N}$ is primitive recursive, then the function $g : \mathbb{N}^{p+1} \to \mathbb{N}$ which to $x_1, \dots, x_p, n$ associates the smallest $t \leqslant n$ such that $f(x_1, \dots, x_p, t) = 0$ (and 0 if such a $t \leqslant n$ does not exist), is primitive recursive. ◇

Let us remember Cantor's bijections as defined in Section 2-3. The encoding of the pairs $\langle x_1, x_2 \rangle$ is a notation for the application of the bijection $\alpha_2 : \mathbb{N}^2 \to \mathbb{N}$ defined by

$$\alpha_2(x, y) = y + \frac{(x + y + 1)(x + y)}{2}$$

The $\langle x_1, \dots, x_k \rangle$ encoding of the n-tuples is a notation for the application of the bijections $\alpha_k : \mathbb{N}^k \to \mathbb{N}$ defined inductively by $\alpha_{k+1}(x_1, x_2, \dots, x_{k+1}) = \alpha_2(x_1, \alpha_k(x_2, \dots, x_{k+1}))$.

Proposition 3.21. For all n the bijection

$$x_1, \ldots, x_n \mapsto \langle x_1, \ldots, x_n \rangle$$

is primitive recursive. For all n and all $i \leqslant n$ the function

$$\langle x_1, \ldots, x_n \rangle \mapsto x_i$$

is primitive recursive. ⋆

Proof. Addition, multiplication and integer division being primitive recursive, the function $(x, y) \mapsto y + \dfrac{(x+y+1)(x+y)}{2}$ is also primitive recursive by the composition scheme. It is therefore the same for all n for the functions $x_1, \ldots, x_n \mapsto \langle x_1, \ldots, x_n \rangle$ which are defined inductively by composition.

The function which to $\langle x_1, x_2 \rangle$ associates x_1 is primitive recursive using bounded minimization and the closure of primitive recursive predicates by bounded existential quantification:

$$f(n) = \min\{x \leqslant n : \exists y \leqslant n \ \langle x, y \rangle = n\}.$$

We leave it to the reader to embroider on this idea to show that all the projections are thus primitive recursive. ∎

We have at this stage all the elements necessary to show an important first theorem. We saw with Theorem 3.11 that primitive recursive functions can be programmed by structured programs with no `while` loops. The converse is true:

Theorem 3.22
Any function computable by a structured program without `while` loops is primitive recursive.

Proof. For $k \in \mathbb{N}$ given, and for a structured program P without `while` loops and using at most the registers $R_0, \ldots, R_k$, we define the function $f_P : \mathbb{N} \to \mathbb{N}$ by $f_P(\langle x_0, \ldots, x_k \rangle) = \langle v_0, \ldots, v_k \rangle$ where v_i is the value of the register R_i at the end of the execution of the program P, when its execution begins with its registers initialized to the values $x_0, \ldots, x_k$.

Let us show that for any structured program P without `while` loops, the corresponding function f_P is primitive recursive. It is clear that this is the case for the empty program. Let Q be the program whose only instruction is $R_i := R_i + 1$. Then, the function f_Q is given by $f_Q(\langle x_0, \ldots, x_k \rangle) = \langle x_0, \ldots, x_i + 1, \ldots, x_k \rangle$. We leave it to the reader to find the primitive recursive function corresponding to the instructions $R_i := R_i - 1$, $R_i := c$ for $c \in \mathbb{N}$ and $R_i := R_j$.

Suppose now that the proposition is true for programs P, P', via functions $f_P, f_{P'}$ and let Q be the program "if $R_j = 0$ then P else P' ". The function f_Q is therefore given by

$$\begin{array}{rcll} f_Q(\langle x_0,\dots,x_k\rangle) & = & f_P(\langle x_0,\dots,x_k\rangle) & \text{if } x_j = 0 \\ & = & f_{P'}(\langle x_0,\dots,x_k\rangle) & \text{otherwise.} \end{array}$$

Suppose now that the proposition is true for programs P, via functions f_P and let Q of the following form: "for $i = 1$ to R_j do P". The function f_Q is given by:

$$f_Q(\langle x_0,\dots,x_k\rangle) = g(\langle x_0,\dots,x_k\rangle, x_j)$$

where

$$\begin{array}{rcl} g(\langle x_0,\dots,x_k\rangle, 0) & = & \langle x_0,\dots,x_k\rangle \\ g(\langle x_0,\dots,x_k\rangle, z+1) & = & f_P(g(\langle x_0,\dots,x_k\rangle, z)). \end{array}$$

Suppose now the proposition true for programs P, P', via functions $f_P, f_{P'}$ and let Q consist of the instructions of P followed by those of P'. Then, $f_Q = f_{P'}(f_P(\langle x_0,\dots,x_k\rangle))$.

Using each of the cases described, we show by induction that the state of the registers of any structured program without `while` loops is a primitive recursive function. All we have to do is retrieve the value of the R_0 register.■

3.4. Study of the Ackermann function

There are several ways of seeing that not all computable functions are primitive recursive. The following is the most natural for the expert in computability theory for whom effective diagonalizations no longer hold any secrets:

Exercise 3.23. **($\star$)** Show that there exists a computable set $A \subseteq \mathbb{N}$ such that $n \mapsto \Phi_e(n)$ is a primitive recursive function for all $e \in A$ and such that every primitive recursive function has a code in A. Deduce that there exists a total computable function which is not primitive recursive. $\diamond$

A commonly given example of a computable non-primitive recursive function is Ackermann's function:

Definition 3.24 (Fonction d'Ackermann [3]). Define the functions $A_n : \mathbb{N} \to \mathbb{N}$ by induction on $n \in \mathbb{N}$ as follows.

- A_0 is the function $x \mapsto 2^x$;

- $A_{n+1}(x)$ is the application x times of the function A_n on 1:
$$A_n(A_n(\dots(A_n(1)))).$$

Formally:

$$\begin{array}{rcl} A_0(x) & = & 2^x \\ A_{n+1}(0) & = & 1 \\ A_{n+1}(x) & = & A_n(A_{n+1}(x-1)). \end{array}$$

Ackermann's function is the $n \mapsto A_n(n)$ function. $\diamondsuit$

The Ackermann function is growing extremely fast: $A(0) = 1$, $A(1) = 2$, $A(2) = 16$, and $A(3)$ is already equal to 65,536 iterations of the function $x \mapsto 2^x$ (starting at 0), that is:

$$A(3) = 2^{\left(2^{\left(\dots(2^0)\right)}\right)} \text{ where the power is iterated } 65,536 \text{ times.}$$

Despite its very strong growth, the Ackermann function is computable: to compute $A_n(n)$, we can use a stack containing either functions A_n (in practice a representation of these functions), or integers. For example, if we stack A_n, A_{n+1} and then k, this corresponds to the computation $A_n(A_{n+1}(k))$. So the top of the stack is always an integer, and the element that follows (if it exists) is always a function. Also to compute $A_n(n)$ we proceed as follows:

1. We stack A_n, then we stack n.
2. As long as the stack contains more than one element:
   (a) We unstack the integer k, then we unstack the function A_m.
   (b) If $m = 0$ we stack 2^k.
   (c) Otherwise, if $k = 0$ we stack 1.
   (d) Otherwise we stack A_{m-1}, then A_m and finally $k - 1$.

The algorithm will halt when the stack contains only one element, the result of the computation of $A_n(n)$.

We leave in exercise the proof that Ackermann's function grows faster than any primitive recursive function, and is therefore not itself primitive recursive.

Exercise 3.25 **(Cori and Lascar[45]). ($\star$)**

(1) Show that $A_n(x) > x$ for all n, x.

(2) Show that the function A_n is strictly increasing for all n.

(3) Show that the function $n \mapsto A_n(x)$ is increasing for all x.

(4) Show that $A_{n+1}(x+y) \geqslant A_n^{(y)}(x)$ for all n, x, y, where $f^{(m)}(x)$ denotes $f(f(\ldots f(x)\ldots))$ where the function f is iterated m times.

(5) Show that for all n, we have $A_{n+2}(x) > A_n^{(x+1)}(x+1)$ for almost all x.

(6) Let $n > 0$ and $f : \mathbb{N}^n \to \mathbb{N}$. We denote by $P(f)$ the predicate

$$\exists k \; \forall^\infty x_1, \ldots, x_n \; A_k(\max(x_1, \ldots, x_n)) > f(x_1, \ldots, x_n).$$

Show that $P(f)$ is true for any primitive recursive function f.

Deduce that Ackermann's function is not primitive recursive. $\diamond$

Loops of type `while` are therefore essential to compute certain functions, and with them comes the possibility of writing programs that do not halt.

3.5. Computable functions are recursive

We are now going to show that the recursive function minimization scheme allows us to compute any programmable function with or without `while` loops. To do this we start by seeing how to simulate list structures of arbitrary size by integers. This indeed seems at the moment to be a lack of our register machine model and structured programs. For example, we need a stack to compute the Ackermann function via the algorithm mentioned above.

Proposition 3.26. There exists a bijection $[] : \bigcup_{n \in \mathbb{N}} \mathbb{N}^n \to \mathbb{N}$ (where we denote by $[x_1, \ldots, x_n]$ the integer corresponding to n-tuple $(x_1, \ldots, x_n)$, such that the following operations are primitive recursive:

1. The function :: to add to the top of the list defined by $a :: [x_1, \ldots, x_n] = [a, x_1, \ldots, x_n]$.

2. The head and tail functions hd and tl, defined by

$$\begin{array}{rclcrcl} \mathrm{hd}([]) & = & 0 & \quad & \mathrm{tl}([]) & = & 0 \\ \mathrm{hd}(x :: l) & = & x & & \mathrm{tl}(x :: l) & = & l. \end{array}$$

3. The $||$ size function of a list defined by $|[x_1, \ldots, x_n]| = n$.

4. The get and set functions such that

$$\begin{array}{rcll} \mathrm{get}([x_0, \ldots, x_{n-1}], i) & = & x_i & \text{if } i < n \\ \mathrm{get}([x_0, \ldots, x_{n-1}], i) & = & 0 & \text{otherwise} \end{array}$$

and

$$\begin{array}{lcll} \text{set}([x_0,\ldots,x_{n-1}],a,i) & = & [x_0,\ldots,x_{i-1},a,x_{i+1},\ldots,x_{n-1}] & \text{if } i<n \\ \text{set}([x_0,\ldots,x_{n-1}],a,i) & = & [x_0,\ldots,x_{n-1}] & \text{otherwise} \end{array}$$

⋆

Proof. The bijection $[] : \bigcup_{n \imath \mathbb{N}} \mathbb{N}^n$ is defined by induction starting with the empty list $[] = 0$ and applying the operation of adding to the head of the list defined by $a :: l = 1 + \alpha_2(a,l)$ where $\alpha_2 : \mathbb{N}^2 \to \mathbb{N}$ is the bijection of Exercise 2-3.7. It is clear that the addition function at the head is primitive recursive, as are the functions hd and tl obtained thanks to the inverse functions of $(x_1,x_2) \mapsto \langle x_1,x_2\rangle$. Let us show that the coding of the lists thus obtained is indeed a bijection.

Let us show by induction that two lists of different sizes cannot be encoded by the same element. The code for an empty list is 0 and the code for a non-empty list is of the form $1+\alpha_2(a,l) \neq 0$. So the code for an empty list is always different from the code for a non-empty list.

Now suppose that all the lists of size n have a different code than the lists of size $m > n$. Let us show that all lists of size $n+1$ have a different code from that of lists of size $m > n+1$. The codes of lists of size $n+1$ are of the form $1+\alpha_2(a,l_1)$ for l_1 the code of a list of size n. The codes of lists of size $m > n+1$ are of the form $1+\alpha_2(b,l_2)$ for l_2 the code of a list of size $m > n$. By induction hypothesis, we necessarily have $l_1 \neq l_2$ and since α_2 is injective, we necessarily have $1+\alpha_2(a,l_1) \neq 1+\alpha_2(b,l_2)$. So the codes of lists of size $n+1$ are different from the codes of lists of size $m > n+1$. By induction, we deduce that the codes of lists of different sizes are different.

Let us now show by induction on k that if $(a_1,\ldots,a_k) \neq (b_1,\ldots,b_k)$, then we have $[a_1,\ldots,a_k] \neq [b_1,\ldots,b_k]$. For $k=1$ we have that $a_1 \neq b_1$ implies $1+\alpha_2(a_1,0) \neq 1+\alpha_2(b_1,0)$ because α_2 is injective. We therefore have $[a_1] \neq [b_1]$. Suppose this is the case for k and show that this is the case for $k+1$. Suppose $(a_1,\ldots,a_{k+1}) \neq (b_1,\ldots,b_{k+1})$. If $a_1 \neq b_1$, then we have $1+\alpha_2(a_1,[a_2,\ldots,a_{k+1}]) \neq 1+\alpha_2(b_1,[b_2,\ldots,b_{k+1}])$ because α_2 is injective. If $(a_2,\ldots,a_{k+1}) \neq (b_2,\ldots,b_{k+1})$, then by induction hypothesis we have $[a_2,\ldots,a_{k+1}] \neq [b_2,\ldots,b_{k+1}]$ and therefore $1+\alpha_2(a_1,[a_2,\ldots,a_{k+1}]) \neq 1+\alpha_2(b_1,[b_2,\ldots,b_{k+1}])$ because α_2 is injective. By induction, for all k we have therefore $(a_1,\ldots,a_k) \neq (b_1,\ldots,b_k)$ implies $[a_1,\ldots,a_k] \neq [b_1,\ldots,b_k]$. The function $[]$ is therefore injective.

Let us now show that $[]$ is surjective. Let us assume absurdly that this is not the case. In this case there exists a smallest n such that n is not the code of any list. Note that we necessarily have $n > 0$ because 0 is the code of the empty list. Also as α_2 is surjective, there exists (a, b) such that $\alpha_2(a,b) = n-1$ and therefore such that $1+\alpha_2(a,b) = n$. We also necessarily have $b \leqslant n-1 < n$. Also by minimality of n, there must exist a list l of which b is the code. So n is the code of the list $a :: l$, which contradicts our hypothesis on n.

In order to show that the functions $||$, get and set are primitive recursive, we give a primitive recursive definition of the function $\mathrm{tl}(l, n)$ which cuts off a list l of its n first elements:

$$\begin{array}{rcl} \mathrm{tl}(l,0) & = & l \\ \mathrm{tl}(l,n+1) & = & \mathrm{tl}(\mathrm{tl}(l,n)). \end{array}$$

The size is then defined via the minimization scheme bounded by $|l| = \min\{n \leqslant l : \mathrm{tl}(l,n) = []\}$. The get function has the following primitive recursive definition: $\mathrm{get}(l,n) = \mathrm{hd}(\mathrm{tl}(l,n))$. The set function is defined in two steps, by first adding an additional parameter:

$$\begin{array}{rcl} \mathrm{set}(l,a,i) & = & \mathrm{set}(l,a,i,i) \\ \mathrm{set}(l,a,n,0) & = & a :: \mathrm{tl}(l,\mathrm{succ}(n)) \\ \mathrm{set}(l,a,n,i+1) & = & \mathrm{get}(l,n-\mathrm{succ}(i)) :: \mathrm{set}(l,a,n,i) \end{array} \qquad \blacksquare$$

We now have all the elements necessary to show that functions computable by structured programs are recursive.

Theorem 3.27
Any function that can be computed by a goto program (and therefore also by a structured program) is a general recursive function.

The rest of the section is devoted to the proof, for which we need to fix an encoding of goto programs and register machines.

Coding of goto programs. We code the instructions of goto programs as follows:

- “$R_i = R_i + 1$” is encoded by $\langle 0, i\rangle$
- “$R_i = R_i - 1$” is encoded by $\langle 1, i\rangle$
- “$R_i = 0$” is encoded by $\langle 2, i\rangle$
- “if $R_i = 0$ goto n_1 else n_2” is encoded by $\langle 3, \langle i, \langle n_1, n_2\rangle\rangle\rangle$
- “goto n” is encoded by $\langle 4, n\rangle$

For uniformity reasons, it will be useful to have a bound on the maximum index of the registers used. The code of a goto program is simply given by the code: $\langle k, [I_1, \ldots, I_n]\rangle$ where k is such that the program uses at most the registers $R_0, \ldots, R_k$, and where I_e is the code of the e-th instruction for $1 \leqslant e \leqslant n$.

Coding of register machines. We now fix an encoding of the state of a k register machine. This state is given by the value of the registers as well as by the index of the next instruction to be executed. For a given number of registers k, this code is $\langle m, [R_0, \ldots, R_k]\rangle$ where m is the instruction number and R_i is the value of register number i for $0 \leqslant i \leqslant k$.

Initialization function. Let us now fix an initialization function init, which given a code $e = \langle k, I\rangle$ of a program (where I is a list of instructions), and given values $x_1, \ldots, x_n$ (for $n \leqslant k$), returns the code representing the state of the k register machine, at the start of the computation.

$$\mathrm{init}(\langle k, I\rangle, x_1, \ldots, x_n) \quad = \quad \langle 0, 0 :: x_1 :: \ldots :: x_n :: \mathrm{aux}(k-n)\rangle$$

with aux defined by

$$\begin{array}{rcl} \mathrm{aux}(0) & = & [] \\ \mathrm{aux}(k+1) & = & 0 :: \mathrm{aux}(k). \end{array}$$

It is clear that the function init is primitive recursive.

Transition functions 1. We define a transition function tr_1, which given the code $e = \langle k, I\rangle$ of a program and the code $p = \langle m, R\rangle$ of the state of a machine, returns the number of the next instruction to be executed at the next computation step. We will use for that a function $\mathrm{cur} : \mathbb{N} \to \mathbb{N}$ which, given the code $e = \langle k, I\rangle$ of a program and the code $p = \langle m, R\rangle$ of the state of a machine, allows to obtain the current instruction of the machine:

$$\mathrm{cur}(\langle k, I\rangle, \langle m, R\rangle) = \mathrm{get}(I, m).$$

Using the function cur, the functions π_1, π_2 such that $n = \langle \pi_1(n), \pi_2(n)\rangle$ we define $\mathrm{tr}_1(e, p)$ as being:

$$\begin{array}{lll} \pi_1(p) + 1 & \text{if} & \pi_1(\mathrm{cur}(e,p)) \leqslant 2 \\ \pi_1(\pi_2(\pi_2(\mathrm{cur}(e,p)))) & \text{if} & \pi_1(\mathrm{cur}(e,p)) = 3 \text{ and} \\ & & \mathrm{get}(\pi_2(p), \pi_1(\pi_2(\mathrm{cur}(e,p)))) = 0 \\ \pi_2(\pi_2(\pi_2(\mathrm{cur}(e,p)))) & \text{if} & \pi_1(\mathrm{cur}(e,p)) = 3 \text{ and} \\ & & \mathrm{get}(\pi_2(p), \pi_1(\pi_2(\mathrm{cur}(e,p)))) \neq 0 \\ \pi_2(\mathrm{cur}(e,p)) & \text{if} & \pi_1(\mathrm{cur}(e,p)) = 4. \end{array}$$

It is clear that tr_1 is primitive recursive.

Transition functions 2. We now define a transition function tr_2 which, given the code e of a program and the code p of the state of a machine, allows to obtain the state of the registers from the machine to the next computation step.

For that, we will use two primitive recursive functions inc : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and dec : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$, such that

$$\begin{array}{rcl} \mathrm{inc}([x_0, \dots, x_n], i) & = & [x_0, \dots, x_i + 1, \dots, x_n] \\ \mathrm{dec}([x_0, \dots, x_n], i) & = & [x_0, \dots, \max(0, x_i - 1), \dots, x_n] \end{array}$$

defined as follows.

$$\begin{array}{rcl} \mathrm{inc}(l, i) & = & \mathrm{set}(l, \mathrm{succ}(\mathrm{get}(l, i)), i) \\ \mathrm{dec}(l, i) & = & \mathrm{set}(l, \mathrm{pred}(\mathrm{get}(l, i)), i). \end{array}$$

We can now define $\mathrm{tr}_2(e, p)$ as being:

$$\begin{array}{rcl} \mathrm{inc}(\pi_2(p), \pi_2(\mathrm{cur}(e, p))) & \text{if} & \pi_1(\mathrm{cur}(e, p)) = 0 \\ \mathrm{dec}(\pi_2(p), \pi_2(\mathrm{cur}(e, p))) & \text{if} & \pi_1(\mathrm{cur}(e, p)) = 1 \\ \mathrm{set}(\pi_2(p), 0, \mathrm{cur}(e, p)) & \text{if} & \pi_1(\mathrm{cur}(e, p)) = 2 \\ \pi_2(p) & \text{otherwise.} & \end{array}$$

It is clear that tr_2 is primitive recursive.

End of the proof. We now define the function tr : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ of transition from one state to another:

$$\mathrm{tr}(e, p) \quad = \quad \langle \mathrm{tr}_1(e, p), \mathrm{tr}_2(e, p) \rangle.$$

Then we define the primitive recursive function st : $\mathbb{N} \times \mathbb{N}^n \times \mathbb{N} \to \mathbb{N}$ such that $\mathrm{st}(e, x_1, \dots, x_n, t)$ returns the state of the machine which executes the program e, with the registers $R_1, \dots, R_n$, initialized respectively to $x_1, \dots, x_n$, after t computation steps.

$$\begin{array}{rcl} \mathrm{st}(e, x_1, \dots, x_n, 0) & = & \mathrm{init}(e, x_1, \dots, x_n) \\ \mathrm{st}(e, x_1, \dots, x_n, t + 1) & = & \mathrm{tr}(e, \mathrm{st}(e, x_1, \dots, x_n, t)). \end{array}$$

We finally arrive at the stage for which we need a minimization scheme, leaving the possibility for a function not to be defined on certain inputs. The recursive function time : $\mathbb{N} \times \mathbb{N}^n \to \mathbb{N}$ gives the smallest computation time necessary for the machine to halt, i.e., arrives at an instruction number greater than the number of instructions in the program. The function will be defined if and only if the machine halts for the program and the corresponding inputs.

$$\mathrm{time}(e, x_1, \dots, x_n) \quad = \quad \min\{t \in \mathbb{N} : \pi_1(\mathrm{st}(e, x_1, \dots, x_n, t)) \geqslant |\pi_2(e)|\}.$$

Finally, here is the recursive function which corresponds to the computation of the code machine e. We launch the transition function for the number of steps necessary before the machine halts, and we return the value of the register R_0:

$$f(x_1,\dots,x_n) \quad = \quad \mathrm{hd}(\pi_2(\mathrm{st}(e,x_1,\dots,x_n,\mathrm{time}(e,x_1,\dots,x_n)))).$$

This concludes the demonstration.

3.6. Consequences

According to the previous proof, given our encoding of a program by an integer e, its execution for t computation steps is a primitive recursive function and therefore itself computable by a structured program. The search for this smaller computation time can be done using a `while` loop. Note further that the proof is uniform: the same primitive recursive function adapts according to any code e of a program.

This makes it possible to obtain Theorem 3-3.1 of the existence of a universal program, used throughout the book, via the notation Φ_e for the code program e.

Theorem (3-3.1)
Let $n \in \mathbb{N}^*$. There exists a code e of a computer program for which $\Phi_e : \mathbb{N}^{n+1} \to \mathbb{N}$ is such that for all $x_1,\dots,x_n$ we have

- $\Phi_e(a,x_1,\dots,x_n)\uparrow$ *iff* $\Phi_a(x_1,\dots,x_n)\uparrow$
- $\Phi_e(a,x_1,\dots,x_n)\downarrow= y$ *iff* $\Phi_a(x_1,\dots,x_n)\downarrow= y$

The code e of the above theorem is a code of the function

$$(x_1,\dots,x_n) \mapsto \mathrm{hd}(\pi_2(\mathrm{st}(a,x_1,\dots,x_n,\mathrm{time}(a,x_1,\dots,x_n))))$$

given at the end of the previous section. The function $(a,x_1,\dots,x_n) \mapsto \mathrm{time}(a,x_1,\dots,x_n)$ which looks for the smallest computation time such that the program halts is the only one which uses the minimization scheme. Note that this also makes it possible to give a precise mathematical definition to the notations $\Phi_a(x_1,\dots,x_n)[t]\downarrow$ and $\Phi_a(x_1,\dots,x_n)[t]\uparrow$: they correspond respectively to the primitive recursive predicates:

$$\begin{array}{rl} & \exists s \leqslant t\ \pi_1(\mathrm{st}(e,x_1,\dots,x_n,s)) \geqslant |\pi_2(e)| \\ \text{et} & \forall s \leqslant t\ \pi_1(\mathrm{st}(e,x_1,\dots,x_n,s)) < |\pi_2(e)| \end{array}$$

Chapter 7

Immunity and function growth

Computability theory studies the computational power of sets of integers, modulo the Turing reduction. In this chapter, we will study in particular two large families of computational properties, namely, the ability to compute sets that are difficult to describe (immune, hyperimmune, effectively immune set), and the ability to compute fast-growing functions (hyperimmune function, dominating function). Let us take a few examples.

Example 1. According to Exercise 3-7.10, any infinite c.e. set contains a computable infinite subset. What computational power is needed to obtain an infinite set that does not have any computable infinite subset? We will study this in Section 1 under the concept of immune set.

Example 2. According to Kleene's fixed point theorem 3-6.2, for any total computable function $f : \mathbb{N} \to \mathbb{N}$, there exists a program code e such that $\Phi_{f(e)} = \Phi_e$. What is the computational power of a function with no fixed point? This notion will be studied in Section 2.

Example 3. Every computable function is trivially dominated by a computable function. How much computational power does it take to compute a function that is not dominated by any computable function? This will be the subject of Section 4.

It is difficult to form an intuition on the *a priori* computational power of properties formulated in such a diverse way, and in particular to compare them. The properties taken from the three preceding examples however admit characterizations which will make this comparison easier. In general, the existence of numerous characterizations of the same computational power with very diverse formulations is a guarantee of the robustness of the concept. This is particularly the case for the properties studied in this chapter.

1. Immune sets

The first family of computational properties on sets relates to the ability to approximate the elements of a set. A set is computable if it is possible to compute effectively which elements belong to it or not. At the next level, a set is computable enumerable if there is a computable procedure for listing all of the elements that belong to it, but potentially out of order, so it is usually not possible to be certain that a element does not belong to the set. We are now going to study the computational power of infinite sets for which it is not even possible to enumerate in a computable way an infinite quantity of its elements.

Definition 1.1. An infinite set $A \subseteq \mathbb{N}$ is *immune* if it does not contain any infinite c.e. subset. ♢

As we have seen, every infinite c.e. set contains a computable infinite subset. Thus, in an equivalent manner, an infinite set is immune if and only if it does not contain a computable infinite subset. In particular, any immune set A is necessarily non-computable, because A would then be its own infinite computable subset contradicting its immunity.

Immunity is a concept of set, but not of degree. Indeed, if A is an immune set, the set $A \oplus \mathbb{N} = \{2n : n \in A\} \sqcup \{2n+1 : n \in \mathbb{N}\}$ has the same Turing degree as A, but $A \oplus \mathbb{N}$ has the infinite computable subset $\{2n+1 : n \in \mathbb{N}\}$. Conversely, any non-computable Turing degree contains an immune set, as the following proposition shows.

Proposition 1.2. Any non-computable set is Turing equivalent to an immune set. ⋆

Proof. Let A be a non-computable set. Let $B = \{\sigma \in 2^{<\mathbb{N}} : \sigma \prec A\}$ be the set of initial segments of A. In particular, $A \equiv_T B$. It is also obvious that any infinite subset of B allows to compute arbitrarily large initial segments of A, and therefore A (as well as B). Since A is not computable, B does not have any computable infinite subset. ■

The notion of immunity has two orthogonal reinforcements, namely effective immunity and hyperimmunity. These two notions are fundamental computational properties in computability theory, and we will see for each of them several equivalent definitions. Recall that W_e denotes the c.e. set of code e: $\{n : \Phi_e(n)\downarrow\}$.

Definition 1.3. An infinite set $A \subseteq \mathbb{N}$ is *effectively immune* if there exists a total computable function $h : \mathbb{N} \to \mathbb{N}$ such that for any code e, if $|W_e| \geqslant h(e)$ then $W_e \not\subseteq A$. $\diamondsuit$

Intuitively, an infinite set A is effectively immune if not only do infinite sets end up erring and enumerating an element outside A, but even more so, this error must occur after sufficiently few elements enumerated, depending on the enumeration code. In particular, any group that is effectively immune is immune.

We will see that the concept of effective immunity is particularly worthy of interest from the point of view of Turing degrees. The computational power corresponding to the capacity to compute an effectively immune set has many characterizations which will be studied in Section 2.

Recall that the *canonical code* of a finite set F is the integer $n = \sum_{i \in F} 2^i$. Let $D_0, D_1, \ldots$ be the collection of finite sets such that D_n is canonically encoded by n for all $n \in \mathbb{N}$.

Definition 1.4. An *array* is a collection of mutually disjoint finite sets $F_0, F_1, \ldots$ An array $F_0, F_1, \ldots$ is *c.e.* if there exists a computable function $f : \mathbb{N} \to \mathbb{N}$ such that for all n, $F_n = D_{f(n)}$. An infinite set A is *hyperimmune* if for any c.e. array $F_0, F_1, \ldots$, there exists an integer n such that $F_n \cap A = \emptyset$. $\diamondsuit$

This more complex definition formalizes the idea according to which not only is it not possible to computably list an infinity of elements of A, but even more so it is not even possible to approximate infinitely many elements by block, that is, to list an infinity of pairwise disjoint finite "blocks", such that each block contains at least one element of A.

As with the concept of effective immunity, it is the extension of hyperimmunity to Turing degrees that will be of particular interest to us in what follows. The computational power corresponding to the capacity to compute a hyperimmune set has many characterizations which will be studied in Section 4.

Exercise 1.5. Show that any hyperimmune set is immune. $\diamond$

2. DNC functions

We now see an example of a remarkable Turing degree, the study of which undoubtedly dates back to the work of Arslanov [8], who studied Turing degrees allowing to escape the famous Kleene fixed point theorem: given a computable function f, there exists e such that $\Phi_e = \Phi_{f(e)}$. What power is needed to compute a function f for which there is no such e? This work was extended by Jockusch, Lerman, Soare, and Solovay who found an equivalent characterization, which today constitutes the modern definition of DNC degree.

Definition 2.1. A function $f : \mathbb{N} \to \mathbb{N}$ is *diagonally non-computable* (DNC) if $f(n) \neq \Phi_n(n)$ for all n. $\diamondsuit$

Let us insist on the fact that the function f must be total in the definition above. Note that if $\Phi_n(n)\uparrow$, there is no restriction on the value of $f(n)$. A Turing degree is DNC if it contains a DNC function.

Exercise 2.2. **($\star$)** Show that the DNC degrees are upward-closed, that is to say that if a set computes a DNC function, its degree is itself DNC. $\diamond$

The notion of DNC degree has many applications in the links between computability theory and algorithmic randomness, as well as in reverse mathematics. We will see in particular with Corollary 18-4.3 that the DNC functions are numerous from the point of view of measure theory, but according to Proposition 10-3.36 rare from the point of view of Baire categories (we will see in particular the existence of non-DNC degrees which are not computable).

Let us observe first of all that there is no computable DNC function, by a simple diagonal argument, which is at the origin of the name "diagonally non-computable". On the other hand, the following proposition shows that we can compute a DNC function using the halting problem.

Proposition 2.3. $\mathbf{0}'$ is of DNC degree. $\star$

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be the $\emptyset'$-computable function, which for an input e, returns $1 - \Phi_e(e)$ if $e \in \emptyset'$ and 0 otherwise. This function is DNC, because it is total, and when $\Phi_e(e)\downarrow$, it returns a different value. Since $\emptyset'$ computes a DNC function, and the DNC degrees are upward-closed, $\mathbf{0}'$ is DNC. ∎

We will see in Section 3 that $\mathbf{0}'$ is the only degree which is both DNC and c.e. It is clear that the DNC degrees are in uncountable quantity, because it is also the case of the degrees above $\mathbf{0}'$. We will see with Proposition 10-3.36 that the non-DNC degrees are also uncountable.

For the moment, we endeavor to show that the notion of DNC degree is natural, in the sense that it has many characterizations via very different formulations (and we will see others still in Part II on algorithmic randomness).

Definition 2.4. A function f is *fixed-point free* if $\Phi_n \neq \Phi_{f(n)}$ for all n.$\diamondsuit$

Theorem 2.5 (Jockusch, Lerman, Soare, and Solovay [105])
Let $X \in 2^{\mathbb{N}}$. Then, the following statements are equivalent:

(1) X computes a diagonally non-computable function.

(2) X computes a free function of fixed point.

Proof. Let us show (1) $\Rightarrow$ (2). Let $g \leqslant_T X$ be such that $g(n) \neq \Phi_n(n)$ for all n. Then also, $f \leqslant_T X$ such that $f(n)$ is a code for a computable function defined only on the input n, and which to n associates $g(n)$. We have in particular $\Phi_{f(n)}(n) \downarrow = g(n)$. Suppose that there exists n such that $\Phi_{f(n)}(m) = \Phi_n(m)$ for all m. Then, $\Phi_n(n) \downarrow = \Phi_{f(n)}(n) \downarrow = g(n)$, which contradicts the definition of g. So f is a free fixed-point function.

Let us show (2) $\Rightarrow$ (1). Let $f \leqslant_T X$ be a free fixed-point function. Then, from X we compute the function $g : \mathbb{N} \to \mathbb{N}$ which on the integer n creates the code e_n of the function $m \mapsto \Phi_{\Phi_n(n)}(m)$, and returns $f(e_n)$. We use here the same abuse of notation as in the proof of the fixed point of Kleene: if $\Phi_n(n) \uparrow$ then $m \mapsto \Phi_{\Phi_n(n)}(m)$ denotes the function nowhere defined. Note that g is total because it does not try to do the computation $\Phi_n(n)$. Suppose that g is not DNC, that is, there is n such that $g(n) = \Phi_n(n)$. By definition of g, $f(e_n) = \Phi_n(n)$. In particular, $\Phi_{f(e_n)} = \Phi_{\Phi_n(n)} = \Phi_{e_n}$. The function f is therefore not free of a fixed point, which contradicts the definition of f. So g is a DNC function. ∎

Let us now see the equivalence between DNC degree and effectively immune degree. The third equivalence of the theorem below is more technical, but is of great interest and will be reused later. It is in fact a reinforcement of the notion of being DNC: Let us consider the sequence $(A_n)_{n\in\mathbb{N}}$ of c.e. sets defined by

$$A_n = \begin{cases} \{\Phi_n(n)\} & \text{if } \Phi_n(n)\downarrow \\ \emptyset & \text{otherwise} \end{cases}$$

Being of DNC degree is the ability to compute a function f such that $f(n) \notin A_n$ for any n. The third equivalence extends this result to the uniform enumeration $(W_n)_{n\in\mathbb{N}}$ of all c.e. sets whose number of elements is known to be bounded by an integer n.

Theorem 2.6
Let $X \in 2^{\mathbb{N}}$. The following statements are equivalent:

(1) X computes a diagonally non-computable function.

(2) X computes an effectively immune set.

(3) X compute a function $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for $e, n \in \mathbb{N}$,

$$|W_e| \leqslant n \Rightarrow h(e,n) \notin W_e$$

PROOF. (1) $\Rightarrow$ (3): Let $f \leqslant_T X$ be a DNC function. We describe a uniform process in e, n in order to compute a value which under the assumption $|W_e| \leqslant n$, is not in W_e. For each $0 \leqslant i < n$ we compute the code $u(e,i)$ of the partial computable function which, for any input, looks for the i-th element k in the enumeration of W_e, s' it exists. If such an element k is found, the function interprets it as the n-tuple $\langle k_0, \ldots, k_{n-1}\rangle$, and returns k_i. Otherwise, the function does not halt. Note that $\Phi_{u(e,i)}$ is either a constant function or the function defined nowhere.

Let us show that the X-computable function

$$h(e,n) = \langle f(u(e,0)), \ldots, f(u(e,n-1))\rangle$$

satisfied (3). Let us reason by the absurd, and suppose that $h(e,n) \in W_e$ with $|W_e| \leqslant n$. Let's say that $h(e,n)$ is the i-th element of W_e in the order of enumeration. Then, $\Phi_{u(e,i)}$ is the constant function which finds

$$k = h(e,n) = \langle f(u(e,0)), \ldots, f(u(e,n-1))\rangle$$

and returns the i-th element of the tuple, that is, $f(u(e,i))$. In particular, $\Phi_{u(e,i)}(u(e,i))\downarrow = f(u(e,i))$, which contradicts the hypothesis that f is DNC.

(3) $\Rightarrow$ (2): Let $h \leqslant_T X$ be a function satisfying (3). Let $D_0, D_1, \ldots$ be an effective enumeration of all finite sets such that n is the canonical code of D_n. Let $g : \mathbb{N} \to \mathbb{N}$ be a partial computable function such that if $|W_e| \geqslant e+1$, then $D_{g(e)} \subseteq W_e$ and $|D_{g(e)}| = e+1$. We are going to define a sequence X-computable infinite increasing of integers $x_0 < x_1 < \ldots$ such that for all s,

$$\forall e \leqslant s\ (|W_e| > e \Rightarrow D_{g(e)} \not\subseteq \{x_i : i \leqslant s\}) \qquad (\star)$$

It follows that $H = \{x_n : n \in \mathbb{N}\}$ is effectively immune, because if $W_e \subseteq H$, then $|W_e| \leqslant e$. The fact that we want $x_i < x_{i+1}$ is simply so that the set H is X-computable. Suppose that we have already defined $x_0 < \cdots < x_s$ satisfying $(\star)$. Let

$$W_{v(s)} = \{y : y \leqslant x_s\} \cup \bigcup_{e \leqslant s+1 \text{ t.q. } g(e)\downarrow} D_{g(e)}$$

The goal is to use our function h to find an element which is not in $W_{v(s)}$. Let x_{s+1} be such an element - we'll see how to get it later. Note first that x_{s+1} is strictly greater than x_s. Note then that for any $e \leqslant s+1$

such that $|W_e| > e$ and such that $D_{g(e)} \not\subseteq \{x_i : i \leqslant s\}$, then also $D_{g(e)} \not\subseteq \{x_i : i \leqslant s+1\}$. Note finally that for $e \leqslant s$ such that $|W_e| > e$ we have $D_{g(e)} \not\subseteq \{x_i : i \leqslant s\}$ by hypothesis, and for $e = s+1$, if $|W_e| > e$, we necessarily have $D_{g(e)} \not\subseteq \{x_i : i \leqslant s\}$ because $D_{g(e)}$ has then $s+2$ elements. In all cases we will have $(\star)$ for the following $(x_i)_{i \leqslant s+1}$. Now let's show how to find x_{s+1} using h. The set $W_{v(s)}$ has at most $x_s + 1$ plus $1+2+3+\cdots+s+2$ elements, which gives by the sum of the terms of an arithmetic sequence $t_s = x_s+1+(s+2)(s+3)/2$. We then define $x_{s+1} = h(v(s), t_s)$. By definition of h, $x_{s+1} \notin W_{v(s)}$. In particular, $x_{s+1} > x_s$, and $(\star)$ is satisfied for $s+1$.

(2) $\Rightarrow$ (1): Let $H \leqslant_T X$ be an effectively immune set. Let $g : \mathbb{N} \to \mathbb{N}$ be a total computable function such that if $W_e \subseteq H$, then $|W_e| < g(e)$. We will show that X computes a free function with a fixed point. Let $f : \mathbb{N} \to \mathbb{N}$ be the X-computable function such that $W_{f(e)}$ is the set of $g(e)$ first elements of H. Note that f is X-computable, but that the set $W_{f(e)}$ is a c.e. set which in particular does not need X for its enumeration: the elements to be enumerated can be seen as "hardcoded" in $f(e)$. Let us show that f is free of a fixed point. Let $e \in \mathbb{N}$. If $W_{f(e)} = W_e$, then $W_e \subseteq H$, but $|W_e| = |W_{f(e)}| = g(e)$, which contradicts the choice of H and g. As X computes a free function of fixed point, then by Theorem 2.5, X computes a DNC function. ■

We will stop there for now. We will see other characterizations of the notion of DNC degree in relation to algorithmic randomness. Finally, let us mention a hierarchy which follows naturally from the definition of DNC degree.

Definition 2.7. Let $f : \mathbb{N} \to \mathbb{N}$ be a function such that $2 \leqslant f(n) \leqslant f(n+1)$. A set $X \subseteq \mathbb{N}$ is of DNC$_f$ degree if X computes a function g such that $g(n) < f(n)$ and $g(n) \neq \Phi_n(n)$ for all n. ♢

An examination of the definition indicates that the slower the function f grows, the more difficult it seems for a function to be DNC$_f$. This is indeed the case, as we can see with the following theorem, due to Ambos-Spies, Kjos-Hanssen, Lempp and Slaman [6]:

Theorem 2.8
Let f be a function such that $2 \leqslant f(n) \leqslant f(n+1)$. There exists a function g such that $2 \leqslant g(n) \leqslant g(n+1)$ and with $f < g$ for which $\mathrm{DNC_f} \subsetneq \mathrm{DNC_g}$, i.e., there exists a set X which computes a $\mathrm{DNC_g}$ function but which does not compute any $\mathrm{DNC_f}$ function.

As announced earlier, we will see many examples of non-DNC and non-computable degrees, but it is informative to ensure their existence by direct construction.

Exercise 2.9. ($\star\star$) Show by the finite extension method that there are Δ^0_2 degrees which are simultaneously non-DNC and non-computable. $\diamond$

3. Arslanov completeness criterion

The Arslanov completeness criterion reflects in a certain way an incompatibility between the c.e. and DNC degrees.

C.e. degree

A Turing degree is said to be *computably enumerable* or *c.e.* if it contains a computably enumerable set. We will see in Chapter 13 that $\mathbf{0}'$ is far from being the only c.e. degree

It is easy to compute a DNC function using the halting problem, as shown in Theorem 2.3. On the other hand, the Arslanov completeness criterion proves that $\mathbf{0}'$ is the only degree at the same time c.e. and DNC.

Theorem 3.1 (Arslanov completeness criterion [8])
Let $A \in 2^{\mathbb{N}}$ be a c.e. set Then, A is Turing complete iff A computes a DNC function.

Proof. By Theorem 2.3, if A is Turing complete, it computes a DNC function. Let's show the converse. Let $(A_s)_{s\in\mathbb{N}}$ be a c.e. approximation of A, that is to say with $\lim_{s\to\infty} A_s = A$, and $A_s \subseteq A_{s+1}$. Note that if $A_s\restriction_n = A\restriction_n$ then also for all $t > s$ we will have $A_t\restriction_n = A\restriction_n$.

Let Φ be a total functional on the oracle A and such that $\Phi(A, n) \neq \Phi_n(n)$ for all n. Uniformly in n, we compute the code a_n of the partial function which on any input k acts as follows: look for the smallest t such that $\emptyset'[t](n) = 1$, then if that happens and if $\Phi(A_t\restriction_m, a_n)[t]$ halts for some $m \leqslant t$, then return the value of $\Phi(A_t\restriction_m, a_n)[t]$ and otherwise diverge. Note that the code a_n is defined as a function of a_n itself, which is made possible by the fixed point theorem.

We claim that for all n, if ever $\emptyset'(n) = 1$, then the smallest t such that $\Phi(A_t\restriction_m, a_n)[t]\downarrow$ for $m \leqslant t$ such that $A_t\restriction_m = A\restriction_m$, is strictly greater than the smallest t such that $\emptyset'[t](n) = 1$. Suppose this is not the case, i.e., there exists n for which $\emptyset'(n) = 1$ and for which given t the smallest integer such that $\emptyset'[t](n) = 1$, we have also $\Phi(A_t\restriction_m, a_n)[t]\downarrow$ for $m \leqslant t$ such that $A_t\restriction_m = A\restriction_m$. In this case, by the procedure described above, we will have $\Phi_{a_n}(a_n) = \Phi(A_t\restriction_m, a_n)$ and therefore $\Phi_{a_n}(a_n) = \Phi(A\restriction_m, a_n)$ which contradicts the fact that Φ computes a DNC function on the oracle A.

The oracle A can therefore compute $\emptyset'$ by simply looking for the smallest computation time t such that $A_t\restriction_m = A\restriction_m$ and $\Phi(A_t\restriction_m, a_n)[t]\downarrow$ for $m \leqslant t$, and then looking if $\emptyset'[t](n) = 1$. If so then $\emptyset'(n) = 1$. Otherwise $\emptyset'(n) = 0$.■

The Arslanov completeness criterion knows several extensions, in particular by Jockusch and al. [105]. We give one of them in an exercise that will allow us to manipulate the principles of the previous proof.

Exercise 3.2. (⋆⋆) (*Bienvenu and al.[17]*). A function g is DNC relative to C, denoted DNC(C), if $g(n) \neq \Phi_n(C, n)$ for all n. Let $X \in 2^{\mathbb{N}}$ of DNC degree, and C a c.e. set Show that either $X \oplus C \geqslant_T \emptyset'$ or X is of DNC degree relative to C.

Indication .– Let g be a DNC function. Define codes $a_{n,m}$ such that $\Phi_{a_{n,m}}(a_{n,m})$ is equal to $\Phi_m(C_s, m)$ for the smallest s such that $n \in \emptyset'[s]$. Show that either there exists n such that the function $m \mapsto g(a_{n,m})$ is DNC relative to C, or $g \oplus C$ computes $\emptyset'$. ◇

Note that the previous exercise implies the Arslanov completeness criterion, because if X and C are of the same degree, either $C \geqslant_T \emptyset'$, or C is of DNC degree relative to C, which is impossible.

4. Hyperimmune functions

We now approach a second family of computational properties, based on the ability to compute fast-growing functions. Exponential functions, or even exponential towers, can be computed, and therefore do not provide additional computational power. We are talking here about functions whose growth rate makes any mental representation difficult.

We have already seen in Section 4-7 that the capacity to grow faster than certain functions allowed to compute the Δ^0_2 sets. We are now going to study the computational power related to functions which are not dominated by any computable function. The study of these functions was initiated by Martin and Miller [162].

Definition 4.1. A function g *dominates* a function f if $g(x) \geqslant f(x)$ for all $x \in \mathbb{N}$. A function $f : \mathbb{N} \to \mathbb{N}$ is *hyperimmune* if it is not dominated by any computable function. ◇

In particular, the computable functions being stable under finite modifications, a function $f : \mathbb{N} \to \mathbb{N}$ is hyperimmune if for any computable function $g : \mathbb{N} \to \mathbb{N}$, $f(x) > g(x)$ for an infinity of values x:

Exercise 4.2. Show that a function f is hyperimmune iff for any total computable function g, there is an infinite number of integers x such that $f(x) > g(x)$. $\diamond$

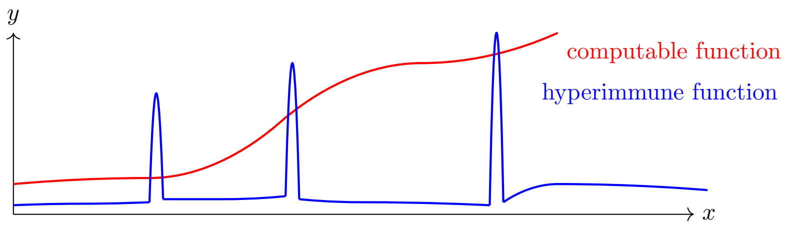


Figure 4.3: Illustration of a hyperimmune function: it does not necessarily grow very fast, but for every computable function f, it is infinitely often above f.

Like DNC degrees, Turing degrees for hyperimmune functions are upward-closed. A Turing degree is *hyperimmune* if it contains a hyperimmune function, or equivalently, if one of its elements computes a hyperimmune function. The name “hyperimmune function” comes from the following correspondence with hyperimmune sets.

Proposition 4.4. An infinite set X is hyperimmune iff the function $p_X : \mathbb{N} \to \mathbb{N}$ which to n associates the n-th element of X is hyperimmune. $\star$

Proof. $\Rightarrow$ Let X be a hyperimmune set and let $g : \mathbb{N} \to \mathbb{N}$ be a total computable function. Let us show that p_X is not dominated by g. Let $h(x) = g(x) + x + 1$, and let $(F_n)_{n\in\mathbb{N}}$, be the c.e. array defined by $F_n = [h^{(n)}(0), h^{(n+1)}(0)[$, where $h^{(n)}$ is the n-th iteration of h, with $h^{(0)}$ the identity function. By hyperimmunity of X, there exists n such that $F_n \cap X = \emptyset$. This means that $p_X(h^{(n)}(0)) \geqslant h^{(n+1)}(0) = h(h^{(n)}(0))$. By taking $x = h^{(n)}(0)$, we have $p_X(x) \geqslant h(x) = g(x) + x + 1$, so p_X is not dominated by g.

$\Leftarrow$ Suppose that p_X is a hyperimmune function. Let us reason by the absurd, supposing that the set X is not hyperimmune. In other words, there exists a c.e. array $(F_n)_{n\in\mathbb{N}}$ such that $F_n \cap X \neq \emptyset$ for all n. Then, the function $g : \mathbb{N} \to \mathbb{N}$ defined by $g(n) = \max \bigcup_{i\leqslant n} F_i$ is total computable, and dominates p_X, contradicting the hyperimmunity of p_X. $\blacksquare$

It follows that a degree is hyperimmune if and only if it contains a hyperimmune set.

Exercise 4.5. Show that hyperimmune degrees are upward-closed. In other words, if X computes a hyperimmune function, then the Turing degree of X contains a hyperimmune function. $\diamond$

Exercise 4.6. Show by the finite extension method that there exists a set of hyperimmune degree. $\diamond$

The existence of non-computable and non-hyperimmune degrees is currently unclear. We will see in the next section that such degrees do exist, even if we will understand later that this is not granted, in the sense that "many" sets are of hyperimmune degree (see Proposition 10-3.35 and Theorem 19-3.4). In fact, we will have to work to show off one that is not. We see in particular right now that the construction of a non-computable and non-hyperimmune degree cannot be done using $\emptyset'$. In particular constructions by finite extensions as seen in Section 4-8, which are all effective in $\emptyset'$, will not work.

Proposition 4.7 (Martin and Miller [162]). Every non-computable Δ^0_2 set is hyperimmune. $\star$

PROOF. Let A be a non-computable Δ^0_2 set, and let $A_0, A_1, \ldots$ be a Δ^0_2 approximation of A. Recall that the *computation function* (Definition 4-7.7) is defined as the function $c_A : \mathbb{N} \to \mathbb{N}$ which to x associates the smallest integer $n \geqslant x$ such that $A_n \upharpoonright_x = A \upharpoonright_x$. In particular, $c_A \leqslant_T A$. According to Theorem 4-7.9, any function dominating c_A computes A. As A is not computable, c_A is not dominated by any computable function, in other words c_A is hyperimmune. $\blacksquare$

We now move on to the existence of non-computable and non-hyperimmune degrees, which are called *computably dominated.*

5. Computably dominated degrees

By upward-closure of the hyperimmune degrees, a Turing degree is not hyperimmune if any function f that it computes is dominated by a computable function g (which depends on f).

Definition 5.1. A set X is *computably dominated* if for any function $f \leqslant_T X$ there exists a computable function g dominating f. A Turing degree $\mathbf{d}$ is computably dominated[a] if all $X \in \mathbf{d}$ is computably dominated. $\diamond$

[a]The terminologies "computably dominated" and "hyperimmune-free" coexist.

Note that being computably dominated is a weakness property and is therefore downward-closed in the Turing degrees. In fact the inverse property — being of hyperimmune degree— is a strength property. The simplest example of a computably dominated degree is the Turing degree of computable sets. The aim of this section is to prove the existence of computably dominated degrees different from $\mathbf{0}$. We will in fact see a little later (Theorem 8 -5.1) that the computably dominated degrees are in uncountable quantity: it is possible to construct an injection $f : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ such that for all X, the set $f(X)$ is of computably dominated degree, and even such that $f(X)$ and $f(Y)$ are in different Turing degrees for $X \neq Y$ (Exercise 8-5.3). The concept at the heart of the construction which will follow is that of *f-tree*.

Definition 5.2. An *f-tree* is a total function $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ such that for all $\sigma, \tau \in 2^{<\mathbb{N}}$, $\sigma \preceq \tau$ if and only if $T(\sigma) \preceq T(\tau)$. ♢

Let T be an f-tree. Note that the image of T ($\operatorname{Im} T$) is not closed under prefix in general. Note also that for all $\sigma \in 2^{<\mathbb{N}}$, $T(\sigma 0)$ and $T(\sigma 1)$ are two incompatible strings extending $T(\sigma)$. Only the image of an f-tree T is important. The tree structure of the domain of T canonically induces that of $\operatorname{Im} T$. The reader may refer to Figure 5.4 for a graphical representation of an f-tree.

Definition 5.3. Let T be an f-tree. We call *nodes* the elements of $\operatorname{Im} T$. A *path* of T is a sequence $P \in 2^{\mathbb{N}}$ of which an infinity of initial segments belong to $\operatorname{Im} T$. We will denote by $[T]$ the set of paths of T. ♢

Figure 5.4 illustrates the fact that an f-tree T induces an injection $f_T : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$, computable using the f-tree: for $X \in 2^{\mathbb{N}}$, the sequence $f_T(X)$ is computed little by little as being $T(X\upharpoonright_1) \prec T(X\upharpoonright_2) \prec \dots$. Moreover by the properties of an f-tree, if $X \neq Y$ then $f_T(X) \neq f_T(Y)$.

Definition 5.5. A *sub-f-tree* of an f-tree T is an f-tree S such that $\operatorname{Im} S \subseteq \operatorname{Im} T$. ♢

It follows that if S is a sub-f-tree of T, then $[S] \subseteq [T]$. We now have the necessary ingredients to proceed to the proof of the announced theorem.

Theorem 5.6 (Martin and Miller [162])
There is a computably dominated degree $\mathbf{d} > \mathbf{0}$.

Proof. We want to build a set A of computably dominated degree by satisfying the following requirements $(\mathcal{R}_e)_{e \in \mathbb{N}}$ and $(\mathcal{S}_e)_{e \in \mathbb{N}}$:

$$\mathcal{R}_e : W_e \neq A \qquad \mathcal{S}_e : \Phi_e^A \text{ total } \Rightarrow \Phi_e^A \text{ dominated by a computable function}$$

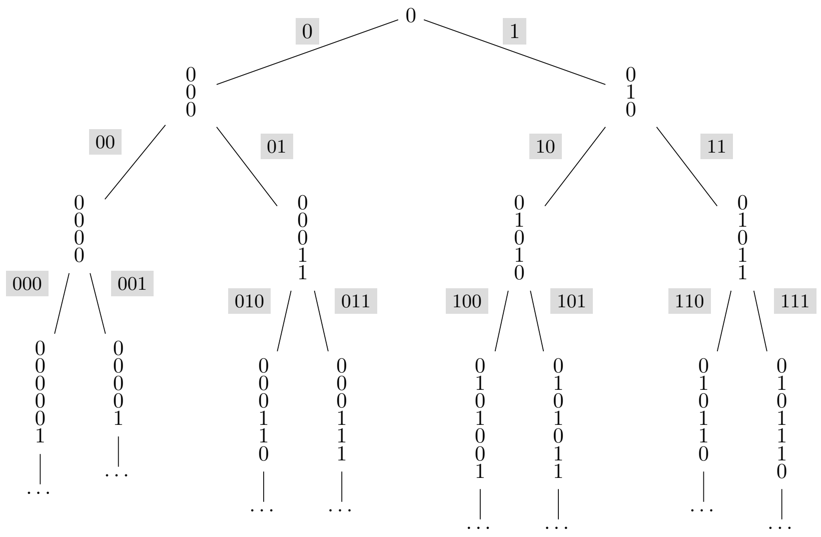


Figure 5.4: Illustration of an f-tree T, whose domain is represented by light grey strings : $T(\epsilon) = 0$, $T(0) = 000$, $T(1) = 010$, …

We are going to build an infinite sequence of computable f-trees $T_0, T_1, T_2, \dots$ such that for all $e \in \mathbb{N}$,

(1) T_{e+1} is a sub-f-tree of T_e;

(2) $|T_e(\epsilon)| \geqslant e$;

(3) For any path $P \in [T_{2e+1}]$, the requirement $\mathcal{R}_e$ is satisfied;

(4) For any path $P \in [T_{2e+2}]$, the requirement $\mathcal{S}_e$ is satisfied.

Satisfaction of a requirement $\mathcal{R}_e$. Let T be a computable f-tree. Since $\sigma_0 = T(0)$ and $\sigma_1 = T(1)$ are incompatible strings, there is some $i \in \mathbb{N}$ such that $\sigma_0(i) \neq \sigma_1(i)$. So either $\sigma_0(i) \neq W_e(i)$ or $\sigma_1(i) \neq W_e(i)$. Suppose we are in the first case, the other being symmetrical. Then, the sub-tree S of T defined by $S(\rho) = T(0\rho)$ ensures that for all paths $P \in [S]$, $\sigma_0 \preceq P$, therefore $P \neq W_e$. Moreover, S is computable in T, therefore computable.

Satisfaction of a requirement $\mathcal{S}_e$. Let T be a computable f-tree. We are going to build a computable sub-tree S such that either Φ_e^P is partial for all $P \in [S]$, or Φ_e^P is total and dominated by the same computable function for all $P \in [S]$. Note that in addition to satisfying the requirement $\mathcal{S}_e$, the functional Φ_e^P will be either partial or total whatever the path P. Two cases arise:

Case 1: There is a node $\sigma \in \operatorname{Im} T$ and an input $x \in \mathbb{N}$ such that $\Phi^\tau(x)\uparrow$ for any $\tau \in \operatorname{Im} T$ such that $\tau \succeq \sigma$. Let $\rho \in 2^{<\mathbb{N}}$ be such that $T(\rho) = \sigma$. Then, the subtree S of T defined by $S(\mu) = T(\rho\mu)$ ensures that for all paths $P \in [S]$, arbitrarily long initial segments τ of P will satisfy $\Phi^\tau(x)\uparrow$. By the use property, it follows that $\Phi^P(x)\uparrow$.

Case 2: For any node $\sigma \in \operatorname{Im} T$ and any input $x \in \mathbb{N}$, there exists $\tau \in \operatorname{Im} T$ such that $\sigma \preceq \tau$ and $\Phi^\tau(x)\downarrow$. We will define S, a computable sub-tree of T such that for all $\rho \in 2^{<\mathbb{N}}$, $\Phi_e^{S(\rho)}(|\rho|)\downarrow$. We compute $S(\epsilon)$ as being the first node of $\operatorname{Im} T$ which we find such that $\Phi_e^{S(\epsilon)}(0)\downarrow$. Suppose we have computed $S(\rho)$. Let μ be such that $S(\rho) = T(\mu)$. We compute $S(\rho 0)$ and $S(\rho 1)$ as follows: for each $i < 2$, $S(\rho i)$ is the first node $\tau \in \operatorname{Im} T$ that we find, with $\tau \succeq T(\mu i)$ and such that $\Phi^\tau(|\rho| + 1)\downarrow$. The algorithm which searches for the nodes $S(\rho 0)$ and $S(\rho 1)$ will always succeed, by the assumption that we are in case 2. By construction, S is indeed an f-tree, and $\operatorname{Im} S \subseteq \operatorname{Im} T$. Even more, Φ_e^P is a total function for all $P \in [S]$.

Let us show that Φ_e^P is dominated by a computable function for all $P \in [S]$. Let $g : \mathbb{N} \to \mathbb{N}$ be the function defined by $g(n) = \max\{\Phi_e^{S(\rho)}(n) : |\rho| = n\}$. Then, for all $n \in \mathbb{N}$ and $P \in [S]$, $\Phi_e^P(n) \leqslant g(n)$. The sub-f-tree S of T satisfies the requirement $\mathcal{S}_e$.

Finally, note to satisfy point (2) above that for any computable f-tree T and any n, there exists a sub-f-tree S such that $S(\epsilon) \geqslant n$. Indeed, it suffices to fix a string ρ of length n, and to define $S(\mu) = T(\rho\mu)$. We can therefore combine the satisfactions of the different requirements and this last observation to construct a sequence of f-trees $T_0, T_1, \ldots$ satisfying the properties (1) (2) (3) and (4) given above.

Remark

Each f-tree T_e of the sequence taken independently is computable, but the sequence $T_0, T_1, \ldots$ is not itself computable.

In order to complete the proof, we need the following lemma.

Lemma 5.7. The intersection $\bigcap_e [T_e]$ contains exactly one element. ⋆

Proof. Since $\operatorname{Im} T_{e+1} \subseteq \operatorname{Im} T_e$ and since any string of $\operatorname{Im} T_e$ is an extension of $T_e(\epsilon)$, it is clear that we have $T_0(\epsilon) \preceq T_1(\epsilon) \prec T_2(\epsilon) \prec \ldots$.

Moreover as $|T_e(\epsilon)| \geqslant e$, the sequence $T_0(\epsilon) \preceq T_1(\epsilon) \prec T_2(\epsilon) \prec \ldots$ converges to a unique infinite sequence $X \in 2^{\mathbb{N}}$. Since X has an infinity of prefixes in each T_e then $X \in [T_e]$ for all e and therefore $X \in \bigcap_e [T_e]$. ■

Let A be the element of $\bigcap_e [T_e]$. For all $e \in \mathbb{N}$, as $A \in [T_{2e+2}]$, then the requirement $\mathcal{R}_e$ is satisfied for A, hence $W_e \neq A$. Moreover, as $A \in [T_{2e+2}]$, then the requirement $\mathcal{S}_e$ is satisfied for A, so if Φ_e^A is total, Φ_e^A is dominated by a computable function. It follows that A is computably dominated and not computable. ■

Remark

A careful analysis of the previous construction shows that the oracle $\emptyset''$ is sufficient to compute the sequence $(T_e)_{e\in\mathbb{N}}$ uniformly. Indeed, the only non-computable parts are the case analysis, which are Σ_2^0 properties. Thus, there exists a degree $\mathbf{d} > \mathbf{0}$ both Δ_3^0 and computably dominated. As we saw with Proposition 4.7, it is not possible to lower this bound to Δ_2^0.

We will find the structure of f-trees in Chapter 14 to show the existence of minimal Turing degrees.

Exercise 5.8. **($\star\star$)** Build an f-tree T, by the finite extension method, such that all $X, Y \in [T]$ are in different Turing degrees. $\diamond$

Note that we had announced the existence of an uncountable quantity of computably dominated degrees. This will be done with Theorem 8-5.1. We now give some equivalences to better understand this notion.

5.1. Truth-table reduction

We suppose in what follows that the functionals considered try to compute sets of integers and not functions, that is, if $\Phi(Y, n)\downarrow$ for a certain oracle Y and a certain integer n, then $\Phi(Y, n)\downarrow \in \{0, 1\}$ (if $\Phi(Y, n)\downarrow\notin \{0, 1\}$ we will consider that the functional diverges).

Consider a functional Φ and an oracle X such that $\forall n\ \Phi(X, n) \downarrow \ \in \{0, 1\}$. Given another set $Y \neq X$, there is no reason why we should also have $\forall n\ \Phi(Y, n)\downarrow$. Totality with one oracle is not necessarily totality with others, and it should not be very hard for the reader to construct such examples. But are such partialities necessary? Assuming $X \geqslant_T Y$, can we still compute Y from X via a total functional over all oracles? We will see that this is not necessarily the case via a restriction of the notion of Turing reduction.

Definition 5.9. For any set $X, Y \subseteq \mathbb{N}$, we say that X is *truth-table reducible* to Y, and we write $X \leqslant_{tt} Y$ if there is a functional Φ such that $\Phi(Y) = X$ and such that $\Phi(Z)$ is total for any oracle Z. We write $X \equiv_{tt} Y$ if $X \leqslant_{tt} Y$ and $Y \leqslant_{tt} X$. We write $X <_{tt} Y$ if $X \leqslant_{tt} Y$

and $Y \not\leqslant_{tt} X$. We call *truth-table degrees* the equivalence classes of the relation $\equiv_{tt}$. ♢

Notice the notation $\Phi(Y) = X$ meaning $\forall n\ \Phi(Y, n) = X(n)$. We will see several definitions equivalent to the truth-table reduction, starting with the one justifying its name.

Definition 5.10. A reduction by truth table is given by a computable sequence of pairs $(\langle C_{0,n}, C_{1,n}\rangle)_{n\in\mathbb{N}}$ such that for all n the set $C_{0,n} \cup C_{1,n} \subseteq 2^{<\mathbb{N}}$ contains exactly all the strings of a certain size m_n, and such that $C_{0,n} \cap C_{1,n} = \emptyset$. The set X computes Y via this reduction if for all n we have $Y(n) = i$ iff $\sigma \in C_{i,n}$ for a prefix σ of X. ♢

The sets $C_{i,n}$ are the "truth tables". Any oracle X has a prefix in $C_{0,n} \cup C_{1,n}$. If the prefix belongs to $C_{0,n}$ then X computes 0 on the input n and if the prefix belongs to $C_{1,n}$ then X computes 1 on the input n. Let us now see the different equivalences to the notion of truth-table reduction.

Theorem 5.11
Let X, Y be sets. The following statements are equivalent:

(1) $Y \leqslant_{tt} X$

(2) X computes Y via a reduction by truth table.

(3) There exists a functional Φ and a total computable function $b : \mathbb{N} \to \mathbb{N}$ such that $\Phi(X, n)[b(n)]\downarrow= Y(n)$ for all n.

Proof. Let us show (1) ⇒ (2). Suppose $\Phi(X) = Y$ via a total functional Φ over all oracles. Given n, we search for the smallest computation time t_n such that for a certain $m_n \leqslant t_n$ and for any string $\sigma \in 2^{\mathbb{N}}$ of size m_n we have $\Phi(\sigma, n)[t_n]\downarrow$. In order to show that such a computing time t_n necessarily exists for all n, we must anticipate a little on Definition 8-1.1 *of tree* and Lemma 8-1.4 of König to come. Let us assume absurdly that for a certain n, one cannot find t_n. This implies in particular that for any m, there exists a string σ of size m such that $\forall t\ \Phi(\sigma, n)[t]\uparrow$. Moreover if $\forall t\ \Phi(\sigma, n)[t]\uparrow$ and $\tau \preceq \sigma$ then also $\forall t\ \Phi(\tau, n)[t]\uparrow$. We can therefore construct an infinite tree T such that $\sigma \in T$ implies $\forall t\ \Phi(\sigma, n)[t]\uparrow$. According to König's lemma T, contains an infinite path Y, which is therefore such that $\Phi(Y, n)\uparrow$ which contradicts the fact that Φ is total over all its oracles. We can therefore at each step find t_n and $m_n \leqslant t_n$, with the set $C_{0,n}$ of strings of sizes m_n on which the computation returns 0 and the set of strings $C_{1,n}$ of sizes m_n on which the computation returns 1.

Let us show (2) ⇒ (3). Given a reduction by truth table given by the computable sequence $(\langle C_{0,n}, C_{1,n}\rangle)_{n\in\mathbb{N}}$, for any input n we can limit the computation time that the functional takes to halt on n with any oracle: this is simply the time required to produce the computation of $\langle C_{0,n}, C_{1,n}\rangle$.

Let us show (3) $\Rightarrow$ (1). Let Φ be a functional and $b : \mathbb{N} \to \mathbb{N}$ a total computable function such that $\Phi(Y, n)[b(n)]\downarrow = X(n)$ for all n. Then, we construct the functional Ψ which on all oracles Z and on any input n launches the computation of $\Phi(Z, n)$ for $b(n)$ steps. If the computation returns a value in $b(n)$ steps, then Ψ returns that value, otherwise Ψ returns 0. The result of the computation is the same between Φ and Ψ on the oracle Y, but Ψ is now total on all oracles. ■

We now show that the sets X for which $X \geqslant_T Y$ implies $X \geqslant_{tt} Y$ are exactly the computably dominated sets.

Theorem 5.12 (Jockusch [101], Martin (non publié))
A set X is computably dominated iff $Y \leqslant_T X \Leftrightarrow Y \leqslant_{tt} X$ for all $Y \in 2^{\mathbb{N}}$.

PROOF. Suppose X is computably dominated. Suppose $X \geqslant_T Y$ via the functional Φ. Let $f : \mathbb{N} \to \mathbb{N}$ be such that $\Phi(X, n)[f(n)]\downarrow = Y(n)$ for all n. Note that f is an X-computable function. There is therefore a computable function $g > f$. We therefore have $\Phi(X, n)[g(n)]\downarrow = Y(n)$. According to Theorem 5.11 we have then $X \geqslant_{tt} Y$.

Suppose now that for all Y we have $Y \leqslant_T X \Leftrightarrow Y \leqslant_{tt} X$. Then, also for any function $f : \mathbb{N} \to \mathbb{N}$, we have $f \leqslant_T X \Leftrightarrow f \leqslant_{tt} X$, via the canonical representation of f by a sequence of $2^{\mathbb{N}}$. Let $f \leqslant_T X$. Let us show that f is dominated by a computable function g. By hypothesis, $f \leqslant_{tt} X$, therefore by Theorem 5.11, there exists a functional Φ and a total computable function $b : \mathbb{N} \to \mathbb{N}$ such that $\Phi(X, n)[b(n)]\downarrow = f(n)$ for all n. We can assume without loss of generality that if $\Phi(X, n)[b(n)]\downarrow$, then the use of the computation is lower than $b(n)$ (if this is not the case we can slow down the computation so that it is), therefore $\Phi(X \upharpoonright_{b(n)}, n)[b(n)]\downarrow$. Let $g : \mathbb{N} \to \mathbb{N}$ be the function which for an input n, executes $\Phi(\sigma, n)[b(n)]$ for any $\sigma \in 2^{<\mathbb{N}}$ of length $b(n)$, and returns the maximum of the values obtained. In particular, $g(n) \geqslant \Phi(X \upharpoonright_{b(n)}, n)[b(n)] = f(n)$. The function g dominates f. It follows that X is of computably dominated degree. ■

We now have three notions of reductions: the many-one reduction, the truth-table reduction and the Turing reduction. The following proposition recapitulates some results seen so far, which attest that none coincides with another.

Proposition 5.13. For all X, Y we have $X \leqslant_m Y \Rightarrow X \leqslant_{tt} Y \Rightarrow X \leqslant_T Y$. No reverse implication is true in the general case. ⋆

Proof. The implications are clear. Let us show that no reciprocal implication holds. The set $\mathbb{N} \setminus \emptyset'$ is *tt*-reducible to $\emptyset'$ but not *m*-reducible to $\emptyset'$ because it would then be Σ^0_1 according to Proposition 5-4.3, and therefore $\emptyset'$ would be computable, which is false. The existence of sets X, Y such that $X \geqslant_T Y$ but $X \ngeqslant_{tt} Y$ is a consequence of Theorem 5.12 and of the fact that non-computably dominated degrees exist (for example $\mathbf{0}'$). ■

6. Martin's domination theorem

Hyperimmune functions are by definition those functions which are not dominated by any computable function. It is natural to wonder about the computational power of the functions which dominate all the computable functions. Of course, no function $f : \mathbb{N} \to \mathbb{N}$ dominates all the computable functions, because the constant function $f(0) + 1$ is not dominated by f. We can however weaken the property, and wonder about the computational power of a function $f : \mathbb{N} \to \mathbb{N}$ such that for any computable function $g : \mathbb{N} \to \mathbb{N}$, f *eventually* dominates g, that is, for all but finitely many inputs.

Notation

We will use the notations $\forall^\infty m$ and $\exists^\infty m$ to signify respectively $\exists n\, \forall m > n$ and $\forall n\, \exists m > n$. Thus, $\forall^\infty m$ means "for all but finitely many m" and $\exists^\infty m$ means "for infinitely many m".

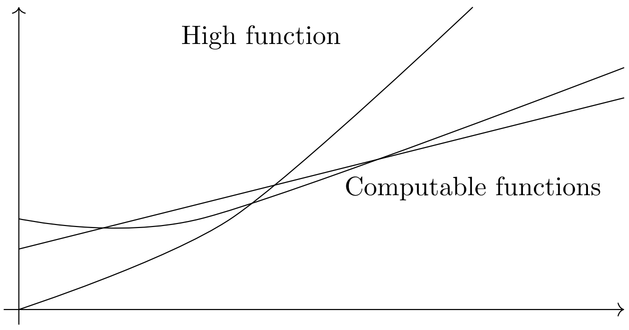


Figure 6.1: Illustration of a dominating function, which for every computable function f, is always above f after some point.

Martin's domination theorem ((1) $\Leftrightarrow$ (2) in the following theorem [152]) gives a magnificent characterization of the Turing degrees of these functions.

The equivalence (2) $\Leftrightarrow$ (3) shown by Jockusch [102] came later and is also of interest. Recall that a set A is high if $A' \geqslant_T \emptyset''$ (see Definition 4-10.1).

Theorem 6.2
Let $A \subseteq \mathbb{N}$ be a set. The following statements are equivalent:

(1) A is high.

(2) A computes a function $g : \mathbb{N} \to \mathbb{N}$ which eventually dominates all computable functions. That is to say that for any computable function f we have $\forall^\infty n \ f(n) \leqslant g(n)$.

(3) A computes a list $(X_n)_{n \in \mathbb{N}}$ containing (possibly with repetitions) exactly the computable sets.

Proof. Let us show (1) $\Rightarrow$ (2). Suppose A is high. We thus have a $\Delta^0_2(A)$ description of $\emptyset''$, that is to say an A-computable function $f : \mathbb{N}^2 \to \mathbb{N}$ such that $\lim_{s \to \infty} f(n, s) = \emptyset''(n)$ for all n. Note that being the code of a total function is a Π^0_2 property. We can in particular, by using the fact that $\emptyset''$ is Σ^0_2-complete (see Proposition 5-5.3), compute for all e a code a_e such that $\emptyset''(a_e) = 0$ iff Φ_e is a total function.

We define g as follows: on the input t, for any functional Φ_e for $e \leqslant t$, we search for the smallest computation time $s \geqslant t$ such that $f(a_e, s) = 1$ or such that $\Phi_e(t)[s] \downarrow$. Note that one of the two events necessarily happens: either Φ_e is total and therefore $\Phi_e(t) \downarrow$, or Φ_e is partial and therefore $\lim_{s \to \infty} f(a_e, s) = \emptyset''(a_e) = 1$. In the first case, we define $v_{t,e} = 0$ and in the second $v_{t,e} = \Phi_e(t)$. We finally define $g(t) = \sum_{e \leqslant t} v_{t,e}$.

It is clear that for any total function Φ_e, starting from the smallest $t \geqslant e$ such that $f(a_e, s) = 0$ for $s \geqslant t$, we will have $g(s) \geqslant \Phi_e(s)$ for all $s \geqslant t$. So g eventually dominates all computable functions.

Let us show (2) $\Rightarrow$ (3). Suppose now that A computes a function g which eventually dominates any computable function. We use the fact that if Φ_e is total with value in $\{0, 1\}$, then the computable function $t : \mathbb{N} \to \mathbb{N}$ which on n returns the smallest computation time such that $\Phi_e(n)[t(n)]\downarrow$ for all n, is eventually dominated by g.

For each functional Φ_e we compute the set Y_e as being $Y_e(n) = \Phi_e(n)[g(n)]$ if $\Phi_e(n)[g(n)]\downarrow \ \in \{0, 1\}$ and $Y_e(n) = 0$ otherwise. The list $(Y_e)_{e \in \mathbb{N}}$ therefore contains, up to finite modification, exactly the computable sets. To obtain them all, we finally compute the sequence $(X_n)_{n \in \mathbb{N}}$ as being all the possible finite modifications of the sets Y_e.

Let us show (3) $\Rightarrow$ (1). The reader can use Figure 6.4 to understand this implication. Suppose that A computes a list $(X_n)_{n \in \mathbb{N}}$ containing (possibly with repetitions) exactly the computable sets. Let $P = \{e : \forall x_1\ \exists x_2\ R(e, x_1, x_2)\}$ be an arbitrary Π^0_2 set. Let us show that P is $\Sigma^0_2(A)$. For that, one defines uniformly for all e a partial computable function $f_e : \mathbb{N} \to \{0, 1\}$ such that:

(a) $e \in P$ implies that f_e is a total computable function.

(b) $e \notin P$ implies that f_e is a partial function which cannot be completed into a total computable function.

Let e be fixed. We describe a uniform process in e. At the stage of computation t, for any value n smaller than t and such that f_e does not halt for the moment on n, we proceed as follows : If $\Phi_n(n)[t]\downarrow \neq 0$ we define $f_e(n) = 0$. Otherwise if $\Phi_n(n)[t]\downarrow = 0$ we define $f_e(n) = 1$. Otherwise if for all $k \leqslant n$ there exists $m_k \leqslant t$ such that $R(e, k, m_k)$ then we define $f_e(n) = 0$.

The process is clearly computable. Let us show (a). Suppose $e \in P$. Then, for all n there exists a smallest t such that for all $k \leqslant n$ there exists $m_k \leqslant t$ for which $R(e, k, m_k)$. When that happens then $f_e(n)$ takes a value at step t if it has not taken one so far. So f_e is total. Let us now show (b). Suppose $e \notin P$. Let n be the largest integer such that for all $k \leqslant n$ there exists m_k for which $R(e, k, m_k)$. For $m > n$, $f_e(m)$ halts iff $\Phi_m(m)$ halts in which case $f_e(m) \neq \Phi_m(m)$. Suppose absurdly that f_e has a computable completion. Then, by the padding lemma (Lemma 3-5.1) it has a computable code completion $a > n$. In this case $f_e(a) = \Phi_a(a)$ which contradicts the definition of f_e. So we have (b).

It follows that P can be written as a $\Sigma^0_2(A)$ set as follows.

$$P = \{e : \exists n\ \forall m\ \forall t\ f_e(m)[t]\uparrow \vee f_e(m)[t]\downarrow = X_n(m)\}.$$

Indeed if $e \in P$ then f_e is computable and therefore coincides with a certain X_n. Conversely if $e \notin P$ then f_e has no computable completion and therefore no completion in $(X_n)_{n\in\mathbb{N}}$. Since P is $\Sigma^0_2(A)$ and $\overline{P}$ is Σ^0_2 by definition, the set P is then $\Delta^0_2(A)$ and therefore P is A' computable. It suffices to apply this for $P = \mathbb{N}\setminus\emptyset''$ to get that $\emptyset''$ is $\Delta^0_2(A)$ and therefore A'-computable. ∎

The implication (3) $\Rightarrow$ (1) of Theorem 6.2 is far from obvious. We hope the reader will appreciate the argument, the subtle complexity of which is characteristic of Jockusch's work. The following exercise gives similar alternative characterizations, simpler to demonstrate:

Exercise 6.3. ($\star$) Show the following equivalences:

(1) A computes a function which eventually dominates any computable function.

(2) A computes a sequence $(f_n)_{n\in\mathbb{N}}$ containing all the computable functions from $\mathbb{N}$ to $\mathbb{N}$.

(3) A compute a sequence $(X_n)_{n\in\mathbb{N}}$ containing only infinite sets, and containing all infinite computable sets.

$\diamond$

f_e	X_0	X_1	X_2	$\ldots$
$f_e(0)$	$X_0(0)$	$X_1(0)$	$X_2(0)$	
$f_e(1)$	$X_0(1)$	$X_1(1)$	$X_2(1)$	
$\uparrow$	$X_0(2)$	$X_1(2)$	$X_2(2)$	
$\uparrow$	$X_0(3)$	$X_1(3)$	$X_2(3)$	
$f_e(4)$	$X_0(4)$	$X_1(4)$	$X_2(4)$	
$\uparrow$	$X_0(5)$	$X_1(5)$	$X_2(5)$	
$f_e(6)$	$X_0(6)$	$X_1(6)$	$X_2(6)$	
$\uparrow$	$X_0(7)$	$X_1(7)$	$X_2(7)$	
$\ldots$	$\ldots$	$\ldots$	$\ldots$	

Figure 6.4: Illustration of the direction (3) $\Rightarrow$ (1) of Theorem 6.2 : if $e \notin P$, one constructs a partial computable function $f_e : \mathbb{N} \to \{0,1\}$ which has no total computable completion. In the illustration, the f_e function cannot equal any of the sets $X_0, X_1, \ldots$ on all the values for which it is defined. Indeed, the list $(X_n)_{n\in\mathbb{N}}$ contains only computable sets. In the inverse case, the f_e function is total computable, hence equals at least one of the sets X_i, since the list $(X_n)_{n\in\mathbb{N}}$ contains all the computable sets.

Let us discuss a little about the characterization (1) $\Leftrightarrow$ (2) of Theorem 6.2, illustrated by Figure 6.1. Being able to compute a very fast growing function from $\mathbb{N}$ into $\mathbb{N}$ is likely to provide a lot of computational power. Thus, for example, as we have seen, computing a function which dominates the halting time of computer programs makes it possible to compute $\emptyset'$.

The previous theorem indicates that there is a first level between the functions able to compute the halting set simply because they grow very fast, and the functions with computable growth: there exists, according to Proposition 4-10.2, high sets that do not compute the halting set, and therefore of functions of "intermediate" growth. We will also see that the faster a function grows, the more computational power it has. A function that grows fast enough will be able to compute $\emptyset''$, one that grows even faster will be able to compute $\emptyset'''$, and so on. In spite of everything, we will see with Theorem 29-5.5 that there is a limit to the computational power conferred by rapid growth. The class of sets computable by any function which grows "sufficiently" fast has a precise characterization and remains countable.

Exercise 6.5. ($\star\star$) The notion of maximal c.e. set was introduced in Exercise 3-7.13. A c.e. set X is maximal if $\mathbb{N} \setminus X$ is infinite, and if any c.e. set $Y \supseteq X$ is such that $Y \setminus X$ is finite or such that $\mathbb{N} \setminus Y$ is finite. Let X be a maximal c.e. set. Show that X is high. $\diamond$

7. High or DNC degrees

We end this chapter with a result that combines high and DNC degrees, in order to obtain a natural characterization in terms of computational power: the possibility of computing a function which differs almost everywhere from any computable function.

Theorem 7.1 (Kjos-Hanssen, Merkle and Stephan [117]) *Let $X \in 2^{\mathbb{N}}$. The following statements are equivalent:*

(1) X is of high or DNC degree.

(2) X computes a function which is different almost everywhere from any computable function.

Proof. Let us show (1) $\Rightarrow$ (2). Suppose first that X is high. Let $g \leqslant_T X$ be a function which eventually dominates any computable function. So also, $m \mapsto g(m) + 1$ is almost everywhere different from any computable function. Now suppose that X is DNC. For each n we compute the code e_n such that W_{e_n} enumerates all the values $\Phi_e(n)$ for $e \leqslant n$ for which $\Phi_e(n)\downarrow$. According to Theorem 2.6, X computes a function $h : \mathbb{N}^2 \to \mathbb{N}$ such that for all $e, n \in \mathbb{N}$, if $|W_e| \leqslant n$ then $h(e, n) \notin W_e$. Let $f : \mathbb{N} \to \mathbb{N}$ be the X-computable function defined by $f(n) = h(e_n, n)$. We then have $f(n) \notin W_{e_n}$ for all n. Thus, $f(n)$ is different from $\Phi_0(n), \Phi_1(n), \dots, \Phi_n(n)$ for all n, so f differs almost everywhere from any computable function (and even from any partial computable function halting on an infinity of values).

Let us show (2) $\Rightarrow$ (1). If X is high there is nothing to check. So suppose that X is not high. Let $g \leqslant_T X$ be such that total Φ_e implies $\forall^\infty m\ g(m) \neq \Phi_e(m)$. We claim that we also have $\forall^\infty e\ g(e) \neq \Phi_e(e)$. Let us assume the contrary to be absurd. Let f be the X-computable function which on n returns the smallest computation time t such that we have $g(m) = \Phi_m(m)[t]\downarrow$ for an integer $m > n$. Since X is not high there exists a computable function b such that $\exists^\infty n\ b(n) \geqslant f(n)$. Note that we can assume without loss of generality $b(n) \leqslant b(n+1)$. We now define the total computable function h by $h(n) = \Phi_n(n)[b(n)]$ if $\Phi_n(n)[b(n)]\downarrow$ and $h(n) = 0$ otherwise. Suppose now $b(n) > f(n)$. By definition of f, there exists a value $m > n$ such that $\Phi_m(m)[f(n)]\downarrow= g(m)$ and therefore such that $\Phi_m(m)[b(n)]\downarrow= g(m)$. As $b(m) \geqslant b(n)$ we will have $h(m) = \Phi_m(m)[b(m)]\downarrow= g(m)$. As we can restart the argument for arbitrarily large values of n, we will have $h(m) = g(m)$ for an infinity of m. This contradicts the fact that g differs almost everywhere from any computable function. So we have $\forall^\infty e\ g(e) \neq \Phi_e(e)$. It suffices then to modify a finite number of values of g to obtain a DNC function. ■

Let us note, to give all its brilliance to the preceding theorem, that it is possible to construct DNC degrees which are not high (by combining Corollary 18-4.3 with Corollary 19-3.9 or more simply by considering Corollary 8 -6.6) just like high degrees which are not are not DNC (see Corollary 10 -3.34).

Chapter 8

Π^0_1 classes and PA degrees

We have so far mainly concentrated on the study of sets of natural integers taken individually, or in an equivalent manner, of functions over the integers. We will now turn to the study of *classes* of sets or functions, i.e., sets of sets of integers. We have already seen many classes of sets, in particular the class of sets of high degree $\{X \in 2^{\mathbb{N}} : X' \geqslant_T \emptyset''\}$, or that of sets of low degree $\{X \in 2^{\mathbb{N}} : X' \equiv_T \emptyset'\}$.

Set vs class

In order to distinguish sets of integers from subsets of Cantor space, we will call them "sets" and "classes" when we are in a context relating to the study of sets $X \in 2^{\mathbb{N}}$ or classes $\mathcal{A} \subseteq 2^{\mathbb{N}}$, respectively.

The classes that we will consider will be defined by predicates, and the study of the complexity of these predicates will allow us to deduce information about the elements that the class contains.

The study of classes will quickly prove to be central in the study of sets of integers. We start here with the classes of the simplest possible complexity: the effectively open and closed classes of Cantor space. The study of classes of higher complexity will be continued in Chapter 17. Despite the apparent simplicity of open and closed classes, we will quickly see the wide range of possibilities and the great richness they contain.

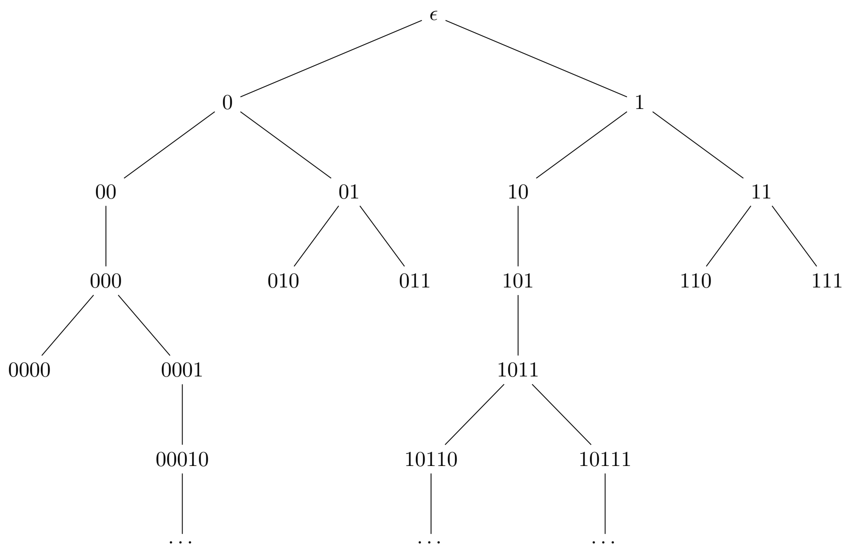


Figure 1.2: Illustration of a tree. The root ϵ is the empty string. Each node has potentially a left and a right successor. If both, then it is a *branching node*, and if none, it is a *leaf*.

1. Binary trees

We have defined the notion of f-tree in order to show the existence of a non-computable computably dominated degree (Theorem 7-5.6). We introduce here a similar notion and in a certain way more primitive, namely, the notion of tree.

Definition 1.1. A set $T \subseteq 2^{<\mathbb{N}}$ is a *tree* if T is closed under prefix, i.e., for all $\sigma \in T$ and $\tau \preceq \sigma$, then $\tau \in T$. $\diamondsuit$

The elements σ of a $T \subseteq 2^{<\mathbb{N}}$ are called *nodes*. A node is *branching* if both $\sigma 0, \sigma 1 \in T$. In the opposite case it is *non branching*. Based on the graphical representation of a tree, we will consider that $\sigma 0$ and $\sigma 1$ are on the left and on the right of σ, respectively. Accordingly, if $\sigma 0 \in T$ then $\sigma 0$ is a *left successor* of σ. If $\sigma 1 \in T$ then $\sigma 1$ is a *right successor of* σ. A node with no successor will be called a *leaf*.

Definition 1.3. Let $T \subseteq 2^{<\mathbb{N}}$ be a tree. A *path* through the tree T is a sequence $P \in 2^{\mathbb{N}}$ such that $P\upharpoonright_n \in T$ for all $n \in \mathbb{N}$. We denote by $[T]$ the class of paths of T. $\diamondsuit$

Intuitively, a path P can be thought of as a sequence of binary instructions, literally indicating "a path" to follow through the tree. A bit at 0 in the path tells us to continue our journey following the left successor and a bit at 1 tells us to follow the right successor. We only consider infinite paths.

Remark

Contrary to graph theory, a path is not represented as a set of nodes in the tree. However, there is a computable bijection between a path P and the set $\{P\upharpoonright_n : n \in \mathbb{N}\}$. Representing a path as an infinite binary sequence is therefore mainly a conventional choice which will prove to be very useful later.

If T is a finite tree, i.e., which has only a finite number of nodes, then $[T]$ is necessarily the empty set. What about the converse? This is a central tool on trees: König's lemma, which states that any infinite, finitely-branching tree has an infinite path. Note that trees $T \subseteq 2^{<\mathbb{N}}$ are necessarily 2-branching. The restriction of König's lemma to binary trees is known as *weak* König's lemma.

Lemma 1.4 (Weak König's lemma). Let $T \subseteq 2^{<\mathbb{N}}$ be an infinite tree, that is, such that $|T| = \infty$. Then, $[T]$ is non-empty. ⋆

Proof. We construct a path X by induction on n. As T is infinite, by the pigeonhole principle there exists $i \in \{0,1\}$ and an infinity of nodes $\sigma \in T$ which extend i (ie with $i \prec \sigma$). We define $X(0) = i$. Suppose that $\tau = X(0)X(1)\dots X(n)$ is defined with $\tau \in T$ and such that there is an infinity of nodes $\sigma \in T$ for which $\tau \preceq \sigma$. By the pigeonhole principle, there exists $i \in \{0,1\}$ and an infinity of nodes $\sigma \in T$ which extend τi. We define $X(n+1) = i$.

By induction on n, we thus define a set X such that $X\upharpoonright_n \in T$ for all n. ■

König's lemma may seem trivial at first glance. Readers with a keen intuition and comfortable with the manipulation of infinite objects may have wondered whether there really was a need to tidy up this statement in a lemma. We will see that despite appearances, this lemma is not as trivial as it seems. Although simple, it constitutes a central tool, particularly interesting for its computational content.

1.1. Computable trees

A tree $T \subseteq 2^{<\mathbb{N}}$ is computable if T is computable as a set, in other words, if there is a procedure to decide whether a node belongs to the tree or not. In this chapter, we will try to answer the following question.

Question 1.5. Given an infinite computable tree, what is the computational power required to compute an infinite path of T? ⋆

The proof of König's lemma provides a clear construction: when we have computed a prefix τ of our infinite path, we determine the next bit to be 0 if the tree contains an infinity of nodes which extend $\tau 0$ and as being 1 otherwise. The problem is that it is not possible to know a priori in a computable way if an infinity of nodes extend $\tau 0$: it is a question for which the halting problem seems necessary. This leads us to define the notion of extendible node.

Definition 1.6. A node σ of a tree $T \subseteq 2^{<\mathbb{N}}$ is *extendible* in T if the set $\{\tau \in T : \tau \succeq \sigma\}$ is infinite. ♢

In other words, a node σ is extendible in a tree if the subtree of nodes compatible with σ is infinite. Let us remember the notation $[\sigma]$ which denotes the class of sets X having σ as a prefix. By König's lemma, a node σ is extendible in T if and only if $[\sigma] \cap [T] \neq \emptyset$. Note that in any infinite tree, the root ϵ is an extendible node, and that if σ is extendible, then so is at least one node among $\sigma 0$ and $\sigma 1$. The following exercise shows that the extendible nodes are sufficient to describe the set of paths of a tree.

Exercise 1.7. Let $T \subseteq 2^{<\mathbb{N}}$ be a tree, and S the set of extendible nodes in T. Show that S is a tree, and that $[T] = [S]$. ⋄

It follows from the definition of extendible node that leaves are not extendible. On the other hand, if the set of leaves of a computable tree is decidable, this is not generally the case for the set of extendible nodes.

Exercise 1.8. Let $T \subseteq 2^{<\mathbb{N}}$ be an infinite computable tree containing only extendible nodes. Show that T contains a computable infinite path. ⋄

In general, determining whether a computable set is infinite or not requires the oracle $\emptyset''$. In the case of trees, we can exploit the closure under prefix to reduce the complexity of the oracle to $\emptyset'$. The notation 2^n of the following proposition denotes the set of strings of size n.

Proposition 1.9. Let $T \subseteq 2^{<\mathbb{N}}$ be a computable tree. The set of its extendible nodes is Π^0_1. ⋆

Proof. The trees being downward-closed, $\{\tau \in T : \tau \succeq \sigma\}$ is infinite iff $\forall n > |\sigma|\ \exists \tau \in 2^n$ such that $\tau \succeq \sigma$ and $\tau \in T$, which is a Π^0_1 predicate. Thus, the set of extendible nodes of T is the following Π^0_1 set.

$$\{\sigma \in T : \forall n > |\sigma|\ \exists \tau \in 2^n \text{ such that } \tau \succeq \sigma \text{ and } \tau \in T\}$$ ■

By complementation, the set of non-extendible nodes of a computable tree is Σ^0_1, which means that if a node is non-extendible, we will end up realizing it after a finite time. We will see how to use the notion of extendible node to create infinite computable trees having no computable path. In the construction of a computable tree, we must be able to decide in a finite time whether a node belongs to it or not. On the other hand, it is possible to defer the decision to make a node extendible or not, by default adding descendants over time to it, until deciding at one point t to stop adding more to it in order to make it non-extendible. This technique known as "time trick" allows to show the following result.

Proposition 1.10. Let $T \subseteq 2^{<\mathbb{N}}$ be a Π^0_1 tree. There exists a computable tree $S \subseteq 2^{<\mathbb{N}}$ such that $[T] = [S]$. ⋆

PROOF. Let $(T_n)_{n\in\mathbb{N}}$ be a Π^0_1 approximation of T, that is to say a uniformly computable sequence of sets decreasing by the inclusion relation (with $T_{n+1} \subseteq T_n$) such that $\bigcap_n T_n = T$. Let $S = \{\sigma \in 2^{<\mathbb{N}} : \forall \tau \preceq \sigma\ \tau \in T_{|\sigma|}\}$. The set S is computable.

Let us show that S is closed under prefix. Let $\sigma \in S$ and $\rho \preceq \sigma$. By definition of S, $\forall \tau \preceq \sigma\ \tau \in T_{|\sigma|}$. As $|\rho| \leqslant |\sigma|$, $T_{|\rho|} \supseteq T_{|\sigma|}$, so $\forall \tau \preceq \sigma\ \tau \in T_{|\rho|}$. In particular, for all $\tau \preceq \rho$, also $\tau \preceq \sigma$, so $\tau \in T_{|\rho|}$. Thus, by definition of S, $\rho \in S$.

Let us now show that $[S] = [T]$. We have $P \in [S]$ iff $\forall \sigma \prec P\ \sigma \in S$ iff $\forall \sigma \prec P\ \forall \tau \preceq \sigma\ \tau \in T_{|\sigma|}$ iff $\forall \tau \prec P\ \forall n \geqslant |\tau|\ \tau \in T_n$ iff $\forall \sigma \prec P\ \sigma \in T$ iff $P \in [T]$. ■

This chapter mainly concerns the study of the set classes corresponding to paths through a computable tree. We will see with Proposition 3.5 that there are infinite computable trees that do not contain any infinite computable path. Then, we will determine the exact computational power that is needed to compute a path in any infinite computable tree. This study constitutes one of the basic building blocks of reverse mathematics, which we will see in Part III.

2. Topology on Cantor space

Topology is a branch of mathematics which abstractly formalizes the notions of limit and continuity, and which by extension studies the properties of geometric objects invariant by continuous deformation. The reader who has never studied this branch can be reassured: for the development of the chapters to come, we only need very basic elements of this theory, which we present here.

Notation

We will denote by i^∞ the infinite sequence which repeats the bit $i \in \{0,1\}$.

2.1. Open and closed classes

As we have already mentioned, Cantor space $2^{\mathbb{N}}$ is similar to the class of reals of the interval $[0,1]$. An element $X \in 2^{\mathbb{N}}$ can also be seen as the binary expansion of the real $0.X(0)X(1)X(2)\ldots$, with however a subtle difference: the elements $\sigma 10^\infty$ and $\sigma 01^\infty$ are two distinct elements of $2^{\mathbb{N}}$ but correspond to the same real number. This difference aside, we can see $2^{\mathbb{N}}$ as an interval, and the simplest subsets of $2^{\mathbb{N}}$ will simply be intervals of the form $[\sigma]$, which we will also call *cylinder*:

Notation

Given a string $\sigma \in 2^{<\mathbb{N}}$ we write $[\sigma]$ for the set $\{X \in 2^{\mathbb{N}} : X \succeq \sigma\}$. We will call *cylinder* a class of the form $[\sigma]$.

Given a string $\sigma \in 2^{<\mathbb{N}}$, we can see $[\sigma]$ as an interval of infinite binary sequences: those which are lexicographically between $\sigma 0^\infty$ and $\sigma 1^\infty$. With this in mind, the so-called *open* classes of Cantor are simply any union of intervals.

Definition 2.1. The *open* classes of Cantor space are arbitrary unions of cylinders, i.e., sets of the form $\bigcup_{\sigma \in W}[\sigma]$ for a set $W \subseteq 2^{<\mathbb{N}}$. The *closed* classes are the complements of the open classes. $\diamondsuit$

The reader not familiar to these concepts can demonstrate, to get his hands on the definitions, that the finite unions of cylinders are both open and closed.

Notation

Given a set $W \subseteq 2^{<\mathbb{N}}$ we will write $[W]$ to denote its corresponding open set, i.e., the class $\bigcup_{\sigma \in W}[\sigma]$.

It is clear from the definition that the open classes are closed under arbitrary union and therefore, by passage to the complement, that the closed are closed under any intersection. We introduce an element of vocabulary that will often come up in the manipulation of open and closed classes:

Notation

Given a countable union of classes $\bigcup_n \mathcal{B}_n$, we will say that the union is *increasing* if $\mathcal{B}_n \subseteq \mathcal{B}_{n+1}$ for all n. In the same way we will say that an intersection $\bigcap_n \mathcal{B}_n$ is *decreasing* if $\mathcal{B}_{n+1} \subseteq \mathcal{B}_n$ for all n.

The mental representation of an open class should be fairly clear to the reader: the unions of simple bricks that are the cylinders. Here is an illustrative example.

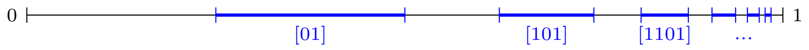


Figure 2.2: Illustration of the open class $[01] \cup [101] \cup [1101] \cup [11101] \cup \ldots$

We now show that we can consider without loss of generality that the intersections of open classes are decreasing and the unions of closed classes increasing (in particular because $\bigcap_n \mathcal{U}_n = \bigcap_n (\bigcap_{m \leqslant n} \mathcal{U}_m)$):

Proposition 2.3. A finite intersection of open classes is an open. By passing to the complement a finite union of closed classes is closed. ⋆

Proof. Let $\mathcal{U}_0, \mathcal{U}_1 \subseteq 2^{\mathbb{N}}$ be two open classes. A set X belongs to $\mathcal{U}_0 \cap \mathcal{U}_1$ iff it belongs to a cylinder $[\sigma_0] \subseteq \mathcal{U}_0$ as well as to a cylinder $[\sigma_1] \subseteq \mathcal{U}_1$. So $\mathcal{U}_0 \cap \mathcal{U}_1 = \bigcup_{[\sigma_0] \subseteq \mathcal{U}_0, [\sigma_1] \subseteq \mathcal{U}_1} [\sigma_0] \cap [\sigma_1]$. ■

Closed sets are more difficult to describe. This is not necessarily surprising. By way of analogy, let's say that you can get to know your neighborhood well, without having a precise idea of the rest of the world. The awareness of this complexity and the way of apprehending it already figure in the work of Cantor, for example through the famous Cantor-Bendixson theorem. However, there is a simple way to represent the closed in $2^{\mathbb{N}}$ as the class of all infinite paths of a tree.

Proposition 2.4. A class $\mathcal{P} \subseteq 2^{\mathbb{N}}$ is closed iff there is a tree $T \subseteq 2^{<\mathbb{N}}$ such that $\mathcal{P} = [T]$. ⋆

Proof. Let $\mathcal{P}$ be a closed class. Let $\mathcal{U} = \bigcup_{\sigma \in W} [\sigma]$ be its complement with $W \subseteq 2^{<\mathbb{N}}$. We define the tree $T \subseteq 2^{<\mathbb{N}}$ as being the set of strings σ having no prefix in W. By definition T is closed under prefix and is therefore a tree. Let us show $[T] = \mathcal{P}$. We have $X \in [T]$ iff no prefix $\sigma \prec X$ is in W iff $X \notin \bigcup_{\sigma \in W} [\sigma]$ iff $X \in \mathcal{P}$. So $[T] = \mathcal{P}$.

Conversely, if $T \subseteq 2^{<\mathbb{N}}$ is a tree, the class $[T]$ of its paths is closed, because it is the complement of the class $\bigcup_{\sigma \notin T} [\sigma]$. ■

By way of example, the reader can consult Figure 2.5, representing the tree corresponding to the complement of the open class described in Figure 2.2.

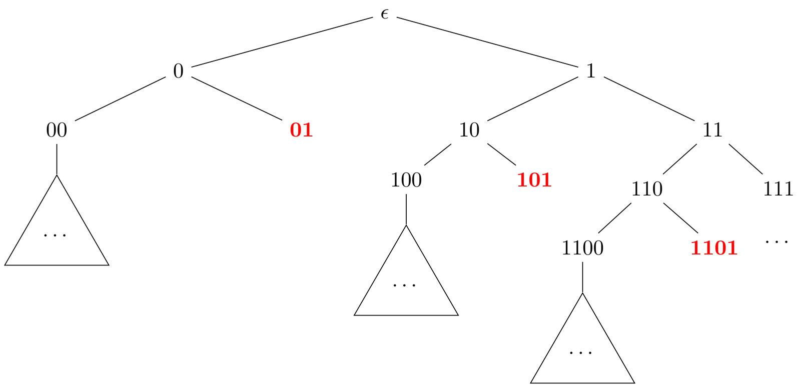


Figure 2.5: Illustration of the tree representing the complement of the open class $[01] \cup [101] \cup [1101] \cup [11101] \cup \ldots$. The bold nodes correspond to the cylinders constituting the basic bricks of the open class. The triangles represent a "full" subtree, starting from the node where they are.

2.2. Compactness

Compactness is a fundamental notion of topology. It is generally defined via the Borel-Lebesgue property, which in Cantor space is formulated as follows:

Definition 2.6. A class $\mathcal{P} \subseteq 2^{\mathbb{N}}$ has the *Borel-Lebesgue property*[a] if for any collection of open classes $(\mathcal{O}_n)_{n\in\mathbb{N}}$ such that $\mathcal{P} \subseteq \bigcup_n \mathcal{O}_n$, there exists a finite set $F \subseteq \mathbb{N}$ such that $\mathcal{P} \subseteq \bigcup_{n\in F} \mathcal{O}_n$. We will say that a class possessing the Borel-Lebesgue property is *compact*. $\diamondsuit$

[a]Also called "Heine-Borel property".

We show with the following proposition that in Cantor space, the compact classes are exactly the closed ones, and the reader will be able to note by reading the proof, that weak König's lemma can be seen as a reformulation of the fact that closed classes are compact.

Proposition 2.7. A class of $2^{\mathbb{N}}$ is closed iff it has the Borel-Lebesgue property. $\star$

Proof. Let $\mathcal{P} \subseteq 2^{\mathbb{N}}$ be closed and let $(\mathcal{O}_n)_{n\in\mathbb{N}}$ be a collection of open classes such that $\mathcal{P} \subseteq \bigcup_n \mathcal{O}_n$. Let $T \subseteq 2^{<\mathbb{N}}$ be a tree such that $[T] = \mathcal{P}$. Let $S \subseteq 2^{<\mathbb{N}}$ be the tree of strings $\sigma \in T$ such that $[\sigma] \not\subseteq \bigcup_{n<|\sigma|} \mathcal{O}_n$. If the tree S is finite, then there exists a length ℓ such that for all $\sigma \in T$ such that $|\sigma| = \ell$, $[\sigma] \subseteq \bigcup_{n<\ell} \mathcal{O}_n$. It follows that $[T] \subseteq \bigcup_{\sigma \in T, |\sigma|=\ell}[\sigma] \subseteq \bigcup_{n<\ell} \mathcal{O}_n$. If S is infinite, by weak König's lemma, $[S] \neq \emptyset$. Let $P \in [S]$. Let us show that $P \notin \bigcup_n \mathcal{O}_n$ to deduce a contradiction, because $[S] \subseteq [T] \subseteq \bigcup_n \mathcal{O}_n$. $i \in \mathbb{N}$. By definition of $[S]$, for all ℓ, $P \upharpoonright_\ell \in S$, so by definition of S we have $[P \upharpoonright_\ell] \nsubseteq \bigcup_{n<\ell} \mathcal{O}_n$. In particular, for all $\ell > i$, $[P \upharpoonright_\ell] \nsubseteq \mathcal{O}_i$. Since $\mathcal{O}_i$ is open, it follows that $P \notin \mathcal{O}_i$.

Suppose now that a class $\mathcal{B}$ admits the Borel-Lebesgue property. For any $X \notin \mathcal{B}$, let $\mathcal{O}_X$ be the open class corresponding to the complement of the class $\{X\}$ (it is the union of the cylinders $[\sigma i]$ for any string σ and all i such that $X(|\sigma|) \neq i$). In particular we have $\mathcal{B} = \bigcap_{X \notin \mathcal{B}} \mathcal{O}_X$. Note that each $\mathcal{O}_X$ is a union of cylinders and that each cylinder is open. So by the Borel-Lebesgue property we can find for any X a finite set of cylinders F_X such that $\mathcal{B} \subseteq \bigcup_{\sigma \in F_X} [\sigma] \subseteq \mathcal{O}_X$. In particular $\mathcal{B} = \bigcap_{X \notin \mathcal{B}} \bigcup_{\sigma \in F_X} [\sigma]$. Each union $\bigcup_{\sigma \in F_X} [\sigma]$ is a closed class as a finite union of cylinders. As an arbitrary intersection of closed classes is closed, we deduce that $\mathcal{B}$ is a closed class. ■

In practice, we will use the following consequence of compactness: any countable and decreasing intersection of non-empty closed classes is non-empty, which we prove here.

Proposition 2.8. Let $\mathcal{P}_0 \supseteq \mathcal{P}_1 \supseteq \dots$ be a decreasing sequence of non-empty closed classes. Then, $\bigcap_n \mathcal{P}_n$ is non-empty. ⋆

Proof. Let T_n be a tree such that $[T_n] = \mathcal{P}_n$. We can assume without loss of generality $T_{n+1} \subseteq T_n$. Let us show that $\bigcap_n [T_n] = [\bigcap_n T_n]$. We have $X \in \bigcap_n [T_n]$ iff $X \upharpoonright_m \in T_n$ for all m, n iff $X \in [\bigcap_n T_n]$. So $\bigcap_n [T_n] = [\bigcap_n T_n]$. Let $T = \bigcap_n T_n$. In particular $[T] = \bigcap_n \mathcal{P}_n$.

Suppose absurdly that $[T] = \bigcap_n \mathcal{P}_n$ is empty. According to König's lemma there is therefore an integer a such that no string σ of size greater than or equal to a is in T. As the strings of size a are in finite quantity and the sequence $(T_n)_{n \in \mathbb{N}}$ is decreasing by inclusion, there must therefore be integer m such that none of these strings belong to T_m. So $[T_m]$ is empty, which contradicts the assumptions. ■

2.3. Continuity

Let's tackle another topological notion that we will mention here and there in the chapters to come. Continuity, another central notion of topology, unexpectedly hides in computability theory under the following idea.

A Turing functional Φ can be seen as a partial function from $2^{\mathbb{N}}$ to $2^{\mathbb{N}}$, whose input is an oracle X, and the result, which we denote by Φ^X or $\Phi(X)$, is the set Y such that $\Phi^X(n)\downarrow= Y(n)$. This function from $2^{\mathbb{N}}$ to $2^{\mathbb{N}}$ is of course only defined for oracles X such that $\forall n\ \Phi^X(n)\downarrow\ \in \{0,1\}$.

We have seen that when $\Phi^X(n)\downarrow= v$, by the use property (see Definition 4-4.2), only a finite initial segment σ of the oracle X is used. More generally, if $\Phi^X \succeq \tau$ (which means $\forall n < |\tau|\ \Phi^X(n)\downarrow= \tau(n)$), only a finite part σ of the oracle is used to realize this. It follows that for all $X \in [\sigma]$, $\Phi^X \succeq \tau$, and therefore $\{\Phi^X : X \in [\sigma]\} \subseteq [\tau]$.

The reader having followed an introductory course in topology will recognize in this idea the notion of continuity: for any open space of the image space as "small" as one wants — in practice a cylinder $[\tau]$ — there is a "small" enough open class in the domain space — in practice a cylinder $[\sigma]$ — such that every $X \in [\sigma]$ is sent to $[\tau]$: concretely the string σ is "sent" to the string τ.

Definition 2.9. A (possibly partial) function $f : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ is *continuous in* $X \in \operatorname{dom} f$ if for any cylinder $[\tau]$ containing $f(X)$, there exists a cylinder $[\sigma]$ containing X such that $f([\sigma]) \subseteq [\tau]$. We will say that a function is *continuous* if it is continuous in X for all $X \in \operatorname{dom} f$. ♢

In general, we will consider continuous functions over their entire domain of definition, which then admit an equivalent characterization in terms of open pre-image:

Proposition 2.10. A (possibly partial) function $f : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ is continuous iff for any open class $\mathcal{U} \subseteq 2^{\mathbb{N}}$, there exists an open class $\mathcal{V} \subseteq 2^{\mathbb{N}}$ such that $f^{-1}(\mathcal{U}) = \mathcal{V} \cap \operatorname{dom} f$. If the function is total we then have $f^{-1}(\mathcal{U})$ open for any open class $\mathcal{U}$. ⋆

Proof. Let $f : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ be a continuous function and $\mathcal{U} \subseteq 2^{\mathbb{N}}$ an open class. As $\mathcal{U}$ is open, for all $X \in f^{-1}(\mathcal{U})$ there exists a cylinder $[\tau_X]$ containing $f(X)$ such that $[\tau_X] \subseteq \mathcal{U}$. By continuity of f, for all X, there exists a cylinder $[\sigma_X]$ containing X such that $f([\sigma_X]) \subseteq [\tau_X]$. Then, $\operatorname{dom} f \cap \bigcup_{X \in f^{-1}(\mathcal{U})} [\sigma_X] = f^{-1}(\mathcal{U})$.

Conversely, suppose that for any open class $\mathcal{U} \subseteq 2^{\mathbb{N}}$, there exists an open class $\mathcal{V}$ such that $\operatorname{dom} f \cap \mathcal{V} = f^{-1}(\mathcal{U})$. Let $Y \in \operatorname{Im} f$ and $[\tau]$ be a cylinder containing Y. Let $\mathcal{V}$ be an open class such that $\operatorname{dom} f \cap \mathcal{V} = f^{-1}([\tau])$. Since $\mathcal{V}$ is open, there is $W \subseteq 2^{<\mathbb{N}}$ such that $\bigcup_{\sigma \in W} [\sigma] = \mathcal{V}$. Note that $W \neq \emptyset$ because $Y \in \operatorname{Im} f \cap [\tau]$, so there exists a string $\sigma \in W$. We have $f([\sigma]) = f(\operatorname{dom} f \cap [\sigma]) \subseteq [\tau]$. ∎

A computable functional Φ is therefore always also a continuous function on its domain of definition, i.e., on the space of X such that $\Phi(X, n) \downarrow \in \{0, 1\}$ for all n. On the other hand, a continuous function has a priori no reason to be computable: it is possible that any finite piece of the output of the function can be determined by a finite piece of the input, but that this "determinism" is not computable. As an example let us consider the function Φ which on any set X associates $X \oplus \emptyset'$. Such a function Φ is continuous, but not computable. On the other hand, it can be computed with the help of $\emptyset'$.

In computability theory, the non-continuous function par excellence is that which to X associates X': indeed to know whether $n \in X'$, one needs to know if $\Phi_n(X, n) \downarrow$ and for that potentially to know an infinity of bits of X (especially if $\Phi_n(X, n) \uparrow$). We will see actual versions of some well-known theorems of analysis, which state that any function that is not continuous, but not too complex, for example $X \mapsto X'$, is nevertheless continuous over a "large" set of points, in particular on a co-meager class (see Theorem 10 -3.20) and on a class of arbitrarily large measure (see Theorem 19-3.8).

2.4. Perfect classes

A last topological notion that we will use is that of perfect classes. These are the classes which are the image of a continuous injection from $2^{\mathbb{N}}$ to $2^{\mathbb{N}}$, that is to say exactly the classes of the form $[T]$ for an f-tree $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ (see Section 7-5). These classes are therefore always closed and can be represented by a tree. By extension we will therefore also speak of a perfect tree.

Definition 2.11. A non-empty tree $T \subseteq 2^{<\mathbb{N}}$ is perfect if any node in T has two incompatible extensions in T. $\diamondsuit$

We have seen with Exercise 1.7 that given a closed class $\mathcal{F}$ represented by a tree T, we can consider without loss of generality — if we do not deal with the effectiveness — that T contains only extendible nodes. On the other hand, an extendible node does not necessarily have two incompatible extensions in the general case. When this happens, it means that there is exactly one infinite path passing through this node. We call such paths *isolated points*. For an arbitrary class $\mathcal{A}$ the corresponding definition is as follows.

Definition 2.12. Let $\mathcal{A} \subseteq 2^{\mathbb{N}}$. An element $X \in \mathcal{A}$ is a *isolated point* if there is a $\sigma \prec X$ prefix such that $[\sigma] \cap \mathcal{A} = \{X\}$. $\diamondsuit$

The usual definition of perfect class then follows from that of isolated point:

Definition 2.13. A non-empty class $\mathcal{F} \subseteq 2^{\mathbb{N}}$ is *perfect* if it is closed and has no isolated point. Equivalently $\mathcal{F} = [T]$ for a perfect tree $T \subseteq 2^{<\mathbb{N}}$.$\diamondsuit$

Perfect classes are of great importance, in particular because they allow the construction of cardinality arguments: any perfect class is by definition in continuous bijection with $2^{\mathbb{N}}$, and therefore has the same cardinality as $2^{\mathbb{N}}$. Moreover if a class $\mathcal{A} \subseteq 2^{\mathbb{N}}$ contains a perfect class, then we have an injection of $2^{\mathbb{N}}$ into $\mathcal{A}$. Identity injection of $\mathcal{A}$ into $2^{\mathbb{N}}$ then gives us $|\mathcal{A}| = |2^{\mathbb{N}}|$. It is in fact roughly speaking *the only way* to show that a class $\mathcal{A} \subseteq 2^{\mathbb{N}}$ has the power of continuum. We will talk about it again in Section 9-4 as well as in Section 30-4. Here is a simple application of the notion of perfect class to the study of cardinality.

Proposition 2.14. Any non-empty countable closed class $\mathcal{F}$ has isolated points. We can inject $2^{\mathbb{N}}$ in any non-empty closed class with no isolated point. $\star$

Proof. Suppose that $\mathcal{F}$ does not contain any isolated point. Let $\mathcal{F} = [T]$ for a tree T having only extendible nodes. Since $\mathcal{F}$ does not contain an isolated point then all nodes of T have two incompatible extensions. We then have an injection of $2^{\mathbb{N}}$ into $\mathcal{F}$. If a closed class $\mathcal{F}$ is countable, we cannot inject $2^{\mathbb{N}}$ into $\mathcal{F}$ and therefore by contraposition, it contains isolated points. ■

Corollary 2.15 (Cantor)
The continuum hypothesis is true for closed classes of $2^{\mathbb{N}}$: they are either countable, or of cardinality $|2^{\mathbb{N}}|$.

Proof. Let $\mathcal{F}$ be closed. Let $W \subseteq 2^{<\mathbb{N}}$ be the set of strings σ such that $\mathcal{F} \cap [\sigma]$ is countable. Let $\mathcal{F}' = \mathcal{F} \setminus \bigcup_{\sigma \in W} [\sigma]$. Note that $\mathcal{F}' \subseteq \mathcal{F}$ is always a closed class. If $\mathcal{F}'$ is empty, then according to König's lemma (or compactness) it is only necessary to remove from $\mathcal{F}$ a finite number of strings $\sigma_0, \dots, \sigma_n \in W$ so that $\mathcal{F}' = \mathcal{F} \setminus [\sigma_0] \cup \dots \cup [\sigma_n]$ is empty. As each class $\mathcal{F} \cap [\sigma_i]$ is countable, we have emptied $\mathcal{F}$ by removing a countable quantity of points. So $\mathcal{F}$ is countable. Otherwise $\mathcal{F}'$ is not empty, and in addition to that it cannot contain isolated points (because if $\mathcal{F}' \cap [\sigma]$ contains only one element then $\mathcal{F} \cap [\sigma]$ is countable). By Proposition 2.14 we have an injection from $2^{\mathbb{N}}$ to $\mathcal{F}'$. ■

This result will be extended with Corollary 30-3.3.

3. Π^0_1 classes

We are now interested in the *effective* version of the open and closed classes. We often use the term *effective* rather than computable for this kind of objects, because they are not necessarily computable in the sense that we can precisely know the smallest details, but they still admit a certain tangible, effective description provided by an algorithm.

Definition 3.1. A class $\mathcal{U} \subseteq 2^{\mathbb{N}}$ is called Σ^0_1 if there exists a c.e. set $W \subseteq 2^{<\mathbb{N}}$ such that $\mathcal{U} = \bigcup_{\sigma \in W}[\sigma]$. A class $\mathcal{P} \subseteq 2^{\mathbb{N}}$ is said to be Π^0_1 if its complement is a Σ^0_1 class. $\diamondsuit$

The Σ^0_1 and Π^0_1 are respectively the *effectively* open and closed classes of Cantor space. We will call *code* of a Σ^0_1 class $\mathcal{U}$ an integer e such that $\mathcal{U} = \bigcup_{\sigma \in W_e}[\sigma]$. Likewise, a *code* of a Π^0_1 class $\mathcal{P}$ is a code of the Σ^0_1 class $\mathcal{U} = 2^{\mathbb{N}} \setminus \mathcal{P}$. This allows us to speak of uniform computability on the sequence of Σ^0_1 or Π^0_1 classes by considering the computability of their sequence of codes. The Π^0_1 classes are an important and well-studied notion in computability theory.

Remark

It is important to distinguish well the Σ^0_1 or Π^0_1 sets of integers which are the first levels of the arithmetic hierarchy (see Chapter 5) from the Σ^0_1 and Π^0_1 classes which are the respective effectively open and closed classes of Cantor space. However, there are links between these concepts.

Let's start with the fact that Proposition 2.3 which states that a finite intersection of open classes is an open one, also works with the Σ^0_1 classes:

Proposition 3.2. Σ^0_1 classes are closed under finite intersection. By passing to the complement, the Π^0_1 classes are closed under finite union. $\star$

Proof. Let $\mathcal{U}_0 = \bigcup_{\sigma \in W_0}[\sigma]$ and $\mathcal{U}_1 = \bigcup_{\sigma \in W_1}[\sigma]$ be two Σ^0_1 classes. Then, the open class $\mathcal{U}_0 \cap \mathcal{U}_1$ is described by the c.e. set which lists the longest string among σ_0, σ_1 for any $\sigma_0 \in W_0$ and $\sigma_1 \in W_1$ such that $\sigma_0 \preceq \sigma_1$ or such that $\sigma_1 \preceq \sigma_0$. The string thus enumerated corresponds to $[\sigma_0] \cap [\sigma_1]$. ■

The first step to better understand the nature of the Π^0_1 classes is undoubtedly to prove the effective version of Proposition 2.4: the Π^0_1 are exactly the infinite paths of computable trees.

Proposition 3.3. A class $\mathcal{P}$ is Π^0_1 iff there is a computable tree $T \subseteq 2^{<\mathbb{N}}$ such that $[T] = \mathcal{P}$. $\star$

Proof. Let $T \subseteq 2^{<\mathbb{N}}$ be a computable tree. The class $[T] = \{X : \forall n\ X\restriction_n \in T\}$ is Π^0_1. Indeed its complement is the Σ^0_1 class described by the union of cylinders $[\sigma]$ such that $\sigma \notin T$.

Suppose that $\mathcal{P}$ is Π^0_1. Let $\mathcal{U} = \bigcup_{\sigma\in W}[\sigma]$ be its complement. We compute the following tree $T \subseteq 2^{<\mathbb{N}}$: in the computation step t, for any string $\sigma \in 2^{<\mathbb{N}}$ of size t, we decide $\sigma \in T$ iff for any prefix $\tau \preceq \sigma$ we have $\tau \notin W[t]$.

It is clear that T is closed under prefix: if σ of size t is in T then no prefix of σ is in W at the computation step t, so for $s \leqslant t$, also no prefix of $\sigma\restriction_s$ is in W at the computation step s. Now if $X \in \mathcal{P}$ then no σ prefix of X is in W and therefore each of those prefixes will be in T. Conversely if $X \notin \mathcal{P}$ then a prefix σ of X goes into W at a certain stage t. By construction no strings $\tau \succeq \sigma$ larger than t will be in T. So $\mathcal{P} = [T]$. ■

A code of a computable tree $T \subseteq 2^{<\mathbb{N}}$ is an integer e such that $\Phi_e = T$. Note that the proof of Proposition 3.3 is uniform, and allows to pass computably from a code of a Π^0_1 class to a code of the corresponding tree, and vice versa. We can therefore consider without distinction the code of Π^0_1 classes and computable trees in the proofs to come. The following proposition establishes a link with the Σ^0_1 and Π^0_1 classes and the arithmetic hierarchy.

Proposition 3.4. Let $\mathcal{P} \subseteq 2^{\mathbb{N}}$ be a class.

(1) $\mathcal{P}$ is Σ^0_1 iff $\mathcal{P} = \{X \in 2^{\mathbb{N}} : \exists n\ R(X\restriction_n)\}$ for a computable predicate $R \subseteq 2^{<\mathbb{N}}$

(2) $\mathcal{P}$ is Π^0_1 iff $\mathcal{P} = \{X \in 2^{\mathbb{N}} : \forall n\ R(X\restriction_n)\}$ for a computable predicate $R \subseteq 2^{<\mathbb{N}}$ ⋆

Proof. For (2), given a Π^0_1 class $\mathcal{P}$, it suffices to consider the computable tree T such that $[T] = \mathcal{P}$. The computable predicate is simply T. Conversely if $\mathcal{P} = \{X \in 2^{\mathbb{N}} : \forall n\ R(X\restriction_n)\}$ then the computable tree given by $\sigma \in T$ iff $\forall \tau \preceq \sigma\ R(\tau)$ is such that $[T] = \mathcal{P}$.

We obtain (1) by passing to the complement. ■

Let us now see some generalities, first of all the proof that König's lemma does not belong to computable mathematics: some non-empty Π^0_1 classes — and therefore some infinite computable trees — do not contain any computable point. We will see throughout the following chapters many examples of Π^0_1 classes not containing any computable point. We anticipate in particular for the following proposition on the simplest example to define: the class of DNC_2 sets of Proposition 6.4.

Proposition 3.5. There are non-empty Π^0_1 classes that do not contain any computable set. ⋆

Proof. We define the class

$$\mathcal{P} = \{X \in 2^{\mathbb{N}} : \forall e\ \forall t\ \Phi_e(e)[t] \uparrow \vee\ \Phi_e(e)[t] \downarrow \neq X(e)\}.$$

The class $\mathcal{P}$ contains all the sets X such that $X(n)$ can take any value if $\Phi_n(n) \uparrow$, and which are always different from $\Phi_n(n)$ if $\Phi_n(n) \downarrow$. It is clear that this class is not empty (the halting problem for example easily computes an element of $\mathcal{P}$). By Proposition 3.4, $\mathcal{P}$ is a Π^0_1 class. Moreover, the class $\mathcal{P}$ does not contain any computable set: if X is computable then there must be some e such that $\Phi_e(e) \downarrow = X(e)$. ■

Let us now continue on another key property of Π^0_1 classes: the non-empty Π^0_1 classes containing no computable points are necessarily uncountable. They cannot contain *isolated points*, that is to say sets X such that for a certain n, no other set than X and extending $X \upharpoonright_n$ does not belong to Π^0_1.

Proposition 3.6. Let $\mathcal{P}$ be a Π^0_1 class containing exactly one X element. Then, X is computable. ⋆

Proof. Let $T \subseteq 2^{<\mathbb{N}}$ be the computable tree such that $[T] = \mathcal{P}$. Consider the following algorithm: search for the smallest t such that either for any string $\sigma \succeq 0$ of size t we have $\sigma \notin T$, or for any string $\sigma \succeq 1$ of size t we have $\sigma \notin T$. Note that exactly one of the two events must necessarily happen: if both events happen, the class is empty. If neither happens there is an infinity of strings in T which extend 0 and also an infinity which extend 1. According to König's lemma T therefore contains at least two infinite paths: one which extends 0 and one which extends 1, which contradicts the hypotheses on $\mathcal{P}$.

Once one of the two events has arrived, we therefore know whether X begins with 0 or 1. We can easily see how to continue by induction: once $X \upharpoonright_n$ has been computed, we look for the smallest t such that for any string $\sigma \succeq X \upharpoonright_n 0$ of size t we have $\sigma \notin T$, or for any string $\sigma \succeq X \upharpoonright_n 1$ of size t we have $\sigma \notin T$. Once one of the two events has occurred, the value of $X(n)$ is known. ■

Corollary 3.7
Isolated points of any Π^0_1 class are computable.

Proof. Let $\mathcal{P}$ be a Π^0_1 class and $X \in \mathcal{P}$ an isolated point. By definition, there is a prefix $\sigma \prec X$ such that $[\sigma] \cap \mathcal{P} = \{X\}$. In particular $[\sigma] \cap$

$\mathcal{P}$ is a Π^0_1 class containing exactly one element, this element is therefore computable. ■

Corollary 3.8
Any countable Π^0_1 class contains a computable set.

Proof. By Proposition 2.14 and Corollary 3.7. ■

4. Basis theorems

Members of a Π^0_1 class can be of very different Turing degrees. For example, Cantor space $2^{\mathbb{N}}$ is a Π^0_1 class containing sets of each Turing degree. Given a non-empty Π^0_1 class, we are mainly interested in the degree of difficulty of computing one of its members.

Definition 4.1. A *basis* for Π^0_1 classes is a class of sets $\mathcal{C}$ such that any non-empty Π^0_1 class contains an element of $\mathcal{C}$. ♢

In this section, we will prove a number of theorems which, given a weakness property P, are of the form "Any non-empty Π^0_1 class contains a member satisfying P." These theorems are called "basis theorems", because they state that the members of P form a basis for the Π^0_1 classes. Conversely, the "anti-basis theorems" state the existence of a non-empty Π^0_1 class containing no member satisfying a weakness property. The very first basis theorem is due to Kreisel [128], and is left as an exercise.

Exercise 4.2. ($\star$) Show that any non-empty Π^0_1 class contains a $\emptyset'$-computable element. ◇

We can do even better than Exercise 4.2 via the central theorem called "low basis theorem", which states that any non-empty Π^0_1 class contains a low set. This theorem has a fundamental importance in computability theory and in reverse mathematics, in particular to provide a number of examples and counter-examples.

Theorem 4.3 (Jockusch et Soare [108])
Any non-empty Π^0_1 class contains a low set.

Proof. Let $\mathcal{P}$ be a non-empty Π^0_1 class. We are going to use $\emptyset'$ to compute an element $Z \in \mathcal{P}$, while computing its Turing jump Z'. For that, let us define a uniformly $\emptyset'$-computable decreasing sequence of non-empty Π^0_1

classes $\mathcal{P} = \mathcal{P}_0 \supseteq \mathcal{P}_1 \supseteq \mathcal{P}_2 \supseteq \dots$ as follows: Let $\mathcal{P}_0 = \mathcal{P}$. Suppose $\mathcal{P}_n$ defined and consider the class

$$\mathcal{B}_n = \{X \in 2^{\mathbb{N}} : \forall t\ \Phi_n(X, n)[t] \uparrow\} \cap \mathcal{P}_n.$$

Note that $\mathcal{B}_n$ is also a Π^0_1 class, and that the code of a computable tree T_n such that $\mathcal{B}_n = [T_n]$ is uniformly computable in n.

We ask $\emptyset'$ the question whether $\mathcal{B}_n$ is empty: according to König's lemma this is the case iff there exists m such that no string σ of size m belongs T_n, which is indeed a Σ^0_1 event. If $\emptyset'$ responds positively, we let $Y(n) = 1$ and we define $\mathcal{P}_{n+1} = \mathcal{P}_n$. Note that in this case all the elements $X \in \mathcal{P}_{n+1}$ are such that $\Phi_n(X, n) \downarrow$. In the opposite case we let $Y(n) = 0$ and we define $\mathcal{P}_{n+1} = \mathcal{B}_n$. Note that in this case all the elements of $X \in \mathcal{P}_{n+1}$ are such that $\Phi_n(X, n) \uparrow$.

For each n, $\mathcal{P}_n$ is a non-empty closed classe and therefore $\bigcap_n \mathcal{P}_n$ is non-empty. By construction the element Y computed by $\emptyset'$ corresponds to the Turing jump of any element of $\bigcap_n \mathcal{P}_n$ (which happens to be a singleton).■

We have already seen with Proposition 4-9.1 the existence of low and non-computable sets. We now have an alternative proof by combining Theorem 4.3 and Proposition 3.5.

We now tackle the second main basis theorem for Π^0_1 classes: computably dominated sets. For this, we need a lemma which also has its own interest.

Lemma 4.4. Let $\mathcal{P}$ be a non-empty Π^0_1 class. Suppose that a functional Φ is total on all the members of $\mathcal{P}$. Then, we can define uniformly in a code of $\mathcal{P}$ a computable function g which dominates $n \mapsto \Phi(X, n)$ for all $X \in \mathcal{P}$.⋆

Proof. Let $T \subseteq 2^{<\mathbb{N}}$ be a computable tree such that $[T] = \mathcal{P}$. Let us show that for any n, there exists t such that $\Phi(\sigma, n)[|\sigma|] \downarrow$ for any string $\sigma \in T$ of size t. Indeed,x in the opposite case, there exists n such that the set $\{\sigma \in T : \Phi(\sigma, n)[|\sigma|] \uparrow\}$ contains for all t a string of size t and is therefore an infinite subtree of T, which therefore contains by König's lemma an infinite path X. We thus have $\forall t\ \Phi(X, n)[t] \uparrow$ which contradicts the totality of Φ on all the oracles of $[T]$.

We can therefore compute the function g which for n searches for the smallest t such that $\Phi(\sigma, n)[t] \downarrow = v_\sigma$ for any string $\sigma \in T$ of size t. Once t is found we define $g(n) = \sum_{|\sigma|=t} v_\sigma + 1$. It is clear that g dominates all functions computable via Φ by an oracle of $[T]$. ■

The following theorem is known as "computably dominated basis theorem". With the existence of a non-empty Π^0_1 class having no computable member, this theorem gives us an alternative proof of the existence of computably dominated sets that are non-computable.

Theorem 4.5 (Jockusch et Soare [108])
Any non-empty Π^0_1 class contains a computably dominated set.

Proof. Let $\mathcal{P}$ be a non-empty Π^0_1 class. We will define an infinite decreasing sequence of non-empty Π^0_1 classes $\mathcal{P} = \mathcal{P}_0 \supseteq \mathcal{P}_1 \supseteq \mathcal{P}_2 \supseteq \dots$ such that $\bigcap_n \mathcal{P}_n$ contains only computably dominated sets. Let $\mathcal{P}_0 = \mathcal{P}$. Suppose $\mathcal{P}_n$ defined. Let $\mathcal{B}_{n,m} = \{X : \Phi_n(X, m) \uparrow\}$. Note that each class $\mathcal{B}_{n,m}$ is Π^0_1. Suppose that there exists m such that $\mathcal{P}_n \cap \mathcal{B}_{n,m} \neq \emptyset$. Then, we define $\mathcal{P}_{n+1} = \mathcal{P}_n \cap \mathcal{B}_{n,m}$. Note that for all $X \in \mathcal{P}_{n+1}$ we have $\Phi_n(X, m) \uparrow$. Suppose now that for all m we have $\mathcal{P}_n \cap \mathcal{B}_{n,m} = \emptyset$. This implies that the functional Φ_n is total for all $X \in \mathcal{P}_n$. We then define $\mathcal{P}_{n+1} = \mathcal{P}_n$. According to Lemma 4.4, there exists a computable function $g : \mathbb{N} \to \mathbb{N}$ which dominates $m \mapsto \Phi_n(X, m)$ for all $X \in \mathcal{P}_{n+1}$.

As a decreasing intersection of non-empty closed classes, the class $\bigcap_n \mathcal{P}_n$ is non-empty. Let $X \in \bigcap_n \mathcal{P}_n$. By construction, for all n, if Φ_n is total on the oracle X then $m \mapsto \Phi_n(X, m)$ is bounded by a computable function. So X is computably dominated. ■

We now see a last basis theorem called "cone avoidance": given a set X, we call *upper cone* of X the class $\mathcal{C}_X = \{Y \in 2^{\mathbb{N}} : Y \geqslant_T X\}$. Jockusch and Soare [108] proved that for each non-computable set X, the class $2^{\mathbb{N}} \setminus \mathcal{C}_X$ is a basis for the Π^0_1 classes. In other words, if X is a non-computable set, any non-empty Π^0_1 class has an element which does not compute X. The more natural contraposition states that if a set is computable by all the members of a non-empty Π^0_1 class, it is necessarily computable. Note that if a Π^0_1 class has a computable member the result is obvious, and it becomes interesting only for non-empty Π^0_1 classes which do not have any. As with the computably dominated basis theorem, we need a lemma to solve the case of a fixed functional.

Lemma 4.6. Let X be a set, $\mathcal{P}$ a non-empty Π^0_1 class and Φ a functional. If $\Phi^Y = X$ for all $Y \in \mathcal{P}$, then X is computable. ⋆

Proof. Let $T \subseteq 2^{<\mathbb{N}}$ be a computable tree such that $[T] = \mathcal{P}$. Suppose that for all $Y \in [T]$, $\Phi^Y = X$. Let us show that for any n, there exists a $t \in \mathbb{N}$ such that $\Phi(\sigma, n)[|\sigma|] \downarrow= X(n)$ for every string $\sigma \in T$ of size t. Indeed, otherwise, the set $S = \{\sigma \in T : \Phi(\sigma, n)[|\sigma|] \neq X(n)\}$ is a subtree of T which contains elements of each length, so by König's lemma, there exists a path $Y \in [S] \subseteq [T]$ such that $\Phi^Y(n) \neq X(n)$, contradicting our hypothesis.

Let $g : \mathbb{N} \to \mathbb{N}$ be the computable function which on input n looks for $t, v_n \in \mathbb{N}$ such that $\Phi(\sigma, n)[|\sigma|] \downarrow= v_n$ for every string $\sigma \in T$ of size t, and returns v_n. We have shown that this function is total. We also necessarily have $v_n = X(n)$ for all n because otherwise each element of $\mathcal{P}$ computes something other than X on the bit n. So X is computed by g and is therefore computable. ∎

Theorem 4.7 (Jockusch et Soare [108])
Let X be a non-computable set and $\mathcal{P}$ a non-empty Π_1^0 class. Then, there exists an element of $\mathcal{P}$ which does not compute X.

Proof. Let X be a non-computable set and $\mathcal{P}$ a non-empty Π_1^0 class. We will define an infinite decreasing sequence of non-empty Π_1^0 classes $\mathcal{P} = \mathcal{P}_0 \supseteq \mathcal{P}_1 \supseteq \mathcal{P}_2 \supseteq \dots$ so that no element of $\bigcap_n \mathcal{P}_n$ computes X. Let $\mathcal{P}_0 = \mathcal{P}$. Suppose $\mathcal{P}_n$ defined. Let $\mathcal{B}_{n,m} = \{Y : \Phi_n(Y, m) \uparrow \vee \Phi_n(Y, m) \neq X(m)\}$. Note that each class $\mathcal{B}_{n,m}$ is Π_1^0 (not uniformly of course because we do not know X). Let us show that there exists m such that $\mathcal{P}_n \cap \mathcal{B}_{n,m} \neq \emptyset$. If this was not the case, then we would have $\Phi_n^Y = X$ for all $Y \in \mathcal{P}_n$, contradicting Lemma 4.6. So there exists m such that $\mathcal{P}_n \cap \mathcal{B}_{n,m} \neq \emptyset$. We then define $\mathcal{P}_{n+1} = \mathcal{P}_n \cap \mathcal{B}_{n,m}$ for such an integer m which gives us $\Phi_n(X, m) \uparrow$ or $\Phi_n(X, m) \downarrow \neq X(m)$ for all $X \in \mathcal{P}_{n+1}$.

As a decreasing intersection of non-empty closed classes, the class $\bigcap_n \mathcal{P}_n$ is non-empty. Let $Y \in \bigcap_n \mathcal{P}_n$. By construction for all n, $\Phi_n^Y \neq X$ because $Y \in \mathcal{P}_n$. So $X \not\leq_T Y$. ∎

Hirschfeldt [91] gave an elegant alternative proof of the cone avoidance basis theorem, as a simple consequence of the low basis theorem (Theorem 4.3) and of the computably dominated basis theorem (Theorem 4.5).

Alternative proof of Theorem 4.7. Two cases arise:

- Case 1: X is Δ_2^0. In particular, by Proposition 7-4.7, X is hyperimmune. By the computably dominated basis theorem (Theorem 4.5), $\mathcal{P}$ contains a computably dominated set P. In particular, P does not compute X.
- Case 2: X is not Δ_2^0. By the low basis theorem (Theorem 4.3), $\mathcal{P}$ contains a low set, so Δ_2^0. In particular, P does not compute X. In each case, $\mathcal{P}$ contains an element which does not compute X. ∎

We will see in the chapters to come many other theorems concerning the Π_1^0 classes.

5. Basis for perfect Π_1^0 classes

We have seen that the non-empty Π_1^0 classes with no computable element are necessarily perfect. These classes admit reinforced basis theorems, and one can in particular construct perfect subclasses all of whose elements have a weakness property fixed in advance. Here, we see an example with computably dominated sets.

The idea is to start again the proof of the computably dominated basis theorem, but by duplicating the construction step by step.

Theorem 5.1
Let $\mathcal{P}$ be a non-empty Π^0_1 class containing no computable element. There exists a perfect class $\mathcal{B} \subseteq \mathcal{P}$ which contains only computably dominated sets.

Proof. Let $P_\epsilon = \mathcal{P}$. Suppose that for n and each $\sigma \in 2^{<\mathbb{N}}$ of size n, we have defined pairwise disjoint non-empty Π^0_1 classes $\mathcal{P}_\sigma \subseteq \mathcal{P}$. We repeat the construction of Theorem 4.5 to define for each σ a non-empty Π^0_1 class $\mathcal{Q}_\sigma \subseteq \mathcal{P}_\sigma$ such that either there is an m such that $\Phi_n(X, m) \uparrow$ for all $X \in \mathcal{Q}_\sigma$, or there is a computable function $g : \mathbb{N} \to \mathbb{N}$ such that $\Phi_n(X, m) < g(m)$ for all m and for all $X \in \mathcal{Q}_\sigma$. As $\mathcal{P}$ does not contain any computable point then for all σ, neither does $\mathcal{Q}_\sigma \subseteq \mathcal{P}$. So according to Corollary 3.7 there must be τ_0, τ_1 incomparable such that $\mathcal{Q}_\sigma \cap [\tau_0]$ and $\mathcal{Q}_\sigma \cap [\tau_1]$ are both non-empty. We define $\mathcal{P}_{\sigma 0} = \mathcal{Q}_\sigma \cap [\tau_0]$ and $\mathcal{P}_{\sigma 1} = \mathcal{Q}_\sigma \cap [\tau_1]$.

For each $X \in 2^{\mathbb{N}}$, the class $\bigcap_n \mathcal{P}_{X \upharpoonright_n} \subseteq \mathcal{P}$ contains exactly one element G_X, this element is computably dominated, and by construction $X \neq Y$ implies $G_X \neq G_Y$. The class of G_X for $X \in 2^{\mathbb{N}}$ in fact forms a perfect tree, whose nodes are determined by the choice of incomparable extensions τ_0, τ_1. ∎

The duplication technique of the previous theorem can also be applied to the cone avoidance basis theorem, but of course it cannot be used with the low basis theorem, because the class of low sets is countable. The reader can try to apply it anyway, in order to see what goes wrong.

Finally, note that it is of course not necessary to go through Π^0_1 classes to build a perfect class of computably dominated sets, and we can apply the same idea of construction duplication to the proof of f-trees:

Exercise 5.2. ($\star$) Construct a perfect class of computably dominated sets via f-trees. $\diamond$

Exercise 5.3. ($\star$) Let $\mathcal{P}$ be a non-empty Π^0_1 class with no computable point. Mix the above construction with the proof of Lemma 4.6 to build a perfect subclass of $\mathcal{P}$ whose elements are computably dominated, and whose Turing degrees are pairwise incomparable. $\diamond$

Exercise 5.4. ($\star\star$) Let $\mathcal{P}$ be a perfect class. Construct a perfect subclass of $\mathcal{P}$ whose elements are pairwise incomparable in terms of Turing degrees. $\diamond$

Exercise 5.5. ($\star$) Let $\mathcal{P}$ be a non-empty Π^0_1 class without isolated point. Show that $\emptyset'$ computes a non-computable element of $\mathcal{P}$. $\diamond$

Note that the last exercise necessarily uses the fact that $\mathcal{P}$ does not contain any isolated point. We will see with Proposition 30-3.5 a simple technique, but very powerful, allowing to build Π^0_1 classes — with isolated points — whose elements are either finite sets, or sets of "very high" computational complexity.

6. PA degrees

We take here a little advance on Chapter 9, in which we expose the notions of first-order logical theory, of the formal system of Peano arithmetic, as well as of the first incompleteness theorem of Gödel: the notion of PA degree was born in direct link with these notions. We will however quickly abstract from this historical aspect to give with Theorem 6.2 a characterization of PA degree involving only computability-theoretic notions already seen.

The study of PA degrees — acronym of "Peano Arithmetic" — goes back to the work of Gödel and his famous incompleteness theorem: there is no computable, complete and consistent extension of the axioms of Peano arithmetic[1]. The study of Turing degrees developing, the question of the power necessary to compute such an extension arose quite naturally. We are going to see that the developments around this question have lead to one of the richest concepts of computability theory, which probably found its climax through the study of reverse mathematics.

In order to speak about the computational power of a theory, we first need to cast the related notions to the setting of computability theory, and in particular to represent theories as sets of integers. In what follows, let us fix a computable enumeration $\psi_0, \psi_1, \psi_2, \ldots$ of all the formulas of arithmetic. Suppose also that there exists a computable function $\texttt{neg} : \mathbb{N} \to \mathbb{N}$ such that $\psi_{\texttt{neg}(n)} = \neg\psi_n$. For the following theorem (Theorem 6.2), we will call *theory* a set $T \subseteq \mathbb{N}$ such that for all m, if $\{\psi_n : n \in T\} \vdash \psi_m$, then $m \in T$. In other words, a theory is a set of arithmetic formulas closed under logical consequence. A theory T is *consistent* if the code of the formula "$0 = 1$" does not belong to T. A theory T is *complete* if for all n, either $n \in T$

[1] This version of the theorem is in fact a reinforcement of that of Gödel, which was proved by Rosser.

or $\texttt{neg}(n) \in T$. The reader who approaches these notions for the first time will find more details in Chapter 9.

Definition 6.1. A *completion of Peano arithmetic* is a complete theory T containing $\{n \in \mathbb{N} : PA \vdash \psi_n\}$. A Turing degree is *PA* if it contains a consistent completion of Peano arithmetic. $\diamondsuit$

The PA degrees being upward-closed, it is equivalent for a degree to be PA and to contain a set which computes a consistent completion of Peano arithmetic. We now show an equivalence which will serve as a characterization for the PA degrees.

Theorem 6.2 (Jockusch et Soare [100], Solovay (non publié)) *Let X be a set. The following statements are equivalent:*

(1) X is of degree PA.

(2) X is of DNC$_2$ *degree, i.e., the set X computes a function $f : \mathbb{N} \to \{0,1\}$ such that $f(n) \neq \Phi_n(n)$ for all n.*

Before going to the proof, we refer the reader to Definition 7-2.7 who introduced the notion of degree DNC$_f$ for a function $f : \mathbb{N} \to \mathbb{N}$ such that $2 \leqslant f(n) \leqslant f(n+1)$. The notion of degree DNC$_2$ is the strongest possible of this order: the computed function f has only two possibilities (0 or 1) to differ from each $\Phi_n(n)$. We will see with the corollaries 18-4.3 and 19-1.8 that many sets of DNC degree are not DNC$_2$.

Proof. The equivalence shown by Jockusch and Soare uses Scott's basis theorem [202] for PA degrees, which states that any PA degree computes an infinite path in any non-empty Π^0_1 class. Here we show the equivalence directly.

The implication (1) $\to$ (2) is essentially the Gödel-Rosser theorem, which extends Gödel's first incompleteness theorem, and which will be formally proved with Theorem 9-3.10 and Corollary 9-3.11.

Let us show (2) $\to$ (1). Let $f \leqslant_T X$ be a $\{0,1\}$-valued function such that $f(n) \neq \Phi_n(n)$ for all n. We are going to define a uniformly f-computable increasing sequence of consistent theories PA $= T_0 \subseteq T_1 \subseteq \dots$ such that $T = \bigcup_n T_n$ is complete. Let $T_0 =$ PA. Suppose T_n is consistent. We consider the arithmetic formula ψ_n of code n and we define the machine code e_n such that $\Phi_{e_n}(e_n) = 1$ si $T + \psi_n \vdash 0 = 1$ and $\Phi_{e_n}(e_n) = 0$ si $T + \neg\psi_n \vdash 0 = 1$. If $\Phi_{e_n}(e_n) \downarrow= 0$ then $T + \neg\psi_n$ is inconsistent and therefore $T + \psi_n$ is consistent. If $\Phi_{e_n}(e_n) \downarrow= 1$ then $T + \psi_n$ is inconsistent and therefore $T + \neg\psi_n$ is consistent. If $\Phi_{e_n}(e_n) \uparrow$ then $T + \psi_n$ and $T + \neg\psi_n$ are both consistent. Now as $f(e_n) \neq \Phi_{e_n}(e_n)$, we can define $T_{n+1} = T_n + \psi_n$

if $f(e_n) = 1$ and $T_{n+1} = T_n + \neg\psi_n$ if $f(e_n) = 0$. In all cases we will have a consistent theory.

The theory $T = \bigcup_n T_n$ is therefore consistent and by construction it is also complete. ■

Remark

Note that the direction (2) → (1) of Theorem 6.2 works for any consistent theory T_0 whose axioms are computable. Thus, any DNC_2 function is able to compute a completion of any consistent theory whose axioms are computable. Direction (1) → (2) is more specific to Peano arithmetic, as it requires a sufficiently expressive theory to encode computations by formulas.

Note that $\emptyset'$ can compute a DNC_2 function and is therefore of PA degree. The following proposition implies that it is not at all necessary to be Turing complete to compute a complete and consistent extension of Peano arithmetic.

Definition 6.3. The *degree spectrum* of a class $\mathcal{P} \subseteq 2^{\mathbb{N}}$ is the set

$$\deg \mathcal{P} = \{\deg_T X : X \in [P]\}$$ ♢

Proposition 6.4. There is a Π^0_1 class whose degree spectrum corresponds to the PA degrees. ⋆

Proof. This is a simple observation, which was already used for the proof of Proposition 3.5. The class of DNC_2 sets is described as follows.

$$\mathcal{P} = \{X \in 2^{\mathbb{N}} : \forall e\ \forall t\ \Phi_e(e)[t] \uparrow \vee\ \Phi_e(e)[t] \downarrow \neq X(e)\}$$ ■

Corollary 6.5
There are low PA degrees.

Proof. According to Proposition 6.4 and Theorem 4.3. ■

Corollary 6.6
There are computably dominated PA degrees.

Proof. According to Proposition 6.4 and Theorem 4.5. ■

Corollary 6.7
Let A be a non-computable set. Then, there exists a PA degree which does not compute A.

Proof. According to Proposition 6.4 and Theorem 4.7. ■

Let us now see another important characterization of the PA degrees, which states that they capture the necessary and sufficient computational power for weak König's lemma.

Theorem 6.8
Let $X \subseteq \mathbb{N}$. The following statements are equivalent:

(1) X is of PA degree.

(2) X computes a set in each non-empty Π^0_1 class.

Moreover for (2) the computation is uniform in a code of the Π^0_1 class.

Proof. For (2) $\rightarrow$ (1) it suffices to notice that there exists a non-empty Π^0_1 class containing only sets of PA degrees (see Proposition 6.4). Let us now show (1) $\rightarrow$ (2). Let $f \leqslant_T X$ be a $\{0,1\}$-valued DNC function, that is, such that $f(n) \neq \Phi_n(n)$ for all n. Let $\mathcal{P}$ be a non-empty Π^0_1 class and T a computable tree such that $[T] = \mathcal{P}$.

Let $\sigma_0 = \epsilon$. Given σ_n defined such that $[\sigma_n] \cap [T]$ is not empty, we compute $\sigma_{n+1} = \sigma_n i$ for $i \in \{0,1\}$ as follows: we first compute the code e_n of a program which on any input m searches for the smallest t such that for $i = 0$ or $i = 1$ no string σ of size t with $\sigma \succeq \sigma_n i$ does not belong to T. If found, the program halts and outputs i. According to König's lemma this condition is equivalent to the fact that $[\sigma_n i] \cap [T]$ is empty. We simply define $\sigma_{n+1} = \sigma_n f(n)$. As $f(e_n) \neq \Phi_{e_n}(e_n)$ we have the guarantee that $[\sigma_{n+1}] \cap [T]$ is non-empty. ■

Classe universelle

Note that according to Proposition 6.4, there exists a non-empty Π^0_1 class whose members are of PA degree, and that according to Theorem 6.8, any PA degree computes a member of each non-empty Π^0_1 class. Such a class is therefore "maximal" in terms of computational complexity, in the sense that if we know how to compute a member of this class, then we know how to compute a member of any non-empty Π^0_1 class. We call *universal* Π^0_1 *class* a non-empty Π^0_1 class all of whose members are of PA degree.

In the same vein as Theorem 7-7.1, we end with a characterization which now combines the fact of being of high or PA degree. Note the difference with (1) $\leftrightarrow$ (3) of Theorem 7-6.2 within which we consider a sequence $(X_n)_{n\in\mathbb{N}}$ containing exactly the computable sets, whereas here we only consider that it contains the computable sets.

Theorem 6.9 (Jockusch [102])
Let $X \subseteq \mathbb{N}$. The following statements are equivalent:

(1) X is of high or PA degree.

(2) X computes a sequence $(X_n)_{n\in\mathbb{N}}$ containing all the computable sets.

Proof. Let us first show (1) implies (2). If X is high then the implication is clear from Theorem 7-6.2. Suppose now that X is of PA degree. Let $g \leqslant_T X$ be such that $g(n) \neq \Phi_n(n)$ for all n. Note that X also computes the function $f(x) = 1 - g(x)$. In particular, $\Phi_n(n)\downarrow \in \{0,1\}$ implies $f(n) = \Phi_n(n)$. Given a computable function Φ_e and an integer n, we can compute the code a_n such that $\Phi_{a_n}(a_n) = \Phi_e(n)$. By using this process and the fact that $\Phi_{a_n}(a_n)\downarrow \in \{0,1\}$ implies $f(a_n) = \Phi_{a_n}(a_n) = \Phi_e(n)$ we easily compute a set X_e such that if $n \mapsto \Phi_e(n)$ is total and has value in $\{0,1\}$ then $X_e(n) = \Phi_e(n)$ for all n. We can therefore compute our sequence $(X_e)_{e\in\mathbb{N}}$ containing all the computable sets.

Let us now show (2) implies (1). Let $(X_n)_{n\in\mathbb{N}}$ be an X-computable sequence containing all the computable sets. The idea is to proceed initially as in the proof of (3) $\rightarrow$ (1) of Theorem 7-6.2. Given a Π^0_2 predicate of the form

$$P = \{e : \forall x_1\ \exists x_2\ R(e, x_1, x_2)\},$$

the idea was to define uniformly in e a partial computable function f_e such that:

(a) $e \in P$ implies that f_e is a total computable function.

(b) $e \notin P$ implies that f_e is a partial function which has no computable completion.

It suffices to notice that in Theorem 7-6.2, the definition of f_e which is given is such that in case (b), not only no completion of f_e is computable, but in addition such a completion is necessarily of PA degree. The definition was as follows: Let e fixed. At the stage of computation t, for any value n smaller than t and such that f_e does not halt for the moment on n, we proceed as follows : if $\Phi_n(n)[t]\downarrow\neq 0$ we define $f_e(n) = 0$. If $\Phi_n(n)[t]\downarrow\neq 1$ we define $f_e(n) = 1$. Otherwise, if for all $k \leqslant n$ there exists $m_k \leqslant t$ such that $R(e, k, m_k)$ then we define $f_e(n) = 0$.

As in the proof of Theorem 7-6.2, if $e \in P$ then f_e is a total function. Otherwise we notice that for almost all the values of n such that $\Phi_n(n) \downarrow$ we have $f_e(n) \neq \Phi_n(n)$. Any completion of f_e is therefore a DNC_2 function, modulo a finite number of values, and is therefore of PA degree.

There are now two possibilities: either $(X_n)_{n\in\mathbb{N}}$ contains a set of PA degree, in which case we have (1). Either this is not the case, in which case we can give a $\Sigma^0_2(X)$ definition of P as in the proof of Theorem 7-6.2, which implies, applied to $P = \mathbb{N} \setminus \emptyset''$ that $\emptyset''$ is $\Delta^0_2(X)$ and therefore X is high. ■

We end this section with an exercise which constitutes an alternative and well-known characterization of the PA degrees.

Exercise 6.10. **($\star$)** Show that X is PA iff for all c.e. sets A, B with $A \cap B = \emptyset$, there exists an X-computable set C such that $A \subseteq C$ and $C \cap B = \emptyset$. ◇

7. Finitely-branching trees

We introduce here the *Baire space*: the class $\mathbb{N}^\mathbb{N}$ of all the infinite valued sequences in $\mathbb{N}$, or in other words the class of all the functions from $\mathbb{N}$ to $\mathbb{N}$. Just as Cantor space has its set of strings $2^{<\mathbb{N}}$, Baire space has its set of strings $\mathbb{N}^{<\mathbb{N}}$: finite sequences with values in $\mathbb{N}$. The different operations that we have seen on binary strings (prefix, concatenation, length, ...) extend without problem to strings in Baire space. In particular, given a string $\sigma \in \mathbb{N}^{<\mathbb{N}}$, we denote by $[\sigma]$ the class of sequences $P \in \mathbb{N}^\mathbb{N}$ such that $\sigma \prec P$. The notion of tree also extends to subsets of $\mathbb{N}^{<\mathbb{N}}$ as follows:

Definition 7.1. A set $T \subseteq \mathbb{N}^{<\mathbb{N}}$ is a *tree* if T is closed under prefix, that is to say for all $\sigma \in T$ and $\tau \preceq \sigma$, then $\tau \in T$. ◇

Unlike binary trees, nodes can have an infinite number of successors. A node $\sigma \in T$ is *branching* if it has at least two successors. A *path* of T is a sequence $P \in \mathbb{N}^\mathbb{N}$ whose initial segments are all in T. We denote by $[T]$ the class of paths of T. König's lemma no longer works on trees in Baire space, as the following counterexample shows.

Example 7.2. Let $T = \{\sigma \in \mathbb{N}^{<\mathbb{N}} : \forall n < |\sigma|\ \sigma(n) \geqslant |\sigma|\}$. The tree T contains nodes of arbitrary length and is infinite, but $[T] = \emptyset$.

The computational power of the paths of an arbitrary computable tree of Baire space will be studied in Part IV on higher computability theory. In this section, we will restrict ourselves to a sub-category of trees falling into the realm of König's lemma: the trees $T \subseteq \mathbb{N}^{<\mathbb{N}}$ which are *finitely-branching*, i.e., within which each node has a finite number of successors.

Lemma 7.3 (König's lemma). Let $T \subseteq \mathbb{N}^{<\mathbb{N}}$ be a finitely-branching tree such that $|T| = \infty$. Then, $[T]$ is non-empty. ⋆

Proof. We construct a path X by induction on n. As T is infinite, but the root ϵ has only a finite number of successors, by the pigeonhole principle, there exists $i \in \mathbb{N}$ and an infinity of nodes $\sigma \in T$ which extend i (ie with $i \prec \sigma$). We define $X(0) = i$. Suppose that $\tau = X(0)X(1)\dots X(n)$ is defined with $\tau \in T$ and such that there is an infinity of nodes $\sigma \in T$ for which $\tau \preceq \sigma$. The node σ having only a finite number of successors, by the pigeonhole principle, there exists $i \in \mathbb{N}$ and an infinity of nodes $\sigma \in T$ which extend τi. We define $X(n+1) = i$.

By induction on n, we thus define in this way a set X such that $X\restriction_n \in T$ for all n. ■

Baire space

As for Cantor space, the open classes of Baire space are the classes $\mathcal{O} \subseteq \mathbb{N}^{\mathbb{N}}$ of the form $\mathcal{O} = \bigcup_{\sigma\in W}[\sigma]$ for a set $W \subseteq \mathbb{N}^{<\mathbb{N}}$, and the closed class $\mathcal{P} \subseteq \mathbb{N}^{\mathbb{N}}$ are of the form $[T]$ for a tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$. On the other hand, unlike Cantor space, the closed classes of Baire space are not compact in general. The compacts of Baire space are precisely the closed classes $\mathcal{P}$ of the form $[T]$ for a finitely-branching tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$.

The proof of König's lemma is almost the same as that of its weak version, and one might expect at first glance that the computational power necessary to compute a path from a computable finitely-branching tree is that of PA degrees. This is not the case, however, as shown by Proposition 7.4.

Proposition 7.4. Let $T \subseteq 2^{<\mathbb{N}}$ be a Δ^0_2 binary tree. There exists a finitely-branching computable tree S such that $\deg([T]) = \deg([S])$. ⋆

Proof. Let $(T_n)_{n\in\mathbb{N}}$ be a Δ^0_2 approximation of T. We can assume without loss of generality that for any n, T_n is closed under prefix and $T_n \subseteq 2^{\leqslant n}$ (the set of strings of size less than or equal to n). We easily show that any union of trees is a tree, which implies that $\bigcup_n T_n$ is a tree.

Let us show that $[\bigcup_n T_n] = [T]$. Clearly, $T \subseteq \bigcup_n T_n$, so $[T] \subseteq [\bigcup_n T_n]$. Let $P \in [\bigcup_n T_n]$. Let $s \in \mathbb{N}$ and show that $P\restriction_s \in T$. $(T_n)_{n\in\mathbb{N}}$ being a Δ^0_2 approximation of T, $P\restriction_s \in T$ iff $\forall t\ \exists n \geqslant t\ P\restriction_s \in T_n$. Let $t \geqslant s$. As $P\restriction_t \in \bigcup_n T_n$ and as by hypothesis $\bigcup_{n<t} T_n \subseteq 2^{<t}$, then $P\restriction_t \in T_n$ for $n \geqslant t$. By downward-closure of T_n we have $P\restriction_s \in T_n$. So $\forall t\ \exists n \geqslant t\ P\restriction_s \in T_n$. So $P\restriction_s \in T$.

Let $\sigma, \tau \in \mathbb{N}^{<\mathbb{N}}$ have the same length. We denote by $\langle \sigma, \tau \rangle$ the string ρ of length $|\sigma|$ such that for all $n < |\rho|$, $\rho(n) = \langle \sigma(n), \tau(n) \rangle$. The operation naturally extends to infinite sequences P, Q for which we will write $\langle P, Q \rangle$. We are going to build a computable finitely-branching tree $S \subseteq \mathbb{N}^{<\mathbb{N}}$ whose paths will be of the form $\langle P, Q \rangle$ with $P \in [\bigcup_n T_n] = [T]$ and Q a "witness" of $P \in [\bigcup_n T_n]$, in the sense where for all s, $P \upharpoonright_s \in T_{Q(s)}$.

Define a partial computable function $f : \bigcup_n T_n \to \mathbb{N}^{<\mathbb{N}}$ which sends strings to strings of the same length inductively as follows: $f(\epsilon) = \epsilon$. If $\sigma i \in \bigcup_n T_n$ then $f(\sigma i) = f(\sigma)^\frown s$ where s is the smallest integer such that $\sigma i \in T_s$. By continuity, the function f extends to infinite sequences of $[\bigcup_n T_n] = [T]$. Let $S = \{\langle \sigma, f(\sigma) \rangle : \sigma \in \bigcup_n T_n\}$. Note that $\bigcup_n T_n$ is not computable in general, but that S is because for any $\rho = \langle \sigma, \mu \rangle$, it is easy to verify that $f(\sigma) = \mu$. The set S is closed under prefix, because $\bigcup_n T_n$ is also closed and $f(\sigma i) \upharpoonright_{|\sigma|} = f(\sigma)$ for all $\sigma \in \bigcup_n T_n$. Thus, S is a computable tree. Note also that S is 2-branching, therefore finitely-branching.

Let us show that $\deg([T]) = \deg([S])$. Let $P \in [T]$. Then, $\langle P, f(P) \rangle \in [S]$ and $P \equiv_T \langle P, f(P) \rangle$. Let $R \in [S]$. Then, $R = P \oplus f(P)$ for a $P \in [\bigcup_n T_n] = [T]$. Likewise, $R \equiv_T P$. This concludes the proof of Proposition 7.4. ■

Corollary 7.5
There exists a finitely-branching computable tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$ and a PA degree P which does not compute a path through T.

Proof. Let $S = \{\emptyset' \upharpoonright_n : n \in \mathbb{N}\}$ be the Δ^0_2 binary tree having $\emptyset'$ for unique infinite path. By Proposition 7.4, there exists a computable finitely-branching tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$ such that $\deg([T]) = \deg([S])$. In particular, any path of T computes $\emptyset'$. By Corollary 6.5, there is a degree both PA and low. In particular, this degree does not compute a path through T. ■

We can relativize the notion of being DNC$_2$ relative to an oracle X: we ask for the computation of a function $f : \mathbb{N} \to \{0, 1\}$ such that $f(n) \neq \Phi_n(X, n)$ for all n. Theorem 6.8 is relativized well in the sense that the DNC$_2$ degrees relative to X, which one will also call PA degrees relative to X or PA (X), coincide with those allowing to compute a path in any non-empty $\Pi^0_1(X)$ class.

Exercise 7.6. Let Y be a PA(X) set. Show that $Y \geqslant_T X$. ◇

We deduce that a PA degree relative to $\emptyset'$ is necessary to compute a path in any infinite computable finitely-branching tree. Note that the situation of Exercise 7.6 is different when we consider DNC degrees instead of DNC$_2$

degrees. More precisely, if X is a non-computable set, there exists a set Y of DNC degree relative to X which does not compute X (see Corollary 18 -4.4). This result brings into play notions of algorithmic randomness that we will discuss in chapter Chapter 18.

Binary tree vs 2-branching tree

The tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$ built in the proof of Proposition 7.4 is 2-branching, in the sense that each node has at most two successors. From a purely structural point of view, it is therefore isomorphic to a binary tree $S \subseteq 2^{<\mathbb{N}}$. However, there are PA degrees that do not compute a path in this tree. The difference between the computational power of this tree and that of a binary tree does not therefore come from a combinatorial difference, but simply stems from a lack of information on the successors of a node: given a computable finitely-branching tree, one cannot limit in a computable and uniform way the maximum value of the successor of a node.

The preceding remark leads us to the following definition.

Definition 7.7. A tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$ is *computably bounded* if there exists a computable function $g : \mathbb{N} \to \mathbb{N}$ such that for all $\sigma \in T$ and $n < |\sigma|$, $\sigma(n) < f(n)$. $\diamondsuit$

It is clear that any computably bounded tree is finitely-branching. The following proposition makes it possible to reconcile the idea according to which combinatorially similar objects should have the same computational power, by showing that as soon as the finitely-branching tree is accompanied by a computable bound on its branching, then the computational power necessary for compute a path is exactly that of PA degrees. Given a function $f : \mathbb{N} \to \mathbb{N}$, we will denote by $f^{<\mathbb{N}}$ the set of strings $\sigma \in \mathbb{N}^{<\mathbb{N}}$ such that for all $n < |\sigma|$, $\sigma(n) < f(n)$.

Proposition 7.8. For any computable, computably bounded tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$, there exists a binary tree $S \subseteq 2^{<\mathbb{N}}$ such that $\deg([T]) = \deg([S])$. $\star$

Proof. The idea of the proof is quite simply to define a binary encoding of the strings, using the computational bound to know how many bits to allocate at each level. To remove any ambiguity, we will denote by $2^{=n}$ the set of binary strings of length n, instead of 2^n, which will denote the n-th power of 2. Let $g : \mathbb{N} \to \mathbb{N}$ be a computable function such that $T \subseteq g^{<\mathbb{N}}$. Without loss of generality, we can assume that $g(n) = 2^{h(n)}$ for all n, with $h : \mathbb{N} \to \mathbb{N}$ a computable function.

For all n, let $e_n : 2^n \to 2^{=n}$ be the canonical bijection. For example, $e_2 : 4 \to \{00, 01, 10, 11\}$ is defined by $e_2(0) = 00, e_2(1) = 01, e_2(2) = 10$

and $e_3(3) = 11$. This coding extends into a computable bijection $e : g^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ defined by

$$e(\sigma) = e_{h(0)}(\sigma(0))^\frown e_{h(1)}(\sigma(1))^\frown \dots ^\frown e_{h(|\sigma|-1)}(\sigma(|\sigma|-1)),$$

where one denotes here for more clarity the concatenation by $^\frown$. For example, if $h(n) = n+1$, then $g(n) = 2^{h(n)} = 2^{n+1}$, and $e(032) = e_1(0)^\frown e_2(3)^\frown e_3(2) = 0^\frown 11^\frown 010 = 011010$.

Note that the set $\hat{S} = \{e(\sigma) : \sigma \in T\}$ is not a tree, because it is closed under prefix only for the initial segments of length exactly $\operatorname{Im} g$. We must therefore define the tree S as the prefix closure of $\hat{S}$, in other words $S = \{e(\sigma) \upharpoonright_n : \sigma \in T \wedge n \in \mathbb{N}\}$. The coding function e being monotonic on the lengths, and the set T being closed under prefix, $\rho \in S$ if and only if there exists a string $\sigma \in g^{<\mathbb{N}}$ of length at most $|\rho|$ such that $\rho \prec e(\sigma)$. Thus, S is an infinite computable binary tree, whose paths are exactly infinite sequences of the form $e_{h(0)}(P(0))^\frown e_{h(1)}(P(1))^\frown \dots$ for $P \in [T]$. Thus, $\deg([T]) = \deg([S])$. ■

Corollary 7.9
Let $T \subseteq \mathbb{N}^{<\mathbb{N}}$ be a computable, computably bounded infinite tree. Any PA degree computes a path of T.

Proof. By Proposition 7.8, there exists a binary tree $S \subseteq 2^{<\mathbb{N}}$ such that $\deg([T]) = \deg([S])$. By Theorem 6.8, any PA degree computes a path of S, so any PA degree computes a path of T. ■

We now see the converse of Proposition 7.4, which shows that the PA degrees relative to $\emptyset'$ are exactly those able to compute a path in a computable finitely-branching tree.

Proposition 7.10. Let $T \subseteq \mathbb{N}^{<\mathbb{N}}$ be an infinite computable finitely-branching tree. There is a Δ^0_2 binary tree $S \subseteq 2^{<\mathbb{N}}$ such that $\deg([T]) = \deg([S])$. ⋆

Proof. Note first that any infinite computable finitely-branching tree $T \subseteq \mathbb{N}^{<\mathbb{N}}$ is $\emptyset'$-computably bounded, that is to say that there exists a $\emptyset'$-computable function $g : \mathbb{N} \to \mathbb{N}$ such that $T \subseteq g^{<\mathbb{N}}$. The proof of Proposition 7.8 is relativized to $\emptyset'$, and allows to define a Δ^0_2 binary tree $S \subseteq 2^{<\mathbb{N}}$ such that $\deg([T]) = \deg([S])$. ■

Let's finish this section with a few exercises. The *prefix closure* of a set $S \subseteq \mathbb{N}^{<\mathbb{N}}$ is the set

$$\hat{S} = \{\tau \in \mathbb{N}^{<\mathbb{N}} : \exists \sigma \in S\ \tau \preceq \sigma\}.$$

Exercise 7.11. Show that for any infinite set $S \subseteq 2^{<\mathbb{N}}$, its prefix closure admits a path. $\diamond$

Exercise 7.12. Show that there exists a computable infinite set $S \subseteq 2^{<\mathbb{N}}$ such that $[\hat{S}] = \{\emptyset'\}$. $\diamond$

Exercise 7.13. ($\star\star$) Show that for any infinite $\emptyset'$-computable tree $T \subseteq 2^{<\mathbb{N}}$, there exists an infinite set $S \subseteq 2^{<\mathbb{N}}$ containing exactly one string of each length, such that $[T] = [\hat{S}]$. $\diamond$

Summary Diagram

Here is a figure which summarizes the various concepts approached so far.

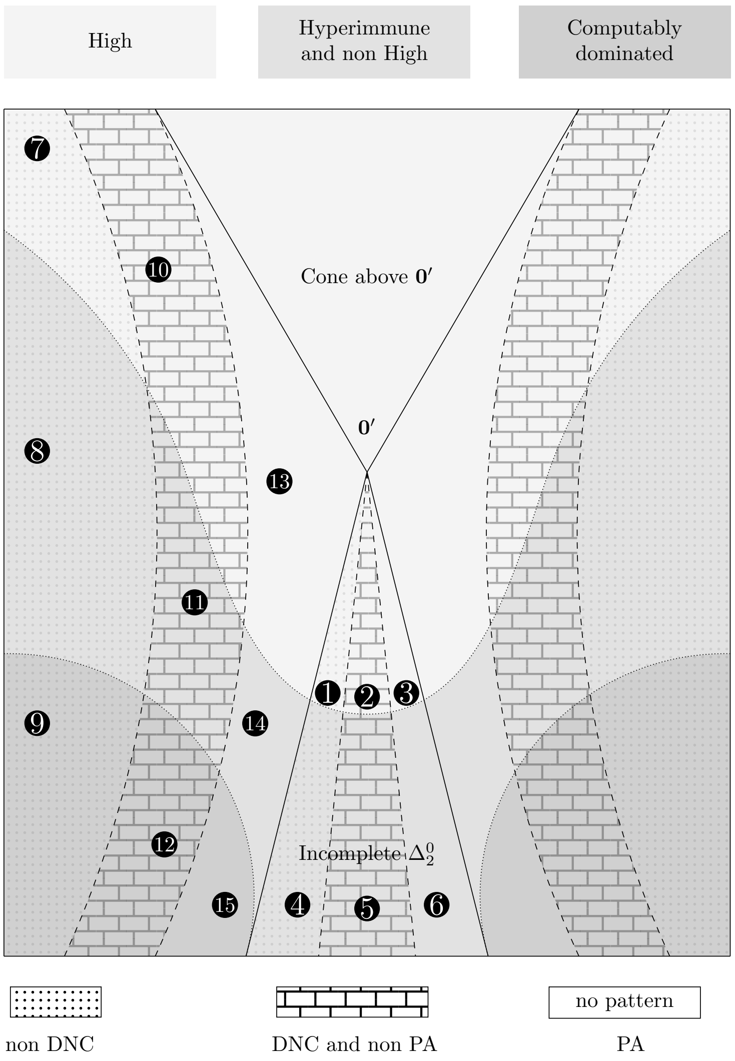


Figure 7.14: Summary on the Turing degrees seen so far. Points (1) to (6) refer to examples of sets of each type. Points (7) to (15) refer to the existence of a perfect class of sets of each type (hence according to Exercise 5.4) to the existence of a perfect class of Turing degrees of each type.)

The points from (1) to (15) which follow use for some of the concepts which will be seen only in the following chapters.

(1) There exists a Δ^0_2 Turing degree not DNC and high: It suffices to mix the proof of Exercise 4-10.6 with that of Exercise 7-2.9 to obtain a high set which in addition to be incomplete, is not DNC.

(2) There exists a Δ^0_2 Turing degree which is DNC, non PA and high: We start from a DNC, non PA and low set X (by Theorem 18-4.1 and Corollary 18-2.3 on can take for example a low random in the sense of Martin-Löf). We can then embroider on the construction of Exercise 4 -10.6 and on that of Exercise 7-2.9 to build a set Y Δ^0_2 high such that $X \oplus Y$ is not PA (using in particular the fact that $X' \leqslant_T \emptyset'$).

(3) There exists an incomplete Δ^0_2 Turing degree which is PA and high: We start from a PA and low set X (see Corollary 6.5). We then embroider on the construction of Exercise 4-10.6 to build a Δ^0_2 high set Y such that $X \oplus Y$ is incomplete.

(4) There exists a Δ^0_2 Turing degree not DNC, hyperimmune and not high: It suffices to build a non DNC and low set, by mixing Proposition 4 -9.1 and Exercise 7-2.9.

(5) There exists a Δ^0_2 Turing degree which is DNC, non PA, hyperimmune and not high: It suffices to consider a random set in the sense of Martin-Löf and low. According to Theorem 18-4.1 such a set is DNC. According to Theorem 19-1.7 it is not PA. According to Proposition 7 -4.7 it is hyperimmune. Finally, since it is low, it cannot be high.

(6) There exists a Δ^0_2 PA, hyperimmune and not high Turing degree: According to Proposition 7-4.7 it suffices to consider a low PA degree given by Corollary 6.5.

(7) There is a perfect class of high and non-DNC sets. Just use Theorem 10 -3.21 and embroider on Posner/Robinson (see Corollary 10-3.34) to build a perfect class of 1-generic and high sets.

(8) There is a perfect class of hyperimmune, non-high and non-DNC sets. According to Theorem 10-3.2, Proposition 10-3.38 and Corollary 10 -3.34 any sufficiently generic set will be in this case there.

(9) There exists a perfect class of computably dominated and non-DNC sets. We have to take again the construction of computably dominated sets via f-trees, and modify it to produce non-DNC sets as it is done in Exercise 7-2.9.

(10) There is a perfect class of high, DNC and non PA sets. We can apply Theorem 18-3.4 of Kučera/Gács relativized to $\emptyset'$ on the 2-random tree to build high and 2-random sets. By Theorem 18-4.1 such sets are DNC. By Theorem 19-1.7 they are not PA.

(11) There is a perfect class of hyperimmune, non-high, DNC and non-PA sets. According to Theorem 18-4.1, Corollary 19-3.9 and Corollary 19 -1.8 this is the case for any sufficiently random set.

(12) There exists a perfect class of computably dominated, DNC and not PA sets. This is Theorem 5.1 applied to a Π^0_1 class containing only random numbers within the meaning of Martin-Löf. According to Theorem 19 -1.7 MLR and computably dominated sets cannot be PA.

(13) There exists a perfect class of incomplete sets, high and PA. We fix a high and incomplete set X. We then develop on Theorem 4.7 the cone avoidance basis relativized to X, to build a perfect class of sets PA Y such that $X \oplus Y$ is incomplete.

(14) There is a perfect class of hyperimmune, non-high and PA sets. Let X be a non-high hyperimmune set. We use the relativization to X of Theorem 5.1, applied to the Π^0_1 class of DNC_2 sets, to construct a perfect class of sets Y such that any function computed by $X \oplus Y$ is dominated by a function computed by X.

(15) There exists a perfect class of computably dominated PA sets. This is Theorem 5.1 applied to the Π^0_1 class of DNC_2 sets.

Chapter 9

Formal interlude

1. A little history: the crisis of foundations

Mathematics have developed naturally over the centuries as a tool at the service of an abstract representation of reality. This state of affairs is, for example, flagrant in physical sciences for which mathematics accurately accounts for a variety of phenomena. Over time, the concepts studied have become more and more complex, more and more abstract, and the connections between mathematics and the real world have become more and more uncertain. For a long time, however, the discipline has been able to rely on the innate logical sense of the human mind to talk about things that we do not "see anymore" while keeping a rigorous framework. Complex numbers constitute a striking example: the negative square numbers, which only exist a priori in the imagination of the mathematician, are baptized in 1545 by Cardan "sophisticated quantities". Sophistication was undoubtedly needed to accept this UFO as a serious object of study. However these "sophisticated quantities" find their utility in the resolution of very concrete problems. Raphaël Bombelli gives a first formalization in 1572, and shows how to use these numbers to solve certain third degree equations. Over the centuries, they will find many uses in mathematics, as well as in physics, where they are used successfully in equations representing the real world.

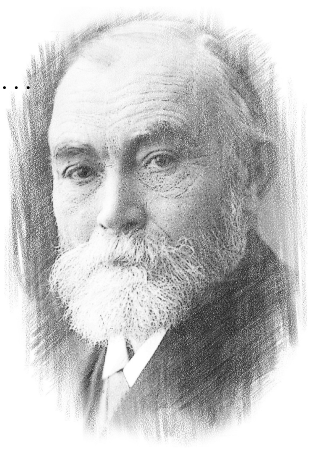
It took a certain conceptual leap to accept the development of a rigorous and consistent mathematical framework around complex numbers. Despite everything, let's say that this concept, however surprising it may be, still remains "relatively simple". The real problems arise with Cantor's work on cardinality and transfinite numbers. Cantor opens the door wide to a bottomless abyss, that takes us far beyond what the human mind can confidently apprehend: the study of the infinite. Of course, infinity has been present in mathematics since antiquity in the first place, through the consideration that there is no greater whole number. But Cantor's epistemological revolution consists in considering the infinite as an object of study in its own right. This consideration will lead to the beginnings of what will become a century later set theory with a capital "S".

With Cantor's work, the question of knowing *what is* mathematical activity has become more and more pressing: can we really reason about everything, and even about the infinite, a concept beyond us? But if we accept, as is the case today, that we can reason about infinity, we certainly cannot do it just any old way. Basically what is *doing* mathematics? Especially when we start to manipulate objects about which we no longer have so much intuition, how can we be sure that what we are talking about really has a meaning? These considerations found their apogee during the famous "crisis of foundations" which goes from the end of the 19th century to the beginning of the 20th. It is then a question of defining rules to frame the mathematical activity. It is in fact a question of precisely defining the mathematical study, not of objects such as integers, reals or even functions, but of defining the mathematical study *of mathematics itself.* It is remarkable a posteriori to note the success of this enterprise: mathematics is a sufficiently powerful tool to be able to define and study itself, with the rigor inherent in the discipline! It was obviously not an easy path. In this enterprise, the three musketeers — who as we know are four — are called Frege, Russel, Zermelo and Hilbert.

Frege

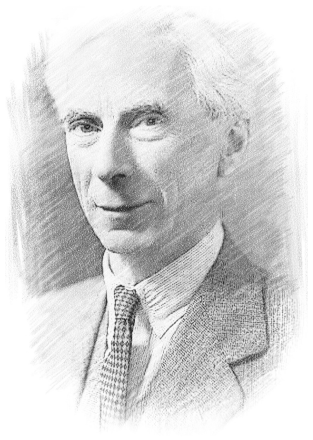
German philosopher and mathematician from the end of the 19th century, Gottlob Frege was moved by one certainty: logic precedes mathematics. But the logic of the time is still very poor, and is essentially confined to the work of George Boole, on what is called today *propositional calculus*: the manipulation of propositions, true or false, that one can connect between them via "and" and "or" logic, well known to computer scientists. This system is too small for Frege's ambition to put mathematics on a logical foundation. In particular, nothing in propositional calculus allows us to speak of specific objects through the relations they maintain with one another. He then formalized in his work "Begriffsschrift" a new language in order to overcome the shortcomings of the logic of the time, a language which will evolve to become what we call today *predicate calculus*, ubiquitous in mathematics.

Frege is considered today as the father of modern logic, notably through his concept of quantified variables $\forall x \ldots, \exists x \ldots$ He uses his formalism to tackle in his following works "Foundations of arithmetic" (1884) and "Fundamental laws of arithmetic I and II" (1893 and 1903) the foundation of arithmetic on logic. For this, he proposed a definition of natural integers which can be seen today as based on the concept of *set* and that of *comprehension scheme*: if $\Psi(x)$ is a mathematical formula which can be true or false according to x, then the set of elements which satisfy this formula: $\{x : \Psi(x)\}$, is well defined.


Gottlob Frege, 1848–1925

Russell

In 1902, Russell in a letter to Frege expressed doubts about his work. Let y be the set of sets which do not belong to each other: $y = \{x : x \notin x\}$. Do we then have $y \in y$ or $y \notin y$? We can easily understand the paradoxical situation we have reached. This example will be famous as *Russell's paradox.* At 53, Frege realizes that Russell's paradox implies the collapse of the system he took years to build. A hard blow from which he will have great difficulty recovering.


Bertrand Russell, 1872–1970

Despite this paradox, Russell welcomes Frege's work with great enthusiasm, and goes a long way in promoting its value. Like Frege, Russell feels the need to put mathematics on a solid foundation. Much like Frege, Russell has this intuition that logic precedes mathematics. He will tackle ten years during, with his former professor Alfred Whitehead, this search for logical foundations, which will lead to their famous work "Principia Mathematica": a titanic work which extends over more than 2,000 pages, whose ambition is to describe a set of logical axioms and inference rules from which any mathematical truth could be demonstrated.

These works lay the foundations of what we call today *type theory*, a system still studied today, presenting very strong links with programming languages. In parallel, the set theory developed, the axiomatization of which was initiated by Zermelo, and completed later by Fraenkel and Skolem independently, to give the axiomatic system ZF, named after Zermelo-Fraenkel.

Zermelo

Ernst Zermelo, 1871–1953

An absolutely remarkable fact is that the ZF system, to which it is sometimes necessary to add the axiom of choice, gives a framework within which can be formalized *the totality of modern mathematics*, if we exclude recent developments in set theory, the objective of which is precisely to get out of this system. Zermelo finds a way to avoid Russell's famous paradox — like Russell himself with his theory of types — by limiting the axiom of comprehension. The set $\{x : \Psi(x)\}$ is no longer valid, one needs to start from an existing set y, in which case we can now define the set of elements of y which satisfy Ψ: $\{x \in y : \Psi(x)\}$. However, this theory does not immediately reach consensus. Poincaré, if he never actively took part in the crisis of foundations, followed its developments with interest. Like all protagonists of the time, he is keenly aware of the danger behind Russell's paradox. He even theorizes the problem as *impredicativity*[1]: An impredicative definition is in substance a circular definition in which the object that is defined is itself likely to be used in the definition. This is what happens when we define $y = \{x : x \notin x\}$: the set y defined to the left of the equality is also concerned to the right of the equal sign, since we potentially consider all the sets. If Zermelo's axiomatic avoids Russell's paradox, it nevertheless remains indirectly impredicative, as Poincaré will notice, who will write [184]:

"*By assuming in advance its set M* [Poincaré then speaks of the bound used by Zermelo in the axiom of comprehension, which makes it possible to define for an existing set M $\{x \in M : \Psi(x)\}$], *he* [Mr. Zermelo] *has erected a wall which stops any disturbers who might come from outside. But he does not ask himself if there may be hindrances from within that he has*

[1]Previously used by Russell in a slightly different sense.

locked up with him in his wall. If the set M has an infinity of elements, this does not mean that these elements can be conceived as existing in advance all at the same time, but that new ones can constantly be born; they will be born inside the wall, instead of being born outside, that's all."

Zermelo's system considers that if a set A exists, then the set of its parts $\mathcal{P}(A)$ also exists. This axiom combined with the axiom of restricted comprehension allows circular definitions to be made. Thus, in the definition $A = \{n \in \mathbb{N} : \forall S \in \mathcal{P}(\mathbb{N})\ n \notin S\}$, the quantifier $\forall S$ will take for value all the subsets of $\mathbb{N}$, and in particular the set A itself. The set A is therefore defined as a function of itself. Poincaré then adds at the end of his argument:

"*But if he* [Mr. Zermelo] *has closed his sheepfold well, I'm not sure the wolf hasn't locked up there. I would only be reassured if he had shown that he is immune to contradiction.*"

We can hardly prove Poincaré wrong: how can we be sure that a Russell's paradox will not appear out of nowhere, at the bend of a hidden circular definition? Zermelo himself is aware of the problem, and will seek to demonstrate without success that his axiomatic system is *consistent*, that is to say free from paradox. This search for proof of the consistency of mathematics reached its peak around 1920, under the leadership of David Hilbert.

Hilbert

Hilbert is certainly —just like Poincaré— one of the last mathematicians to have an in-depth knowledge of all the mathematics of his time. His work is considerable, and he profoundly influenced the developments of the discipline during the 20th century. He takes an active part in the crisis of the foundations by opposing to Russell a *formalist* vision of mathematics, rather than a *logicist* vision. For Hilbert, mathematics must be able to be reduced to a set of rules, which we must be applicable in a purely mechanical way and disconnected from any psychology of the mathematician. He thus imagines proof systems, within which he differentiates *axioms*, which are the mathematical sentences assumed to be true, for example the Zermelo-Fraenkel axioms, and *deduction rules*, which allow the axioms to be combined together to deduce theorems. It is Hilbert's vision that will eventually prevail, even if it does not happen without a stir. The ultimate objective for Hilbert is to show via deduction systems considered to be

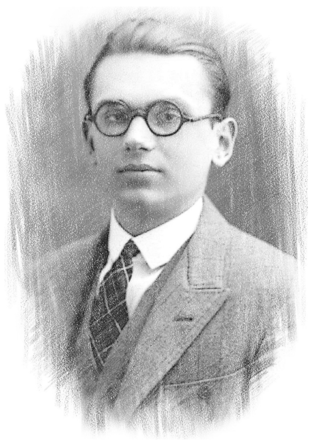
reliable — in particular via finite reasoning on finite objects — that the set of mathematics, which they call on infinite objects whose relevance is subject to caution, forms a consistent system, that is to say free from paradox: this is what we will call *Hilbert's program*, of which the Entscheindungsproblem mentioned in Section 6-1 constitutes one of the aspects. This program will come to a sudden stop with the work of Gödel, who demonstrates ten years later his famous incompleteness theorem: arithmetic itself is powerless to demonstrate that it is free from paradox.

The Fifth Musketeer

Gödel's work brings a conclusion as masterful as it is unexpected to the crisis of foundations. Gödel shows two things: even the simplest and best understood systems, like arithmetic, which speaks only of finite objects, contain unprovable truths. In particular, and supposing it to be true that the axioms of arithmetic form a system free from paradoxes, then the consistency of arithmetic is itself one of those unprovable truths. Gödel finally shows — with the help of Rosser — that the addition of axioms changes nothing: any consistent system of axioms, containing arithmetic, and "whose axioms can be known" cannot demonstrate its own consistency. Gödel developed for this the first versions of what would later be computability theory: "whose axioms can be known" means computable, in a similar sense to the modern one.

The repercussions in the mathematical world are colossal. Hilbert's program is down, and mathematics will never have a fully satisfactory foundation.

Even today, we do not know whether the ZF system, which axiomatizes all mathematics, is consistent: and for good reason, if, as we hope, it is indeed free from paradox, we will never be able to demonstrate it mathematically, or by anyway as long as one confines oneself to the axioms of ZF. The epistemological impact is considerable. Mathematics, mother of the exact sciences, is not only dependent on a *belief*, but is also able to demonstrate that it will always be so!


Kurt Gödel, 1906–1978

2. First-order logic

From Frege and Russell, we will retain the modern logical language used in mathematics, from Hilbert we will retain a proof system based on axioms and deduction rules applicable to mathematical statements, and from Zermelo, we will retain the axiomatic system ZF or ZFC, sufficient to formalize all of traditional mathematics. We now present without going into too much detail the basic principles of first-order logic, our objective being to present in a more precise way Gödel's theorem and its consequences. For this reason, the common thread of our presentation will be the specific example of Peano arithmetic.

2.1. Arithmetic language

The first step to formalize our mathematical demonstrations is to fix the language. We will therefore define the language of Peano arithmetic.

Definition 2.1. The language $\mathcal{L}_{\mathrm{PA}}$ of Peano arithmetic includes symbols specific to predicate calculus:

(1) *Variable* symbols $x, y, z, \ldots$: they represent natural numbers.

(2) Parentheses () and symbols of *logical connectors*: $\wedge, \vee, \rightarrow, \neg$.

(3) *Quantifier* symbols: $\forall, \exists$.

And those specific to Peano arithmetic:

(1) The following symbols of *binary functions*: $+$, $\times$.

(2) The following symbols of *binary relations*: $=, <$.

(3) *Constant* symbols $\dot{0}, \dot{1}$. ♢

A language is nothing but a list of symbols. However, these symbols are intended to be used with a specific meaning. Regarding predicate calculus, this is the usual meaning: for example "$\wedge$" is the logical *and*, while "$\exists$" is the existential quantification. Regarding the symbols specific to Peano arithmetic, there are first the functions $+$ and $\times$ which respectively represent addition and multiplication, the symbols of equality and inequality. which have their usual meanings on integers, and finally the constants $\dot{0}, \dot{1}$ which each represents the corresponding respective integer.

First-order languages

It is easy to see how to generalize the previous definition to obtain other languages. The symbols specific to predicate calculus are the same for all first-order languages, to which we add an arbitrary number of function symbols (n-ary for arbitrary integers n), an arbitrary number of symbols of relations (also n-ary for arbitrary integers n) and an arbitrary number of constant symbols.

The function symbols are subject to arrangement rules to form what are called the *terms* of the language:

Definition 2.2. The *terms* of predicate calculus for arithmetic are inductively defined as follows.

(1) A variable or a constant symbol is a term.

(2) If t_1, t_2 are terms, then $(t_1 + t_2)$ and $(t_1 \times t_2)$ are terms. ♢

Example 2.3. The following expressions are terms: x, $((((x + \dot{1}) + \dot{1}) + \dot{1}) + \dot{1} + \dot{1})$, $(\dot{1} + \dot{0})$, $(x + (y \times z))$

As we can see, the language of arithmetic is quite minimalist, and the valid expressions are very structured to remove any ambiguity. In practice, a number of notation shortcuts will be used to improve readability, as long as the translation in valid terms is unambiguous. For example, $t_0 + t_1 + t_2$ is a shorthand for $((t_0 + t_1) + t_2)$. Likewise, $x + \dot{3}$ is a shorthand for $(x + ((\dot{1} + \dot{1}) + \dot{1}))$.

Definition 2.4. A term is *closed* if it does not contain any variable and therefore only constants and the operations $+$ and $\times$. ♢

Intuitively, a closed term in arithmetic is a way of representing a natural number. For example, $(\dot{1} + \dot{1}) \times \dot{0}$ is a name for the integer 0. The integers each have an infinite number of names.

Example 2.5. The term $x + \dot{1}$ is not closed, unlike $(\dot{1} + \dot{1}) \times \dot{0}$.

If the function symbols are used to create the terms of the language, the relation symbols are used to create the *formulas* of the language:

Definition 2.6. The *arithmetic formulas* are defined as follows:

(1) For all terms t_1, t_2, then $t_1 = t_2$ and $t_1 < t_2$ are formulas. These formulas are called *atomic formulas*. The atomic formulas and their

negations, here, $\neg t_1 = t_2$ and $\neg t_1 < t_2$, are called *literals*.

(2) For all formulas F_1, F_2, then $(F_1 \wedge F_2), (F_1 \vee F_2), (F_1 \to F_2)$ and $\neg F_1$ are formulas.

(3) For any formula F, then $\forall x F$ and $\exists x F$ are formulas. ♢

There again, we will resort to syntactic sugar by writing $t_1 \leqslant t_2$ for the formula $(t_1 < t_2) \vee (t_1 = t_2)$ and $F_1 \leftrightarrow F_2$ for the formula $(F_1 \to F_2) \wedge (F_2 \to F_1)$.

First-order formulas

Here as well, we can easily generalize the formation of formulas and terms in any language: the function symbols of the language are used to create the terms, which then serve with the help of the relation symbols to create atomic formulas, which can then be composed between them with the help of the symbols of predicate calculus as in (2) and (3) of the previous definition.

With the quantifiers appear the notions of *bounded* and *free* variables: the *bounded* variables are unsurprisingly those which are bound to a quantifier, and the free variables are those which are not. The formal definition is quite heavy, but a few examples are enough to create an intuition.

Example 2.7. In the following formula: "$\forall x\ \exists y\ y = x + \dot{1}$" the variables x and y are bounded while in the following formula: "$\exists y\ y = x + 1$" only the variable y is bounded, unlike the variable x which is free.

Definition 2.8. A *closed formula* or a *statement* is a formula in which no variable is free. ♢

Notation

Given a formula F having for free variables $x_1, \ldots, x_n$, we will write $F(x_1, \ldots, x_n)$ to signify that the free variables of F are $x_1, \ldots, x_n$.

Intuitively, a closed formula is an affirmation which will have a truth value (true or false) when evaluated on integers. Formulas with free variables define predicates on integers.

2.2. Hilbert-style deduction systems

In order to mathematically formalize the notion of proof, Hilbert imagined a very precise system of rules, which suffice to show "everything that is demonstrable". How do we know? This idea will be made precise with

Gödel's completeness theorem to come. Subsequently, many other demonstration systems were developed, all equivalent and more or less suited to certain objectives.

2.2.1. Axioms and rules

In a Hilbert-style system, a proof is a finite list of mathematical sentences $F_0, F_1, F_2, \ldots, F_n$ —formulas in the considered language— satisfying the following rules: for any $i \leqslant n$, either F_i is a axiom, or F_i is produced from inference rules applied to formulas $F_{j_1}, \ldots, F_{j_m}$ for $j_1, \ldots, j_m < i$. Each sentence F_i in this list will then be demonstrated, the objective being normally to obtain F_n, the last of them. Let us now see a specific example of a system à la Hilbert powerful enough to demonstrate all that is demonstrable.

Axioms: The axioms that we can always use are the tautologies of first-order logic. Thus, for example $A \vee \neg A$ could be used as an axiom. There are three types:

1. The tautologies of propositional logic. For example $(F \to G) \to (\neg G \to \neg F)$ is a tautology of propositional logic: it will be true for any formula F or G regardless of their truth value.

2. The tautologies of predicate calculus. In practice, only four axiom schemes are necessary:

   (a) $\forall x(F \to G) \to (F \to \forall x G)$ for any formula F not containing the variable x, and any formula G.

   (b) $\exists x(F \to G) \to (\exists x F \to G)$ for any formula F, and any formula G not containing the variable x.

   (c) $\forall x F \to F_{t/x}$ for any term t and any formula F containing no variable of t.

   (d) $F_{t/x} \to \exists x F$ for any term t and any formula F containing no variable of t.

   Above $F_{t/x}$ denotes the formula F for which each occurrence of x is replaced by the term t.

3. The axioms of equality:

   (e) $t = t$ for any term t.

   (f) $t_1 = q_1 \wedge \cdots \wedge t_n = q_n \to f(t_1, \ldots, t_n) = f(q_1, \ldots, q_n)$ for all n, all terms $(t_i)_{1 \leqslant i \leqslant n}, (q_i)_{1 \leqslant i \leqslant n}$ and any n-ary function symbol f.

   (g) $t = q \to (F(t/z) \to F(q/z))$ for any terms t, q and any formula $F(z)$ not involving variables of t or q.

Above, $F(t/z)$ and $F(q/z)$ denote the formula F in which each occurrence of z is replaced by t and q, respectively.

Note that these axiom schemes depend on the language considered, each language using symbols of functions and relations which are specific to them, to construct the atomic terms and formulas respectively.

Equality symbol

We consider here that the symbol of equality is necessarily part of the language that we use, and will always have its usual meaning, which justifies the axioms of equality mentioned above.

Inference rules. Inference rules allow us to combine sentences already demonstrated in our list, to obtain new ones. The following two rules are sufficient.

1. *Rule 1: Modus Ponens* - the basis of all deductive reasoning. If $A \to B$ is proved and if A is proved, then we can deduce B.
2. *Rule 2: Generalization.* If $F(x)$ is proved for a variable x free in F, then can deduce $\forall x F(x)$. This rule is widely used in mathematics: if we want to prove for example that for all rationals $x < y$, there exists a rational z such that $x < z < y$, we start by fixing rational variables x, y on which we assume nothing other than $x < y$. If we manage to deduce the existence of a rational z such that $x < z < y$, without using any specific property of x, y, we deduce by the generalization rule that for all rational $x < y$, there exists a rational z such that $x < z < y$.

This concludes the description of our system à la Hilbert. Let's see an example of a demonstration right away.

Example 2.9. Let us show $\forall x F(x) \to \exists x F(x)$. For more readability we will note $A \equiv \forall x F(x)$, $B \equiv F(y)$ and $C \equiv \exists x F(x)$.

(1) $A \to B$ (axiom (c)).

(2) $B \to C$ (axiom (d)).

(3) $(A \to B) \to ((B \to C) \to ((A \to B) \wedge (B \to C)))$ (tautology).

(4) $(B \to C) \to ((A \to B) \wedge (B \to C))$ (Modus Ponens on (1) and (3)).

(5) $(A \to B) \wedge (B \to C)$ (Modus Ponens on (2) and (4)).

(6) $((A \to B) \wedge (B \to C)) \to (A \to C)$ (tautology).

(7) $A \to C$ (Modus Ponens on (5) and (6)).

Quid of empty universes?

The reader may be surprised by the sentence $\forall x F(x) \to \exists x F(x)$—that we have demonstrated. What happens if we place ourselves in an empty universe? At this time, $\forall x F(x)$ is correct, but not $\exists x F(x)$. The axiom (d) above effectively implies that the demonstrated formulas will only be valid if there is at least one element in our universe. The notion of universe will be made precise in Section 2.4 to come. It is in fact necessary to have such a restriction if one wants to demonstrate that any formula is equivalent to a formula in prenex form (see Definition 2.12).

2.2.2. First tools

We claim that the Hilbert-style system described above allows us to show everything that is demonstrable, and we will see this formally with the completeness theorem to come. The system is intentionally minimalist, and difficult to handle as it is. The reader can for example try to demonstrate $\neg\forall x F(x) \to \exists x\, \neg F(x)$ to realize the difficulty of using the system as it is. The mathematician who wants to do this kind of proof will proceed naturally as for any proof: assuming that $\neg\forall x F(x)$ is true, and trying to deduce $\exists x\, \neg F(x)$. The problem is that if we want to respect the formalism of a Hilbert system, $\neg\forall x F(x)$ is not necessarily an axiom that we can assume to be true in order to derive a conclusion. We then see our first fundamental tool, which will allow us to proceed as we are used to.

Lemma 2.10 (Deduction lemma). Let F be a closed formula. If we can prove G using F as an axiom, then there exists a proof of $F \to G$ (which does not use F as an axiom). ⋆

Proof. Let $G_1, \ldots, G_n$ be a proof of G_n using F as an axiom. Let us show by induction on the size of a proof that we can do some insertions in the sequence $F \to G_1, \ldots, F \to G_n$ in order to make a valid proof of it not using F as an axiom. If G_i is the statement F, then $F \to F$ is an axiom of propositional logic. If G_i is an axiom of propositional logic, then it is also the case for $F \to G_i$. If G_i is one of the axioms (a), (b), (c), (d) of predicate calculus, then $G_i \to (F \to G_i)$ is an axiom of propositional logic. By using the Modus Ponens on G_i and $G_i \to (F \to G_i)$, we get $F \to G_i$. If $G_i = \forall x\ G_j$ for $j < i$ is obtained by the generalization rule, then $\forall x\ (F \to G_j)$ is obtained from $F \to G_j$ (which we have by induction hypothesis) by the generalization rule. We obtain $F \to \forall x G_j$ by Modus Ponens and by axiom (a)

$$\forall x(F \to G_j) \to (F \to \forall x\ G_j).$$

Finally, if $G_i = G_b$ is obtained by Modus Ponens on G_a, then $G_a \to G_b$

for $a, b < i$. Then, we have $F \to G_a$ and $F \to (G_a \to G_b)$, by induction hypothesis. The formula

$$\big(F \to (G_a \to G_b)\big) \;\to\; \big((F \to G_a) \to (F \to G_b)\big)$$

is a tautology of predicate calculus. By Modus Ponens, we deduce $(F \to G_a) \to (F \to G_b)$ and, by a second application of Modus Ponens, we deduce $F \to G_b$. ■

Let us see immediately an example of application of the deduction lemma to prove $\neg\forall x F(x) \to \exists x\, \neg F(x)$.

Example 2.11. Let us show $\neg\exists x F(x) \to \forall x\, \neg F(x)$. From the deduction lemma, we can assume $\neg\exists x F(x)$ as an axiom.

(1) $\neg\exists x F(x)$ (axiom).

(2) $F(x) \to \exists x F(x)$ (axiom (d)).

(3) $(F(x) \to \exists x F(x)) \to (\neg\exists x F(x) \to \neg F(x))$ (tautology).

(4) $\neg\exists x F(x) \to \neg F(x)$ (Modus Ponens on (2) and (3)).

(5) $\neg F(x)$ (Modus Ponens on (1) and (4)).

(6) $\forall x\, \neg F(x)$ (generalization on (5)).

Let us now show $\forall x\, \neg\neg F(x) \to \forall x\, F(x)$.

(1) $\forall x\, \neg\neg F(x)$ (axiom).

(2) $\forall x\, \neg\neg F(x) \to \neg\neg F(x)$ (axiom (c)).

(3) $\neg\neg F(x)$ (Modus Ponens on (1) and (2)).

(4) $\neg\neg F(x) \to F(x)$ (tautology).

(5) $F(x)$ (Modus Ponens on (3) and (4)).

(6) $\forall x F(x)$ (generalization on (5)).

We leave it to the reader to use the contrapositive to deduce

$$\neg\forall x F(x) \to \exists x \neg F(x).$$

2.2.3. Prenex form

This proof system allows us to show that any formula — in any language — is provably equivalent to a formula in *prenex form.*

Definition 2.12. A formula is in *prenex form* if it is of the form

$$Q_1x_1 \dots Q_nx_n \ F(x_1, \dots, x_n, y_1, \dots y_m)$$

where each Q_i is a $\forall$ or $\exists$ quantifier and $F(x_1, \dots, x_n, y_1, \dots y_m)$ is a quantifier-free formula. $\diamondsuit$

We leave it to the reader to show the following equivalences:

- $\forall xF \ \wedge \ G \equiv \forall x(F \wedge G)$;
- $\forall xF \ \vee \ G \equiv \forall x(F \vee G)$;
- $\exists xF \ \wedge \ G \equiv \exists x(F \wedge G)$;
- $\exists xF \ \vee \ G \equiv \exists x(F \vee G)$.

These equivalences, coupled with Example 2.11 allow to transform any formula into a provably equivalent prenex formula in our system of deduction, by gradually shifting the quantifiers to the left.

Note that the equivalences above only hold if one places oneself in a universe having at least one element. Then, for example, we will have $(\forall x \ x = x) \wedge (\exists y \ y \neq y)$ false in the empty universe, but $\forall x \ (x = x \wedge (\exists y \ y \neq y))$ always true.

2.3. Logical and arithmetic theories of Peano

Once a language fixed — in our case, that of arithmetic — and the proof system specified, we can then consider a *mathematical theory* in this language, and use it to prove theorems concerning the structure described by this theory.

Definition 2.13. A *theory* T in a language $\mathcal{L}$ is a collection of closed formulas of that language. We also often use the term *axiomatic system* or more simply *system* to denote a theory. $\diamondsuit$

The theory is then seen as a list of axioms, which we can use in our proofs, in addition to the axioms present in the proof system. Let us see immediately the axioms of arithmetic which were developed by Peano towards the end of the 19th century.

2.3.1. Axioms of Peano arithmetic

Peano axioms allow us to specify the behavior of natural numbers. The first series of axioms defines the behavior of integers with respect to the successor.

(1) $\forall x \ \neg(x + \dot{1} = \dot{0})$: 0 has no predecessor.

(2) $\forall x\ (x = \dot{0} \vee \exists y\ (x = y + \dot{1}))$: Any integer other than 0 has a predecessor.

(3) $\forall x\ \forall y\ (x + \dot{1} = y + \dot{1} \to x = y)$: The successor function for integers is injective.

The following axioms give rules for computing addition and multiplication:

(4) $\forall x\ (x + \dot{0} = x)$;

(5) $\forall x \forall y\ (x + (y + \dot{1}) = (x + y) + \dot{1})$;

(6) $\forall x\ (x \times \dot{0} = \dot{0})$;

(7) $\forall x \forall y\ (x \times (y + \dot{1}) = (x \times y) + x)$.

Finally, we define the behavior of integers with respect to order:

(8) $\forall x \forall y\ (x < y \leftrightarrow (\exists z\ (z \neq \dot{0} \wedge x + z = y)))$

Notation

We denote by Q the theory composed of axioms (1) - (8), which form what we call *Robinson arithmetic.*

To obtain Peano arithmetic, we add the following axiom, for any arithmetic formula $F(x)$:

(9) $\Big(F(0) \wedge \big(\forall x \big(F(x) \to F(x + \dot{1})\big)\big)\Big) \to \forall x F(x)$

Note that axiom (9) is not a unique axiom. As for axioms (a), (b), (c), (d) of our proof system, it is an *axiom scheme*, that is to say of an infinity of axioms parameterized by a formula, here $F(x)$.

Statement (9) is the well-known axiom of induction on integers: if a formula F is true for the integer 0, and if the fact that it is true for n implies that it is true for $n + 1$, then it is true for all integer n.

Notation

We denote by PA the theory composed of Q and the axiom scheme (9) for any formula of arithmetic. It is known as *Peano arithmetic.*

We will see in Chapter 23 how to use the axioms of PA to show some elementary facts about natural numbers. We will see in particular that the induction scheme is equivalent to the following scheme: for any formula F true for at least one integer, there exists a smaller integer x such that $F(x)$ is true. This may of course seem perfectly obvious, because we have in mind the structure of the natural numbers $\mathbb{N}$ that we know well, but a proof does not use this structure: it uses only the axioms, and in the case of the arithmetic, these are precisely made so that any mathematical structure verifying them behaves like the natural numbers. This will bring us to the notion of *model*, in the next section.

2.3.2. Proofs in a theory

Once we have fixed a theory, we can use its axioms within a Hilbert system to prove mathematical statements.

Notation

We will note $T \vdash F$ to signify that there is a proof of the formula F from the axioms of T via the Hilbert system exposed in Section 2.2. If there is no such proof, then we write $T \nvdash F$.

Among the tautologies of propositional logic, we find for all formulas F, G the formula $(F \wedge \neg F) \to G$, called "ex falso quodlibet", meaning that from a contradiction $(F \wedge \neg F)$ we can deduce anything. It follows that if a theory proves a formula and its opposite, any statement is provable in this theory, which removes all interest from it. We will therefore expect above all from a theory that it be *consistent*.

Definition 2.14. A theory T is *consistent* if there is no formula F such that $T \vdash F \wedge \neg F$. ♢

Notation

We will write $T \vdash \perp$ to mean $T \vdash F \wedge \neg F$ for a certain formula F. The notation $T \nvdash \perp$ then logically signifying that for any formula F we have $T \nvdash F \wedge \neg F$. As the equality symbol is part of our language, we can consider without loss of generality $\perp$ as being $\neg x = x$. Since $x = x$ is an axiom, if $T \vdash \neg x = x$ then $T \vdash x = x \wedge \neg x = x$.

We saw in the introduction to our interlude that the inconsistency Russell found in Frege's work was a cornerstone in the crisis of foundations. So mathematicians would like as much as possible to be certain that they are working only with consistent theories. But how can we verify the consistency of a theory? Without even talking about theory, what about the consistency of the Hilbert system itself and its axioms of logic, presented in Section 2.2? The notion of *model* answers these questions.

2.4. Structures, models and consistency theorem

The notion of structure can be defined very generally for any fixed language. Let us start with the example which interests us more specifically, namely the language of arithmetic.

Definition 2.15. A *structure* $\mathcal{M} = (M, +^{\mathcal{M}}, \times^{\mathcal{M}}, <^{\mathcal{M}}, =^{\mathcal{M}}, 0^{\mathcal{M}}, 1^{\mathcal{M}})$ in $\mathcal{L}_{\mathrm{PA}}$ is given by:

- A non-empty set M
- $+^{\mathcal{M}}, \times^{\mathcal{M}} : M \times M \to M$ functions corresponding to $+, \times$ function symbols.
- A relation $<^{\mathcal{M}} \subseteq M \times M$ corresponding to the relation symbol $<$.
- A relation $=^{\mathcal{M}} \subseteq M \times M$ corresponding to the relation symbol $=$, and which corresponds to "true equality", i.e., such that $(x, y) \in \;=^{\mathcal{M}} \leftrightarrow x = y$.
- Elements $0^{\mathcal{M}}, 1^{\mathcal{M}} \in \mathcal{M}$ corresponding to the constant symbols $\dot{0}$ and $\dot{1}$. ◊

A structure is therefore a set, as well as functions, relations and constants on this set, constituting an interpretation of the symbols of language.

By abuse of notation, we will sometimes identify $\mathcal{M}$ with its *underlying set* M (we write for example $x \in \mathcal{M}$). To simplify the notations, we will sometimes remove the exponents ${}^{\mathcal{M}}$ when it is clear that we are talking about the functions and relations of the structure and not of symbols of the language.

First-order structure

It is easy to see how to generalize the previous definition to obtain structures for any first-order language. We always have a non-empty set M. Each n-ary function symbol f corresponds to a function of $f^{\mathcal{M}} : M^n \to M$, each n-ary relation symbol R corresponds to a relation $R^{\mathcal{M}} \subseteq M^n$ and each constant symbol c corresponds to a constant $c^{\mathcal{M}} \in M$. The equality relation will always be present and will always correspond to "the true equality".

A formula, like for example $F(x) = \exists y \; y \times (\dot{1} + \dot{1}) = x$ is only a sequence of symbols. Once a structure $\mathcal{M}$ has been fixed, each symbol is intended to be interpreted by the object which corresponds to it in $\mathcal{M}$; moreover, the free variables can also be replaced by various *parameters* — that is to say various elements — of the structure.

Definition 2.16 (Parametric formulas). Let $\mathcal{L}$ be a language and $\mathcal{M}$ a structure for $\mathcal{L}$. Given a formula $F(x_1, \ldots, x_n)$ of $\mathcal{L}$ having $x_1, \ldots, x_n$ as free variables, and given $a_1, \ldots, a_n \in \mathcal{M}$, the expression $F(a_1, \ldots, a_n)$ denotes a formula *parameterized* by $a_1, \ldots, a_n$: this is the formula F

in which each free occurrence of x_i is replaced by a_i for $1 \leqslant i \leqslant n$. A parametric formula without free variable will be a *closed parametric formula*. ♢

A closed parametric formula is no longer a simple sequence of symbols, but a statement which will be *true* or *false* in the considered structure.

Remark

Note that the notion of parametric formula induces that of parametric term. Thus, in the usual structure of integers for the language of arithmetic, $(5+4)\times 2$ will be a term parameterized by the elements $5, 4$ and 2 of our structure. This should not be confused with the closed term $(\dot{5}+\dot{4})\times\dot{2}$ with no parameter.

We now define satisfaction in a structure, for closed formulas or else parameterized in this structure.

Definition 2.17. Let $\mathcal{L}$ be a language and $\mathcal{M} = (M, \dots)$ a structure for $\mathcal{L}$. We say that a formula $F(x_1, \dots, x_n)$ of $\mathcal{L}$ is *true* in $\mathcal{M}$ for parameters $a_1, \dots, a_n \in \mathcal{M}$, and we write $\mathcal{M} \models F(a_1, \dots, a_n)$, if $F(a_1, \dots, a_n)$ is actually satisfied in the structure. The definition is formally done by induction on the formulas (in what follows $\overline{x}$ and $\overline{a}$ are shortcuts for $x_1, \dots, x_n$ and $a_1, \dots, a_n$):

- Base case: $\mathcal{M} \models R(t_1(\overline{a}), \dots, t_m(\overline{a}))$ where R is an m-ary relation symbol of the language corresponding to the relation $R^{\mathcal{M}} \subseteq M^m$ and each $t_i(\overline{x})$ is a term, iff $(t_1^{\mathcal{M}}(\overline{a}), \dots, t_n^{\mathcal{M}}(\overline{a})) \in R^{\mathcal{M}}$ where each $t_i^{\mathcal{M}}(\overline{a})$ is the element of M obtained by applying the functions corresponding to each function symbol used in $t_1(\overline{a})$.
- Universal quantification: $\mathcal{M} \models \forall y\ G(y, \overline{a})$ iff $\mathcal{M} \models G(b, \overline{a})$ for all $b \in M$.
- Existential quantification: $\mathcal{M} \models \exists y\ G(y, \overline{a})$ iff there is $b \in M$ such that $\mathcal{M} \models G(b, \overline{a})$.
- Negation: $\mathcal{M} \models \neg G(\overline{a})$ iff $\mathcal{M} \not\models G(\overline{a})$.
- Conjunction: $\mathcal{M} \models G_1(\overline{a}) \wedge G_2(\overline{a})$ iff $\mathcal{M} \models G_1(\overline{a})$ and $\mathcal{M} \models G_2(\overline{a})$.
- Disjunction: $\mathcal{M} \models G_1(\overline{a}) \vee G_2(\overline{a})$ iff $\mathcal{M} \models G_1(\overline{a})$ or $\mathcal{M} \models G_2(\overline{a})$.

The satisfaction of implication is deduced from that of negation and disjunction. ♢

Once a theory is fixed, we can consider models of this theory, that is, structures within which each axiom of the theory will be true.

Definition 2.18. Let T be a theory in a language $\mathcal{L}$. A structure $\mathcal{M}$ in $\mathcal{L}$ is a *model* of T if every axiom of T is true in $\mathcal{M}$. ♢

Example 2.19.

- The set $\mathbb{Z}$ equipped with the usual operations is not a model of Peano arithmetic because it does not satisfy axiom (1): 0 has no predecessor.
- The set $2\mathbb{N}$ of even integers where $\dot{0}$ is interpreted by 0 and $\dot{1}$ is interpreted by 2 is also not a model of the arithmetic of Peano because it does not verify axiom (7): $2 \times (2+2) \neq (2 \times 2) + 2$.
- The model par excellence of Peano arithmetic is of course that of natural numbers $\mathbb{N}$, equipped with the usual operations and relations.

While theories form the *syntactic* aspect of mathematics, the models form the *semantic* aspect. There are many advantages to thinking about models of a theory.

In the first place, models are what mathematicians have naturally worked with since the beginning. Today mathematics is so advanced in abstraction that it is an aspect of things that we sometimes lose sight of, but this science is not initially that far from physics, in the sense that it is first of all an *observation* work of the reality of certain phenomena in order to extract the logical laws which govern them. Every mathematician knows from experience that he does not decide the truth, which sometimes resides well hidden in the abstract structures studied, in other words in the models. This methodology of observation and research based on logic gives its universal and transcendent character to mathematical truth, constituting in a way the glue which binds the community of mathematicians. If syntax is of course important, because it constitutes the language allowing mathematics to be communicated, *semantics precedes syntax*[2]: it is from there that intuitions start and about it that the theorems bear.

Finally, there is a more prosaic advantage to the study of models: in essence, if a closed formula F is true in a model, then its negation $\neg F$ cannot be true there. A model is always a consistent and complete structure: each closed formula is either true or false, and no formula can be true at the same time as its negation. We can use this to show the consistency theorem.

[2]Aphorism dear to Professor René Cori, who taught the authors of this book the principles of present chapter.

Theorem 2.20 (Soundness theorem)
Let T be a theory and $F(x_1, \ldots, x_n)$ a formula in the language of that theory. If $T \vdash F(x_1, \ldots, x_n)$ then any model of T is also a model of $\forall x_1 \ldots \forall x_n \ F(x_1, \ldots, x_n)$.

Proof. The proof is easily done by induction on the size of a proof. Suppose this is the case for any proof $G_1, \ldots, G_n$ in T. Let $G_1, \ldots, G_{n+1}$ be a proof in T. By induction hypothesis any model of T is a model of each formula $\forall \overline{x} G_i$ for $i < n+1$ where the notation $\forall \overline{x} G_i$ means that we universally quantify on each free variable of G_i (when there is). Let $\mathcal{M}$ be a model of T. If G_{n+1} is an axiom of T, then it is a closed formula and $\forall \overline{x} G_{n+1}$ is obviously true in $\mathcal{M}$. If G_{n+1} is an axiom of equality (of type (e) (f) or (g) above), then $\forall \overline{x} G_{n+1}$ is true in $\mathcal{M}$ by the fact that the equality of $\mathcal{M}$ is always true equality.

If G_{n+1} is a tautology of propositional logic or an axiom of type (a) (b) (c) or (d) of predicate calculus, then $\forall \overline{x} G_{n+1}$ is true by definition of the satisfaction in a model (the details are left to the reader, note that for (d) we use the fact that the model is not empty).

If G_{n+1} is obtained by generalization on G_i for $i < n+1$ — in particular G_{n+1} is of the form $\forall y G_i$ — then $\mathcal{M}$ satisfying $\forall \overline{x} G_i$, it also satisfies $\forall \overline{x} G_{n+1}$ (which is in fact the same formula). Finally if G_{n+1} is obtained by Modus Ponens via $G_i \to G_{n+1}$ and G_i for $i < n+1$ then $\mathcal{M}$ satisfies $\forall \overline{x} G_i$ and $\forall \overline{x}(G_i \to G_{n+1})$. By definition of satisfaction, $\mathcal{M}$ thus satisfies $\forall \overline{x} G_{n+1}$. ∎

The previous theorem can be summarized as follows: "one can only prove true things". This is good news, from which we can deduce our consistency theorem:

Corollary 2.21 (Consistency theorem)
If an axiomatic system admits a model, then it is consistent.

Proof. This is shown by contraposition. If $T \vdash F \wedge \neg F$ for a closed formula F, then any model of T is a model of $F \wedge \neg F$. Since there is no model for $F \wedge \neg F$ then T has no model. ∎

We easily verify the existence of a model of the axioms of logic: the set $\{1\}$ with the relation of equality $1 = 1$. This shows that the axioms of logic are consistent and cannot prove $\perp$.

We also easily verify the existence of a model for Peano arithmetic, namely $\mathbb{N}$ equipped with the usual functions of addition and multiplication, as well as the usual relations $<$ and $=$ on integers. This is the second good news, the axioms of Peano arithmetic are also consistent. We will examine the meaning of this statement a little further on, notably in the light of Gödel's second incompleteness theorem and its implications.

2.5. Models and completeness theorem

If the system of deduction à la Hilbert that we have given, with its axioms of logic and its rules of deduction is indeed sound, how on the other hand know that it is sufficiently powerful? After all, the two inference rules seem to form a very poor working tool. Can we really demonstrate everything with this system? We will see that this is the case, via a theorem demonstrated by Gödel in his doctoral thesis which can be seen as the converse of the consistency theorem: everything that is universally true is demonstrable.

Theorem 2.22 (Théorème de complétude de Gödel)
Let T be a theory in a countable language. If T is consistent, then T has a model.

Note that the completeness theorem for uncountable languages can also be demonstrated using the axiom of choice. We only show a countable version which will be more than enough for us. Gödel's completeness theorem is more difficult to prove than the consistency theorem which is a simple routine check. It is a question here of building a model of a theory T, from the simple fact that $T \nvdash \bot$. The idea of the proof passes by the creation of a *complete* theory.

Definition 2.23. A theory T is called *complete* if $T \vdash F$ or $T \vdash \neg F$ for any closed formula F in the language of T. ♢

Proposition 2.24. Let $\mathcal{L}$ be a countable language. Any consistent theory of $\mathcal{L}$ can be extended into a complete and consistent theory. ⋆

The proof of the above proposition uses Lemma 2.10 which we reformulate here with the notation $\vdash$ introduced since:

Lemma (2.10). Let $T \cup \{F\}$ be a theory and G a formula. Suppose $T \cup \{F\} \vdash G$. Then, $T \vdash F \to G$. ⋆

Preuve de Proposition 2.24. Given a consistent theory T in a countable language, we inductively construct a complete and consistent extension T'. Let $T_0 = T$. At step n, suppose that a consistent theory T_n is defined. Let F_n be the n-th closed formula of our language. If $T_n \vdash F_n$ we define $T_{n+1} = T_n \cup \{F_n\}$. If $T_n \vdash \neg F_n$ we define $T_{n+1} = T_n \cup \{\neg F_n\}$. In these first two cases, the consistency of T_{n+1} follows from the consistency of T_n and from the deduction lemma.

If ever T_n proves neither F_n nor $\neg F_n$, then we define $T_{n+1} = T_n \cup \{F_n\}$. Let us assume by the absurdity $T_n \cup \{F_n\} \vdash \perp$. Then, according to the deduction lemma $T_n \vdash F_n \to \perp$ and therefore $T_n \vdash \neg F_n$ by contraposition and Modus Ponens, which contradicts the fact that T does not prove $\neg F_n$. Note that we could just as well define $T_{n+1} = T_n \cup \{\neg F_n\}$.

Finally, we let $T' = \bigcup_n T_n$. The consistency of T' then comes from the fact that a proof in a theory uses only a finite number of its axioms: if $T' \vdash \perp$ then there necessarily exists n such that $T_n \vdash \perp$, Since each theory T_n is consistent, then T' must be consistent. ■

Proof of the completeness theorem. The proof that we give, is due to Leon Henkin [89], and rests on the creation of a complete, consistent theory, and having what one calls *Henkin witnesses.* We will first show the following statement.

"Let T be a theory in a language $\mathcal{L}$ and c a constant symbol which does not appear in $\mathcal{L}$. Suppose $T \cup \{\exists x F(x) \to F(c)\} \vdash \perp$. Then, $T \vdash \perp$."

As $T \cup \{\exists x F(x) \to F(c)\} \vdash \perp$ then by the deduction lemma, contraposition and Modus Ponens we have $T \vdash \exists x F(x) \wedge \neg F(c)$. In particular $T \vdash \neg F(c)$. Let now be a variable z which does not intervene in the proof of $\neg F(c)$. As the constant c does not appear in the theory T it can only be introduced into the proof by an axiom of logic. Each of these axioms remains valid by replacing c by z. We leave it to the reader to check that if we replace c by z for each step of the proof, we obtain a valid proof of $\neg F(z)$. By the generalization rule we finally get $T \vdash \forall x\ \neg F(x)$. As also $T \vdash \exists x F(x)$ then $T \vdash \perp$.

Let us now proceed to the proof of the completeness theorem. Let T be a theory in a countable language $\mathcal{L}$. Let's show how to build a model. We define $T_0 = T$ and $\mathcal{L}_0 = \mathcal{L}$. At step $n \in \mathbb{N}$, suppose that we have defined a consistent theory T_{2n} in a language $\mathcal{L}_{2n}$. Let $\mathcal{L}_{2n+1}$ be the language $\mathcal{L}_{2n}$ to which we add a new constant symbol c_G for any formula G of T_{2n} of the form $\exists x F(x)$. Let T_{2n+1} be the theory T_{2n} to which we add the statements $\exists x F(x) \to F(c_G)$ for any statement G of T_{2n} of the form $\exists x F(x)$. Note that by the above statement, as T_{2n} is consistent, each time an axiom of the form $\exists x F(x) \to F(c_G)$ is added, the theory remains consistent. So T_{2n+1} is consistent. Finally let $\mathcal{L}_{2n+2} = \mathcal{L}_{2n+1}$ and using Proposition 2.24, let T_{2n+2} be the completion of T_{2n+1} for the language $\mathcal{L}_{2n+2}$. Let $\mathcal{T}_\omega = \bigcup_n T_n$ and $\mathcal{L}_\omega = \bigcup_n \mathcal{L}_n$. Note that $\mathcal{T}_\omega$ is a complete and consistent theory in the language $\mathcal{L}_\omega$, which also contains a statement of the form $\exists x F(x) \to F(c)$ for each of the statements of the form $\exists x F(x)$ of T_ω, where c is a constant symbol of $\mathcal{L}_\omega$. The famous *Henkin's witnesses* are the new constant symbols thus introduced.

The underlying set M of our model $\mathcal{M}$ is the set of closed terms of $\mathcal{L}_\omega$, quotiented by the equality relation. Formally let $(t_n)_{n\in\mathbb{N}}$ be the list of closed terms of $\mathcal{L}_\omega$. Then, $M = \{t_n : \forall i < n\ (\neg t_i = t_n) \in T_\omega\}$.

Note that the function symbols of $\mathcal{L}$ have a clear interpretation in M. For example if f is a unary function symbol of $\mathcal{L}$ then its corresponding function $f^{\mathcal{M}} : M \to M$ is defined by $f^{\mathcal{M}}(t) = q$ for q the element of M which is equal to closed term $f(t) \in \mathcal{L}_\omega$, that is to say such that $(f(t) = q) \in T_\omega$.

The theory T_ω being complete and consistent, for any symbol of m-ary relation R of $\mathcal{L}$ and any element $t_1, \ldots, t_m \in M$, exactly one of the statements among $R(t_1, \ldots, t_m)$ or $\neg R(t_1, \ldots, t_m)$ is in T_ω. This induces an interpretation $R^{\mathcal{M}}$ of the relation symbol R of $\mathcal{L}$ in $\mathcal{M}$.

It remains to show by induction on the formulas that all the statements of T_ω are satisfied in $\mathcal{M}$ (and therefore also those of T). Without loss of generality, only the formulas in prenex form are treated, and we can assume that the negation symbol only appears in front of the atomic formulas. By definition of relations in $\mathcal{M}$ this is indeed the case for atomic formulas and their negations. If $F_1 \wedge F_2 \in T_\omega$ then as T_ω is complete $F_1, F_2 \in T_\omega$. By induction hypothesis $\mathcal{M} \models F_1$ and $\mathcal{M} \models F_2$ therefore $\mathcal{M} \models F_1 \wedge F_2$. If $F_1 \vee F_2 \in T_\omega$ then as T is complete $F_1 \in T_\omega$ or $F_2 \in T_\omega$ (otherwise by completeness $\neg F_1, \neg F_2 \in T_\omega$ which contradicts $F_1 \vee F_2$). By induction hypothesis $\mathcal{M} \models F_1 \vee F_2$. If $\forall x F(x) \in T_\omega$ then as T_ω is complete $F(t)$ is in T_ω for any closed term t of $\mathcal{L}_\omega$ and therefore any element of M. By induction hypothesis $\mathcal{M} \models F(t)$ for all $t \in M$ and therefore $\mathcal{M} \models \forall x\ F(x)$. If $\exists x F(x) \in T_\omega$ then $\exists x F(x) \to F(c) \in T_\omega$ for a symbol with constant $c \in \mathcal{L}_\omega$. In particular $F(c) \in T_\omega$ by Modus Ponens. Let $t \in M$ be a closed term of $\mathcal{L}_\omega$ such that $(t = c) \in T_\omega$. By the axioms of equality we have $F(t)$. By induction hypothesis $\mathcal{M} \models F(t)$. So $\mathcal{M} \models \exists x F(x)$. ■

The completeness theorem is often used in the form of the following corollary, which can be seen as a reverse of Theorem 2.20:

Corollary 2.26
If every model of a theory T is a model of a formula F, then $T \vdash F$.

PROOF. If we have $T \cup \{\neg F\} \vdash \perp$ then $T \vdash \neg F \to \perp$ by the deduction lemma and therefore $T \vdash F$.

Suppose now $T \nvdash F$, then by the line above $T \cup \{\neg F\} \nvdash \perp$. According to the completeness theorem there exists a model of $T \cup \{\neg F\}$, that is to say a model of T which is not a model of F. ■

The completeness theorem implies in particular that one cannot do better than the Hilbert system that we have presented: suppose that in this system the axioms of logic alone are not sufficient to prove a formula F, in other words $\nvdash F$ (from an empty theory). Suppose that a more powerful and consistent proof system exists such that $\vdash^* F$, where $\vdash^*$ is the notion of proof in this system. So also, $\vdash^* \neg F \rightarrow\perp$ and therefore $\neg F \vdash^* \perp$. Now as $\nvdash F$, according to the completeness theorem there is a model of $\neg F$ and according to the consistency theorem for $\vdash^*$ our model is therefore a model of $\perp$ which is impossible.

3. Incompleteness theorems of Gödel

We now get to the heart of the matter, via Gödel's incompleteness theorems, which are based, among other things, on a coding of computably enumerable sets by arithmetic formulas.

3.1. Peano arithmetic formulas

We have defined in Chapter 5 a complexity hierarchy on sets, called *arithmetic hierarchy*. We are now going to give all its meaning to this name by defining a syntactic hierarchy of arithmetic formulas, which coincides with the arithmetic hierarchy, in the sense that a set A is Σ^0_n iff it is definable by a Σ_n formula of Peano arithmetic.

Definition 3.1. A formula of Peano arithmetic is Δ_0 if the quantifications it comprises are all bounded, that is to say of the form $\exists x < t$ and $\forall x < t$, with t a term where the variable x does not appear freely. Note that the formulas $\exists x < t\ F(x)$ and $\forall x < t\ F(x)$ translate respectively to $\exists x(x < t \wedge F(x))$ and $\forall x(x < t \rightarrow F(x))$. $\diamondsuit$

Given a Δ_0 formula of Peano arithmetic $F(x_1, \ldots, x_n)$, the restriction on quantifications makes the set $\{(x_1, \ldots, x_n) \in \mathbb{N} : \ F(x_1, \ldots, x_n)\}$ computable. This follows for example directly from the closure of primitive recursive predicates by conjunction, disjunction and bounded quantification (see Example 6-3.16 and Exercise 6-3.19). We define a complexity hierarchy on the formulas of Peano arithmetic analogous to the arithmetic hierarchy of Definition 5-1.1.

Definition 3.2.

1. A formula $F(x_1,\dots,x_m)$ of Peano arithmetic is Σ_n if
$$F(x_1,\dots,x_m) = \overbrace{\exists y_1 \forall y_2 \dots Q y_n}^{n \text{ quantifiers}}\ G(x_1,\dots,x_m,y_1,\dots,y_n)$$
for a Δ_0 formula $G(x_1,\dots,x_m,y_1,\dots,y_n)$, where Q is $\exists$ if n is odd, and $\forall$ if n is even.

2. A formula $F(x_1,\dots,x_m)$ of Peano arithmetic is Π_n if
$$F(x_1,\dots,x_m) = \overbrace{\forall y_1 \exists y_2 \dots Q y_n}^{n \text{ quantifiers}}\ G(x_1,\dots,x_m,y_1,\dots,y_n)$$
for a Δ_0 formula $G(x_1,\dots,x_m,y_1,\dots,y_n)$, where Q is $\forall$ if n is odd, and $\exists$ if n is even. ♢

We will see that the sets definable by a Σ_n (resp. Π_n) formula of Peano arithmetic coincide with the Σ^0_n (resp. Π^0_n) sets of Definition 5-1.1. We start for that by showing some closure properties similar to those of propositions 5-1.6 to 5-1.9.

Proposition 3.3. Let $F(\overline{a},x), F_1(\overline{a},x), F_2(\overline{a},x)$ be Σ_n (resp Π_n) formulas. Then, each of the following formulas is provably equivalent (using the axioms of arithmetic) to a Σ_n (resp. Π_n) formula:

- $F_1(\overline{a},x) \wedge F_2(\overline{a},x)$, $F_1(\overline{a},x) \vee F_2(\overline{a},x)$
- $\exists x < b\ F(\overline{a},x)$, $\forall x < b\ F(\overline{a},x)$,
- $\exists x\ F(\overline{a},x)$ (resp. $\forall x\ F(\overline{a},x)$) ⋆

Proof. Let $F(\overline{a},x) \equiv \exists y G(\overline{a},x,y)$, $F_1(\overline{a},x) \equiv \exists y G_1(\overline{a},x,y)$ and $F_2(\overline{a},x) \equiv \exists y G_2(\overline{a},x,y)$. Then, we have the following equivalences:

$$\begin{array}{rcl}
F_1(\overline{a},x) \wedge F_2(\overline{a},x) & \leftrightarrow & \exists y\ \exists y_1, y_2 < y\ (G_1(\overline{a},x,y_1) \wedge G_2(\overline{a},x,y_2)) \\
F_1(\overline{a},x) \vee F_2(\overline{a},x) & \leftrightarrow & \exists y\ (G_1(\overline{a},x,y) \vee G_2(\overline{a},x,y)) \\
\exists x < b\ F(\overline{a},x) & \leftrightarrow & \exists y\ \exists x < b\ G(\overline{a},x,y) \\
\forall x < b\ F(\overline{a},x) & \leftrightarrow & \exists z\ \forall x < b\ \exists y < z\ G(\overline{a},x,y) \\
\exists x\ F(\overline{a},x) & \leftrightarrow & \exists z\ \exists x < z\ \exists y < z\ G(\overline{a},x,y).
\end{array}$$

Now, if F, F_1, F_2 are Σ_1 with G, G_1, G_2 Δ_0, the above equivalences show the proposition for the Σ_1 case. By passing to the negation, the equivalences also hold for the Π_1 case. Suppose the proposition is true for the Σ_n and Π_n cases. Then, the above equivalences for F, F_1, F_2 of the Σ_{n+1} formulas with G, G_1, G_2 Π_n imply —using the induction hypotheses on G, G_1, G_2— the proposition for the Σ_{n+1} case. By passing to negation, the proposition is true for the Π_{n+1} case. ■

Note that the fourth equivalence in the above proof (the equivalence $\forall x < b\, F(\overline{a}, x) \leftrightarrow \exists z\, \forall x < b\, \exists y < z\, G(\overline{a}, x, y)$) is the least trivial of all: the other four simply use the fact that two integers always have an upper bound, while this one requires the use of induction. This is something we will study in detail in Section 23-3.

Let's now move on to the announced equivalence. Given a Σ_1 formula of Peano arithmetic $\exists y\, F(x_1, \dots, x_n, y)$ where F is Δ_0, the set $\{(x_1, \dots, x_n) \in \mathbb{N} : \exists y\, F(x_1, \dots, x_n, y)\}$ is computably enumerable: we test the formula F little by little on all $(n+1)$-tuples $x_1, \dots, x_n, y$ and when we find one for which F is true, we enumerate $(x_1, \dots, x_n)$. Gödel showed that any computably enumerable set could in fact be represented in this form.

Theorem 3.4 (Gödel)
A set of integers $A \subseteq \mathbb{N}$ is c.e. iff there exists a Σ_1 formula $F(n)$ of $\mathcal{L}_{\mathrm{PA}}$ such that $n \in A$ iff $\mathbb{N} \vDash F(n)$.

We will show Theorem 3.4 based on the model of general recursive functions which coincide, as we saw in Chapter 6, with computable functions. We will show that any general recursive partial function $f : \mathbb{N}^k \to \mathbb{N}$ is *represented* by a Σ_1 formula of arithmetic $F(n_1, \dots, n_k)$, that is, to say

$$\{(\overline{n}, r) \in \mathbb{N}^{k+1} : f(\overline{n}) \downarrow= r\} = \{(\overline{n}, r) \in \mathbb{N}^{k+1} : \mathbb{N} \vDash F(\overline{n}, r)\}.$$

As any c.e. set is the domain of a partial function, this proves the theorem. The main difficulty lies in the management of the primitive recursion scheme, for which we need to encode lists of integers by arithmetic formulas. Gödel resorts to a clever use of a result of modular arithmetic: the Chinese Remainder Theorem.

Lemma 3.5 (Chinese Remainder Theorem). Let $(a_0, \dots, a_n)$ be an arbitrary sequence of integers. Let $(p_0, \dots, p_n)$ be a sequence of pairwise prime integers with $p_i \geqslant a_i$ for $i \leqslant n$. Then, there exists an integer b such that a_i is the remainder of the Euclidean division of b by p_i for all i. ⋆

The Chinese Remainder Theorem will allow Gödel to code lists of integers of arbitrary size. Note that this is not the only way to establish a system for coding/decoding lists by arithmetic formulas, and we will see another one in Section 23-4.

Lemma 3.6 (Gödel's β function). There exists a function $\beta : \mathbb{N}^3 \to \mathbb{N}$ represented by a Δ_0 formula, such that for all n and any sequence of integers $(a_0, \dots, a_n)$, there exist integers $a, b \in \mathbb{N}$ for which $\beta(a, b, i) = a_i$ for all $i \leqslant n$. ⋆

Proof. The formula $B(a, b, i, r)$ which represents β is as follows: "r is the remainder of the Euclidean division of b by $a(i+1)+1$". The formula is indeed Δ_0: $r < a \times (i+1) + 1 \ \wedge \ \exists c < b \ c \times (a \times (i+1) + 1) + r = b$. Let $(a_0, \ldots, a_n)$ be a sequence of integers. Let us show the existence of integers a, b such that this formula defines the function $(a, b, i) \mapsto a_i$.

Let m be such that $m > \max\{a_i : i \leqslant n\}$ and $m > n$. Let $a = m!$. We will use the Chinese Remainder Theorem with $p_i = a(i+1)+1$. Let us show that these numbers are pairwise prime. Suppose absurdly that a prime number p divides p_i and p_j with $i < j$. So p also divides $p_j - p_i = a(j-i)$. Since p is prime then p divides a or p divides $j-i$. As $a = m!$ with $m > n \geqslant j > i$ then $j-i$ divides a and therefore in all cases p divides a. So p divides $a(i+1)$. Since p also divides $a(i+1)+1$, then p divides $a(i+1)+1-a(i+1) = 1$ which is a contradiction. So the numbers p_i are mutually prime.

We have $p_i = a(i+1)+1 > m > a_i$ for all i. According to the Chinese Remainder Theorem, there exists an integer b such that for all i, the integer a_i is the remainder of the Euclidean division of b by $a(i+1)+1$. ■

Proof of Theorem 3.4. We show that any general recursive partial function is represented by a Σ_1 formula of arithmetic. We leave it to the reader to show that this is indeed the case for the basic functions (projections, constant functions and successor function). We use Proposition 3.3 without mentioning it for each of the three schemes to come, in order to obtain a Σ_1 formula which represents our function.

Composition scheme. Let

$$f(\overline{x}) = g(h_1(\overline{x}), \ldots, h_k(\overline{x}))$$

for functions $g, h_1, \ldots, h_k$ represented by formulas $G, H_1, \ldots, H_k$. Then, f is represented by the formula

$$F(\overline{x}, r) \equiv \exists y_1, \ldots, y_k \ H_1(\overline{x}, y_1) \wedge \cdots \wedge H_k(\overline{x}, y_k) \wedge G(y_1, \ldots, y_k, r).$$

Minimization scheme. Let

$$f(\overline{x}) = \min\{a \in \mathbb{N} : \forall i \leqslant a \ g(\overline{x}, i)\downarrow \ \wedge \ g(\overline{x}, a) = 0\}$$

for g represented by a formula G. Then, f is represented by the formula:

$$F(\overline{x}, a) \equiv G(\overline{x}, a, 0) \wedge \forall i < a \ \exists r \neq 0 \ G(\overline{x}, i, r).$$

Primitive recursion scheme. This is where we will need Gödel's β function, represented by the formula B. Let

$$\begin{array}{rcl} f(\overline{x}, 0) & = & g(\overline{x}) \\ f(\overline{x}, n+1) & = & h(\overline{x}, n, f(\overline{x}, n)) \end{array}$$

for functions g, h represented by formulas G, H. Then, f is represented by the following formula $F(\overline{x}, n, r)$:

$$\begin{aligned}\exists a, b \quad & B(a,b,n,r) \wedge \exists a_0\ (B(a,b,0,a_0) \wedge G(\overline{x}, a_0)) \wedge \forall i < n \\ & \exists a_i, a_{i+1}\ (B(a,b,i,a_i) \wedge B(a,b,i+1,a_{i+1}) \wedge H(\overline{x}, i, a_i, a_{i+1})).\end{aligned}$$

In order to see that f is well represented by F, it should be noted that B, as defined in the previous lemma, is always a functional formula: whatever the values of a, b, i, there are always at most one element r such that $B(a,b,i,r)$ is true. Thus, if the formula $F(\overline{x}, n, r)$ holds, there does exist a sequence $a_0, \ldots, a_n$ such that $g(\overline{x}) = a_0$ and $h(\overline{x}, i, a_i) = a_{i+1}$, with $a_n = r$, the result of $f(\overline{x}, n)$. According to the previous lemma, there are indeed, for all n integers a, b which code via the function β, the sequence of values $(f(\overline{x}, 0), f(\overline{x}, 1), \ldots, f(\overline{x}, n))$. The formula F therefore represents the function f. ■

We easily show from Theorem 3.4 that a set of integers is Σ^0_n (resp. Π^0_n) iff it is described by a Σ_n (resp. Π_n) formula of arithmetic.

Coding of finite sequences

There are many ways to encode finite sequences of integers using natural numbers. Most of these techniques use primitive recursive functions, which requires showing beforehand that they are representable by simple arithmetic formulas. The Chinese Remainder Theorem allows a simple encoding of finite sequences based on Euclidean division, which is expressed by an immediate Δ_0 predicate (see Lemma 3.6).

3.2. Proofs and computation

A proof in our system of deduction à la Hilbert relies on a very precise system of rules, and it is easy to create a computer program which takes as parameter a proof — via an appropriate coding — and which verifies in a finite time whether the demonstration is valid or not. Indeed, the inference rules are clear, as for the axioms of logic, we have the tautologies of propositional calculus, four schemes of axioms for predicate calculus, and three schemes of axioms for equality. It is easy to check whether a sentence of propositional calculus — for example of the form $(F \rightarrow G) \leftrightarrow (\neg G \rightarrow \neg F)$ — is a tautology, by ensuring that the sentence is always true for any truth value for F and G. It is also easy to check whether a sentence matches one of the equality axiom schemes or predicate calculus. We deduce the following theorem:

Theorem 3.7 (Gödel)
Given a computably enumerable theory T in the language $\mathcal{L}_{PA}$, the set

of formulas F such that $T \vdash F$ is computably enumerable.

It suffices to do a search on all the possible proofs from the axioms of T and to list all the formulas they prove. Hilbert's goal was to show that any arithmetical truth was provable, Peano arithmetic supposedly being a sufficient system to do so. If this were the case, then we could create an algorithm allowing to decide whether a mathematical statement F is provable or refutable: it would suffice to list all the statements proved by Peano arithmetic until we find F or $\neg F$.

Gödel showed that this was not possible, neither for Peano arithmetic, nor for any computably enumerable and consistent theory containing Peano arithmetic (however Rosser's help was needed for this last step). We start by proving a lemma which will help us in the following:

Lemma 3.8. If $\mathbb{N} \vDash F$ where F is a closed Σ_1 formula, then $\mathrm{PA} \vdash F$. ⋆

Proof. The formula F is of the form $\exists x_1 \dots \exists x_n\ G(x_1, \dots, x_n)$ for a Δ_0 formula G. If F is true in $\mathbb{N}$ then there are integers $a_1, \dots, a_n \in \mathbb{N}$ such that $G(a_1, \dots, a_n)$ is true in $\mathbb{N}$. It is easily shown by induction that if a Δ_0 formula and parameterized in $\mathbb{N}$ is true in $\mathbb{N}$, then it is provable in PA. Informally, for a bounded existential quantification, it suffices to take a witness integer for this quantification and to show the formula with this witness, and for a universal quantification bounded by an integer a, it suffices to show that the formula is true for each integer less than a. ■

We now have the necessary ingredients to prove the first incompleteness theorem.

Theorem 3.9 (Gödel's first incompleteness theorem)
Let $T \supseteq \mathrm{PA}$ be a consistent c.e. theory, such that if T proves a Σ_1 formula, then $\mathbb{N}$ is a model of this formula. Then, there exists a Σ_1 formula F such that $T \nvdash F$ and $T \nvdash \neg F$.

Proof. Let $(\Phi_e)_{e \in \mathbb{N}}$ be an enumeration of all computable functions. According to Theorem 3.4 there exists a Σ_1 formula $F(e)$ of $\mathcal{L}_{\mathrm{PA}}$ such that

$$\{e \in \mathbb{N} : \mathbb{N} \vDash F(e)\} = \{e \in \mathbb{N} : \exists t\ \Phi_e(e)[t] \downarrow\}.$$

Suppose that for all e, we have $T \vdash F(e)$ or $T \vdash \neg F(e)$. Note that if $\exists t\ \Phi_e(e)[t] \downarrow$ then $\mathbb{N} \vDash F(e)$ and therefore $\mathrm{PA} \vdash F(e)$ according to Lemma 3.8. As $\mathrm{PA} \subseteq T$ we also have $T \vdash F(e)$.

Now, if $\forall t\ \Phi_e(e)[t] \uparrow$ we have $\mathbb{N} \vDash \neg F(e)$. According to our hypothesis we cannot have $T \vdash F(e)$ because we would then have $\mathbb{N} \vDash F(e)$ which contradicts $\mathbb{N} \vDash \neg F(e)$. Since T is complete, we therefore have $T \vdash \neg F(e)$.

It follows that the c.e. set $\{e \in \mathbb{N} : T \vdash \neg F(e)\}$ coincides with the set $\{e \in \mathbb{N} : \forall t\ \Phi_e(e)[t] \uparrow\}$ which makes the complement of the halting problem a c.e. set. Contradiction. ■

We notice several things. In the first place we had to restrict ourselves to 1-*consistent* theories T, that is to say theories which do not prove Σ_1 formulas false in $\mathbb{N}$. We will soon see that there are many other possible models than $\mathbb{N}$ for PA, and as many non 1-consistent theories as we want. So this is a very annoying restriction.

Then the proof shows in substance that a complete and 1-consistent theory allows to compute $\emptyset'$. We have already seen that any PA degree allows to compute a complete and consistent extension of PA, and since there are PA degrees which do not compute $\emptyset'$, there is in fact no hope of showing Gödel's theorem by reducing the halting problem to any complete and consistent theory that extends PA. The trick to get out of this situation was found by Rosser, the idea being in essence to use the fact that if a theory proves $\exists t\ \Phi_e(e)[t] \downarrow= 0$ (whether this is true in $\mathbb{N}$ or not), then it cannot show at the same time $\exists t\ \Phi_e(e)[t] \downarrow= 1$, assuming of course that the Σ_1 formulas allowing to talk about computable functions, integrate well the fact that a function has at most one value on its input, which is the case in practice.

Theorem 3.10 (Gödel-Rosser's incompleteness theorem)
Let $T \supseteq \mathrm{PA}$ be a consistent c.e. theory. Then, there exists a Σ_1 formula F such that $T \nvdash F$ and $T \nvdash \neg F$.

Proof. Let $(\Phi_e)_{e\in\mathbb{N}}$ be an enumeration of all computable functions. Let us assume absurdly that we have $T \vdash F$ or $T \vdash \neg F$ for any Σ_1 formula F. We will then compute a total function $f : \mathbb{N} \to \{0, 1\}$ which is DNC_2, i.e., such that for any integer e we have $\Phi_e(e) \downarrow$ implies $f(e) \neq \Phi_e(e)$. Note that this is then a contradiction: if f is computable then there exists a code e such that $\Phi_e(n) \downarrow= f(n)$ for all n and therefore in particular such that $\Phi_e(e) \downarrow= f(e)$.

The sets $\{\exists t\ \Phi_e(e)[t] \downarrow= 0\}$ and $\{\exists t\ \Phi_e(e)[t] \downarrow= 1\}$ are c.e. and therefore according to Theorem 3.4, there exist Σ_1 formulas $F_0(e)$ and $F_1(e)$ such that

$$\begin{aligned}\{e \in \mathbb{N} : \mathbb{N} \models F_0(e)\} &= \{e \in \mathbb{N} : \exists t\ \Phi_e(e)[t] \downarrow= 0\}\\ \{e \in \mathbb{N} : \mathbb{N} \models F_1(e)\} &= \{e \in \mathbb{N} : \exists t\ \Phi_e(e)[t] \downarrow= 1\}.\end{aligned}$$

Note that PA also proves $F_0(e) \to \neg F_1(e)$ and $F_1(e) \to \neg F_0(e)$: in other words $e \mapsto \Phi_e(e)$ is a partial function which cannot have both 0 and 1 as a value for the same element.

To compute the value of $f(e)$, we then enumerate using Theorem 3.7 all the formulas demonstrated by T, until we find $F_0(e)$ or $\neg F_0(e)$. By hypothesis, one of these two eventualities necessarily occurs. If $T \vdash F_0(e)$ then we define $f(e) = 1$. Otherwise we define $f(e) = 0$. Let us show that our function has the expected property. If $\exists t\ \Phi_e(e)[t] \downarrow = 0$ then $\mathbb{N} \vDash F_0(e)$ and therefore, according to Lemma 3.8, $T \vdash F_0(e)$. So $f(e) = 1 \neq \Phi_e(e)$. If now $\exists t\ \Phi_e(e)[t] \downarrow = 1$ then $\mathbb{N} \vDash F_1(e)$ and therefore, according to Lemma 3.8, $T \vdash F_1(e)$. We then have $T \nvdash F_0(e)$ and therefore $T \vdash \neg F_0(e)$. So $f(e) = 0 \neq \Phi_e(e)$. Finally if $\forall t\ \Phi_e(e)[t] \uparrow$ the value of f does not matter. ■

Corollary 3.11
Let $T \supseteq \mathrm{PA}$ be a complete and consistent theory. Then, T computes a DNC_2 set.

Proof. From the proof of the previous theorem. ■

Let us dwell for a moment on what the Gödel-Rosser theorem tells us: there exists a Σ_1 formula F which is neither provable nor refutable in PA. According to Lemma 3.8 if a Σ_1 formula is true in $\mathbb{N}$ it is provable in PA. We deduce $\mathbb{N} \nvDash F$ and therefore $\mathbb{N} \vDash \neg F$: that makes $\neg F$ a true formula, in the sense that it is true in $\mathbb{N}$, but that we cannot prove using axioms of PA.

Finally, the theorem tells us that adding axioms is hopeless: as long as we keep the theory computably enumerable, it will remain incomplete. Note that we have shown with Proposition 2.24 that T could quite be extended into a complete and consistent theory, but this will be at the cost of no longer knowing its axioms, and we would then struggle using it to demonstrate anything ...

Let us now move on to Gödel's second theorem, even more surprising than the first, and which put an abrupt halt to Hilbert's program.

Notation

For a computably enumerable theory T in $\mathcal{L}_{PA}$, let $\mathrm{Coh}(T)$ the closed Π_1 formula corresponding to "T is a consistent theory", that is -to say "for any proof p in T, p is not a proof of $\bot$".

The existence of such a formula follows from the correspondence between Σ^0_n/Π^0_n sets and Σ_n/Π_n formulas.

Theorem 3.12 (Gödel's second incompleteness theorem)
Let T be a c.e. consistent theory containing PA. Then, $T \nvdash \mathrm{Coh}(T)$.

PROOF. The first step is to see that the proof of Theorem 3.10 can be formalized, via an appropriate coding, in Peano arithmetic: there exists a Σ_1 formula F such that

$$\mathrm{PA} \vdash \mathrm{Coh}(T) \to (\ulcorner T \nvdash F \urcorner \wedge \ulcorner T \nvdash \neg F \urcorner).$$

Notations $\ulcorner \Psi \urcorner$ indicate the transformation of Ψ into a statement of arithmetic.

Let us assume by the absurdity that $T \vdash \mathrm{Coh}(T)$. Then, by the Modus Ponens rule, we have $T \vdash \ulcorner T \nvdash F \urcorner \wedge \ulcorner T \nvdash \neg F \urcorner$ and therefore $T \vdash \ulcorner T \nvdash \neg F \urcorner$ as well as $T \vdash \ulcorner T \nvdash F \urcorner$.

It should then be noted that the proof of Lemma 3.8 can also be done in Peano arithmetic, that is to say that Peano arithmetic shows that if a fixed Σ_1 formula is true, then it is demonstrable in Peano arithmetic - and therefore in T. This therefore gives formally with the formula F

$$T \vdash F \to \ulcorner T \vdash F \urcorner.$$

We then have by contraposition (using the equivalence between the negation and the encoding of the negation):

$$T \vdash \ulcorner T \nvdash F \urcorner \to \neg F.$$

As $T \vdash \ulcorner T \nvdash F \urcorner$ then by Modus Ponens we get

$$T \vdash \neg F.$$

Finally, it suffices to see that if T proves any formula, then PA — and therefore T — shows that T proves this formula. This is once again an application of the formalization of the proof of Lemma 3.8 in T, the sentence $T \vdash F$ being Σ_1. So we finally have

$$T \vdash \ulcorner T \vdash \neg F \urcorner$$

which contradicts $T \vdash \ulcorner T \nvdash \neg F \urcorner$. ∎

3.3. Consequence of the incompleteness theorems

Let us remember Gödel's completeness theorem: $T \vdash F$ iff any model of T is a model of F. According to the second incompleteness theorem, no consistent and computably enumerable theory T containing arithmetic can demonstrate its own consistency: $T \nvdash \mathrm{Coh}(T)$. We therefore deduce for example that there are models of Peano arithmetic within which the sentence Coh(PA) is false: in these models, there exists in particular a proof of $0 = 1$. We have however shown that PA had models and therefore according to the consistency theorem, that PA could not prove $0 = 1$. The situation which seems paradoxical is resolved with the following consideration: the proof of $0 = 1$ in a model of PA $\cup \{\neg\text{Coh(PA)}\}$ is not a real proof. The sentence $\neg$Coh(PA) when expressed in the language of arithmetic is of the form: "there is an integer which codes for a valid proof of $0 = 1$ in PA". Such a model will therefore contain such an integer, but this integer will not be a true integer - otherwise by unwinding the proof encoded by this integer we would have a true proof of $0 = 1$.

Such a PA model is called *non-standard*. Any PA model contains 0 and 1. By the axioms governing addition, we easily show that any PA model contains of course the standard integers: $0, 1, 2, 3, 4, \ldots$. A non-standard model of PA will contain integers *larger* than all standard integers, and from the point of view of the model, there is nothing to distinguish these integers from others. These integers also called non-standard.

Let us take a non-standard integer a of such a model. Using the axioms of Peano arithmetic we see that the integers $a+1, a+2, a+3, \ldots$ also exist in the model. By the axiom which states that any integer different from 0 has a predecessor we also see that the integers $a-1, a-2, a-3, \ldots$ are in the model. As $a > n$ for all $n \in \mathbb{N}$ then also $a + a > a + n$ for all $n \in \mathbb{N}$. There is therefore also a non-standard integer greater than all $a + n$. By playing with the axioms of PA in this way, we arrive at the following theorem:

Theorem 3.13
For any countable non-standard models of arithmetic, the order $<$ consists of a copy of $\mathbb{N}$, followed by $\mathbb{Q}$ copies of $\mathbb{Z}$.

We therefore know what the order of the elements looks like in a non-standard model. Unfortunately, however, we do not know what the addition or the multiplication looks like:

Theorem 3.14 (Tennenbaum)
Let $\mathcal{M}$ be a countable non-standard model of PA. Let $+, \times : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be functions which represent the addition and multiplication functions of $\mathcal{M}$, once established a bijection between $\mathcal{M}$ and $\mathbb{N}$. Then, neither $+$ nor $\times$ is computable.

We will talk a little more about non-standard models of arithmetic in the part on reverse mathematics, and in particular in Section 23-3.

4. ZFC system

We have given a proof of the consistency of Peano arithmetic, using the consistency theorem (see Corollary 2.21) and providing a model for this theory. According to Gödel's second incompleteness theorem, we necessarily used a theory more powerful than PA to create this model. What theory is it? We can give several answers to this question. The most natural theory to show the consistency of first-order arithmetic is undoubtedly the theory of *second-order* arithmetic, which will be discussed in more detail in Chapter 22.

4.1. Motivation

In second-order arithmetic, we allow ourselves not only to work with integers, but also with sets of integers. The existence of arbitrary sets of integers is questionable, at least more than the existence of the integers themselves: they are infinite objects, which are in uncountable quantity, and if there is a "legitimate" aspect of accepting the existence of computable sets of integers (after all we can write algorithms that produce them, their theoretical existence is therefore doubled by a certain form of practical existence), we can more easily question that of other sets. Some of them are still accessible, in the sense that they have a clear definition, for example the Turing jump. What makes $\emptyset'$ more legitimate than another arbitrary non-computable set is the fact that it is *definable*, and what is more by a fairly simple formula - Σ_1 in particular: $\{e \in \mathbb{N} : \Phi_e(e) \downarrow\}$. In second-order arithmetic, we allow ourselves the existence of all sets definable by an arbitrary arithmetic formula, in particular the Σ^0_n formulas for a certain n: this is called the *axiom of comprehension*. Once we have admitted the existence of a set X, it would be absurd not to accept the existence of computable sets with X as an oracle, and if we accept the axiom of comprehension, it would be just as absurd not to accept the existence of sets definable by a formula of arbitrary arithmetic, which could use X as oracle.

Second-order arithmetic therefore consists of Peano arithmetic, to which we add the axiom of comprehension which makes it possible to validate the existence of definable sets via a formula of arithmetic, possibly using a oracle — an already existing set —. A very large part of mathematics can already be formalized in this way, but not all of mathematics. In particular, nothing in second-order arithmetic allows us to speak of the set of all the subsets of integers, or even of arbitrary subsets of the latter. For that, we need another axiom: given a set X, the *power set* $\mathcal{P}(X)$ of X — that is to say the set of all the subsets of X — is legitimate, it exists and we can use it. The power set of X is not of the same nature as X. One is a set of integers and the other is a set of sets of integers. Set theory makes it possible to treat these two elements in a homogeneous way: every element will be a set, including the integers. Informally the integer 0 will be the empty set, the integer 1 will be the set which contains 0, the integer 2 will be the set which contains 0 and 1, and inductively the set $n+1$ will be the set which contains m for all $m \leqslant n$. We can of course imagine other ways of representing integers by sets, and with practice these do not matter, but we must choose one. This representation is the one that has become standard, under the mpulse of the mathematician John von Neumann. We will discuss this again with the study of ordinals in Chapter 27.

4.2. Zermelo system

Now that we only work with sets, we are no longer in the language of arithmetic. Our only relation symbols will be that of ownership $\in$ and of course the symbol of equality $=$. We need some basic axioms in order to govern the elementary manipulations of sets:

(1) *Axiom of empty set*: the empty set exists.

(2) *Axiom of pairing*: if a and b are sets, then $\{a, b\}$ is a set.

(3) *Axiom of union*: the idea is that if $(a_i)_{i \in I}$ is a collection of sets indexed by a set I, then the set $\bigcup_{i \in I} a_i$ exists. The notion of "indexed set" is not formally defined in set theory, we will say instead that if b is a set whose elements are a_i, then the union of all these a_i exists. To be quite formal, the axiom is: $\forall b\ \exists c\ \forall x\ (x \in c \leftrightarrow \exists a \in b\ x \in a)$.

We also need an axiom which governs the equality between two sets.

(4) *Axiom of extensionality*: two sets are equal iff they have the same elements.

Finally, there is our troublemaking axiom, which pushed us into this set theory allowing, for example, to deal with the power set of $\mathbb{N}$.

(5) *Axiom of power set*: for any set a, the power set of a, denoted $P(a)$ exists. Formally: $\forall a\ \exists b\ \forall c\ (c \in b \leftrightarrow c \subseteq a)$ where $c \subseteq a$ can be written as $\forall x(x \in c \rightarrow x \in a)$.

We can finally add our axiom of comprehension, allowing to build sets from first-order formulas, possibly using other sets as parameters. As there is an infinity of first-order formulas, it is not a question of a single axiom, but of an axiom scheme: one axiom per formula.

(6) *Comprehension scheme*: for a formula $F(y, x_1, \dots, x_n)$ fixed in the language of set theory, for all n-tuples of sets $b_1, \dots, b_n$ and for any set a, the set $\{y \in a : F(y, b_1, \dots, b_n)\}$ exists.

We can easily verify that the preceding axioms imply the existence of all hereditarily finite sets, that is to say finite sets, the elements of which are themselves finite sets, etc., until arriving, by scrolling down the tree describing the membership relations of a set with its elements, to the empty set for any leaf of this tree. Nothing allows us for the moment to speak of the set of all integers. In fact, we need an axiom for this.

(7) *Axiom of infinity*: there exists an infinite set. Formally

$$\exists x \ (\emptyset \in x \land \forall y \in x \quad y \cup \{y\} \in x).$$

The axiom of infinity morally asserts the existence of $\mathbb{N}$ as a set. Via the encoding that we have given of integers, we can check that the set encoding the integer $n+1$ is equal to the set $n \cup \{n\}$, where n is the set encoding the integer n. The axiom of infinity therefore tells us that there exists a set containing all integers. It could possibly contain other elements, but using the other axioms, we define $\mathbb{N}$ as the smallest set x —smallest for inclusion — such that $\emptyset \in x \land \forall y \in x \quad y \cup \{y\} \in x$.

4.3. Replacement axiom and Borel games

The axioms from (1) to (7) form Zermelo's theory Z. They are sufficient to develop a large part of mathematics. Fraenkel and Skolem will introduce a new axiom, both intuitive and formally necessary to develop the theories of ordinals and hierarchies of infinities. It is more precisely a scheme of axioms, which essentially says that the image of any functional formula exists. A formula $F(y, r)$ is *functional* if for all y, the formula $F(y, r_y)$ is satisfied for exactly one set r_y. Formally:

(8) *Replacement scheme*: for a functional formula $F(y, x_1, \dots x_n, z)$ fixed in the language of set theory, for any n-tuple of sets $b_1, \dots, b_n$, for any set a, the set

$$\{z : \exists y \in a \ F(y, b_1, \dots, b_n, z)\}$$

exist.

Set theory becomes strictly more powerful with the replacement scheme, which allows in particular to show the existence of the set $\mathbb{N} \cup P(\mathbb{N}) \cup P(P(\mathbb{N}))\dots$ which it is impossible to build without this axiom.

It is remarkable to note that a joint use of the axiom of power set and of the replacement scheme, is essential to construct certain sets of integers - which will be necessarily extreme complexity in terms of Turing degree. The emblematic example is Martin's determination theorem for Borel games. Let's see what it is: consider a class $\mathcal{B} \subseteq 2^{\mathbb{N}}$ for the moment arbitrary, and consider the following two-player game: Player 1 chooses a bit $x_0 \in \{0,1\}$, then Player 2 in turn chooses a bit $x_1 \in \{0,1\}$, and so on, in step $2n$, Player 1 chooses bit x_{2n} and in step $2n+1$ Player 2 chooses bit x_{2n+1}. At the "end" of the game, we get a set $X = x_0x_1x_2\dots$. Player 1 wins the game if $X \in \mathcal{B}$, otherwise Player 2 wins. The question then is: does one of the two players have a winning strategy? A *strategy* for Player 1 is a function f which takes as parameter a string σ of even size, corresponding to what has been played so far, the last move being the last bit of σ played by Player 2, and who returns the next move $f(\sigma)$. Such a strategy is winning if the set obtained by playing the moves given by the function f is always in $\mathcal{B}$, whatever the moves made by Player 2.

We will see in Chapter 17 that a large variety of classes $\mathcal{B} \subseteq 2^{\mathbb{N}}$ do not require the axiom of power set to be handled: we already have an example with the Π^0_1 or Σ^0_1 classes. We will carry out iterated constructions of increasingly complex classes, which can be encoded by a countable object, in a manner analogous to the encoding of Π^0_1 classes by trees: these will be the so-called *Borel classes*. The mathematician Donald A. Martin, of whom we will speak again in Chapter 12, showed the following remarkable theorem:

Theorem 4.1 (Martin [153])
Let $\mathcal{B}$ be a Borel class. Then, for the game described above with the class $\mathcal{B}$, one of the two players has a winning strategy.

The strategy of a Borel game is a function $f : 2^{<\mathbb{N}} \to \{0,1\}$ and can therefore be represented by a set of integers. Friedman [70] showed that for some Borel classes, of relatively simple complexity, such a strategy could not be constructed without the use of the axiom of power set, and even without arbitrary iteration of the application of this axiom. Thus, to show the existence of certain reals, we must appeal to the axiom of power set used jointly with the replacement scheme. More precisely, it is used to construct $\mathbb{N} \cup P(\mathbb{N}) \cup P(P(\mathbb{N}))\dots$, essential for the definition of our function - the winning strategy for a certain Borel class.

4.4. Axiom of foundation and coding

Another axiom was added by Fraenkel and Skolem, as well as by von Neumann. Unlike the other axioms, the latter is almost useless for the construction of the mathematical universe, but we add it simply because it corresponds to our conception of things: the axiom says in essence that given a set x, if we consider an element $x_1 \in x$, then an element $x_2 \in x_1$, thus continuing inductively with an element $x_{n+1} \in x_n$, we will necessarily eventually obtain the empty set for some $n \in \mathbb{N}$. In other words, there are no sets other than those that we can build inductively via the other axioms starting from the empty set. In particular, there is no set x such that $x \in x$. This may or may not seem obvious to everyone, but in any case reflects the conception generally adopted in the community on the nature of sets.

In order to support our point, let us recall that one of the interests of set theory is the ability to formalize the totality of mathematics in it. This formalization involves the coding of the usual mathematical structures by sets, and this coding is done anyway only with sets which respect the axiom of foundation - whether the latter is adopted or not. So if it is not contradictory to think that there exist other sets which do not respect this axiom, in practice we do not need them, and such objects do not correspond *a priori* to anything tangible.

Let us end by insisting on the fact that if we can code the rest of mathematics by sets, this is on the other hand not a reason for doing so, and we are obviously much more comfortable working with integers, reals or other. What is interesting is the *existence* of such an encoding, and the fact that if a statement is undecidable in set theory, it then becomes undecidable.

4.5. Axiom of choice and cardinality

Zermelo's theory Z plus the replacement axiom and the foundation axiom gives ZF theory, from Zermelo/Fraenkel.

A final axiom, arguably the most famous, is the axiom of choice, which was originally part of Zermelo's theory. The need for this axiom should be easy to understand via the analogy that can be made of it in computability theory. We have for example seen that any non-empty and countable Π^0_1 class contains a computable set. On the other hand it is not possible, given the code of a non-empty Π^0_1 class, to find uniformly an algorithm allowing to compute an element of it. In particular, given a countable sequence $(\mathcal{F}_n)_{n\in\mathbb{N}}$ of non-empty Π^0_1 classes, there does not necessarily exist a computable function allowing to *choose* an element in each of these classes. The key question here is that of uniformity. Obviously, we can create this choice function using $\emptyset'$, but what happens if we consider more complex classes? Can we still build a choice function? The answer is no, without the help of a new axiom.

(9) *Axiom of choice*: given a collection $(A_i)_{i\in I}$ of non-empty sets, there exists a function $f : I \to \bigcup_{i\in I} A_i$ such that $f(i) \in A_i$ for all i.

The axiom of choice appears necessary to develop a complete theory of the cardinality of sets. In particular with the axiom of choice, we can show that for any set A, B, we have $|A| \leqslant |B|$ or $|B| \leqslant |A|$ (we will come back to this with the detailed study of ordinals in Chapter 27). This is no longer true without the axiom of choice, the example par excellence in computability theory being certainly that of Turing degrees. It is easy to construct an injection of $2^{\mathbb{N}}$ in the Turing degrees, it is however impossible to construct an injection of Turing degrees in $2^{\mathbb{N}}$ without the axiom of choice (in particular it is impossible to show the existence of a function $f : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ such that $X \equiv_T Y \leftrightarrow f(X) = f(Y)$).

A first reaction is to simplify your life and use the axiom of choice if necessary. However, this approach is not in the spirit of computability theory, from which set theory is less distant than one might think: axioms other than the axiom of choice allow us to *construct* more and more complex objects starting from existing objects, a bit like the way one constructs more and more complex Turing degrees by iterating the jump. The axiom of choice is fundamentally non-constructive, and from this point of view is not as legitimate as the others. It also leads to theorems which seem paradoxical, the best known example being the *Banach/Tarski paradox*, a construction, which using the axiom of choice, shows how to cut a ball of the space $\mathbb{R}^3$ into a finite number of pieces, and how to reassemble these pieces to obtain two balls strictly identical to the first.

By not accepting the axiom of choice, we are of course leaving this ideal world where the cardinality of any set is comparable, but it is in fact an "artificial" situation of which we have do not really need.

4.6. Independence results

The theory obtained with axioms (1) - (8) is the ZF theory, if we add the axiom of choice to it, we then have the so-called ZFC theory.

4.6.1. Axiom of choice

The question of knowing whether the axiom of choice is provable from the other axioms, or even that of knowing whether it does not risk introducing contradiction, has long remained open. Gödel's completeness theorem allows the following technique: if we assume that ZF is consistent, and therefore has a model, and that using this model we can build a model of ZFC, we will have showed that the consistency of ZF implies that of ZFC, and in particular that ZF cannot prove that the axiom of choice is false (unless of course ZF is inconsistent). This is exactly what Gödel did a few years later [77], via his model called *constructible universe*. It was only later in 1962 with his famous forcing technique, some aspects of which we will see in Chapter 11, that Cohen [40] succeeded in the feat of building a model of ZF in which the axiom of choice is false: the axiom of choice can therefore neither be proved nor refuted in ZF.

4.6.2. Continuum hypothesis

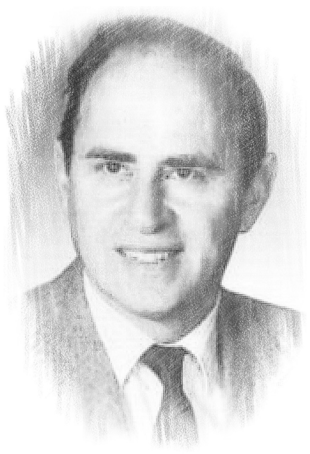
The question that obsessed Cantor throughout his life (and many mathematicians for almost a century) is also independent of the other axioms of set theory (with or without the axiom of choice). Gödel's constructible universe also constitutes a ZFC model in which the continuum hypothesis is verified, i.e., there is no set A for which $|\mathbb{N}| < |A| < |2^{\mathbb{N}}|$. Still with his forcing technique, Cohen built models of ZFC in which we have an arbitrary number of infinities strictly between $|\mathbb{N}|$ and $|2^{\mathbb{N}}|$.

Note here that we can give several versions of the continuum hypothesis. The one originally formulated by Cantor concerned sets of reals: does there exist a set $\mathcal{A} \subseteq \mathbb{R}$ such that $|\mathbb{N}| < |A| < |\mathbb{R}|$? Later the question will be extended to any set, and in particular to ordinals, which we will see formally in Chapter 26. Regardless of the version of the continuum hypothesis, this is a question independent of the other axioms of set theory.

Chapter 10

Cohen forcing

Forcing is a technique invented by the mathematician Paul Cohen in the early 1960s, to show the independence of the continuum hypothesis and the axiom of choice from Zermelo/Fraenkel's theory. This technique has revolutionized set theory, and also plays a preponderant role in computability theory, where it has established itself as one of the two main techniques of set construction, alongside the priority method (see Chapter 13).


Paul Joseph Cohen, 1934–2007

Historically, this technique was designed to extend a model $\mathcal{M}$ of ZF theory by adding a new object G to form a new model $\mathcal{M}[G]$, while allowing the elements of the model $\mathcal{M}$ to control the properties of the extended model $\mathcal{M}[G]$. More generally, the forcing technique makes it possible to create mathematical objects using increasingly precise approximations, while being able to control certain properties of the final object before the construction is finished. A forcing notion is above all the definition of a partial order of approximations $(\mathbb{P}, \leqslant)$, where the relation $d \leqslant c$ means that the approximation d is more precise than c. When a property (or requirement) $\mathcal{R}$ on the final object is already determined by an approximation $c \in \mathbb{P}$, in other words when whatever the sequence of the construction, the final object will satisfy $\mathcal{R}$, we then say that c *forces* $\mathcal{R}$. The heart of the technique of forcing lies in the ability to force complex properties on the final object from simple approximations.

Forcing is often considered a difficult technique to grasp at first glance. Its application to computability theory, which is simpler, makes it possible on the one hand to approach it smoothly, and on the other hand to more easily understand the details of its underlying mechanisms. This simplicity is mainly due to two aspects:

- Like the finite extension method, we will mainly use forcing to create sets of integers, while controlling their computational powers. We will therefore be interested in forcing simple formulas, of the type "$\Phi^G(n)\downarrow$" or "$\Phi^G(n)\uparrow$", where G denotes the final set. It is therefore a question of forcing Σ^0_1 or Π^0_1 formulas. We will see that several constructions, in particular those with the finite extension method, are simplified uses of forcing. We will also see how to extend the forcing to arbitrarily complexity formulas, which presents a first level of additional complexity.
- Contrary to set theory, we will only force properties whose quantifiers relate to integers. The creation of a set of integers by forcing does not modify the nature of the integers, and therefore the scope of the quantifiers. In set theory on the other hand, quantifiers relate to sets, and creating a new set adds a large amount of sets to the model, and therefore changes the scope of quantifiers. It is then necessary to use *names*, to talk about the objects of our extended model inside our base model. We will not need to use names in computability theory, which makes the presentation more accessible.

1. Formulas of second-order arithmetic

The statements that one "forces" in computability theory are always formulas of first-order arithmetic *with free set variables.* These formulas are a special case of second-order arithmetic, which we will talk about in more detail in Chapter 22. A formula of second-order arithmetic has two types of variables: *first-order* variables, representing integers and which will be noted in lowercase, and *second-order* variables, representing sets of integers and which will be denoted in upper case. The language is enrich with the membership symbol $\in$, and the atomic formula "$x \in A$". In second-order arithmetic, quantifications can be on integers, and on sets of integers. For example, the statement "$\forall A\ \forall n\ \exists m\ (m > n \wedge m \in A)$" states that any set of integers is infinite.

First-order arithmetic with free set variables is a restriction of second-order arithmetic, where only the quantifications on the integers are allowed[1]. During the evaluation of a second-order arithmetic formula in a given model, its free set variables are replaced by parameters, i.e., elements of the model considered. Thus the statement $F(G) = \forall n\ \exists m\ (m > n \wedge m \in G)$ is a formula of first-order arithmetic with G as free set variable, and any infinite set $X \subseteq \mathbb{N}$ is a parameter for which $F(X)$ is true.

We have seen in Section 9-3 the Δ^0_0 formulas of arithmetic: those containing only bounded quantifications, that is to say quantifications of the form $\forall x \leqslant y$ and $\exists x \leqslant y$. We can define a hierarchy on arithmetic formulas with free set variables, similar to the hierarchy of Definition 9-3.2.

Definition 1.1.

1. A formula $F(Y_1, \dots, Y_k, x_1, \dots, x_m)$ of second-order arithmetic is Σ^0_n if

$$F(Y_1, \dots, Y_k, x_1, \dots, x_m) = \overbrace{\exists y_1 \forall y_2 \dots Q y_n}^{n \text{ quantifiers}} G(Y_1, \dots, Y_k, x_1, \dots, x_m, y_1, \dots, y_n)$$

   for a Δ^0_0 formula $G(Y_1, \dots, Y_k, x_1, \dots, x_m, y_1, \dots, y_n)$, where Q is $\exists$ if n is odd, and $\forall$ if n is even.

2. A formula $F(Y_1, \dots, Y_k, x_1, \dots, x_m)$ of second-order arithmetic is Π^0_n if

$$F(Y_1, \dots, Y_k, x_1, \dots, x_m) = \overbrace{\forall y_1 \exists y_2 \dots Q y_n}^{n \text{ quantifiers}} G(Y_1, \dots, Y_k, x_1, \dots, x_m, y_1, \dots, y_n)$$

   for a Δ^0_0 formula $G(Y_1, \dots, Y_k, x_1, \dots, x_m, y_1, \dots, y_n)$, where Q is $\forall$ if n is odd, and $\exists$ if n is even. $\diamondsuit$

We have so far implicitly used the formulas of second-order arithmetic, via the use of functionals. Theorem 9-3.4 comes in the following equivalence.

Theorem 1.2
Let $A, Z \in 2^{\mathbb{N}}$. The following statements are equivalent:

(1) $A \subseteq \mathbb{N}$ is Z-c.e.

(2) There exists a Σ^0_1 formula of second-order arithmetic $F(X, n)$ such

[1] We will see only very late in this work, in Part IV the great computational complexity hidden behind second-order quantifications.

that $A = \{n \in \mathbb{N} : \mathbb{N} \models F(Z, n)\}$.

(3) There exists a Turing functional $\Phi(Z, n)$ such that $A = \{n \in \mathbb{N} : \Phi(Z, n)\downarrow\}$.

2. Σ^0_1/Π^0_1 forcing

We will now begin our immersion in the world of forcing by studying a specific forcing notion, namely Cohen forcing. As explained in the introduction, a forcing notion is specified by its partial order of approximations. In our case, it will be the partial order of the strings, equipped with the prefix relation. It is one of the simplest concepts of forcing conceptually, but which already contains the fundamental concepts of forcing.

There are several ways to approach forcing, with different levels of abstraction. It is possible to see it as an elaboration of the finite extension method, seeking to systematize the satisfaction of requirements, and to extract a general construction from it. We will present this approach in Section 2.1.

It is also possible to formulate forcing in a topological framework. Topology makes it possible to define a notion of "negligible" or *meager* class of sets. The construction of a set by forcing then consists in choosing an element "typical", that is to say avoiding a negligible set of undesirable properties. We will see this approach in Section 2.2.

2.1. The finite extension approach

Let us reconsider the finite extension method developed in Section 4-8. The goal is to build a set $G \in 2^{\mathbb{N}}$ satisfying an infinity of requirements $(\mathcal{R}_e)_{e \in \mathbb{N}}$. Each requirement is treated independently, and must therefore be satisfied while leaving enough degrees of freedom in the construction to satisfy the other requirements.

Check-in on requirements

The requirements —which we have seen so far informally— can be seen as formulas in the language of first-order arithmetic with set variables, and in particular with a free second-order variable representing the set G which must fulfill the requirement. The finite extension method consists of building $G \in 2^{\mathbb{N}}$ by specifying increasingly long initial segments, represented as strings $\sigma \in 2^{<\mathbb{N}}$. The Σ^0_1 requirements are those corresponding to Σ^0_1 formulas, or in an equivalent way those which one can always put in the form "$\Phi_e(G, 0)\downarrow$" for a functional Φ_e. The Π^0_1 requirements are those corresponding to Π^0_1 formulas or in an equivalent way those which one can always put in the form "$\Phi_e(G, 0)\uparrow$" for a functional Φ_e.

Satisfying a requirement. The general procedure for satisfying a requirement $\mathcal{R}_e$ is as follows: given a string $\sigma \in 2^{<\mathbb{N}}$ representing an initial segment of G already specified to satisfy previous requirements, we must find a string $\tau \succeq \sigma$ such that the requirement $\mathcal{R}_e$ is satisfied. The final set G then not being known, the requirement must be satisfied whatever the rest of the construction, in other words it must be satisfied for all $G \in [\tau] = \{X \in 2^{\mathbb{N}} : \tau \prec X\}$.

Partial order of strings. Let's take some height, and consider the finite extension method from a more abstract point of view. We have a partial order $(2^{<\mathbb{N}}, \preceq)$, which corresponds to the set $2^{<\mathbb{N}}$ of finite strings, equipped with the prefix relation. We also have an interpretation function $[\cdot] : 2^{<\mathbb{N}} \to \mathcal{P}(2^{\mathbb{N}})$ defined by $[\sigma] = \{X \in 2^{\mathbb{N}} : \sigma \prec X\}$. Intuitively, the elements of $2^{<\mathbb{N}}$ represent approximations of the set G under construction. Given an approximation σ, $[\sigma]$ is the class of sets that we could potentially obtain at the end of the construction. The further we advance in the construction, the more the approximation is refined and the class of candidate sets is restricted. Thus, we have the following property: if $\sigma \preceq \tau$, then $[\tau] \subseteq [\sigma]$. Let us see a first definition of the forcing relation.

Definition 2.1. Let $\mathcal{R}$ be a Σ^0_1 or Π^0_1 requirement. A string σ *forces* $\mathcal{R}$, denoted by $\sigma \Vdash^* \mathcal{R}$, if the requirement is satisfied for all $G \in [\sigma]$. ♢

Density. We can abstract ourselves from the notion of requirement by representing a requirement $\mathcal{R}_e$ as the set $P_e \subseteq 2^{<\mathbb{N}}$ of the strings which force it. Note that if σ forces $\mathcal{R}_e$, then every $\tau \succeq \sigma$ also forces $\mathcal{R}_e$ because $[\tau] \subseteq [\sigma]$. The set P_e is therefore closed under suffix. The procedure for satisfying the requirement $\mathcal{R}_e$ consists in showing that for any string $\sigma \in 2^{<\mathbb{N}}$, there exists an extension $\tau \succeq \sigma$ such that $\tau \in P_e$. If this is the case, we will say that the set P_e is dense:

Definition 2.2. A set $W \subseteq 2^{<\mathbb{N}}$ is *dense* if for all $\sigma \in 2^{<\mathbb{N}}$, there exists an extension $\tau \succeq \sigma$ such that $\tau \in W$. ♢

Intuitively, a set $W \subseteq 2^{<\mathbb{N}}$ is dense if whatever the current construction $\sigma_0 \preceq \sigma_1 \preceq \cdots \preceq \sigma_n$ using the finite extension method, it is never too late to find an extension $\sigma_{n+1} \succeq \sigma_n$ in W. It follows that if we have a countable set of requirements represented by dense sets $(P_n)_{n\in\mathbb{N}}$, since these sets are dense, there exists an infinite sequence $\sigma_0 \preceq \sigma_1 \preceq \sigma_2 \preceq \dots$ such that for all n, there exists an integer m for which $\sigma_m \in P_n$. In particular, let $\{G\} = \bigcap_n [\sigma_n]$, then G will have initial segments in each set P_e.

Remark

If the sets $(P_n)_{n\in\mathbb{N}}$ are uniformly c.e, then the construction of the infinite sequence $\sigma_0 \preceq \sigma_1 \preceq \sigma_2 \preceq \dots$ can be done in a computable way, and the resulting set G is also computable.

Genericity. We can formalize the construction of the finite extension method into a trivial theorem in view of the previous intuitions.

Definition 2.3. We say that a set $G \in 2^{\mathbb{N}}$ *meets* a set $W \subseteq 2^{<\mathbb{N}}$ if $G\upharpoonright_n \in W$ for a some $n \in \mathbb{N}$. Let $\vec{D} = (D_n)_{n\in\mathbb{N}}$ be a sequence of sets of strings. A set $G \in 2^{\mathbb{N}}$ is $\vec{D}$*-generic* if it meets every D_n. ♢

Theorem 2.4
Let $\vec{D} = (D_n)_{n\in\mathbb{N}}$ be a countable sequence of dense sets of strings and let $\sigma \in 2^{<\mathbb{N}}$. There is a $\vec{D}$-generic set which extends σ.

Proof. Let $\sigma_0 \prec \sigma_1 \prec \sigma_2 \prec \dots$ be the strictly increasing infinite sequence of strings defined inductively as follows. Initially, $\sigma_0 = \sigma$. If σ_n is defined, σ_{n+1} is a strict extension of σ_n in D_n. Such an extension exists by density of D_n. Let $\{G\} = \bigcap_n [\sigma_n]$. Then, G is $\vec{D}$-generic and extends σ. ■

We will see the topological counterpart of the previous theorem with Lemma 2.14. Theorem 2.4 says, among other things, that if $(D_n)_{n\in\mathbb{N}}$ is a sequence of dense sets corresponding to the requirements $(\mathcal{R}_n)_{n\in\mathbb{N}}$, then there exist $\vec{D}$-generic sets, which therefore all satisfy the requirements simultaneously.

There is an uncountable amount of dense sets of strings, and a set $G \in 2^{\mathbb{N}}$ cannot be generic for all of these sets simultaneously, simply because the set $\{\sigma \in 2^{<\mathbb{N}} : \sigma \not\prec G\}$ is a dense set that G does not meet. The notion of genericity is therefore dependent on a countable collection $\vec{D}$ of dense sets of strings.

Sufficiently generic

It is common to state results of the form "any *sufficiently generic* set satisfies such and such property". This means that there exists a countable sequence $\vec{D} = (D_n)_{n\in\mathbb{N}}$ of dense sets of strings such that any $\vec{D}$-generic set satisfies the property. Note that given any other countable sequence of dense sets of strings $\vec{E} = (E_n)_{n\in\mathbb{N}}$, the sequence $\{\vec{D}, \vec{E}\}$ is again countable, and we can therefore construct a $\{\vec{D}, \vec{E}\}$-generic set. This is what justifies the name of *sufficiently generic*.

We will reformulate the proof of Proposition 4-8.2 in terms of density and genericity. For that, we need to extend the forcing relation to Σ^0_2 requirements, which does not present any difficulty: a string σ *forces* such a requirement if the latter is satisfied for all $G \in [\sigma]$. We will see in Section 4 that this idea no longer works for Π^0_2 requirements.

Proposition (4-8.2). For any non-computable set A, there exists a set B such that $B \not\leq_T A$ and $A \not\leq_T B$. ⋆

Proof. Let A be a non-computable set. We want to build a set B satisfying the requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$:

$$\mathcal{R}_e : \exists x \Phi^A_e(x)\uparrow \vee \exists x \Phi^A_e(x)\downarrow \neq B(x) \qquad \mathcal{S}_e : \exists x \Phi^B_e(x)\uparrow \vee \exists x \Phi^B_e(x)\downarrow \neq A(x).$$

Let $R_e \subseteq 2^{<\mathbb{N}}$ and $S_e \subseteq 2^{<\mathbb{N}}$ be the set of strings forcing respectively $\mathcal{R}_e$ and $\mathcal{S}_e$.

Density of the set R_e. Let $\sigma \in 2^{<\mathbb{N}}$ and let $x = |\sigma|$. Two cases arise:

- Case 1: $\Phi^A_e(x)\downarrow = i$ for an $i \in \{0,1\}$. It is then sufficient to define τ as the unique string of length $|\sigma|+1$ extending σ such that $\tau(x) = 1-i$. For all $X \in [\tau]$, $X(x) = 1 - i \neq \Phi^A_e(x)$, so $\tau \in R_e$.
- Case 2: $\Phi^A_e(x)\uparrow$. In this case, $R_e = 2^{<\mathbb{N}}$ and $\sigma \in R_e$.

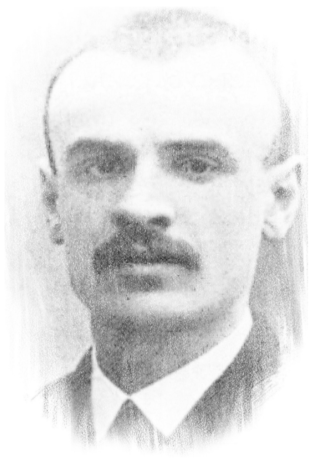
Density of the set S_e. Let $\sigma \in 2^{<\mathbb{N}}$. Three cases arise:

- Case 1: there exists an input x and a set $X \succeq \sigma$ such that $\Phi^X_e(x)\downarrow \neq A(x)$. In this case, by the use property, there exists a finite string $\tau \succeq \sigma$ such that $\Phi^X_e(x)\downarrow \neq A(x)$ for all $X \in [\tau]$. The string τ therefore forces the requirement $\mathcal{S}_e$, hence $\tau \in S_e$.
- Case 2: there exists an input x such that for all the sets $X \succeq \sigma$, $\Phi^X_e(x)\uparrow$. In this case, the string σ already forces the requirement $\mathcal{S}_e$ ensuring that $\Phi^B_e(x)\uparrow$. So $\sigma \in S_e$.
- Case 3: neither of the two previous cases appears. We then showed in the initial proof of Proposition 4-8.2 that this case could not happen, because the set A would be computable, contrary to our hypothesis.

Let B be a $(R_e, S_e)_{e\in\mathbb{N}}$-generic set. Such a set exists by Theorem 2.4. In particular, B satisfies all the requirements $\mathcal{R}_e$ and $\mathcal{S}_e$ simultaneously, so $B \not\leq_T A$ and $A \not\leq_T B$. This concludes the proof of Proposition 4-8.2. ■

2.2. The topological approach

The genesis of the finite extension method, of genericity, and more generally of forcing, can be found in the work of René Baire, at the beginning of the 20th century. One of Baire's motives at the time was to understand a little better certain "bizarre" functions, but which arise naturally in analysis. If we go back a bit, in the first half of the 19th century, the awareness emerges, notably through the work of Cauchy and Bolzano, that a sequence of continuous functions $(f_n : \mathbb{R} \to \mathbb{R})_{n\in\mathbb{N}}$ which pointwise converges does not necessarily have a continuous function as its limit, and for good reason: once the notions of computability have been correctly extended to work in $\mathbb{R}$, the limits of *effectively continuous* functions — that is, computable — are exactly the Δ^0_2 functions, i.e., those for which $f(r)$ is uniformly computable in r', the jump of r. This is a simple application of Schoenfield's lemma.


René Baire, 1874–1932

A few decades later, Baire looks at this phenomenon, and tries to better understand these functions which are the limits of continuous functions, and whose irregularities make them difficult to manipulate and even to grasp with clarity. He will present his results in the Peccot course "Lessons on discontinuous functions" where he develops his famous mathematical tools of *Baire categories* to show in particular that a limit function of continuous functions, if it is no more necessarily continuous, will be continuous despite everything on a "large set of points". According to modern terminology, the points of discontinuity of such a function will be a class *meager* or even of *category 1*. Theorem 3.20 to come can be seen as an actual version of this result.

We have defined in Section 8-2 the notion of open class of Cantor space, as being a class of the form $\bigcup_{\sigma\in U}[\sigma]$ for any set $U \subseteq 2^{<\mathbb{N}}$. The density of a set of strings results in a notion of density of the corresponding open class in Cantor space.

Definition 2.7. A class $\mathcal{B} \subseteq 2^{\mathbb{N}}$ is said to be *dense* if it intersects any cylinder $[\sigma]$, ie $[\sigma] \cap \mathcal{B} \neq \emptyset$ for all $\sigma \in 2^{<\mathbb{N}}$. $\diamondsuit$

The following exercise links the notion of density on the open classes of Cantor space, and on the partial order of binary strings.

Exercise 2.8. Given $U \subseteq 2^{<\mathbb{N}}$ we denote by $U^{\prec}$ the suffix closure of U, that is $U^{\prec} = \{\tau \in 2^{<\mathbb{N}} : \exists \sigma \in U\ \tau \succeq \sigma\}$.

Let $U \subseteq 2^{<\mathbb{N}}$. Show that $U^{\prec}$ is dense in $2^{<\mathbb{N}}$ iff the open class $\bigcup_{\sigma \in U}[\sigma]$ is dense in Cantor space. $\diamond$

Our goal is to define a notion of "negligible" or *meager* class, to give a topological definition of forcing. We start from the intuition that a cylinder $[\sigma]$ is not thin. The notion of negligible class should be closed under subclass. Thus, if a class contains a non-empty open class of Cantor space, it is not negligible. To formalize this, we introduce the notion of interior.

Definition 2.9. The *interior* $\operatorname{int}(\mathcal{F})$ of a class $\mathcal{F} \subseteq 2^{\mathbb{N}}$ is the largest open class included in $\mathcal{F}$, that is, the union of all cylinders $[\sigma]$ such that $[\sigma] \subseteq \mathcal{F}$. $\diamondsuit$

A negligible class must therefore in particular have an empty interior.

Example 2.10. The closed class $\{X \in 2^{\mathbb{N}} : \forall n\ X(2n) = 0\}$ has an empty interior: it is indeed clear that it does not contain any cylinder $[\sigma]$ because there are always $X \in [\sigma]$ such that $X(2n) = 1$ for n sufficiently large. Its complement $\{X \in 2^{\mathbb{N}} : \exists n\ X(2n) \neq 0\}$ is therefore a dense open class, described by the union of the cylinders $[\sigma 1]$ for any string σ of even size.

We now have the elements in hand to define the notion of meager class.

Definition 2.11. A class $\mathcal{B} \subseteq 2^{\mathbb{N}}$ is said to be *meager* if $\mathcal{B}$ is included in a countable union of closed classes with empty interior. The complement of a meager class is called *co-meager*. $\diamondsuit$

Note that by passing to the complement, a class is co-meager if it contains a countable intersection of dense open classes. We leave to the reader the care to show in the two following exercises that for a sequence $\vec{W} = (W_n)_{n\in\mathbb{N}}$ of dense sets of strings, the class of $\vec{W}$-generic sets is a co-meager class.

Exercise 2.12. Let $\vec{W} = (W_n)_{n\in\mathbb{N}}$ be a sequence of sets of strings such that each $W_n^{\prec}$ is dense (using the notation of Exercise 2.8).

Show that the class $\bigcap_n [W_n]$ is co-meager, where $[W_n] = \bigcup_{\sigma \in W_n}[\sigma]$. $\diamond$

Exercise 2.13. Let $\vec{W} = (W_n)_{n\in\mathbb{N}}$ be a sequence of dense sets of strings. Show that the class $\bigcap_n [W_n]$ contains exactly the $\vec{W}$-generic sets. $\diamond$

The two previous exercises therefore establish a link between the approach of forcing by the finite extension method and the topological approach:

if $(\mathcal{R}_n)_{n\in\mathbb{N}}$ is a sequence of requirements such that the corresponding sets of strings $(W_n)_{n\in\mathbb{N}}$ are dense, then the class of $\vec{W}$-generic — sets which satisfy all requirements simultaneously — is co-meager.

Digression

Historically Baire calls *classes of category 1* the meager classes, and *classes of category 2* those that are not. This will give the name of "Baire category theory" to the study of these notions. Note that a class is not necessarily meager or co-meager. In particular, category 2 classes are not necessarily co-meager. As far as we are concerned, it is really the notions of meager and co-meager that interest us, and it is therefore this vocabulary that we will use.

Fact

From Definition 2.11, it is clear that a countable union of meager classes is meager, and that a countable intersection of co-meager classes is co-meager.

One intuition that we will support in future developments is that meager classes are "small" and co-meager classes are "large". The attribution of these adjectives should not be taken as absolute. There are other ways of judging class size, which do not coincide with the fact that the meager classes are small and the co-meager ones are large. One can for example find meager classes of measure 1 and co-meager classes of measure 0 (see Part II).

What should be understood rather, is that the co-meager classes are large enough to always be stable under countable intersection, and at the same time always dense, and even in a strong sense: if $\mathcal{B}$ is a co-meager class then $\mathcal{B} \cap [\sigma]$ is uncountable for any cylinder $[\sigma]$. This is in fact a reinforcement of Theorem 2.4. To see it, let us remember the following fact demonstrated with Proposition 8-2.3 and Proposition 8-3.2:

Fact

The intersection of a finite number of open classes is open. Therefore we can always assume that a countable open intersection $\bigcap_n \mathcal{U}_n$ is decreasing. By passing to the complement, one can always assume that a countable union of closed classes is increasing.

Recall that a class $\mathcal{F} \subseteq 2^{\mathbb{N}}$ is *perfect* if it is the image of a continuous injection from $2^{\mathbb{N}}$ to $2^{\mathbb{N}}$ (see Section 8-2.4), that is, $\mathcal{F}$ is of the form $[T]$

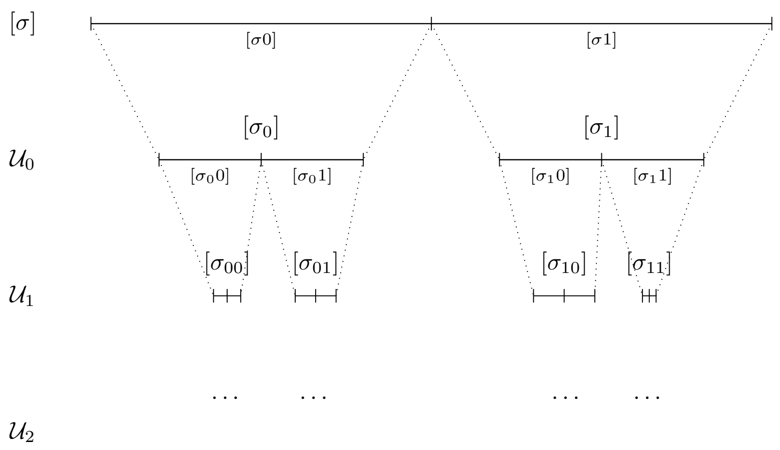


Figure 2.15: Illustration of the construction of a perfect sub-class of points in $[\sigma] \cap \bigcap_n \mathcal{U}_n$. As each open class $\mathcal{U}_n$ is dense, one can find for every string $\sigma_\tau i$ an extension $\sigma_{\tau i} \succeq \sigma_\tau i$ such that $[\sigma_{\tau i}] \subseteq \mathcal{U}_n$.

for an f-tree $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ (see Section 7-5). The following lemma shows that a co-meager class has the power of the continuum.

Lemma 2.14. A co-meager class of Cantor space contains a perfect sub-class of points in each cylinder $[\sigma]$. ⋆

Proof. Let $\mathcal{B}$ be a co-meager class. Let $\bigcap_n \mathcal{U}_n \subseteq \mathcal{B}$ be an intersection of dense open classes. We can assume without loss of generality $\mathcal{U}_{n+1} \subseteq \mathcal{U}_n$. The reader can use Figure 2.2 for a graphical representation of the following construction.

Let $\sigma \in 2^{<\mathbb{N}}$. Since $\mathcal{U}_0$ is dense, $\mathcal{U}_0 \cap [\sigma 0]$ is non-empty and so there is a string $\sigma_0 \succ \sigma 0$ such that $[\sigma_0] \subseteq \mathcal{U}_0$. Likewise there is a string $\sigma_1 \succ \sigma 1$ such that $[\sigma_1] \subseteq \mathcal{U}_0$. Suppose that for any string τ of size $n+1$ we have defined strings $\sigma_\tau \succeq \sigma$ that are pairwise incomparable and such that $[\sigma_\tau] \subseteq \mathcal{U}_n$. For each of these strings τ and for each $i \in \{0,1\}$ we define $\sigma_{\tau i}$ as being a string which extends $\sigma_\tau i$ and such that $[\sigma_{\tau i}] \subseteq \mathcal{U}_{n+1}$, which is possible because $\mathcal{U}_{n+1}$ is dense.

It is clear that for any set X the class $\bigcap_{\tau \prec X} [\sigma_\tau] \subseteq \bigcap_n \mathcal{U}_n$ contains a single element $Y_X \in [\sigma] \cap \bigcap_n \mathcal{U}_n$. We easily verify that the function $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ defined by $T(\tau) = \sigma_\tau$ is an f-tree whose paths are the elements $Y_X = T(\sigma_0) \prec T(\sigma_1) \prec \ldots$ for $X = \sigma_0 \prec \sigma_1 \prec \ldots$.

So $\mathcal{B}$ contains a perfect subclass of points in the cylinder $[\sigma]$. ■

Note in particular as a consequence of the previous lemma that a countable union of closed classes with empty interior has empty interior: if such a union of closed classes contained a cylinder $[\sigma]$, its complement — a co-meager class — could not contain any point in $[\sigma]$, which would be a contradiction.

We now see the topological equivalent of Definition 2.1 of forcing for the Σ^0_1 and Π^0_1 requirements:

Definition 2.16. Let $\mathcal{R}$ be a Σ^0_1 or Π^0_1 requirement. Let $\mathcal{B}_\mathcal{R}$ be the class of elements which satisfy $\mathcal{R}$. Then, σ forces $\mathcal{R}$ iff $[\sigma] \subseteq \mathcal{B}_\mathcal{R}$. Note that if $\mathcal{R}$ is Σ^0_1 then $\mathcal{B}_\mathcal{R}$ is an effectively open class and if $\mathcal{R}$ is Π^0_1 then $\mathcal{B}_\mathcal{R}$ is an effectively closed class. $\diamondsuit$

Note for example that if the class of elements satisfying a Π^0_1 requirement has empty interior, then no string forces this requirement: the open class of elements not satisfying the requirement is a dense, and will contain any sufficiently generic element. This brings us to the study of the sets which are in "sufficiently dense open classes": the 1-generic and the weakly 1-generic, which we see now.

3. Effective genericity

Jockusch was undoubtedly one of the first to understand the usefulness of Cohen's ideas in computability theory, and he initiated the study of an effective version of Cohen's concepts, with the notions of 1-genericity and weak 1-genericity, resulting from the application of forcing to all Σ^0_1 and Π^0_1 requirements.

Genericity can be seen as both a notion of strength and weakness. A sufficiently generic set will, for example, always have hyperimmune degree. On the other hand, we will see that sufficiently generic sets cannot compute the halting problem, nor even a DNC function. Generally speaking, the finite extension method satisfies strength and weakness properties in the same way: by proving the density of some well-chosen sets.

3.1. Weakly 1-generic sets

We begin by presenting the weakly 1-generic sets, introduced by Kurtz during his doctoral thesis, carried out under the supervision of Jockusch.

Definition 3.1 (Kurtz [130]). A set $G \in 2^\mathbb{N}$ is *weakly 1-generic* if it is generic for all dense c.e. sets. In other words, G is weakly 1-generic if G belongs to all the dense Σ^0_1 classes of Cantor space. $\diamondsuit$

Note that there is only a countable quantity of dense Σ^0_1 classes. The notion of weakly 1-generic set turns out not to be restrictive enough to hold properties normally inherent to generics, but in terms of Turing degree, the notion is of some interest, in particular via the following characterization.

Theorem 3.2 (Kurtz [130])
Let $G \subseteq \mathbb{N}$. The following statements are equivalent:

(1) G is of hyperimmune degree.

(2) G computes a function which is often infinitely equal to any computable function.

(3) G computes a weakly 1-generic set.

Proof. Let us first show (1) $\rightarrow$ (3), the most difficult implication. Let $f \leqslant_T G$ be a function which is not bounded by any computable function. We assume without loss of generality that f is increasing. Note that for any computable function g, there is an infinity of values n such that $f(n) > g(n)$. We compute from f a set $G \in 2^{\mathbb{N}}$ which belongs to any dense Σ^0_1 class. Let $(W_e)_{e\in\mathbb{N}}$ be an enumeration of the Σ^0_1 subsets of $2^{<\mathbb{N}}$. We construct G by successive approximation $\sigma_0 \preceq \sigma_1 \preceq \sigma_2 \preceq \dots$.

We first describe a recursive procedure to be performed each time we want to concatenate a string τ to a string σ that we have so far computed. This procedure, which we will name R, takes a third parameter: an integer e which corresponds to the smallest integer such that $\sigma\tau$ is enumerated in W_e at the computation step $f(|\sigma|)$. We will note $R(\sigma, \tau, e)$ for the result of the call to this procedure. Finally, note that some integers are marked as "satisfied" when the procedure is called: these are the integers e such that σ extends a string of W_e at the current computation step.

The $R(\sigma, \tau, e)$ procedure does the following: for each prefix $\tau' \preceq \tau$ in order, it searches for the smallest integer $e' < e$ that is not satisfied and such that a string of the form $\sigma\tau'\rho$ is listed in $W_{e'}[f(|\sigma\tau'|)]$. If such an integer is found then the procedure returns the result of the recursive call to $R(\sigma\tau', \rho, e')$. Otherwise it returns $\sigma\tau$. Note that reducing the value of the last parameter in recursive calls causes the procedure to stop necessarily.

In step 0 we define $\sigma_0 = \epsilon$. Suppose σ_t defined in step t. In step $t+1$, we look for the smallest unsatisfied integer $e \leqslant t+1$ such that a string of the form $\sigma_t\tau$ is listed in $W_e[f(|\sigma_t|)]$. If we find such an integer e we define σ_{t+1} as being $R(\sigma_t, \tau, e)$. Otherwise σ_{t+1} to be $\sigma 0$. This concludes the construction.

Note that if W_e describes a dense open class, then the function f_e which to n associates the smallest computation time t such that all the strings of size n have an extension in $W_e[t]$, is a total computable function. We have in particular $f_e(n) < f(n)$ for an infinity of values n. Suppose that W_e describes a dense open class, that e is not satisfied at time t, and that all $e' < e$ which are satisfied at some point in the construction are satisfied at time t. Let n be the smallest integer greater than or equal to $|\sigma_t|$ such that $f(n) > f_e(n)$. Let $s \geqslant t$ be the smallest integer such that $|\sigma_s| \leqslant n < |\sigma_{s+1}|$. If $|\sigma_s| = n$ then by minimality of e, the algorithm defines $\sigma_{s+1} = \sigma_s\tau$ with $\sigma_s\tau \in W_e[f(n)]$. If not then by construction when defining $\sigma_{s+1} = \sigma_s\tau$ for a certain string τ, the algorithm checks for any prefix $\tau' \preceq \tau$, that we do not have an extension of $\sigma_s\tau'$ listed in $W_e[f(|\sigma_s\tau'|)]$, and in particular for the prefix τ' such that $|\sigma_s\tau'| = n$. If this is the case, the algorithm is restarted on this extension. As it is indeed the case by hypothesis, and by minimality of e, we will in fact have $\sigma_s\tau \in W_e[f(n)]$ for $\sigma_{s+1} = \sigma_s\tau$. We conclude that the set $G = \sigma_0 \prec \sigma_1 \prec \sigma_2 \prec \dots$ belongs to all the dense open classes.

Let us show (3) $\rightarrow$ (2). Let G be a weakly 1-generic set. Note that if f is a total computable function then the set $W_f = \{\sigma 0^{f(|\sigma|)}1 : \sigma \in 2^{<\mathbb{N}}\}$ describes a dense Σ^0_1 class. We compute from G the function g which to n associates the maximum number $g(n)$ of 0 such that $G\upharpoonright_n 0^{g(n)} \prec G$. Since for any total computable function f the set G belongs to W_f, it is clear that g is equal at least once to any computable function. If g were equal only a finite number of times to a given computable function, one could modify a finite number of values of this function to have a computable function which is never equal to g. So g is infinitely often equal to any computable function.

Let us show (2) $\rightarrow$ (1). Let f be a function infinitely equal to any computable function. Then, $f + 1$ is infinitely often above any computable function. ■

We can relativize the notion of weak 1-genericity to any oracle:

Definition 3.3 (Kurtz [130]). Let $A \in 2^{\mathbb{N}}$. A set G is *weakly* 1*-generic relative to* A if G meets W for any $\Sigma^0_1(A)$ dense set of strings W. ♢

The hierarchy of Turing jumps allows to define a hierarchy of genericity as follows:

Definition 3.4 (Kurtz [130]). A set $G \in 2^{\mathbb{N}}$ is *weakly* n*-generic* if it is weakly 1-generic relative to $\emptyset^{(n-1)}$, that is to say if it meets all the $\Sigma^0_1(\emptyset^{(n-1)})$ dense sets of strings, or equivalently all Σ^0_n dense sets of strings. ♢

Some implications of Theorem 3.2 generalize to any oracle A:

Exercise 3.5. Show that any set G weakly 1-generic relative to A computes a function $g : \mathbb{N} \to \mathbb{N}$ which is often infinitely equal to any A-computable function. ⋄

Exercise 3.6. Let $n \geqslant 1$ and $G \in 2^{\mathbb{N}}$. Show that if G computes a function which is infinitely often equal to all A-computable functions, then G computes an A-hyperimmune function. ⋄

The direction (1) $\to$ (3) of Theorem 3.2 does not hold in general. In the Turing jump hierarchy, this only works at the first level.

Proposition 3.7 (Andrews, Gerdes and Miller [7]). Any hyperimmune function relative to $\emptyset'$ computes a weakly 2-generic set. ⋆

The idea to show the previous proposition is to use the fact that $\emptyset'$-computable objects are limit-computable. On the other hand, from $n \geqslant 3$, weakly n-generic sets can no longer be constructed simply using functions escaping collections of functions, in the following sense.

Definition 3.8. Let $\mathcal{F}$ be a collection of functions. A function $f : \mathbb{N} \to \mathbb{N}$ is $\mathcal{F}$-*escaping* if for all $g \in \mathcal{F}$, there exists $n \in \mathbb{N}$ such that $f(n) > g(n)$.♢

The following theorem expresses a deep structural difference between escaping functions and generic degrees, in the sense that whatever an oracle A, there exists an A-hyperimmune function which computes no weakly 3-generic set.

Theorem 3.9 (Andrews, Gerdes and Miller [7])
For any countable collection of $\mathcal{F}$ functions, there exists a $\mathcal{F}$-escaping function which is not of weakly 3-generic degree.

Proof. We will use a variant of the notion of f-tree defined in Section 7-5. We are going to define a total Δ^0_3 function $T : \mathbb{N}^{<\mathbb{N}} \to \mathbb{N}^{<\mathbb{N}}$ such that:

(1) $\operatorname{dom} T$ (the domain of T) is a prefix-closed set;

(2) for all $\sigma, \tau \in \operatorname{dom} T$, $\sigma \preceq \tau$ if and only if $T(\sigma) \preceq T(\tau)$;

(3) for all $\sigma \in \mathbb{N}^{<\mathbb{N}}$ and $n \in \mathbb{N}$, there exists an $m \geqslant n$ such that $T_s(\sigma n)$ extends $T_s(\sigma)m$.

The function T extends as a function from $\mathbb{N}^{\mathbb{N}}$ to $\mathbb{N}^{\mathbb{N}}$, defining for all $X \in \mathbb{N}^{\mathbb{N}}$, $T(X)$ as the only element of the class $\bigcap_{\sigma \prec X}[T(\sigma)]$. A *path* of T is a sequence $P \in \mathbb{N}^{\mathbb{N}}$ of which an infinity of initial segments belong to $\operatorname{Im} T$. In other words, a path is a sequence $P \in 2^{\mathbb{N}}$ of the form $P = T(X)$ for an $X \in \mathbb{N}^{\mathbb{N}}$. We denote by $[T]$ the set of paths of T. By (3), for any countable collection of functions $\mathcal{F}$, there exists a path $f \in [T]$ which is $\mathcal{F}$-escaping. We are going to construct T such that for any path $f \in [T]$, f is not of weakly 3-generic degree.

The reader can use Figure 3.10 to understand the following construction. Let $(\sigma_s)_{s\in\mathbb{N}}$ be a computable enumeration of $\mathbb{N}^{<\mathbb{N}}$ which only shows each string after enumerating its prefixes. In particular, $\sigma_0 = \epsilon$. At the start of step s, we will have already defined T over $\sigma_0, \dots, \sigma_s$. We will also assume defined for each $t \leqslant s$, an infinite c.e. reservoir $V_{t,s} \subseteq \mathbb{N}^{<\mathbb{N}}$ of strings extending $T(\sigma_s)$. We'll make sure that for all $n \in \mathbb{N}$, $T(\sigma_s n)$ extends a string of $V_{t,s}$. Simultaneously, we will create for each e a dense $\Sigma^0_1(\emptyset'')$ set $U_e \subseteq 2^{<\mathbb{N}}$ such that if Φ_e^P is total for a path $P \in [T]$, then P does not meet U_e. Thus, whatever the path $P \in [T]$, Φ_e^P will not be a weakly 3-generic set.

Initially, $f(\epsilon) = \epsilon$ and $V_{0,0} = \mathbb{N}$. At step $s = \langle e, i\rangle$, we will make sure that any string of length i has an extension in U_e. The set $\{\sigma_t : t \leqslant s\}$ forms a finite tree, and for each leaf τ of this tree, $\Phi_e^{T(\tau)}$ can have different values. The set U_e must therefore add extensions to all strings of length i, while ensuring that it will avoid $\Phi_e^{T(\sigma_t)}$ for all $t \leqslant s$. We therefore fix a sufficiently large length, $k = i + s + 1$, so that if we construct a set of "forbidden" binary strings $\rho_0, \dots, \rho_s$ each of length k, any binary string of length i admits an extension of length k avoiding the forbidden strings.

For each $t \leqslant s$, we ask $\emptyset''$ if there exists a string ρ_s of size k such that there is an infinity of binary strings $\mu \in V_{t,s}$ having an extension $\mu' \succeq \mu$ for which $\Phi_e^{\mu'} \upharpoonright_k = \rho_s$. If this is the case, we define $V_{t,s+1}$ as the set of these μ'. Note that $V_{t,s+1}$ is then still computably enumerable. Otherwise for any string ρ of size k, only a finite number of strings of $V_{t,s}$ have an extension which will be sent to an extension of ρ via Φ_e. In particular, one can remove a finite number of strings from $V_{t,s}$ so that no extension of the remaining strings will ever be sent to a string larger than k. We then take an arbitrary string ρ_s, and define $V_{t,s+1}$ as the restriction of $V_{t,s}$ to strings μ which have no extension μ' sent to a string larger in size than k via Φ. Since we remove a finite number of strings, $V_{s,t+1}$ is still c.e.

So we end up with a set of binary strings $\rho_0, \dots, \rho_s$ each of length k, such that for any path $P \in [T]$, if Φ_e^P is total, then $\Phi_e^P \upharpoonright_k = \rho_t$ for one $t \leqslant s$. We list in U_e all the strings of length k other than $\rho_0, \dots, \rho_s$. Excluding these strings, we make sure that for any path $P \in [T]$, if Φ_e^P is total, then it will not meet U_e. The length k being sufficiently large, any string of length i has an extension in U_e.

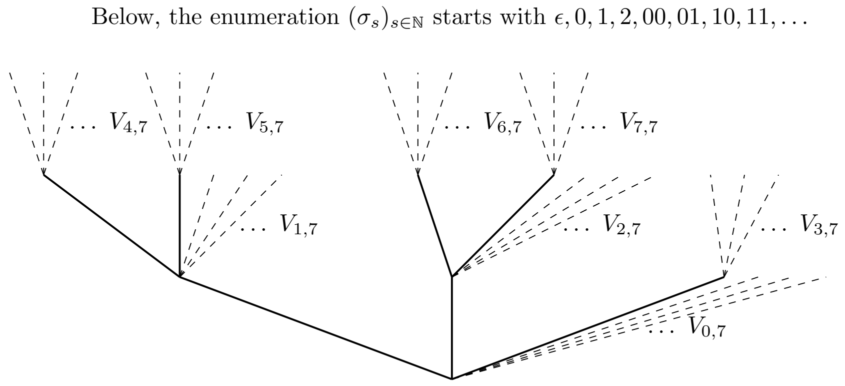


Figure 3.10: Illustration of the proof : at step 8 we will restrict each of the reservoirs from $V_{0,7}$ to $V_{7,7}$ by suppressing some of their branches, and by extending others, so that the current functional Φ_e avoids, for each of the branches of the reservoirs $V_{i,8}$, a dense open class that we are currently enumerating thanks to $\emptyset''$. Once the reservoirs restricted, we complete our tree by taking an extension in each of them, which creates for each extension a new reservoir, and so on.

Finally, we define $T(\sigma_{s+1})$. Suppose that $\sigma_{s+1} = \sigma_t n$ for a $t \leqslant s$ and $n \in \mathbb{N}$. We choose $\tau \in V_{t,s+1}$, remove it from $V_{t,s+1}$, and we define $T(\sigma_{s+1}) = \tau$. Finally, we define $V_{s+1,s+1} = \{\tau m : m \in \mathbb{N}\}$. This completes the proof of Theorem 3.9. ■

Exercise 3.11. **($\star\star$)** Modify the proof of the previous theorem to show that for any countable collection of functions $\mathcal{F}$, there exists an $\mathcal{F}$-escaping function which does not compute any function which is often infinitely equal to any $\emptyset''$-computable function. ◇

3.2. 1-generic sets

We now see 1-genericity, a somewhat stronger and much richer notion than weak 1-genericity. The weakly 1-generic sets are those resulting from forcing for dense Σ^0_1 requirements. But what about Σ^0_1 requirements which are not dense?

The 1-genericity corresponds to the first level of forcing making it possible to tackle any Σ^0_1/Π^0_1 requirements, via the following theorem, which will be demonstrated at the end of the subsection which follows (we refer the reader to Definition 2.1 for the notation $\Vdash^*$):

Theorem 3.12
Let G be a 1-generic set. Let $\mathcal{R}$ be a Σ^0_1 or Π^0_1 requirement. Then, G satisfies $\mathcal{R}$ iff there is a prefix $\sigma \prec G$ such that $\sigma \Vdash^* \mathcal{R}$.

3.2.1. Nature of 1-generic sets

Recall the motivation of the definition of density: a dense set W is such that whatever the stage of the construction by the finite extension method, it is always possible to meet W. If W is not dense, then in the worst case, when we decide to meet it, our initial segment constructed so far may have no extension in W. This motivates the following definition.

Definition 3.13. A string $\sigma \in 2^{<\mathbb{N}}$ *avoids* a set $W \subseteq 2^{<\mathbb{N}}$ (denoted $\sigma \perp W$) if not only $\sigma \notin W$, but also no extension of σ is in W. ♢

Notation
We note $W^\perp = \{\tau \in 2^{<\mathbb{N}} : \tau \perp W\}$.

Lemma 3.14. Let $W \subseteq 2^{<\mathbb{N}}$ be an arbitrary set. Then the set $W \cup W^\perp$ is dense. ⋆

Proof. Let $\sigma \in 2^{<\mathbb{N}}$. Either σ has an extension in W, and therefore in $W \cup W^\perp$, or σ avoids W, in which case $\sigma \in W^\perp$. ∎

Note that if W is a dense set, then $W^\perp = \emptyset$. Remember that the density of a set of strings W closed under suffix on $2^{<\mathbb{N}}$ corresponds to the density on $2^{\mathbb{N}}$ of its corresponding open class $[W] = \bigcup_{\sigma \in W}[\sigma]$. The set $W^\perp$ also corresponds to an open one: the interior of the complement of the set $[W]$, that is to say the union of all the cylinders $[\sigma]$ included in the complement of $[W]$.

If we reformulate it in Cantor space, Lemma 3.14 states that for any open class $\mathcal{U} \subseteq 2^{\mathbb{N}}$, the class $\mathcal{U} \cup \operatorname{int}(2^{\mathbb{N}} \setminus \mathcal{U})$ is a dense open class. If $\mathcal{U}$ is basically dense then $\operatorname{int}(2^{\mathbb{N}} \setminus \mathcal{U}) = \emptyset$, otherwise we “densify” $\mathcal{U}$ by adding the interior of its complement. We are now ready to define the notion of 1-generic set.

Definition 3.15 (Jockusch [104]). A set $G \in 2^{\mathbb{N}}$ is 1-*generic* if it is $\{W_e \cup W_e^\perp : e \in \mathbb{N}\}$-generic, where W_e is the computably enumerable set of strings of code e. Equivalently, G is 1-generic if $G \in \mathcal{U} \cup \operatorname{int}(2^{\mathbb{N}} \setminus \mathcal{U})$ for any Σ^0_1 class $\mathcal{U}$. ♢

As mentioned above, if $W \subseteq 2^{<\mathbb{N}}$ is a dense set, then $W^\perp = \emptyset$. It follows that every 1-generic set meets every dense Σ^0_1 set, and is therefore weakly

1-generic. In particular, the degrees of weakly 1-generic sets coinciding with hyperimmune degrees, any 1-generic set is of hyperimmune degree, and therefore not computable.

If we look at the contraposition of the notion of 1-genericity, a set G is not 1-generic if there exists a c.e. set $W \subseteq 2^{<\mathbb{N}}$ containing no prefix of G and such that W "is dense along G", that is, for all $\sigma \prec G$ there exists $\tau \succeq \sigma$ with $\tau \in W$ and $\tau \not\prec G$. This idea is illustrated in Figure 3.16.

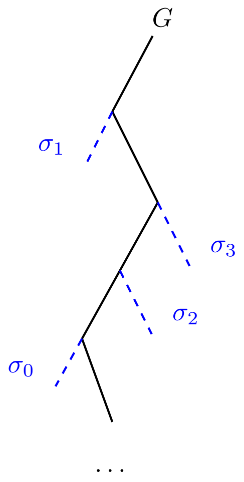


Figure 3.16: Illustration of a set G which is not 1-generic : one can enumerate strings $\sigma_0, \sigma_1, \dots$ *densely along* G, without ever enumerating a prefix of G.

We now see formally why Theorem 3.12 claimed above is true.

Proposition 3.17. Let $\mathcal{R}_e$ be a Σ^0_1 requirement. Then the set $W \subseteq 2^{<\mathbb{N}}$ of strings forcing $\mathcal{R}_e$ is Σ^0_1. Moreover $W^\perp$ is the set of strings forcing $\neg\mathcal{R}_e$ and it is Π^0_1. $\star$

Proof. $\mathcal{R}_e$ being Σ^0_1, it can be written in the form $\Phi(G, 0)\downarrow$ for a functional Φ. A string σ forces $\mathcal{R}_e$ if $\Phi(X, 0)\downarrow$ for all $X \in [\sigma]$. By König's lemma, σ forces $\mathcal{R}_e$ if there exists a length n such that for all $\tau \succeq \sigma$ of length n, $\Phi(\tau, 0)\downarrow$. The set

$$W = \{\sigma \in 2^{<\mathbb{N}} : \exists n\ \forall \tau \in 2^n\ (\tau \succeq \sigma \to \Phi(\tau, 0)\downarrow)\}$$

is Σ^0_1 (where 2^n denotes the set of strings of size n).

Let us show that $W^\perp$ is the set of strings σ which force $\neg\mathcal{R}_e$. If σ does not force $\neg\mathcal{R}_e$, then there exists an $X \in [\sigma]$ such that $\Phi(X, 0)\downarrow$. By the use property, for n sufficiently large and for all $Y \in [X\restriction_n]$, $\Phi(Y, 0)\downarrow$. We can assume $n \geqslant |\sigma|$. Let $\tau = X\restriction_n$. Then, $\tau \in W$, so $\sigma \notin W^\perp$. By contraposition, if $\sigma \in W^\perp$, then σ forces $\neg\mathcal{R}_e$. Now suppose that $\sigma \notin W^\perp$. Then, there exists $\tau \succeq \sigma$ such that $\tau \in W$. In particular, there is an $X \in [\tau] \subseteq [\sigma]$ such that $\Phi(X, 0)\downarrow$. It follows that σ does not force $\neg\mathcal{R}_e$.

As the set W is Σ^0_1, it is clear that the set $W^\perp$ is Π^0_1. ■

Corollary 3.18
Let $\mathcal{R}_e$ be a Σ^0_1 requirement. Then, the set D of strings which force $\mathcal{R}_e$ or which force $\neg\mathcal{R}_e$ is dense.

Proof. Immediate by Lemma 3.14 and Proposition 3.17. ■

Note that any set satisfies the formula "$\mathcal{R}_e \vee \neg\mathcal{R}_e$", and therefore that any string forces "$\mathcal{R}_e \vee \neg\mathcal{R}_e$". On the other hand, it is much stronger for a string σ to force $\mathcal{R}_e$ or to force $\neg\mathcal{R}_e$, because any set $A \in [\sigma]$ must have the same behavior towards $\mathcal{R}_e$.

We can now show Theorem 3.12 announced: if G is a 1-generic set and $\mathcal{R}$ a Σ^0_1 or Π^0_1 requirement, then G satisfies $\mathcal{R}$ iff there is a $\sigma \prec G$ prefix such that $\sigma \Vdash^* \mathcal{R}$.

Proof of Theorem 3.12. Let G be a 1-generic set. Let $\mathcal{R}$ be a Σ^0_1 requirement. If a prefix $\sigma \prec G$ forces $\mathcal{R}$ or $\neg\mathcal{R}$ then by definition G satisfies respectively $\mathcal{R}$ or $\neg\mathcal{R}$.

Conversely, suppose that G satisfies $\mathcal{R}$. Let W be the c.e. set of strings that force $\mathcal{R}$. Since G is 1-generic it meets $W \cup W^\perp$. If G meets $W^\perp$ then σ forces $\neg\mathcal{R}$ for a prefix $\sigma \prec G$ and therefore G satisfies $\neg\mathcal{R}$ which is a contradiction. So G meets W and a prefix $\sigma \prec G$ forces $\mathcal{R}$.

Symmetrically if G satisfies $\neg\mathcal{R}$ then a prefix $\sigma \prec G$ forces $\neg\mathcal{R}$. ■

3.2.2. Properties of 1-generic sets

In general, the function which to X associates X' is not continuous. Indeed, it is sometimes necessary to know an infinity of bits of X to know if $n \in X'$. René Baire showed — in another form of course — that this function was, on the other hand, continuous over a co-meager class of points. The 1-genericity is the level of genericity required to account for this theorem.

Definition 3.19. A set $G \in 2^{\mathbb{N}}$ is *generalized low* if

$$G' \leqslant_T G \oplus \emptyset'$$ ♢

If a set G is generalized low, then not only the function which to X associates X' is continuous in G, but even more it is computable in $G \oplus \emptyset'$. Any set has a priori no reason to be generalized low. For example $\emptyset'$ is not, but it is the case for 1-generic sets.

Theorem 3.20
The 1*-generic sets are generalized low.*

Proof. For any e, define the class $\mathcal{U}_e = \{X : \Phi_e(X, e)\downarrow\}$. Let $W_e \subseteq 2^{<\mathbb{N}}$ be a Σ^0_1 set which represents $\mathcal{U}_e$, i.e., such that $[W_e] = \mathcal{U}_e$ (where $[W_e] = \bigcup_{\sigma \in W_e}[\sigma]$). Note that $e \in G'$ iff $G \in \mathcal{U}_e$. We also have $G \in \mathcal{U}_e$ iff there exists $\tau \preceq G$ such that $\tau \in W_e$, and by definition of the 1-genericity of G, we have $G \notin \mathcal{U}_e$ iff there is $\tau \preceq G$ such that $\sigma \notin W_e$ for any σ compatible with τ. The question of whether $\sigma \notin W_e$ for any σ compatible with τ is Π^0_1 uniformly in τ and therefore can be asked to $\emptyset'$. To know if $e \in G'$, it suffices therefore to look for a prefix τ of G such that we are in one case or the other, which will necessarily happen. ∎

We have seen that any 1-generic set has a hyperimmune degree, and therefore cannot be computed. Therefore, 1-genericity is not a weakness property. We will now see that it is not a strength property either, in the sense that certain computational powers can never be reached by the 1-generic degrees. In particular, no 1-generic degree computes $\emptyset'$.

Theorem 3.21 (Demuth and Kučera [48])
No 1*-generic set is of DNC degree.*

Proof. Suppose $n \mapsto \Phi(G, n)$ is a DNC function. We consider the class $\mathcal{U} = \{X : \exists n\ \Phi(X, n)\downarrow = \Phi_n(n)\downarrow\}$. By hypothesis $G \notin \mathcal{U}$. Consider a string σ. We define by Kleene's fixed point theorem the code e_σ of the function which, on the input e_σ, searches for an extension $\tau_\sigma \succeq \sigma$ such that $\Phi(\tau_\sigma, e_\sigma)\downarrow$ and assigns $\Phi(\tau_\sigma, e_\sigma)$ to $\Phi_{e_\sigma}(e_\sigma)$. The process being uniform, we enumerate all the strings of the form τ_σ in a c.e. set W.

Note that for all $\sigma \prec G$ the function code e_σ halts on the input e_σ because there exists at least one extension of σ —namely G itself— for which Φ halts on the input e_σ. Since $n \mapsto \Phi(G, n)$ is DNC, we have $\Phi(G, e_\sigma) \neq \Phi_{e_\sigma}(e_\sigma)$ and therefore $\sigma \prec \tau_\sigma \not\prec G$.

We therefore have a c.e. set W of which no element is a prefix of G, but which contains an extension of each prefix of G. It follows that G is not 1-generic. ∎

The restriction of genericity makes it possible to construct sets benefiting from certain advantages of genericity while not being too complex from a computational point of view.

Exercise 3.22. **($\star$)** Show by a direct argument that no 1-generic set is computable. ◇

Exercise 3.23. ($\star$) Show that there exists a 1-generic Δ^0_2 set. Deduce that there exists a 1-generic set of low degree. $\diamond$

We will see in the next section with Corollary 3.34 that there are also 1-generic sets of high degree. Let us end with an interesting exercise, which requires elaborating on the techniques of Theorem 3.28, and which uses the notion of effective join of Definition 4-5.6, but extended to a countable sequence of sets:

Definition 3.24. The *effective join* $\bigoplus_{n\in\mathbb{N}} A_n$ of a sequence of sets $(A_n)_{n\in\mathbb{N}}$ is the set Y such that $\langle n, m\rangle \in Y$ iff $m \in A_n$. $\diamondsuit$

In the following exercises, the notation $\bigoplus_{j\neq i} G_j$ therefore corresponds to the effective join of the sequence $(G_n)_{n\in\mathbb{N}}$ from which we remove the set G_i.

Exercise 3.25. ($\star\star$) Let $G = \bigoplus_{n\in\mathbb{N}} G_n$ be a 1-generic set. Show that G_i is hyperimmune relative to $\bigoplus_{j\neq i} G_j$ for all $i \in \mathbb{N}$. $\diamond$

Exercise 3.26. ($\star\star$) (*Miller*). Let $X = \bigoplus_{n\in\mathbb{N}} X_n$ be a set such that X_i is hyperimmune relative to $\bigoplus_{j\neq i} X_j$ for all $i \in \mathbb{N}$. Show that X computes a 1-generic set. $\diamond$

3.2.3. Relativization of 1-generic sets

Just as we relativized weak 1-genericity, we now relativize 1-genericity.

Definition 3.27 (Jockusch [104]). Let $A \in 2^{\mathbb{N}}$. A set $G \in 2^{\mathbb{N}}$ is *1-generic* relative to A if $G \in \mathcal{U} \cup \operatorname{int}(2^{\mathbb{N}} \setminus \mathcal{U})$ for any class $\mathcal{U}$ which is $\Sigma^0_1(A)$. We will say that G is n-generic if it is 1-generic with respect to $\emptyset^{(n-1)}$, or in an equivalent way if it meets $W \cup W^{\perp}$ for any Σ^0_n set of strings W. $\diamondsuit$

We will study this relativization in more detail in Section 5. Let us see for the moment a first key theorem.

Theorem 3.28
Let X be non-computable and G be a 1-generic set relative to X. Then, we have $G \not\geq_T X$.

Proof. We are going to build a $\Sigma^0_1(X)$ class $\mathcal{U}$ such that $Y \in \mathcal{U} \cup \operatorname{int}(2^{\mathbb{N}} \setminus \mathcal{U})$ implies $\Phi(Y) \neq X$.

We simply enumerate in the set X-c.e. which describes $\mathcal{U}$, all strings σ such that $\exists n\ \Phi(\sigma, n){\downarrow} \neq X(n)$. Suppose now that τ is a string for which $[\tau] \subseteq 2^{\mathbb{N}} \setminus \mathcal{U}$, that is, no extension $\sigma \succeq \tau$ is such that $\exists n\ \Phi(\sigma, n){\downarrow} \neq X(n)$. Let us show that for all $Y \succeq \tau$ we have $\exists n\ \Phi(Y, n){\uparrow}$. Suppose this is not the case.

Then, we can compute X as follows: to know $X(n)$, it suffices to look for an extension $\sigma \succeq \tau$ such that $\Phi(\sigma, n)\downarrow$. By hypothesis, such an extension exists, and still by hypothesis, it is such that $\Phi(\sigma, n)\downarrow = X(n)$. As this is true for every n, this contradicts the fact that X is non-computable. So if $[\tau] \subseteq 2^{\mathbb{N}} \setminus \mathcal{U}$ then for all $Y \succeq \tau$ we have $\exists n\ \Phi(Y, n)\uparrow$. We deduce that for all $Y \in \mathcal{U} \cup \operatorname{int}(2^{\mathbb{N}} \setminus \mathcal{U})$ we have $\Phi(Y) \neq X$. As G is 1-generic relative to X then $G \in \mathcal{U} \cup \operatorname{int}(2^{\mathbb{N}} \setminus \mathcal{U})$ and the same is true for any functional Φ. So $G \not\geqslant_T X$. ■

Let us now see the various implications between the notions of n-genericity and n-weak genericity.

Proposition 3.29. Let G be a set. Then, for all $n > 0$, G weakly $(n+1)$-generic implies G n-generic implies G weakly n-generic. The implications are strict. ⋆

Proof. If G is weakly $(n+1)$-generic, then it meets any $\Sigma^0_1(\emptyset^{(n)})$ dense set. For any $\Sigma^0_1(\emptyset^{(n-1)})$ set W, $W \cup W^{\perp}$ is dense and $\Sigma^0_1(\emptyset^{(n)})$, so G meets $W \cup W^{\perp}$. Thus G is n-generic. If G is n-generic, then for any $\Sigma^0_1(\emptyset^{(n-1)})$ set W, G meets $W \cup W^{\perp}$. If W is dense, then $W^{\perp} = \emptyset$, so G meets W. Thus G is weakly n-generic.

To see that the first implication is strict, it suffices to construct an n-generic $\emptyset^{(n)}$-computable set and to see that no weakly $(n+1)$-generic set is $\emptyset^{(n)}$-computable because $\{\sigma : \sigma \not\prec X\}$ is a Σ^0_{n+1} dense set of strings for any set X which is Δ^0_{n+1}. We can consult Exercise 3.30 for an example of an n-generic set which is not weakly n-generic. ■

Exercise 3.30. (⋆⋆) A set X is *left-c.e.* relative to A if there exists an A-computable sequence of sets $(X_s)_{s\in\mathbb{N}}$ such that X_s is lexicographically smaller than X_{s+1} for all s, and such that $\lim_s X_s = X$.

Show that for all A there exists a weakly 1-generic set relative to A and left-c.e. relative to A. Show that no 1-generic set relative to A is left-c.e. relative to A. ◇

3.3. Posner/Robinson theorem

In a 1981 article, Posner and Robinson study degrees strictly under the halting set, but the join of which makes it possible to compute the halting set. A generalized and modern version of their main theorem is the following: for any set non-computable A —and in particular as "weak" as computably possible— there exists a set G such that $A \oplus G \geqslant_T G'$. Informally, there always exists a set G whose computational distance between itself and its jump, is "reduced" to A, and this for any A.

The modern presentation of the theorem is that of Jockusch and Shore, who show something more general:

Theorem 3.31 (Jockusch and Shore [106])
Let A, Z be non-computable sets. There exists a 1-generic set G such that $A \oplus G \geqslant_T Z$. Moreover, we can obtain G in a computable way from $A \oplus Z \oplus \emptyset'$.

Proof. The idea is to build a 1-generic set G, which will encode Z, so that G and A allow to find the construction sequence. The construction itself will be computable in $A \oplus Z \oplus \emptyset'$. We can assume without loss of generality that A is not a c.e. set (otherwise, one replaces A by its complement). Let $(W_e)_{e\in\mathbb{N}}$ be an enumeration of the Σ^0_1 subsets of $2^{<\mathbb{N}}$.

We define $\sigma_0 = \epsilon$, the empty word. Suppose σ_e defined. We consider the set

$$D_e = \{m : \exists \tau \text{ such that } \sigma_e Z(e) 0^m 1 \tau \in W_e\}.$$

Note that D_e is a c.e. set. In particular as A is not c.e. there is some $m \in D_e$ with $m \notin A$ or some $m \notin D_e$ with $m \in A$. We consider the smallest m such that we are in one case or the other. Note that $\emptyset' \oplus A$ allows to find uniformly this integer m.

In the first case, we define σ_{e+1} as being $\sigma_e Z(e) 0^m 1\tau$ for the first string τ such that $\sigma_e Z(e) 0^m 1 \tau$ is listed in W_e. In the second case, we define σ_{e+1} as being $\sigma_e Z(e) 0^m 1$. Note that in this case no string of W_e can extend σ_{e+1}. We define G as being $\sigma_0 \preceq \sigma_1 \preceq \sigma_2 \preceq \dots$. This completes the construction.

It is clear that G is 1-generic and computable in $A \oplus Z \oplus \emptyset'$. How do you now compute Z from $G \oplus A$? Suppose we know the string σ_e. We then necessarily know the e-th bit of Z: it is the bit i such that $\sigma_e i \prec G$. We can then find σ_{e+1} as follows: we look at the number m of 0 which follows $\sigma_e i$ in G. If $m \in A$, this means that $\sigma_{e+1} = \sigma_e i 0^m 1$. If $m \notin A$, this means that $\sigma_{e+1} = \sigma_e i 0^m 1 \tau$ for the first string τ found in W_e. Finding this string τ is then a computable process. We can therefore in all cases find σ_{e+1}, and by repeating the process, compute Z from $A \oplus G$. ■

Corollary 3.32 (Posner and Robinson [188])
Let A be a non-computable set. There exists a set G such that $A \oplus G \geqslant_T G'$. If A is Δ^0_2 then we have $A \oplus G \equiv_T G'$.

Proof. We apply the previous theorem with $Z = \emptyset'$. We therefore have a 1-generic set G such that $G \leqslant_T \emptyset' \oplus A$ and such that $G \oplus A \geqslant_T \emptyset'$. By using the fact that the 1-generics are generalized low, we thus have $G \oplus A \geqslant_T G \oplus \emptyset' \equiv_T G'$. It is clear that if A is Δ^0_2 then $A \oplus G \equiv_T G'$. ■

Another interesting corollary is the jump inversion theorem: any set that computes the halting problem can be seen as the Turing degree of the jump of a set.

Corollary 3.33 (Jump inversion theorem, Friedberg [65])
Let $Z \geqslant_T \emptyset'$. There exists a set G such that $Z \equiv_T G'$.

Proof. We apply the previous theorem with Z and $A = \emptyset'$. We therefore have a 1-generic set G such that $G \oplus \emptyset' \leqslant_T Z \oplus \emptyset' \equiv_T Z$ and such that $G \oplus \emptyset' \geqslant_T Z$. By using the fact that the 1-generics are generalized low, we thus have $G' \equiv_T Z$. ■

Theorem 3.31 also allows us to deduce the existence of high degrees which are Turing incomplete, and even of non-DNC degree.

Corollary 3.34
There is a high set of non-DNC degree, and in particular Turing incomplete.

Proof. It suffices to apply Theorem 3.31 to find a 1-generic set G such that $\emptyset' \oplus G \geqslant_T \emptyset''$, which implies $G' \geqslant_T \emptyset''$. Note that as G is 1-generic, it is not of DNC degree and in particular does not compute $\emptyset'$. ■

3.4. Meager/co-meager nature of computational properties

Let us go back for a moment to the topological notions of meager and co-meager classes introduced by Baire. We have already mentioned that not every class is necessarily meager or co-meager. On the other hand, it is the case for the classes having good closure properties, and in particular for all those known as Borel (we will precisely define this term in Chapter 17) and closed under Turing equivalence. This will be the case in particular for all the notions of computability that we will see, and we will ask ourselves the question of whether these latter are meager or co-meager. Also a class is without loss of generality co-meager if it contains any *sufficiently generic* element. In practice, 1-genericity is sufficient for the various computational properties seen so far.

We have the high degrees — the low degrees being a countable class, they are not of interest — the computably dominated vs hyperimmune degrees, and finally the DNC and PA degrees. Is the class of sets of each of these degrees meager or co-meager? We already have the answer when it comes to computably dominated and hyperimmune degrees:

Proposition 3.35. The class of computably dominated sets is meager. That of hyperimmune sets is co-meager. ⋆

Proof. According to Theorem 3.2 if G is weakly 1-generic then it is not of computably dominated degree. ■

Proposition 3.36. The class of DNC sets is meager, as is of course the class of PA sets. ⋆

Proof. According to Theorem 3.21 if X is 1-generic then it is not of DNC degree, and the class of 1-generic sets is co-meager. ■

We are now tackling the high degrees. For this, we use our cone avoidance theorem 3.28.

Proposition 3.37. If X is non-computable, then the class of sets which compute X is meager. ⋆

Proof. The class of 1-generic sets relative to X is an intersection of dense open classes, and is therefore co-meager. By Theorem 3.28, none of them computes X and therefore the class of sets computing X is in its complement, a meager class. ■

We can finally show by combining what we have seen that the class of high sets is also meager:

Proposition 3.38. The class of high sets is meager. ⋆

Proof. For this, we use a relativized version of Theorem 3.28: if X is not $\emptyset'$-computable then for any set G which is 1-generic relative to $X \oplus \emptyset'$ we have $G \oplus \emptyset' \not\geqslant_T X$. This relativized version is shown in the same way and does not present any particular difficulty.

We now use Theorem 3.20: if G is 1-generic then $G' \leqslant_T G \oplus \emptyset'$. We deduce that if G is 1-generic then G is high iff $G \oplus \emptyset' \geqslant_T \emptyset''$. Moreover, according to the relativized version of Theorem 3.28, no 1-generic relative to $\emptyset''$ is such that $G \oplus \emptyset' \geqslant_T \emptyset''$. As any 1-generic set relative to $\emptyset''$ is also 1-generic, one therefore has that no 1-generic relative to $\emptyset''$ is high. So the class of high sets is meager. ■

4. Σ^0_n/Π^0_n forcing

The concept of 1-genericity can be seen as an effective formal framework around the finite extension method, which enables to control the halting or not of functionals, that is to say to control the truth value of Σ^0_1 or Π^0_1 predicates. This is a first level of forcing: given a Σ^0_1 or Π^0_1 requirement $\mathcal{R}_e$, according to Corollary 3.18, any string σ admits an extension $\tau \succeq \sigma$ such that any set $X \in [\tau]$ satisfies $\mathcal{R}_e$ or such that any set $X \in [\tau]$ satisfies $\neg\mathcal{R}_e$.

From level Σ^0_2/Π^0_2, things get more complex and the finite extension method can no longer work in the same way, as the following example shows.

Example 4.1. Consider the requirement $\mathcal{R}$: "$\exists x \forall y \geqslant x\ G(y) = 0$" which expresses the finiteness of the set G. For all $\sigma \in 2^{<\mathbb{N}}$, $[\sigma]$ contains both a finite set and an infinite set. There is therefore no cylinder of which all the elements satisfy $\mathcal{R}$ or of which all the elements satisfy $\neg\mathcal{R}$.

Check-in on requirements

We now tackle the Σ^0_n (resp. Π^0_n) requirements, that is to say the requirements relating to a set G and which are expressed by a Σ^0_n (resp . Π^0_n) formula of second-order arithmetic, with G as a free set variable. Equivalently, a requirement is Σ^0_n if there exists a functional $\Phi_e(G, x_1, \dots, x_{n-1})$ such that the sets G satisfying the requirement are those such that:

$$\begin{array}{ll} \exists x_1\ \forall x_2 \dots \forall x_{n-1}\ \Phi_e(G, x_1, x_2, \dots, x_{n-1})\downarrow & \text{for } n \text{ odd} \\ \exists x_1\ \forall x_2 \dots \exists x_{n-1}\ \Phi_e(G, x_1, x_2, \dots, x_{n-1})\uparrow & \text{for } n \text{ even.} \end{array}$$

The Π^0_n requirements have the analogous equivalence starting with a universal quantification. By convention the negation $\neg\mathcal{R}$ in front of a Σ^0_n formula (resp. Π^0_n) will not be considered as a symbol of the language, but as a transformation operation over $\mathcal{R}$, which reverses the quantifiers, and replaces the final Δ^0_0 predicate by its negation, to make $\neg\mathcal{R}$ a Π^0_n formula (resp. Σ^0_n).

We must therefore abstract things a little, in order to give a more general definition of forcing allowing to control the truth value of predicates of arbitrary complexity. Before we begin, let's introduce a definition that will be used in future proofs.

Definition 4.2. A set $D \subseteq 2^{<\mathbb{N}}$ is *dense below* σ if for all $\tau \succeq \sigma$, there exists a $\rho \succeq \tau$ such that $\rho \in D$. $\diamondsuit$

4.1. The semantic approach

The forcing relation will be defined by induction on n, between finite strings and Σ^0_n predicates. One of the objectives will then be to keep the following property.

(D): The set of strings forcing $\mathcal{R}$ or forcing $\neg\mathcal{R}$ is dense.

In order to examine what we need, let's take the previous example, namely an arbitrary Σ^0_2 requirement $\mathcal{R}$ "$\exists x\ \Phi(G, x)\uparrow$", and see what does not work when we try to prove the density of the set $Q \subseteq 2^{<\mathbb{N}}$ of strings all of whose elements satisfy $\mathcal{R}$ or all of whose elements satisfy $\neg\mathcal{R}$.

Let $\sigma \in 2^{<\mathbb{N}}$ be a string. Let's do a case analysis. Case 1: there is an extension $\tau \succeq \sigma$ and an $x \in \mathbb{N}$ such that for all $\rho \succeq \tau$, $\Phi(\rho, x)\uparrow$. In this case, by the use property, for all $A \in [\tau]$, $\Phi(A, x)\uparrow$. It follows that $\mathcal{R}$ is satisfied for all $A \in [\tau]$, so that $\tau \in Q$. Case 2: for any extension $\tau \succeq \sigma$ and any $x \in \mathbb{N}$, there exists a string $\rho \succeq \tau$ such that $\Phi(\rho, x)\downarrow$. This is where our attempt fails. Yet, intuitively, in this case, we should be able to continue building a sequence while ensuring that the resulting set satisfies $\neg\mathcal{R}$. Indeed, whatever the state of progress of the construction, we end up with a string $\tau \succeq \sigma$, so by hypothesis, for any $x \in \mathbb{N}$, it is always possible to find an extension $\rho \succeq \tau$ such that $\Phi(G, x)\downarrow$ for all $G \in [\rho]$: for all x, the set Q_x of strings ρ such that $\Phi(\rho, x)\downarrow$ is dense below σ, that is, for all $\tau \succeq \sigma$ there exists $\rho \succeq \tau$ such that $\rho \in Q_x$. So if a sufficiently generic set G extends σ, then it will meet each of Q_x. We will therefore have $\forall x\ \Phi(G, x)\downarrow$, in other words G will satisfy $\neg\mathcal{R}$.

This analysis therefore motivates the following definition for Cohen forcing:

Definition 4.3. A string σ *semantically forces* a requirement $\mathcal{R}$, in which case we will write $\sigma \Vdash \mathcal{R}$, if any "sufficiently generic" set which extends σ satisfies $\mathcal{R}$: there exists a countable sequence of dense sets of strings $(W_n)_{n\in\mathbb{N}}$ such that if $G \in [\sigma]$ meets every W_n, then the requirement will be satisfied for G. ♢

Note that the relation $\Vdash$ is more general than the relation $\Vdash^*$ for the Σ^0_1 and Π^0_1 requirements. For example, the empty string ϵ semantically forces the requirement $\mathcal{R}$: "$\exists x\ G(x) = 1$", because the set of strings containing a 1 is dense, and therefore any sufficiently generic set will satisfy $\mathcal{R}$. On the other hand, the infinite sequence of zeros 0^∞ belongs to $[\epsilon]$ and does not satisfy the requirement $\mathcal{R}$.

4.2. The syntactic approach

Definition 4.3 is simple to express in natural language, but escapes the arithmetic hierarchy: a direct translation requires existentially quantifying on all the countable sequences of dense sets of strings, then universally quantifying on the generic sets for those sets of strings. We are going to define a relation much simpler syntactically speaking, which will make it possible to reason more easily and in particular to prove the essential properties which one expects from the forcing relation, namely that it is closed under extension and that the set of strings forcing a requirement or its negation is dense.

Let us now define a syntactic forcing relation for any arithmetic requirement, by induction on its quantifications. The first level will be the one we are already used to: let $\mathcal{R}$ a Σ^0_1 requirement — and therefore of the form $\Phi_e(G)\downarrow$ — then a string σ forces $\mathcal{R}$ if $\Phi_e(\sigma)\downarrow$ and therefore if all $X \in [\sigma]$ satisfies the requirement. The same goes Π^0_1 for requirements.

Definition 4.4.

(1) $\sigma \Vdash^* \mathcal{R}$ for a Σ^0_1 or Π^0_1 requirement $\mathcal{R}$ iff all $X \in [\sigma]$ satisfy $\mathcal{R}$.

(2) $\sigma \Vdash^* \exists x \mathcal{R}(x)$ for a Π^0_k requirement $\mathcal{R}(x)$ for $k > 0$ with free variable x iff there is an $n \in \mathbb{N}$ such that $\sigma \Vdash^* \mathcal{R}(n)$.

(3) $\sigma \Vdash^* \forall x \mathcal{R}(x)$ for a Σ^0_k requirement $\mathcal{R}(x)$ for $k > 0$ with free variable x iff for all $\tau \succeq \sigma$ and all $n \in \mathbb{N}$, $\tau \nVdash^* \neg\mathcal{R}(n)$. $\diamondsuit$

First of all, note that (3) of the previous definition admits an equivalent formulation, sometimes more suited to what we want to demonstrate:

Lemma 4.5. Let $\mathcal{R}$ be a Π^0_n requirement, we have $\sigma \Vdash^* \mathcal{R}$ iff

$$\forall \tau \succeq \sigma\ \tau \nVdash^* \neg\mathcal{R}$$

Proof. This is a simple reformulation of the definition.

Case 1: $\mathcal{R}$ is a Π^0_1 requirement of the form $\Phi(G)\uparrow$. Then, $\sigma \Vdash^* \mathcal{R}$ iff $\Phi(X)\uparrow$ for all $X \in [\sigma]$ iff $\Phi(X)\uparrow$ for all $\tau \succeq \sigma$ and all $X \in [\tau]$ iff for all $\tau \succeq \sigma$, there exists $X \in [\tau]$ such that $\Phi(X)\uparrow$ (by the use property) iff for all $\tau \succeq \sigma$, $\tau \nVdash^* \Phi(G)\downarrow$.

Case 2: $\mathcal{R}$ is a Π^0_{k+1} requirement of the form $\forall x \mathcal{Q}(x)$ for a Σ^0_k requirement $\mathcal{Q}(x)$ for $k \geqslant 1$. Then, $\sigma \Vdash^* \mathcal{R}$ iff for all $\tau \succeq \sigma$ and all $n \in \mathbb{N}$, $\tau \nVdash^* \neg\mathcal{Q}(n)$ iff $\forall \tau \succeq \sigma\ \tau \nVdash^* \exists x\ \neg\mathcal{Q}(x)$ iff $\forall \tau \succeq \sigma\ \tau \nVdash^* \neg\mathcal{R}$. ■

The first property that we expect from a forcing relation is its closure under extension. Indeed, "σ forces $\mathcal{R}$" means that the property $\mathcal{R}$ is already decided on the final constructed object, which should not change during the following stages of the construction.

Proposition 4.6. Let $\sigma, \tau \in 2^{<\mathbb{N}}$ and $\mathcal{R}$ be an arithmetic requirement. If $\sigma \Vdash^* \mathcal{R}$ and $\sigma \preceq \tau$, then $\tau \Vdash^* \mathcal{R}$. $\star$

Proof. By induction on the arithmetic complexity of the requirement.

Case 1: $\mathcal{R}$ is Σ^0_1 or Π^0_1. By definition, for all $X \in [\sigma]$, $\mathcal{R}(X)$ is true. As $\tau \succeq \sigma$, then $[\tau] \subseteq [\sigma]$, so for all $X \in [\tau]$, $\mathcal{R}(X)$ is true. So $\tau \Vdash^* \mathcal{R}$.

Case 2: $\mathcal{R}$ is of the form $\exists x \mathcal{S}(x)$ for a Π^0_k requirement $S(x)$ for $k > 0$. By definition, there exists an $n \in \mathbb{N}$ such that $\sigma \Vdash^* \mathcal{S}(n)$. By induction hypothesis, $\tau \Vdash^* \mathcal{S}(n)$, therefore $\tau \Vdash^* \exists x \mathcal{S}(x)$.

Case 3: $\mathcal{R}$ is of the form $\forall x \mathcal{S}(x)$ for a Σ^0_k requirement $S(x)$ for $k > 0$. According to Lemma 4.5, $\sigma \Vdash^* \mathcal{R}$ implies $\forall \rho \succeq \sigma\ \rho \nVdash^* \neg\mathcal{R}$. If $\tau \succeq \sigma$ then we also have $\forall \rho \succeq \tau\ \rho \nVdash^* \neg\mathcal{R}$, which again according to Lemma 4.5 implies $\tau \Vdash^* \mathcal{R}$. ■

The second property, and perhaps the most important, is the density of the set of strings forcing a requirement or its negation. Proposition 4.7 means in particular that the truth value of any arithmetic requirement will be decided after a finite moment of the construction. This is what gives all the power of the forcing relationship.

Proposition 4.7. Let $\mathcal{R}$ be an arithmetic requirement. The set

$$\{\sigma \in 2^{<\mathbb{N}} : \sigma \Vdash^* \mathcal{R} \text{ or } \sigma \Vdash^* \neg\mathcal{R}\}$$

is dense. ⋆

Proof. We can assume without loss of generality that $\mathcal{R}$ is Σ^0_n (in the opposite case we repeat the argument with $\neg\mathcal{R}$). Let σ be a string. According to Lemma 4.5, either $\tau \Vdash^* \mathcal{R}$ for an extension $\tau \succeq \sigma$, or $\sigma \Vdash^* \neg\mathcal{R}$. ■

A last property that we also expect is of course the validity of the definition of the forcing relation, that is to say that if a string σ forces a requirement, then this requirement will be effectively satisfied for any sufficiently generic set that extends σ. Let us insist again on what it means to be "sufficiently generic" in this context: there exists a countable sequence of dense sets of strings $(W_n)_{n\in\mathbb{N}}$ such that if $G \in [\sigma]$ meets every W_n, then the requirement will be satisfied for G.

Proposition 4.8. Let $\mathcal{R}$ be an arithmetic requirement and $\sigma \in 2^{<\mathbb{N}}$. If $\sigma \Vdash^* \mathcal{R}$, then $\sigma \Vdash \mathcal{R}$: if $G \in [\sigma]$ is sufficiently generic then $\mathcal{R}$ is satisfied for G. ⋆

Proof. By induction on the arithmetic complexity of the requirement.

Case 1: $\mathcal{R}$ is Σ^0_1 or Π^0_1. Suppose that $\sigma \Vdash^* \mathcal{R}$. By definition, any set $G \in [\sigma]$ satisfies $\mathcal{R}$, so *a fortiori* any sufficiently generic set $G \in [\sigma]$. Thus, $\sigma \Vdash \mathcal{R}$.

Case 2: $\mathcal{R}$ is of the form $\exists x \mathcal{S}(x)$ for a Π^0_k requirement $S(x)$ for $k > 0$. Suppose $\sigma \Vdash^* \exists x \mathcal{S}(x)$. By definition, there is an $n \in \mathbb{N}$ such that $\sigma \Vdash^* \mathcal{S}(n)$. By induction hypothesis, σ forces $\mathcal{S}(n)$, therefore $\sigma \Vdash \exists x \mathcal{S}(x)$.

Case 3: $\mathcal{R}$ is of the form $\forall x \mathcal{S}(x)$ for a Σ^0_k requirement $S(x)$ for $k > 0$. Suppose that $\sigma \Vdash^* \forall x \mathcal{S}(x)$. By definition, for all $\tau \succeq \sigma$ and $n \in \mathbb{N}$, $\tau \nVdash^* \neg\mathcal{S}(x)$. By Proposition 4.7, for all n, the set $D_n = \{\tau \in 2^{<\mathbb{N}} : \tau \Vdash^* \mathcal{S}(n)\}$ is dense below σ: for all $\tau \succeq \sigma$ there exists $\rho \succeq \tau$ such that $\rho \Vdash^* \mathcal{S}(n)$. Let G be a sufficiently generic set extending σ. We suppose in particular that the level of genericity of G guarantees that it meets every set D_n. So for all n, there exists a prefix $\tau_n \prec G$ such that $\tau_n \Vdash^* \mathcal{S}(n)$. For such a string τ_n, by induction hypothesis, there exists a countable sequence of dense sets of strings $(D_{n,m})_{m\in\mathbb{N}}$ such that if $G \in [\tau_n]$ meets each $D_{n,m}$ then it satisfies $\mathcal{S}(n)$. We have $G \in [\tau_n]$ and since G is sufficiently generic it meets every $D_{n,m}$ and it therefore satisfies $\mathcal{S}(n)$. As is the case for all n, then G satisfies $\forall x \mathcal{S}(x)$, so $\sigma \Vdash \forall x \mathcal{S}(x)$. ∎

The heart of forcing is undoubtedly in the previous proposition, and in particular in case 3 of its proof: it is there that the mechanism of the relation $\sigma \Vdash^* \mathcal{R}$ is exerted, which guarantees that if G belongs to $[\sigma]$ and if G is sufficiently generic, then the requirement $\mathcal{R}$ will be satisfied for G. This is actually a sophisticated modification of the finite extension method, the key point being the following: regardless of the prefix σ of G that we have built so far, we can expand σ to meet any dense set of strings fixed in advance.

We now see in the following section that this idea is not new: the mathematician René Baire had already formalized all of these mechanisms at the beginning of the 20th century, in particular via the following theorem: "every Borel class has the Baire property".

4.3. The topological approach: the Baire property

An important technique for the study of complex objects in mathematics consists in reducing oneself to simpler objects while controlling the approximation margin of error. In particular, in the study of sets, whether from the point of view of measure theory or category theory, there are a number of theorems of the form "any complex set is equivalent to a simple set modulo a negligible quantity of elements". In measure theory for example, we have the three Littlewood principles [149], which state that any measurable set is "almost" a finite union of intervals, any function is "almost" continuous, and any convergent sequence is "almost" uniformly convergent. In this context "almost" means: "except on a set of measure less than ε for ε as small as we want".

The Baire property expresses the fact that a class S is "almost" open, where almost means in this context: "except on a meager class".

Definition 4.9. A class $\mathcal{B} \subseteq 2^{\mathbb{N}}$ has the *Baire property* if there is an open class $\mathcal{U} \subseteq 2^{\mathbb{N}}$ such that $\mathcal{B}\Delta\mathcal{U}$ is a meager class. Here $\mathcal{B}\Delta\mathcal{U}$ is the class of elements on which $\mathcal{B}$ and $\mathcal{U}$ do not match, ie $(\mathcal{B} \setminus \mathcal{U}) \cup (\mathcal{U} \setminus \mathcal{B})$.$\diamondsuit$

Consider a requirement $\mathcal{R}$ and let $\mathcal{B}_{\mathcal{R}}$ and $\mathcal{B}_{\neg\mathcal{R}}$ be the classes of the elements which respectively satisfy and do not satisfy this requirement (in particular $\mathcal{B}_{\mathcal{R}} \cap \mathcal{B}_{\neg\mathcal{R}} = \emptyset$ and $\mathcal{B}_{\mathcal{R}} \cup \mathcal{B}_{\neg\mathcal{R}} = 2^{\mathbb{N}}$). Suppose that $\mathcal{B}_{\mathcal{R}}$ and $\mathcal{B}_{\neg\mathcal{R}}$ both have the Baire property and fix two open classes $\mathcal{U}_{\mathcal{R}}$ and $\mathcal{U}_{\neg\mathcal{R}}$ such that $\mathcal{B}_{\mathcal{R}}\Delta\mathcal{U}_{\mathcal{R}}$ and $\mathcal{B}_{\neg\mathcal{R}}\Delta\mathcal{U}_{\neg\mathcal{R}}$ are both meager. This means that there is a countable union of closed classes of empty interior containing $\mathcal{B}_{\mathcal{R}}\Delta\mathcal{U}_{\mathcal{R}}$ and $\mathcal{B}_{\neg\mathcal{R}}\Delta\mathcal{U}_{\neg\mathcal{R}}$. By passing to the complement, there exists a countable intersection of dense open classes $\bigcap_n \mathcal{U}_n$ such that for all $X \in \bigcap_n \mathcal{U}_n$, we have $X \in \mathcal{B}_{\mathcal{R}}$ iff $X \in \mathcal{U}_{\mathcal{R}}$ and $X \in \mathcal{B}_{\neg\mathcal{R}}$ iff $X \in \mathcal{U}_{\neg\mathcal{R}}$.

In particular $X \in \mathcal{B}_{\mathcal{R}}$ iff there is a prefix $\sigma \prec X$ such that $[\sigma] \subseteq \mathcal{U}_{\mathcal{R}}$. We will then say that σ forces the requirement $\mathcal{R}$. Note that if $X \notin \mathcal{B}_{\mathcal{R}}$ then $X \in \mathcal{B}_{\neg\mathcal{R}}$ and there then a prefix $\sigma \prec X$ such that $[\sigma] \subseteq \mathcal{U}_{\neg\mathcal{R}}$. At this point σ forces the requirement $\neg\mathcal{R}$. The following definition is simply a reformulation of Definition 4.3 which defines semantic forcing.

Definition 4.10. Let $\mathcal{R}$ be a requirement. We say that σ *semantically forces* $\mathcal{R}$ and we write $\sigma \Vdash \mathcal{R}$ if $[\sigma] \setminus \mathcal{B}_{\mathcal{R}}$ is a meager class, where $\mathcal{B}_{\mathcal{R}}$ is the class of the elements which satisfy $\mathcal{R}$. $\diamondsuit$

As mentioned earlier, the fundamental property expected of the forcing relation for a requirement is as follows.

(D): The set of strings forcing $\mathcal{R}$ or forcing $\neg\mathcal{R}$ is dense.

We therefore start with a characterization of the requirements for which this property is true, using the Baire property.

Proposition 4.11. Let $\mathcal{R}$ be a requirement and $\mathcal{B}_{\mathcal{R}}$ the class of elements which satisfy $\mathcal{R}$. The following statements are equivalent:

(1) $\mathcal{B}_{\mathcal{R}}$ has the Baire property

(2) $\{\sigma \in 2^{<\mathbb{N}} : \sigma \Vdash \mathcal{R} \text{ or } \sigma \Vdash \neg\mathcal{R}\}$ is dense $\star$

Proof. Let $U_{\mathcal{R}} = \{\sigma \in 2^{<\mathbb{N}} : \sigma \Vdash \mathcal{R}\}$ and $U_{\neg\mathcal{R}} = \{\sigma \in 2^{<\mathbb{N}} : \sigma \Vdash \neg\mathcal{R}\}$.

(1) $\Rightarrow$ (2). Suppose that $\mathcal{B}_{\mathcal{R}}$ has the Baire property. Let $\mathcal{U}$ be an open class such that $\mathcal{B}_{\mathcal{R}}\Delta\mathcal{U}$ is meager. In particular, $\mathcal{U} \setminus \mathcal{B}_{\mathcal{R}}$ is meager, so $\mathcal{U} \subseteq [U_{\mathcal{R}}]$. Also, $\mathcal{B}_{\mathcal{R}} \setminus \mathcal{U} = (2^{\mathbb{N}} \setminus \mathcal{U}) \setminus \mathcal{B}_{\neg\mathcal{R}}$ is meager, so $\text{int}(2^{\mathbb{N}} \setminus \mathcal{U}) \setminus \mathcal{B}_{\neg\mathcal{R}}$ is meager. It follows that $\text{int}(2^{\mathbb{N}} \setminus \mathcal{U}) \subseteq [U_{\neg\mathcal{R}}]$. Since $\mathcal{U} \cup \text{int}(2^{\mathbb{N}} \setminus \mathcal{U})$ is dense in $2^{\mathbb{N}}$, $[U_{\mathcal{R}}] \cup [U_{\neg\mathcal{R}}]$ is also dense, so by Exercise 2.8, $U_{\mathcal{R}} \cup U_{\neg\mathcal{R}}$ is dense in $2^{<\mathbb{N}}$.

(2) $\Rightarrow$ (1). Suppose that $U_{\mathcal{R}} \cup U_{\neg\mathcal{R}}$ is dense in $2^{<\mathbb{N}}$. By density, the complement of $[U_{\mathcal{R}}] \cup [U_{\neg\mathcal{R}}]$ is a closed class of empty interior, therefore meager. Let us show that $\mathcal{B}_R \Delta [U_{\mathcal{R}}]$ is meager. By definition, for all $\sigma \in U_{\mathcal{R}}$, $[\sigma] \setminus \mathcal{B}_R$ is meager, so $[U_{\mathcal{R}}] \setminus \mathcal{B}_R$ is a countable union of meager classes, so is meager. By the same reasoning, if $\mathcal{B}_{\neg\mathcal{R}}$ is the class of elements which satisfy $\neg\mathcal{R}$, $[U_{\neg\mathcal{R}}] \setminus \mathcal{B}_{\neg\mathcal{R}}$ is meager. As $[U_{\neg\mathcal{R}}] \setminus \mathcal{B}_{\neg\mathcal{R}} = \mathcal{B}_R \cap [U_{\neg\mathcal{R}}]$, and since the complement of $[U_{\mathcal{R}}] \cup [U_{\neg\mathcal{R}}]$ is meager, then $\mathcal{B}_R \setminus [U_{\mathcal{R}}]$ is meager. So $\mathcal{B}_R \Delta [U_{\mathcal{R}}]$ is meager. ∎

Since the syntactic forcing relation implies the semantic relation, we can deduce the density of the semantic forcing relation for arithmetic requirements:

Lemma 4.12. Let $\mathcal{R}$ be an arithmetic requirement. The set

$$\{\sigma \in 2^{<\mathbb{N}} : \sigma \Vdash \mathcal{R} \vee \sigma \Vdash \neg\mathcal{R}\}$$

is dense. ⋆

Proof. For all σ there exists an extension $\tau \succeq \sigma$ such that $\tau \Vdash^* \mathcal{R}$ or such that $\tau \Vdash^* \neg\mathcal{R}$. If $\tau \Vdash^* \mathcal{R}$ then $\tau \Vdash \mathcal{R}$ and if $\tau \Vdash^* \neg\mathcal{R}$ then $\tau \Vdash \neg\mathcal{R}$. ∎

The following corollary follows directly from the characterization of the classes having the Baire property.

Corollary 4.13
Let $\mathcal{R}$ is an arithmetic requirement. The class $\mathcal{B}_{\mathcal{R}}$ of elements which satisfy $\mathcal{R}$, has the Baire property.

Proof. Immediate by Lemma 4.12 and Proposition 4.11. ∎

We will develop in Chapter 17 and Section 27-3.1 the theory of Borel classes, which formalizes and generalizes the notion of arithmetic requirements, and which all have the Baire property.

Let us end this section by clarifying the links between the semantic forcing relation and the syntactic relation $\Vdash^*$ that we defined in the previous section:

Theorem 4.14
Let $\mathcal{R}$ be an arithmetic requirement. Then $\sigma \Vdash \mathcal{R}$ iff $\{\tau \in 2^{<\mathbb{N}} : \tau \Vdash^* \mathcal{R}\}$ is dense below σ.

To show the theorem we will use two lemmas. The first corresponds to Proposition 4.6 for the relation $\Vdash^*$.

Lemma 4.15. Suppose $\sigma \Vdash \mathcal{R}$ for a string σ and a requirement $\mathcal{R}$. If $\tau \succeq \sigma$ then $\tau \Vdash \mathcal{R}$. ⋆

Proof. Let $\mathcal{B}_{\mathcal{R}}$ be the class of elements which satisfy $\mathcal{R}$. We have $[\tau] \setminus \mathcal{B}_{\mathcal{R}} \subseteq [\sigma] \setminus \mathcal{B}_{\mathcal{R}}$. So if $[\sigma] \setminus \mathcal{B}_{\mathcal{R}}$ is meager then $[\tau] \setminus \mathcal{B}_{\mathcal{R}}$ is meager. ■

Lemma 4.16. Suppose $\sigma \Vdash \mathcal{R}$ for a string σ and a requirement $\mathcal{R}$. Then, $\sigma \nVdash \neg\mathcal{R}$ ⋆

Proof. Let $\mathcal{B}_{\mathcal{R}}$ be the class of elements which satisfy $\mathcal{R}$ and Let $\mathcal{B}_{\neg\mathcal{R}}$ be the class of elements which satisfy $\neg\mathcal{R}$.

Let us assume by the absurdity $\sigma \Vdash \mathcal{R}$ and $\sigma \Vdash \neg\mathcal{R}$. Then, $[\sigma] \setminus (\mathcal{B}_{\mathcal{R}} \cap [\sigma])$ and $[\sigma] \setminus (\mathcal{B}_{\neg\mathcal{R}} \cap [\sigma])$ are both meager. Moreover $([\sigma] \setminus (\mathcal{B}_{\mathcal{R}} \cap [\sigma])) \cup ([\sigma] \setminus (\mathcal{B}_{\neg\mathcal{R}} \cap [\sigma])) = [\sigma]$, which contradicts the fact that the union of two meager classes is meager and therefore of empty interior. ■

Proof of Theorem 4.14. Recall Proposition 4.8: for any arithmetic requirement $\mathcal{R}$ and for all σ, if $\sigma \Vdash^* \mathcal{R}$ then $\sigma \Vdash \mathcal{R}$.

Let us show that if $\{\tau \in 2^{<\mathbb{N}} : \tau \Vdash^* \mathcal{R}\}$ is dense below σ, then $\sigma \Vdash \mathcal{R}$. According to Proposition 4.8, $A = \{\tau \in 2^{<\mathbb{N}} : \tau \Vdash \mathcal{R}\}$ is dense below σ. So the class $\{X \in [\sigma] : \forall n\ X \upharpoonright_n \nVdash \mathcal{R}\}$ is therefore a closed class of empty interior, which we denote by $\mathcal{F}$. Let $\mathcal{B}_{\mathcal{R}}$ be the class of elements which satisfy $\mathcal{R}$. By definition of A, for all $\tau \in A$, the class $\mathcal{M}_\tau = [\tau] \setminus (\mathcal{B}_{\mathcal{R}} \cap [\tau])$ is meager. It follows that the class $[\sigma] \setminus (\mathcal{B}_{\mathcal{R}} \cap [\sigma])$ is included in the union of $\mathcal{F}$ and all the classes $\mathcal{M}_\tau$. It is therefore meager.

Finally, let us show that if $\sigma \Vdash \mathcal{R}$ then $\{\tau \in 2^{<\mathbb{N}} : \tau \Vdash^* \mathcal{R}\}$ is dense below σ. By contraposition, let us suppose that there exists $\tau \succeq \sigma$ such that for all $\rho \succeq \tau$ we have $\rho \nVdash^* \mathcal{R}$. By Proposition 4.7, there must then exist $\rho \succeq \tau$ such that $\rho \Vdash^* \neg\mathcal{R}$. By Proposition 4.8, we then have $\rho \Vdash \neg\mathcal{R}$. According to Lemma 4.16, $\rho \nVdash \mathcal{R}$ and therefore according to Lemma 4.15, $\sigma \nVdash \mathcal{R}$. ■

5. Arbitrarily generic sets

Genericity can be seen as a notion of "typicality", in the sense that anything that can happen infinitely often will end up happening. In the case of Cohen forcing, a dense set has infinitely often the possibility of being met, and if a set is typical this will happen, which corresponds to the notion of generic set.

We have seen relativizations of weakly 1-generic and 1-generic sets, and in particular the concepts of n-generic and weakly n-generic set. In this section, we study the properties of typical sets, that is, the properties that sufficiently generic sets will have.

Note that if a property is verified by any sufficiently generic set, it is without loss of generality verified for any weakly 1-generic set relative to A for a certain oracle A sufficiently powerful: it suffices that A encodes the intersection of dense open classes corresponding to the required level of genericity. We will therefore endeavor to determine "the right level" of genericity necessary to satisfy such and such a property. In practice, this will often correspond to being n-generic for some n.

5.1. Properties of sufficiently generic sets

We have seen that 1-genericity was the level of genericity corresponding to the forcing of Σ^0_1/Π^0_1 requirements. Unsurprisingly, we now see that the n-genericity is the level of genericity corresponding to the forcing of Σ^0_n/Π^0_n requirements, and we will show the following theorem.

Theorem 5.1
Let G be an n-generic set. Let $\mathcal{R}$ be a Σ^0_n or Π^0_n requirement. Then, G satisfies $\mathcal{R}$ iff there is a prefix $\sigma \prec G$ such that $\sigma \Vdash^* \mathcal{R}$.

Let us note that this iteration is not trivial: we need to appeal for that to the developments of the forcing relation of the preceding sections.

We will now proceed to the proof of Theorem 5.1 by exploiting the definitional simplicity of the syntactic forcing relation. Let's start with the following proposition.

Proposition 5.2. Let $\mathcal{R}$ be an arithmetic requirement.

(1) If $\mathcal{R}$ is Σ^0_n, then the predicate $\sigma \Vdash^* \mathcal{R}$ is Σ^0_n

(2) If $\mathcal{R}$ is Π^0_n, then the predicate $\sigma \Vdash^* \mathcal{R}$ is Π^0_n ⋆

Proof. We will prove (1) and (2) simultaneously by induction on the arithmetic complexity of the requirement. The Σ^0_1 and Π^0_1 case has already been treated with Proposition 3.17.

If $\mathcal{R}$ is Σ^0_{m+1} with $m > 0$, then it can be expressed in the form $\exists x \mathcal{S}(x)$ where $\mathcal{S}$ is a Π^0_m requirement. We have $\sigma \Vdash^* \mathcal{R}$ iff there exists an $n \in \mathbb{N}$ such $\sigma \Vdash^* \mathcal{S}(n)$. By induction hypothesis, the predicate $\sigma \Vdash^* \mathcal{S}(n)$ is Π^0_m, so the predicate $\sigma \Vdash^* \mathcal{R}$ is Σ^0_{m+1}.

If $\mathcal{R}$ is Π^0_{m+1} with $m > 0$, then it can be expressed in the form $\forall x \mathcal{S}(x)$ where $\mathcal{S}$ is a Σ^0_m requirement. We have $\sigma \Vdash^* \mathcal{R}$ iff for all $\tau \succeq \sigma$ and all $n \in \mathbb{N}$, $\tau \nVdash^* \neg\mathcal{S}(n)$. In particular, $\neg\mathcal{S}(n)$ is Π^0_m, so by induction hypothesis, the predicate $\tau \Vdash^* \neg\mathcal{S}(n)$ is Π^0_m, therefore $\tau \nVdash^* \neg\mathcal{S}(n)$ is Σ^0_m, and the predicate $\sigma \Vdash^* \mathcal{R}$ is Π^0_{m+1}. ■

We now see the level of genericity required to make Proposition 4.8 true.

Proposition 5.3. Let $\mathcal{R}$ be an arithmetic requirement such that $\sigma \Vdash^* \mathcal{R}$ for $\sigma \in 2^{<\mathbb{N}}$.

(1) If $\mathcal{R}$ is Σ^0_n, then $\mathcal{R}$ is satisfied for all weakly $(n-2)$-generic sets which extend σ.

(2) If $\mathcal{R}$ is Π^0_n, then $\mathcal{R}$ is satisfied for all weakly $(n-1)$-generic sets which extend σ. ⋆

Proof. By definition of the forcing relation, if $\mathcal{R}$ is Σ^0_1 or Π^0_1 then it is satisfied for any set which extends σ. Suppose the proposition is true for n. Let $\mathcal{R} = \exists x \mathcal{Q}(x)$ be a Σ^0_{n+1} requirement with $\mathcal{Q}(x)$ a Π^0_n requirement. We have $\sigma \Vdash^* \mathcal{R}$ iff there exists $m \in \mathbb{N}$ such that $\sigma \Vdash^* \mathcal{Q}(m)$. By induction hypothesis $\mathcal{Q}(m)$ is satisfied for any weakly $(n-1)$-generic set which extends σ and therefore it is the same for $\mathcal{R}$.

Suppose now that $\mathcal{R} = \forall x \mathcal{Q}(x)$ is a Π^0_{n+1} requirement with $\mathcal{Q}(x)$ a Σ^0_n requirement. We have $\sigma \Vdash^* \mathcal{R}$ iff for all $m \in \mathbb{N}$ and for all $\tau \succeq \sigma$ $\tau \nVdash^* \neg Q(m)$. According to Proposition 4.7 this means that for m fixed, the set $A_m = \{\tau : \tau \Vdash^* Q(m)\}$ is dense below σ. According to Proposition 5.2 each set A_m is Σ^0_n. In particular if G is weakly n-generic and extends σ then G meets A_m. By induction hypothesis if G meets A_m and is weakly n-generic then $Q(m)$ is true for G. As is the case for all m then $\mathcal{R}$ is true for weakly n-generic set G which extends σ. ■

We can finally show Theorem 5.1.

Proof of Theorem 5.1. The Σ^0_1/Π^0_1 case has already been treated with Theorem 3.12. Let $n > 1$. We can assume without loss of generality that $\mathcal{R}$ is Σ^0_n. In the opposite case, we reproduce the following argument with $\neg\mathcal{R}$ instead of $\mathcal{R}$.

Let $U = \{\sigma \in 2^{<\mathbb{N}} : \sigma \Vdash^* \mathcal{R}\}$. According to Lemma 4.5, $U^\perp = \{\sigma \in 2^{<\mathbb{N}} : \sigma \Vdash^* \neg\mathcal{R}\}$. According to Proposition 5.2, the set U is Σ^0_n and the set $U^\perp$ is Π^0_n. Note that as G is n-generic, it meets $U \cup U^\perp$.

Suppose that $\mathcal{R}$ is satisfied for G. Then, G cannot meet $U^\perp$ because in this case we would have a prefix $\sigma \prec G$ such that $\sigma \Vdash^* \neg\mathcal{R}$ and by Proposition 5.3, $\neg\mathcal{R}$ would therefore be satisfied on G, which contradicts the fact that $\mathcal{R}$ is satisfied on G. So G meets U and we have a prefix $\sigma \prec G$ such that $\sigma \Vdash^* \mathcal{R}$.

Suppose that $\neg\mathcal{R}$ is satisfied for G. Symmetrically, G meets necessarily $U^{\perp}$ and we have a prefix $\sigma \prec G$ such that $\sigma \Vdash^* \mathcal{R}$.

Conversely if a prefix $\sigma \prec G$ is such that $\sigma \Vdash^* \mathcal{R}$ or $\sigma \Vdash^* \neg\mathcal{R}$ then respectively $\mathcal{R}$ or $\neg\mathcal{R}$ is satisfied on G according to Proposition 5.3. ■

5.2. Turing degree of sufficiently generic sets

Let us first recap a result that we have already established: given a set A, if G is sufficiently generic then A does not compute G and G does not compute A.

Theorem 5.4
Let $A \in 2^{\mathbb{N}}$ be a non-computable set. If G is 1-generic relative to A, then A and G are of incomparable Turing degrees.

Proof. According to Exercise 3.5 and Exercise 3.6, if G is weakly 1-generic relative to A, it is of degree A-hyperimmune and therefore cannot be computed by A. According to Theorem 3.28, if G is 1-generic relative to A, it does not compute A. ■

Note that Theorem 5.4 can be generalized to show that for any countable sequence of sets fixed in advance $A_0, A_1, \dots$, any set G sufficiently generic for Cohen forcing is incomparable with the items from this list. Indeed if G is sufficiently generic, it will be 1-generic relative to A_i for all i. This result, despite its simplicity, admits of several interesting consequences:

Corollary 5.5
Let G be a sufficiently generic set for Cohen forcing. Then, G is not arithmetic and does not compute any non-computable arithmetic set.

Let's now see how to reinforce and iterate Theorem 5.4: for A not $\emptyset^{(n)}$-computable, if a set G is sufficiently generic, not only will it not compute A, but also the n-th Turing jump will not compute A. We will actually see something a little more precise: if A is not Σ^0_n and if G is sufficiently generic, then A will not be $\Sigma^0_n(G)$. Let us first make sure of the complexity of the requirement $a \in G^{(n)}$ via the following lemma.

Lemma 5.6. For all $a \in \mathbb{N}$, the requirement $a \in X^{(n)}$ is Σ^0_n uniformly in a. ⋆

Proof. For the case $n = 1$ we have $a \in X'$ iff $\Phi_a(X, a)\downarrow$, which is indeed a Σ^0_1 requirement. In the case n, suppose that $F_n(G, x)$ is a Σ^0_n formula of second-order arithmetic such that for all $X \in 2^{\mathbb{N}}$ and for all $a \in \mathbb{N}$, $F(X, a)$

is true iff $a \in X^{(n)}$. We then have:

$$\begin{aligned} a \in X^{(n+1)} &\leftrightarrow \exists \sigma\ \forall i < |\sigma| \left(\begin{array}{cl} & (\sigma(i) = 0 \wedge i \notin X^{(n)}) \\ \vee & (\sigma(i) = 1 \wedge i \in X^{(n)}) \end{array} \right) \wedge \Phi_a(\sigma, a)\downarrow \\ &\leftrightarrow \exists \sigma\ \forall i < |\sigma| \left(\begin{array}{cl} & (\sigma(i) = 0 \wedge \neg F_n(X, i)) \\ \vee & (\sigma(i) = 1 \wedge F_n(X, i)) \end{array} \right) \wedge \Phi_a(\sigma, a)\downarrow . \end{aligned}$$

By using the fact that $\neg F_n(G, x)$ and $F_n(G, x)$ are both Σ^0_{n+1} formulas, and using the closure of the Σ^0_{n+1} formulas under bounded quantification, finite union and finite intersection, we can define a Σ^0_{n+1} formula $H(G, \tau)$ such that

$$H(X, \sigma) \leftrightarrow \forall i < |\sigma|\ ((\sigma(i) = 0 \wedge \neg F_n(X, i)) \vee (\sigma(i) = 1 \wedge F_n(X, i))).$$

Then,

$$a \in X^{(n+1)} \leftrightarrow \exists \sigma\ H(X, \sigma)\ \wedge \Phi_a(\sigma, a)\downarrow,$$

which is indeed a Σ^0_{n+1} formula. ■

Theorem 5.7
Let A be a non Σ^0_{n+1} set for $n \geqslant 0$. Then, for any 1-generic set G relative to $A \oplus \emptyset^{(n)}$, A is not $\Sigma^0_{n+1}(G)$.

Proof. Let G be a 1-generic set relative to $A \oplus \emptyset^{(n+1)}$. We must show that for any e, the set A is different from $W_e^{G^{(n)}}$, the set $G^{(n)}$-c.e. code e. Let

$$U = \{\sigma : \exists m \notin A\ \sigma \Vdash^* m \in W_e^{G^{(n)}}\}.$$

According to Lemma 5.6, the requirement $m \in W_e^{G^{(n)}}$ is Σ^0_{n+1}. So according to Proposition 5.2, the set U is $\Sigma^0_1(A \oplus \emptyset^{(n)})$. If G meets U then according to Proposition 5.3, we will have $m \in W_e^{G^{(n)}}$ for $m \notin A$ and therefore $A \neq W_e^{G^{(n)}}$.

If G does not meet U, since G is 1-generic relative to $A \oplus \emptyset^{(n+1)}$, G meets $U^\perp = \{\sigma : \forall m \notin A\ \forall \tau \succeq \sigma\ \tau \nVdash^* m \in W_e^{G^{(n)}}\}$. According to Lemma 4.5, we have $U^\perp = \{\sigma : \forall m \notin A\ \sigma \Vdash^* m \notin W_e^{G^{(n)}}\}$. Let $\sigma \in U^\perp$ be a fixed string. Then, the set $D_\sigma = \{m : \sigma \Vdash^* m \notin W_e^{G^{(n)}}\}$ is a Π^0_{n+1} set which by hypothesis contains $\mathbb{N} \setminus A$. As $\mathbb{N} \setminus A$ is not Π^0_{n+1}, there is necessarily an element $m \in A$ such that $m \in D_\sigma$. So if G meets $U^\perp$ it also meets the set $\{\sigma : \exists m \in A\ \sigma \Vdash^* m \notin W_e^{G^{(n)}}\}$. Then, according to Proposition 5.3, we will have $m \notin W_e^{G^{(n)}}$ for $m \in A$ and therefore $A \neq W_e^{G^{(n)}}$. ■

Corollary 5.8
Let A be a non-$\emptyset^{(n)}$-computable set for $n \geqslant 0$. Then, if G is 1-generic relative to $A \oplus \emptyset^{(n)}$, the set A is not $G^{(n)}$-computable.

Proof. As A is not $\emptyset^{(n)}$-computable it is not Δ^0_{n+1}. Since A is not Δ^0_{n+1}, either A is not Σ^0_{n+1}, or $\overline{A}$ is not Σ^0_{n+1}. By Theorem 5.7, for any 1-generic set G relative to $A \oplus \emptyset^{(n)}$, A is not $\Sigma^0_{n+1}(G)$ in the first case, and $\overline{A}$ is not $\Sigma^0_{n+1}(G)$ in the second case. In any case, A is not $\Delta^0_{n+1}(G)$ and therefore it is not $G^{(n)}$-computable. ∎

We saw with Theorem 3.20 that the Turing jump of 1-generic sets was a continuous function on the class of 1-generic sets: the 1-generics are all generalized low. This result can be put into perspective: the n-th Turing jump is a continuous function on the class of n-generic sets:

Theorem 5.9
Let G be an n-generic set. Then, $G^{(n)} \leqslant_T G \oplus \emptyset^{(n)}$.

Proof. Let $n > 0$ and G be an n-generic set. Let $U_e = \{\sigma : \sigma \Vdash^* e \in G^{(n)}\}$. According to Lemma 5.6, the requirement $e \in G^{(n)}$ is Σ^0_n and therefore U_e is a Σ^0_n set. Consider $U_e^\perp = \{\sigma : \forall \tau \succeq \sigma\ \tau \nVdash^* e \in G^{(n)}\}$. In particular, $U_e^\perp$ is a Π^0_n set. According to Lemma 4.5, we have $U_e^\perp = \{\sigma : \sigma \Vdash^* e \notin G^{(n)}\}$. As G is n-generic, it meets $U_e \cup U_e^\perp$. It suffices using $\emptyset^{(n)}$ to search for a prefix of G in U_e or $U_e^\perp$. According to Proposition 5.3, in the first case we have $e \in G^{(n)}$ and in the second $e \notin G^{(n)}$. ∎

Note that Corollary 5.8 is also deduced from the previous theorem and from a relativized version of Theorem 3.28. Beware, in the literature, the notion of generalized low$_n$ does not correspond to the property of theorem 5.9.

Definition 5.10. A set $G \in 2^{\mathbb{N}}$ is *generalized low$_n$* if $G^{(n)} \leqslant_T (G \oplus \emptyset')^{(n-1)}$. ♢

Note that if X is generalized low$_n$ it is also generalized low$_{n+1}$. Each n-generic set is indeed generalized low$_n$, but Theorem 5.9 proves something stronger.

Chapter 11

Effective forcing

On the strength of the intuitions created with the study of Cohen forcing, we can then introduce the abstract forcing notions on an arbitrary partial order, and develop all the associated machinery.

1. Fundamentals of forcing

Now that we have familiarized ourselves with the notions of density and genericity on the partial order of binary strings, equipped with the extension relation, we are ready to approach the concepts of forcing in all their generality, while keeping the intuitions of the finite extension method. The benefits of this abstraction will only appear from the introduction of the forcing relation, which gives all its power to the formalism. So far, we have seen the notion of genericity in the partial order of binary strings as a systematization of constructions with the finite extension method.

Partial order. In all its generality, a forcing notion is quite simply a partial order $(\mathbb{P}, \leqslant)$, whose elements are called *conditions*. A condition intuitively represents an approximation of the object that we are constructing. A *extension* of $c \in \mathbb{P}$ is a condition $d \leqslant c$.

Remark

Beware, for historical reasons, the order relation of the forcing is reversed. An extension of a condition c is therefore smaller condition d. The underlying idea comes from the fact that d is a more precise approximation than c, and that therefore the set of "candidate" objects that we are building is a subset of the candidates of c, because the more constraints we add, the more candidates we exclude.

Filter. In the case of the finite extension method, the constructed object is an element of $2^{\mathbb{N}}$, using a strictly increasing infinite sequence of strings. In the language of an arbitrary partial order, we are going to construct an infinite decreasing sequence of conditions. The produced object is a maximal filter.

Definition 1.1. Two conditions c_0, c_1 are *compatible* if there is a condition d which extends both c_0 and c_1. Otherwise, c_0 and c_1 are *incompatible*. A *filter* is an upward-closed set $F \subseteq \mathbb{P}$, such that for any $c_0, c_1 \in F$, there exists a condition $d \in F$ such that $d \leqslant c_0, c_1$. A filter is *maximal* if it is not included in a strictly larger filter. ♢

If we consider the partial order on the strings, equipped with the suffix relation, the maximal filters are in one-to-one correspondence with Cantor space. Indeed, for any $X \in 2^{\mathbb{N}}$, the set $\{X \upharpoonright_n : n \in \mathbb{N}\}$ is a maximal filter, and conversely, any maximal filter is of this form.

In the context of a countable partial order, it may be more intuitive to think of a filter as the upward closure of an infinite decreasing sequence of conditions. In particular, for any decreasing infinite sequence of conditions $c_0 \geqslant c_1 \geqslant c_2 \geqslant \dots$, the set

$$F = \{d \in \mathbb{P} : \exists n\ c_n \leqslant d\}$$

is a filter. This intuition corresponds more to the construction of the finite extension method.

Notation

As explained, a condition $c \in \mathbb{P}$ can be seen as an approximation of the object being constructed, namely a maximal filter. We can therefore associate with each condition the set $[c]_\in$ of maximum filters containing c, representing the candidate objects. In particular, according to intuition, if $d \leqslant c$, then the approximation d is more precise than the approximation c, thus reducing the number of candidates. We therefore have $[d]_\in \subseteq [c]_\in$.

Note that for any maximal filter F, $\bigcap_{c \in F} [c]_\in = \{F\}$. In other words, F is the unique candidate of all the conditions of the filter simultaneously. We have already encountered several forcing notions in the previous chapters. Here are a few. We will see that for each of these forcing notions, the maximum filters can be interpreted as sets of integers.

Example 1.2.

1. *Cohen forcing* is the partial order of strings equipped with its suffix relation $(2^{<\mathbb{N}}, \succeq)$. For any $\sigma \in 2^{<\mathbb{N}}$, $[\sigma]_\in$ is in bijection with the set $[\sigma] = \{X \in 2^{\mathbb{N}} : \sigma \prec X\}$, by the function which to $F \in [\sigma]_\in$ associates the unique element of $\bigcap_{\sigma \in F}[\sigma]$.

2. *Jockusch-Soare forcing* is the partial order of non-empty Π^0_1 classes, ordered by the inclusion relation. For any Π^0_1 class $\mathcal{P}$, the set $[\mathcal{P}]_\in$ is in bijection with $\mathcal{P}$. A maximal filter F containing $\mathcal{P}$ can therefore be seen as the only element of $\bigcap_{\mathcal{Q} \in F} \mathcal{Q}$, which is a member of $\mathcal{P}$.

3. *Sacks forcing* is the partial order of computable f-trees, ordered by the sub-f-tree relation. For any f-computable tree T, the set $[T]_\in$ is in bijection with the set

$$[T] = \{\bigcup_n T(X\restriction_n) : X \in 2^{\mathbb{N}}\} = \{Y \in 2^{\mathbb{N}} : \exists^\infty n\; Y\restriction_n \in \operatorname{Im} T\}$$

In each of the preceding examples, for any maximal filter F, we will denote by $\dot{F}$ the corresponding element of $2^{\mathbb{N}}$. We therefore have for each of these forcing notions the equality $[c] = \{\dot{F} : F \in [c]_\in\}$.

Density, genericity. Recall that in the finite extension method, requirements can be represented as the set of strings that force them. This representation has the advantage of being abstracted from the notion of requirement and is generalized to any partial order.

Definition 1.3. Let $(\mathbb{P}, \leqslant)$ be a partial order. A set $D \subseteq \mathbb{P}$ is *dense* in $(\mathbb{P}, \leqslant)$ if for all $c \in \mathbb{P}$, there exists a $d \leqslant c$ such that $d \in D$. ♢

The intuition that it is useful to keep with the notion of density is that if a set D is dense, then whatever the finite piece of the decreasing sequence of conditions that we have already constructed, it will never be too late to integrate an element of D in the sequence.

Definition 1.4. Let $\vec{D} = (D_n)_{n \in \mathbb{N}}$ be a collection of sets of conditions. A filter $F \subseteq \mathbb{P}$ is *$\vec{D}$-generic* if it intersects D_n for all $n \in \mathbb{N}$. ♢

The following proposition shows that if the sets of conditions are dense, there is a generic filter for these sets. The proof of this proposition corresponds to the construction, by the finite extension method, of an increasing infinite sequences of strings satisfying each requirement.

Proposition 1.5. Let $\vec{D} = (D_n)_{n\in\mathbb{N}}$ be a countable collection of dense sets, and $c \in \mathbb{P}$ a condition. There is a $\vec{D}$-generic filter containing c. ⋆

Proof. Let us define an infinite decreasing sequence of conditions $c_0 \geqslant c_1 \geqslant c_2 \geqslant \dots$ inductively as follows: $c_0 = c$. If c_n is defined, $c_{n+1} \leqslant c_n$ is a condition belonging to D_n. Such an extension of c_n exists by density of the set D_n. Let $F = \{d \in \mathbb{P} : \exists n\ d \geqslant c_n\}$. The set F is a filter containing c and meeting D_n for all n. ■

As explained in Section 10-2, there is in general an uncountable quantity of dense sets, and a filter F cannot be generic for all these sets simultaneously. The notion of genericity is therefore dependent on a countable collection $\vec{D}$ of dense sets. We will say that any *sufficiently generic* filter for a forcing notion satisfies such property if there exists a countable collection of dense sets $\vec{D}$ such that any filter $\vec{D}$-generic satisfies this property.

Definition 1.6. Let $(\mathbb{P}, \leqslant)$ be a partial order, $c \in \mathbb{P}$ and let $D \subseteq \mathbb{P}$ be a set. We say that D is *dense below* c if for all $d \leqslant c$, there exists an $e \leqslant d$ such that $e \in D$. ♢

The following exercise will be useful in the rest of this development.

Exercise 1.7. Suppose that a set $D \subseteq \mathbb{P}$ is dense below $c \in \mathbb{P}$. Show that any sufficiently generic filter F containing c intersects D. ⋄

2. Forcing relation

So far, we have developed notions expressed purely in terms of partial order, namely, the notions of density, filter and genericity. We will now define a generalization of the forcing relation introduced in Section 10-4. For that, we will restrict ourselves to forcing notions producing sets of integers.

Definition 2.1. A *Cantor forcing* is a partial order $(\mathbb{P}, \leqslant)$ equipped with a function $F \mapsto \dot{F}$ from maximal filters towards Cantor space $2^{\mathbb{N}}$, such for all $c \in \mathbb{P}$, and all $\sigma \in 2^{<\mathbb{N}}$ for which $[c] \cap [\sigma] \neq \emptyset$, there exists a condition $d \leqslant c$ such that $[d] \subseteq [\sigma]$. Here, $[c]$ denotes the set $\{\dot{F} : F \in [c]_{\in}\}$. ♢

2.1. Semantic forcing relation

By abuse of language, we will say that a set $G \in 2^{\mathbb{N}}$ is *sufficiently generic* for a Cantor forcing if it is of the form $\dot{F}$ for a sufficiently generic filter F.

Definition 2.2. A condition c *semantically forces* a requirement $\mathcal{R}$, in which case we write $c \Vdash \mathcal{R}$ if $\dot{F}$ satisfies $\mathcal{R}$ for any maximal filter F sufficiently generic and containing c. ♢

This semantic definition gives us "for free" some properties of the relation:

Proposition 2.3. Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing, $c, d \in \mathbb{P}$ and $\mathcal{R}$ a requirement.

(1) If $c \Vdash \mathcal{R}$ and $d \leqslant c$, then $d \Vdash \mathcal{R}$.

(2) If $c \Vdash \mathcal{R}$, then $c \nVdash \neg\mathcal{R}$. ⋆

Proof. (1) Let F be a sufficiently generic filter containing d. By upward-closure of the filters, $c \in F$, so as c forces $\mathcal{R}$, $\dot{F}$ satisfies $\mathcal{R}$. It follows that d forces $\mathcal{R}$.

(2) If c forces $\mathcal{R}$ and $\neg\mathcal{R}$, then for any sufficiently generic filter F containing c, $\dot{F}$ satisfies $\mathcal{R}$ and $\neg\mathcal{R}$, contradiction. ∎

Exercise 2.4. Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing, $c \in \mathbb{P}$ and $\mathcal{R}$ be a requirement. Show that if the set $\{d \in \mathbb{P} : d \Vdash \mathcal{R}\}$ is dense below c, then $c \Vdash \mathcal{R}$. ◇

On the other hand, some properties are much less obvious to prove. We will see for example that for any arithmetic requirement $\mathcal{R}$, the set of conditions which force $\mathcal{R}$ or which force $\neg\mathcal{R}$ is dense. As in the case of Cohen forcing with the partial order of the strings $(2^{<\mathbb{N}}, \succeq)$, we will define a syntactic forcing relation which will allow us to account for this phenomenon.

We first insist on the three fundamental properties that a syntactic forcing relation must have.

Definition 2.5 (Forcing relation). Given a Cantor forcing $(\mathbb{P}, \leqslant)$, a relation $\Vdash^\circ$ is a *forcing relation* if it satisfies the following properties for all $c, d \in \mathbb{P}$ and any arithmetic requirement $\mathcal{R}$.

(1) If $c \Vdash^\circ \mathcal{R}$ then $c \Vdash \mathcal{R}$.

(2) If $c \Vdash^\circ \mathcal{R}$ and $d \leqslant c$, then $d \Vdash^\circ \mathcal{R}$.

(3) The set $\{c \in \mathbb{P} : c \Vdash^\circ \mathcal{R} \text{ or } c \Vdash^\circ \neg\mathcal{R}\}$ is dense. ♢

Properties (1-3) correspond to propositions 10-4.8, 10-4.6 and 10-4.7 for Cohen forcing. It follows immediately from (1) that if $c \Vdash^\circ \mathcal{R}$, then $c \nVdash^\circ \neg\mathcal{R}$. Recall that a set $S \subseteq \mathbb{P}$ is dense below $c \in \mathbb{P}$ if for all $d \leqslant c$, there exists an $e \leqslant d$ such that $e \in S$.

Proposition 2.6. Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing, $c \in \mathbb{P}$ and $\mathcal{R}$ an arithmetic requirement. Let $\Vdash^\circ$ be a forcing relation. Then $c \Vdash \mathcal{R}$ iff $\{d \in \mathbb{P} : d \Vdash^\circ \mathcal{R}\}$ is dense below c. ⋆

Proof. Suppose that $c \Vdash^\circ \mathcal{R}$. Let $d \leqslant c$. By Definition 2.5 (3), there exists an $e \leqslant c$ such that $e \Vdash^\circ \mathcal{R}$ or $e \Vdash^\circ \neg\mathcal{R}$. If the second case arises, then by Definition 2.5 (1), $e \Vdash \neg\mathcal{R}$. So we have $e \Vdash \mathcal{R}$ and $e \Vdash \neg\mathcal{R}$, which contradicts Theorem 2.3 (2). The first case therefore arises. We have shown the density of the set $\{d \in \mathbb{P} : d \Vdash^\circ \mathcal{R}\}$ under c.

Conversely, suppose that the set $\{d \in \mathbb{P} : d \Vdash^\circ \mathcal{R}\}$ is dense below c. Let F be a sufficiently generic filter containing c. By genericity, there exists $d \in F$ such that $d \Vdash^\circ \mathcal{R}$, and by Definition 2.5 (1), $d \Vdash \mathcal{R}$, so $\dot{F}$ satisfies $\mathcal{R}$. It follows that $c \Vdash \mathcal{R}$. ■

2.2. Syntactic forcing relation

We now define our syntactic forcing relation $\Vdash^*$, which directly generalizes the relation $\Vdash^*$ defined in Section 10-4 for Cohen forcing. As for Cohen forcing, the interest of the relation $c \Vdash^* \mathcal{R}$ is that it is simple to define: relative to $\mathbb{P}$, it has the same arithmetic complexity as that of the requirement $\mathcal{R}$. We will see in the following sections that $\mathbb{P}$ as a partial order is unfortunately rarely computable, which leads to additional computational complexities which must be dealt with on a case-by-case basis.

Definition 2.7. Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing. We define the relation $\Vdash^*$ for all $c \in \mathbb{P}$ and any arithmetic requirement $\mathcal{R}$:

(1) $c \Vdash^* \mathcal{R}$ for a Σ^0_1 or Π^0_1 requirement $\mathcal{R}$ iff all $X \in [c]$ satisfy $\mathcal{R}$.

(2) $c \Vdash^* \exists x \mathcal{R}(x)$ for a Π^0_k requirement $\mathcal{R}(x)$ with $k \geqslant 1$ iff there is an $n \in \mathbb{N}$ such that $c \Vdash^* \mathcal{R}(n)$.

(3) $c \Vdash^* \forall x \mathcal{R}(x)$ for a Σ^0_k requirement $\mathcal{R}(x)$ with $k \geqslant 1$ iff for all $d \leqslant c$ and all $n \in \mathbb{N}$, $d \nVdash^* \neg\mathcal{R}(n)$. ♢

Note that just as in the case of Cohen forcing, we have $c \Vdash^* \forall x \mathcal{R}(x)$ iff $\forall d \leqslant c$ $d \nVdash^* \exists x \neg\mathcal{R}(x)$. We leave as an exercise the proof that the relation $\Vdash^*$ respects the items (1-3) of Definition 2.5. Each time, it is a simple proof by induction on the complexity of the requirements.

Exercise 2.8. (⋆) Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing, $c, d \in \mathbb{P}$ and $\mathcal{R}$ be an arithmetic requirement. Show that if $c \Vdash^* \mathcal{R}$ and $d \leqslant c$, then $d \Vdash^* \mathcal{R}$. ⋄

Exercise 2.9. ($\star$) Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing and let $\mathcal{R}$ be an arithmetical requirement. Show that $\{c \in \mathbb{P} : c \Vdash^* \mathcal{R} \text{ or } c \Vdash^* \neg\mathcal{R}\}$ is dense. $\diamond$

Exercise 2.10. ($\star$) Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing, $c \in \mathbb{P}$ and $\mathcal{R}$ an arithmetic requirement. Show that if $c \Vdash^* \mathcal{R}$, then $c \Vdash \mathcal{R}$. $\diamond$

Complexity of $\Vdash$

The complexity of $\Vdash^*$ has consequences on the complexity of the semantic forcing relation $\Vdash$. According to the previous exercises, $\Vdash^*$ respects the items (1-3) of Definition 2.5. So according to Proposition 2.6, the relation $c \Vdash \mathcal{R}$ is equivalent to $\forall d \leqslant c\ \exists e \leqslant d\ e \Vdash^* \mathcal{R}$, which for example for a Σ^0_n requirement will be a $\Pi^0_{n+1}(\mathbb{P})$ predicate. This is a considerable simplification of the semantic relation, but which remains —relative to $\mathbb{P}$— of arithmetical complexity greater than that of the requirements which it forces, which will make us prefer the relation $\Vdash^*$.

3. Forcing with trees

Tree-based forcing is one of the major families of forcing. We detail here two examples, already encountered in the previous chapters: Jockusch-Soare forcing and the computable Sacks forcing. These notions were originally created to control the single jump, via the forcing of Σ^0_1/Π^0_1 requirements or the double jump, via the forcing of Σ^0_2/Π^0_2 requirements. We will discuss this again in Section 4, and we will content ourselves for the moment with seeing how these forcings have been used implicitly on several occasions in this book.

3.1. Jockusch-Soare forcing

The Jockusch-Soare forcing corresponds to the partial order of non-empty Π^0_1 classes, partially ordered by the inclusion relation. We can associate with any maximal filter F on this order a set of integers $\dot{F} \in 2^{\mathbb{N}}$ which is the element of the singleton $\bigcap_{\mathcal{P} \in F} \mathcal{P}$. Equipped with this interpretation of the filters, Jockusch-Soare forcing is a Cantor forcing (see Definition 2.1).

We have already encountered several uses of Jockusch-Soare forcing in Chapter 8 on Π^0_1 classes and PA degrees. Here is a reformulation of the computably dominated basis theorems and the cone avoidance basis theorems, via the forcing vocabulary:

Theorem (Reformulation of Theorem 8-4.5 and Theorem 8-4.7) *Let A be a non-computable set. Let G be a sufficiently generic set for Jockusch-Soare forcing. Then, G is computably dominated and does not compute A.*

Proof. Let $(\mathbb{P}, \leqslant)$ be Jockusch-Soare forcing, i.e., the set of non-empty Π^0_1 classes ordered by inclusion. The proof of Theorem 8-4.5 shows the following: for any functional Φ_e and all $c \in \mathbb{P}$, there exists $d \leqslant c$ such that $d \Vdash^* \exists n\ \Phi_e(G, n)\uparrow$ or there exists $d \leqslant c$ and a computable function $f : \mathbb{N} \to \mathbb{N}$ such that $d \Vdash^* \Phi_e(G, n)\downarrow< f(n)$ for all n. So the set C_e of the conditions which force Φ_e to be partial or else bounded by a computable function is dense. Any sufficiently generic filter contains a condition in each of C_e. So if $G \in 2^{\mathbb{N}}$ is sufficiently generic it is computably dominated.

The proof of Theorem 8-4.7 shows the following: for any functional Φ_e, the set $D_e = \{c \in \mathbb{P} : c \Vdash^* \exists n\ \Phi_e(G, n)\uparrow \text{ or } \exists n\ c \Vdash^* \Phi_e(G, n)\downarrow\neq A(n)\}$ is dense. So if G is sufficiently generic, it does not compute A. ■

Regarding the low basis theorem, things are different: it is indeed Jockusch-Soare forcing that we use to build a low set, but it is an effective use of this forcing, where the passage from a step n to a step $n + 1$ is controlled using $\emptyset'$. It is therefore not strictly speaking a forcing result, in the sense that it is not a property satisfied by any sufficiently generic set, but on the contrary by a small countable class of sets which are not very generic.

We have seen with Cohen forcing that any sufficiently generic set differs from a countable quantity of sets fixed in advance (see Theorem 10-5.4). In particular, no sufficiently generic set for Cohen forcing is arithmetic. This is not the case with Jockusch-Soare forcing. Indeed, for any computable set X, the singleton $\{X\}$ is a Π^0_1 class and the unique filter containing $\{X\}$ —namely the filter of all Π^0_1 classes having X as an infinite path— is maximal. The set X is therefore as generic as we want under the condition $\{X\}$.

However, it is possible to slightly modify Jockusch-Soare forcing in order to avoid computable elements.

Proposition 3.2. Let $(\mathbb{P}, \leqslant)$ be the partial order of non-empty Π^0_1 classes without computable element, ordered by inclusion. Let $(A_n)_{n\in\mathbb{N}}$ be any sequence of sets. If $G \in 2^{\mathbb{N}}$ is generic enough for $\mathbb{P}$ it is different from each A_n. ⋆

Proof. Let us fix any set A and show that if G is sufficiently generic, it is different from A. The result will follow automatically for the continuation $(A_n)_{n\in\mathbb{N}}$.

Let $D \subseteq \mathbb{P}$ be the set of Π^0_1 classes of $\mathbb{P}$ not containing A. Let us show that D is dense in $\mathbb{P}$. Let $\mathcal{P} \in \mathbb{P}$. By Proposition 8-3.6, the class $\mathcal{P}$ contains at least two elements. Let $B \in \mathcal{P}$ be such that $B \neq A$, and let $n \in \mathbb{N}$ be such that $A(n) \neq B(n)$. Then the class $\mathcal{Q} = \{X \in \mathcal{P} : X(n) = B(n)\}$ is a non-empty Π^0_1 class included in $\mathcal{P}$ and such that $\mathcal{Q} \in D$. So D is dense. ■

The modification of Jockusch-Soare forcing of the preceding proposition can be declined in multiple ways to obtain different results: one can consider the partial order of the non-empty Π^0_1 classes containing only PA degrees, or even those of the non-empty Π^0_1 classes containing only random sets in the sense of Martin-Löf (see Chapter 18).

3.2. Computable Sacks forcing

Sacks forcing generally designates the partial order of perfect trees of $2^{<\mathbb{N}}$, without any particular effectiveness restriction. We restrict ourselves in computability theory to computable perfect trees, which have already been approached via the notion of *f-tree*.

Recall that an *f-tree* is a total function $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ such that for all $\sigma, \tau \in \operatorname{dom} T$, $\sigma \preceq \tau$ if and only if $T(\sigma) \preceq T(\tau)$. A *sub-f-tree* of an f-tree T is an f-tree S such that $\operatorname{Im} S \subseteq \operatorname{Im} T$. An f-tree $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ extends into a function $\hat{T} : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ defined by $\{\hat{T}(X)\} = \bigcap_n [T(X \restriction_n)]$. A *path* of T is an element of $\operatorname{Im} \hat{T}$. We denote by $[T]$ the set of paths of T.

We call *computable Sacks forcing* the set of computable f-trees partially ordered by the sub-f-tree relation. As there will be no possible ambiguity in this book, we will sometimes simply say *Sacks forcing*. We can associate with any maximal filter F on this order a set of integers $\dot{F} \in 2^{\mathbb{N}}$ which is the element of the singleton $\bigcap_{T \in F} [T]$. Equipped with this interpretation of the filters, Sacks forcing is a Cantor forcing (see Definition 2.1).

Remark

Recall that a class $\mathcal{P} \subseteq 2^{\mathbb{N}}$ is perfect if $\mathcal{P} = [T]$ for an f-tree T. Sacks forcing is not, however, the restriction of Jockusch-Soare forcing to perfect Π^0_1 classes: some perfect Π^0_1 classes cannot necessarily be represented by a computable f-tree. Indeed, for any computable f-tree T, $[T]$ contains an infinity of computable elements, namely $T(X)$ for any computable set X, which is for example not the case of the Π^0_1 class of DNC_2 functions, which is nevertheless a perfect class.

As explained in the previous remark, any computable f-tree possesses an infinity of computable paths. On the other hand, any sufficiently generic set for Sacks forcing will be non-computable:

Exercise 3.3. **($\star$)** Let $(A_n)_{n \in \mathbb{N}}$ be any sequence of sets. Show that any sufficiently generic set G for Sacks forcing is different from each A_n. ◇

A careful examination of the first proof that we have given of the existence of a computably dominated degree different from $\mathbf{0}$, using f-trees, in fact shows that we have the following result.

Theorem (reformulation du théorème 7-5.6)
Let G be a sufficiently generic set for Sacks forcing. Then, G is non-computable and computably dominated.

Proof. Let $(\mathbb{P}, \leqslant)$ be Sacks forcing. The proof of Theorem 7-5.6 shows the following: for any functional Φ_e and any $c \in \mathbb{P}$, there exists $d \leqslant c$ such that $d \Vdash^* \exists n\ \Phi_e(G, n) \uparrow$ or there exists $d \leqslant c$ and a computable function $f : \mathbb{N} \to \mathbb{N}$ such that $d \Vdash^* \Phi_e(G, n) \downarrow < f(n)$ for all n. So the set C_e of the conditions which force Φ_e to be partial or else bounded by a computable function is dense. Any sufficiently generic filter contains a condition of each of C_e. So if $G \in 2^{\mathbb{N}}$ is sufficiently generic, it is computably dominated.

Moreover, Exercise 3.3 shows that if G is sufficiently generic, it is not computable. ■

Exercise 3.5. ($\star$) Let A be a non-computable set. Show that if G is sufficiently generic for Sacks forcing, it does not compute A. ◇

Exercise 3.6. ($\star\star$) A set X is *computably traceable* (Terwijn and Zambella [231]) if there exists a computable bound $h : \mathbb{N} \to \mathbb{N}$ such that for any X-computable function $f : \mathbb{N} \to \mathbb{N}$, there exists a computable sequence $(T_n)_{n \in \mathbb{N}}$ of finite sets such that $|T_n| \leqslant h(n)$ and such that $f(n) \in T_n$ for all n.

1. Show that if X is sufficiently generic for Sacks forcing, it is computably traceable.
2. Show that if X is computably traceable via a computable bound h, then it is computably traceable for any computable bound h' such that $h'(n) \leqslant h'(n+1)$ and $\lim_n h(n) = +\infty$ (Terwijn and Zambella [231]).
3. Let $(2^n)^{\mathbb{N}}$ be the set of functions $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) < 2^n$. Let $(2^n)^{<\mathbb{N}}$ be the set of function prefixes of $(2^n)^{\mathbb{N}}$. Let $(\mathbb{P}, \leqslant)$ be the forcing conditions given by $T \in \mathbb{P}$ if $T \subseteq (2^n)^{<\mathbb{N}}$ is a computable tree which satisfies the following property: for all $\sigma \in T$ there exists $\tau \in T$ with $\tau \succeq \sigma$ such that for all $i < 2^{|\tau|}$ the string τi is in T. In other words, each node has a maximally branching extension. Show that any set sufficiently generic for this forcing is computably dominated and not computably traceable.

◇

4. Computational complexity and forcing question

Cohen forcing presents a particularity that distinguishes it from other computability-theoretic forcings: the partial order $(2^{<\mathbb{N}}, \succeq)$ is computable: the set $2^{<\mathbb{N}}$ is computable and given $\sigma \in 2^{<\mathbb{N}}$, we can compute the set of strings $\tau \succeq \sigma$. Another of its particularities is that the forcing relation of Σ^0_1 and Π^0_1 requirements is respectively Σ^0_1 and Π^0_1. These two properties make it possible to obtain Proposition 10-5.2: if $\mathcal{R}$ is a Σ^0_n (resp. Π^0_n) requirement, the predicate $\sigma \Vdash^* \mathcal{R}$ is Σ^0_n (resp. Π^0_n). This very fine control of the complexity of the forcing relation then allows a fine control of the iterated jumps of the generic sets, in order to obtain for example the following results:

1. Corollary 10-5.8: if X is not $\emptyset^{(n)}$-computable and if G is sufficiently generic then $G^{(n)}$ does not compute X.

2. Theorem 10-5.9: if G is sufficiently generic then $G \oplus \emptyset^{(n)} \geqslant_T G^{(n)}$.

It is in general the kind of property that one seeks to obtain with any forcing notion: the control of the truth value of arithmetic requirements. The finir this control is, the better the theorems we get. Unfortunately, things will rarely be as simple as with Cohen forcing. Let us examine the computational complexity of Jockusch-Soare forcing and that of Sacks forcing.

4.1. Complexity of partial orders

To talk about the computability of partial orders of abstract objects, one needs to first agree on their representation by sets of integers. For Cohen forcing, the partial order of the strings admits a natural bijective coding, while the notions of Jockusch-Soare and Sacks forcing involve more complex objects.

Jockusch-Soare forcing. A natural idea is to identify $\mathbb{P}$ with the set of codes of non-empty Π^0_1 classes. Note that we then have repetitions because several integers code for the same class, which in practice is not a problem.

Proposition 4.1. The partial order of Jockusch-Soare forcing is Π^0_2. ⋆

Proof. Note that the set of codes of non-empty Π^0_1 classes is Π^0_1. Indeed if a Π^0_1 class is empty then the computable tree $T \subseteq 2^{<\mathbb{N}}$ which represents it has no infinite path, and according to König's lemma there exists an integer n such that no length string n belongs to T, which is a Σ^0_1 event.

Now, given two non-empty Π^0_1 classes $\mathcal{P}, \mathcal{Q}$ we have $\mathcal{P} \subseteq \mathcal{Q}$ iff $\forall t\ \exists s\ \mathcal{P}[s] \subseteq \mathcal{Q}[t]$, which is a Π^0_2 predicate. ■

Note that it is possible to improve the complexity of the partial order of Jockusch-Soare forcing, by working only on a well-chosen part of the codes of non-empty Π^0_1 classes. There is indeed a computable function $f : \mathbb{N} \to \mathbb{N}$ such that $f(e)$ always codes for a non-empty Π^0_1 class and such that if e codes for a non-empty Π^0_1 class then e and $f(e)$ code for the same class: given e, it suffices to stop the co-enumeration of the corresponding Π^0_1 class if this one is about to make the class empty.

We can also in practice improve the complexity of the partial order, by restricting there too the set of codes on which we work: given the code e of a non-empty Π^0_1 class $\mathcal{P}$, we can consider the set A of the codes of all the Π^0_1 classes $\mathcal{Q}$ such that $\mathcal{P} \cap \mathcal{Q}$ is not empty. The set A is Π^0_1 and contains at least one code corresponding to each non-empty Π^0_1 class included in $\mathcal{P}$. This does not of course make it possible to decide whether two codes represent comparable forcing conditions, but it simplifies the computational complexity of finding the list of all the conditions of $\mathbb{P}$ found under a given condition.

Computable Sacks Forcing. Here, the natural idea is to identify the conditions of the computable Sacks forcing with codes of functions $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ corresponding to f-trees. Againn the coding is not injective, but in practice this is not a problem.

Proposition 4.2. The partial order of computable Sacks forcing is Π^0_2. $\star$

Proof. The set of conditions is the set of codes e of functions $T_e : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ such that $\forall \sigma \in 2^{<\mathbb{N}}\ \exists t \in \mathbb{N}\ T_e(\sigma)[t]\downarrow$ and such that $\forall \sigma_0, \sigma_1 \in 2^{<\mathbb{N}}\ \sigma_0 \prec \sigma_1 \leftrightarrow T_e(\sigma_0) \prec T_e(\sigma_1)$. These are Π^0_2 conditions to check.

Given a computable f-tree T, the set of codes e of computable f-trees S_e such that $[S_e] \subseteq [T]$ is the set of codes e of f-tree (which is a Π^0_2 condition), which checks, for any string σ and for any integer n sufficiently large such that $|T(\tau)| \geqslant |S_e(\sigma)|$ for any string τ of size n, the existence of a string τ such that $|\tau| \leqslant n$ for which $S_e(\sigma) = T(\tau)$. This is a Π^0_1 condition. ■

4.2. Forcing Σ^0_1/Π^0_1 requirements

The complexity of the forcing relation for the Σ^0_1 and Π^0_1 requirements is not directly linked to the complexity of the partial order. We will see in particular that the syntactic forcing relation of Jockusch-Soare forcing is more complex than that of Sacks forcing.

Jockusch-Soare forcing. The complexity of the forcing relation does not follow that of the complexity of the requirements to be forced, including already for the Σ^0_1/Π^0_1 requirements.

Proposition 4.3. Let $(\mathbb{P}, \leqslant)$ be Jockusch-Soare forcing and $\mathcal{R}$ a requirement. Let $\mathcal{P}_e$ be the non-empty Π^0_1 class of code e.

(1) If $\mathcal{R}$ is Σ^0_1, then the predicate $\mathcal{P}_e \Vdash^* \mathcal{R}$ is Σ^0_1

(2) If $\mathcal{R}$ is Π^0_1, then the predicate $\mathcal{P}_e \Vdash^* \mathcal{R}$ is Π^0_2 ⋆

Proof. Let $T_e \subseteq 2^{<\mathbb{N}}$ be a computable tree such that $[T_e] = \mathcal{P}_e$.

(1) Let $\mathcal{R}$ be of the form $\Phi(G, 0)\downarrow$ for a functional Φ. Then, $\mathcal{P}_e \Vdash^* \mathcal{R}$ iff $\Phi(X, 0)\downarrow$ for all $X \in \mathcal{P}_e$ iff (by the use property and König's lemma) $\exists n\, \forall \sigma \in T_e \cap 2^n,\ \Phi(\sigma, 0)\downarrow$.

(2) Let $\mathcal{R}$ be of the form $\Phi(G, 0)\uparrow$ for a functional Φ. Then, we can obtain the code $a \in \mathbb{N}$ of the Π^0_1 class such that $\mathcal{P}_a = \{X : \Phi(G, 0)\uparrow\}$. We then have $\mathcal{P}_e \Vdash^* \mathcal{R}$ iff $\mathcal{P}_e \subseteq \mathcal{P}_a$. As seen in the proof of Proposition 4.2, the inclusion relation on the codes of Π^0_1 classes is Π^0_2. ■

We will see in the next two sections how to get around the problem of complexity raised by the previous proposition.

Exercise 4.4. **(⋆)** Let $(\mathbb{P}, \leqslant)$ be the partial order of infinite computable trees $T \subseteq 2^{<\mathbb{N}}$. Let $\Vdash^\circ$ be the relation defined by

(1) $T \Vdash^\circ \exists n \Phi(G, n)\downarrow$ if there exists $n, t \in \mathbb{N}$ such that for any $\sigma \in T$ of length t, $\Phi(\sigma, n)\downarrow$

(2) $T \Vdash^\circ \forall n \Phi(G, n)\uparrow$ if for all $\sigma \in T$ and all $n < |\sigma|$, $\Phi(\sigma, n)\uparrow$

To show that

(a) $(\mathbb{P}, \leqslant)$ is a Cantor forcing, where $[T]$ is the class of the paths of T.

(b) The sufficiently generic sets for Jockusch-Soare forcing and for $(\mathbb{P}, \leqslant)$ coincide.

(c) The set $\{T \in \mathbb{P} : T \Vdash^\circ \exists n \Phi(G, n)\downarrow$ or $T \Vdash^\circ \forall n \Phi(G, n)\uparrow\}$ is dense in $(\mathbb{P}, \leqslant)$

(d) The relations $T \Vdash^\circ \exists n \Phi(G, n)\downarrow$ and $T \Vdash^\circ \forall n \Phi(G, n)\uparrow$ are respectively Σ^0_1 and Π^0_1.

◇

Computable Sacks Forcing. The forcing of Σ^0_1 and Π^0_1 requirements is in this case of minimal complexity.

Proposition 4.5. Let $(\mathbb{P}, \leqslant)$ be Sacks forcing and let $\mathcal{R}$ be a requirement. Let T_e be the computable f-tree of code e.

(1) If $\mathcal{R}$ is Σ^0_1, then the predicate $T_e \Vdash^* \mathcal{R}$ is Σ^0_1

(2) If $\mathcal{R}$ is Π^0_1, then the predicate $T_e \Vdash^* \mathcal{R}$ is Π^0_1 ⋆

Proof. (1) Let $\mathcal{R}$ be of the form $\Phi(G, 0)\downarrow$ for a functional Φ. We have $T_e \Vdash^* \mathcal{R}$ iff there exists n, t such that $\Phi(\sigma, 0)[t]\downarrow$ for any string $\sigma \in \operatorname{Im} T_e$ of size n.

(2) Let $\mathcal{R}$ be of the form $\Phi(G, 0)\uparrow$ for a functional Φ. We have $T_e \Vdash^* \mathcal{R}$ iff for all $\sigma \in \operatorname{Im} T_e$ and for all t, we have $\Phi(\sigma, 0)[t]\uparrow$. ■

Continuity of the Turing jump. We now see a theorem that contrasts with the fact that every sufficiently generic set for Cohen forcing is generalized low. This is generally not the case for tree-based forcings, and it is in particular not the case for computable Sack forcing.

Theorem 4.6
Let X be any set. Let G be a sufficiently generic set for computable Sacks forcing. Then, $X \oplus G \not\geq_T G'$.

Proof. Let $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ be a computable f-tree. Let Φ_e be a Turing functional. We will build a subf-tree S of T such that for each of the paths G of S we have $\Phi_e(X \oplus G, n) \neq G'(n)$ for a certain n. First consider a computable sub-tree S of T such that for all $\sigma \in \operatorname{Im} S$ there exists $\tau \succeq \sigma$ with $\tau \in \operatorname{Im} T$ and $\tau \notin \operatorname{Im} S$.

Now consider the code a of the partial computable functional Φ_a such that $\Phi_a(Y, a)\uparrow$ for all $Y \in [S]$ and such that $\Phi_a(Y, a)\downarrow$ for all $Y \notin [S]$. Note in particular that $\Phi_a(Y, a)\downarrow$ for all $Y \in [T] \setminus [S]$. Suppose first that $\Phi_e(X \oplus \sigma, a)\uparrow\notin \{0, 1\}$ for all $\sigma \in \operatorname{Im} S$. Then, we can take S as a forcing extension of T in order to force the partiality of Φ_e on the input a. Otherwise let $\sigma \in \operatorname{Im} S$ be such that $\Phi_e(X \oplus \sigma, a)\downarrow= i$ for $i \in \{0, 1\}$. If $i = 0$, then we choose a string $\tau \succeq \sigma$ such that $\tau \in \operatorname{Im} T$ and $\tau \notin \operatorname{Im} S$, and we take as a forcing extension a subf-tree of T whose image only contains extensions of τ. Note that we then have $\Phi_a(Y, a)\downarrow$ for any path Y of our forcing extension. If $i = 1$, then we take as a forcing extension the subf-tree of S whose image only contains extensions of σ. Note that we then have $\Phi_a(Y, a)\uparrow$ for any path Y of our forcing extension.

In both cases we force $\Phi_e(X \oplus G, a)$ to be different from $G'(a)$ for any set G of our forcing condition. ■

An analogous theorem for Jockusch-Soare forcing will not necessarily be true, for example because of the existence of Π^0_1 classes containing a single computable element. Even if we restrict ourselves to Π^0_1 classes which do not contain computable points, things are not so simple and will depend on the Π^0_1 class with which we start the forcing.

Exercise 4.7. **($\star\star\star$)** Let $\mathcal{P}$ be a non-empty Π^0_1 class. Then, $\mathcal{P}$ is *thin* (definition due to Downey [49]) if for any non-empty Π^0_1 subclass $\mathcal{Q} \subseteq \mathcal{P}$, there exists a finite sequence of cylinders $[\sigma_0], \dots, [\sigma_n]$ such that $\mathcal{Q} = \mathcal{P} \cap ([\sigma_0] \cup \dots \cup [\sigma_n])$.

1. Show the existence of a perfect thin Π^0_1 class (this is an algorithm of the priority method type as explained in Chapter 13).
2. Show that if $\mathcal{P}$ is thin and if $X \in \mathcal{P}$ then $X \oplus \emptyset'' \geqslant_T X'$.

$\diamond$

Exercise 4.8. **($\star\star$)** This exercise anticipates on Part II within which Lemma 18-3.3 should be useful. We consider a variant of Jockusch-Soare forcing with Π^0_1 classes containing only random ones, in the sense of Martin-Löf. Show that for any X and any set Z sufficiently generic for this forcing we have $Z \oplus X \not\geqslant_T Z'$. $\diamond$

4.2.1. Forcing Σ^0_2/Π^0_2 requirements

Tree-based forcing are generally suitable, not to force Σ^0_1 or Π^0_1 requirements (as shown by Theorem 4.6 they do not allow to obtain generalized low sets), but to force Σ^0_2 or Π^0_2 requirements, on which they work particularly well. We saw in Section 10-4 that in the case of Cohen forcing, if we define the forcing relation by "any maximal filter satisfies the requirement", then the set of conditions forcing a Π^0_2 requirement or its negation, is not dense in general. This led us to define the forcing relation for any *sufficiently generic* maximal filter.

Unlike Cohen forcing, tree-based forcings, like Jockusch-Soare forcing or Sacks forcing, generally the existence of forcing conditions whose members all satisfy Σ^0_2 or Π^0_2 formulas. The reader will be able to note that it is indeed this mechanism which is at work to force a generic set to be computably dominated. To see this, we introduce the notion of *forcing question*, denoted $?\vdash$, which will be developed and studied in the following sections for arbitrary arithmetic requirements.

Definition 4.9. Let $\mathcal{R}$ be a Σ^0_2 requirement corresponding to $\exists n\ \Phi(G,n)\uparrow$ for a functional Φ.

(1) Let $\mathcal{P}$ be a condition of Jockusch-Soare forcing. We define

$$\mathcal{P} \mathrel{?{\vdash}} \exists n\ \Phi(G,n)\uparrow$$

if there is n such that $\mathcal{P} \nVdash^* \Phi(G,n)\downarrow$.

(2) Let T be a computable Sacks forcing condition. We define

$$T \mathrel{?{\vdash}} \exists n\ \Phi(G,n)\uparrow$$

if there exists n and σ such that $T\restriction_\sigma \Vdash^* \Phi(G,n)\uparrow$, where $T\restriction_\sigma$ is the f-tree S defined by $S(\tau) = T(\sigma\tau)$. ◊

The first interest of the forcing question relation that we have defined is its complexity, which is the same as that of the requirement concerned.

Proposition 4.10. Let c be a condition of Jockusch-Soare forcing or Sacks forcing. Let $\mathcal{R}$ be a Σ^0_2 requirement corresponding to $\exists n\ \Phi(G,n)\uparrow$ for a functional Φ. The predicate $c \mathrel{?{\vdash}} \exists n\ \Phi(G,n)\uparrow$ is Σ^0_2. ⋆

Proof. Let's start with Jockusch-Soare forcing. According to Proposition 4.3, given n, the predicate $\mathcal{P} \Vdash^* \Phi(G,n)\downarrow$ is Σ^0_1 and therefore the predicate $\mathcal{P} \nVdash^* \Phi(G,n)\downarrow$ is Π^0_1. So the predicate $\exists n\ \mathcal{P} \nVdash^* \Phi(G,n)\downarrow$ is Σ^0_2 and the predicate $\mathcal{P} \mathrel{?{\vdash}} \Phi(G,n)\uparrow$ is therefore Σ^0_2.

Let's move on to the computable Sacks forcing. According to Proposition 4.5, given n, and an f-tree S, the predicate $S \Vdash^* \Phi(G,n)\uparrow$ is Π^0_1. So the predicate $\exists n\ \exists\sigma\ T\restriction_\sigma \Vdash^* \Phi(G,n)\uparrow$ is Σ^0_2 and the predicate $T \mathrel{?{\vdash}} \exists n\ \Phi(G,n)\uparrow$ is therefore Σ^0_2. ■

The second interest of the forcing question relation, is that it allows to *decide* if a condition c can be extended into a condition d to force a Σ^0_2 requirement, or the negation of this requirement. Moreover, in the case of Σ^0_2/Π^0_2 formulas, we can find an extension $d \leqslant c$ such that the requirement will be satisfied *for all the elements* of $[d]$ (As for the Σ^0_1/Π^0_1 requirements).

Proposition 4.11. Let c be a condition of Jockusch-Soare forcing or Sacks forcing. Let $\mathcal{R}$ be a Σ^0_2 requirement corresponding to $\exists n\ \Phi(G,n)\uparrow$ for a functional Φ.

1. If $c \mathrel{?{\vdash}} \exists n\ \Phi(G,n)\uparrow$ then there exists $d \leqslant c$ such that $\exists n\ \Phi(X,n)\uparrow$ is true for all $X \in [d]$.

2. If $c \mathrel{?{\nvdash}} \exists n\ \Phi(G,n)\uparrow$ then there exists $d \leqslant c$ such that $\forall n\ \Phi(X,n)\downarrow$ is

true for all $X \in [d]$.

In both cases, we can find d uniformly in c and Φ using $\emptyset''$. ⋆

Proof. Let's start with Jockusch-Soare forcing. Let $\mathcal{P}$ be a non-empty Π^0_1 class.

1. If $\mathcal{P} \mathrel{?}\!\vdash \exists n\ \Phi(G,n)\uparrow$ then there exists n such that $\mathcal{P} \nVdash^* \Phi(G,n)\downarrow$. This means that the class $\mathcal{Q} = \{X \in \mathcal{P} : \Phi(X,n)\uparrow\}$ is a non-empty Π^0_1 class. This is then our forcing extension.

2. If $\mathcal{P} \mathrel{?}\!\nvdash \exists n\ \Phi(G,n)\uparrow$ then $\mathcal{P} \Vdash^* \Phi(G,n)\downarrow$ for all n. In this case for all $X \in \mathcal{P}$ we already have $\forall n\ \Phi(X,n)\downarrow$.

Note that $\emptyset''$ can decide between the two cases and therefore find the appropriate forcing extension. Let's move on to the computable Sacks forcing. Let $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ be a computable f-tree.

1. If $T \mathrel{?}\!\vdash \exists n\ \Phi(G,n)\uparrow$ then there exists $\sigma \in 2^{<\mathbb{N}}$ and $n \in \mathbb{N}$ such that $T\restriction_\sigma \Vdash^* \Phi(G,n)\uparrow$. In this case the f-tree $T\restriction_\sigma$ is our forcing extension.

2. If $T \mathrel{?}\!\nvdash \exists n\ \Phi(G,n)\uparrow$ then for all $\sigma \in 2^{<\mathbb{N}}$ and for all $n \in \mathbb{N}$ we have $T\restriction_\sigma \nVdash^* \Phi(G,n)\uparrow$. This implies that for all $\sigma \in 2^{<\mathbb{N}}$ and for all $n \in \mathbb{N}$, there exists $\tau \succeq \sigma$ such that $\Phi(T(\sigma\tau),n)\downarrow$. Then, we proceed as in the proof of Theorem 7-5.6 to compute a subf-tree S of T such that $\forall n\ \Phi(X,n)\downarrow$ for all $X \in [S]$.

Note that there again $\emptyset''$ can decide between the two cases and therefore find the appropriate forcing extension. ■

Before seeing how to extend the forcing question relation to requirements of arbitrary complexity, let us see how to use the developments obtained so far to show that any set sufficiently generic for Sacks forcing or Jockusch-Soare forcing is generalized low$_2$.

Theorem 4.12
If G is sufficiently generic for Sacks forcing or Jockusch-Soare forcing, it is generalized low$_2$ in a strong sense: $\emptyset'' \oplus G' \geqslant_T G''$.

Proof. For all n, according to Lemma 10-5.6 the requirement $n \in G''$ is Σ^0_2. According to Proposition 4.10 and Proposition 4.11 we can enumerate with the help of $\emptyset''$ a set D_1 of conditions c such that $n \in X''$ for all $X \in [c]$, and a set D_2 of conditions c such that $n \notin X''$ for all $X \in [c]$, the whole such that $D_1 \cup D_2$ is a dense set of conditions. If G is sufficiently generic, there exists a condition of $c \in D_1 \cup D_2$ such that $G \in [c]$. The condition c being a tree, we need G' to know if $G \in [c]$. Once the condition c has been found such that $G \in [c]$, if $c \in D_1$ then $n \in G''$ and if $c \in D_2$ then $n \notin G''$. ■

4.2.2. Forcing Σ^0_n/Π^0_n requirements

For more complex requirements, it is not possible to find conditions in which all the elements satisfy the requirement, and one needs to return to the inductive definition of forcing. Our objective is now to show an analogue of the theorem 10-5.7 of preservation of the arithmetic hierarchy. As the relation $\Vdash^*$ for the Sacks and Jockusch-Soare forcings is too complex, we will extend and instead use the forcing question relation defined previously for the Σ^0_2 requirements.

Definition 4.13. Let c be a condition of the Jockusch-Soare or Sacks forcing. Let $\mathcal{R}$ be an arithmetic requirement. We define the relation $?\vdash$ between c and $\mathcal{R}$ as follows:

(1) $c \,?\vdash \mathcal{R}$ for a Σ^0_1 requirement $\mathcal{R}$ iff $c \Vdash^* \mathcal{R}$.

(2) $c \,?\vdash \exists x \mathcal{R}(x)$ where $\mathcal{R}(x)$ is a Π^0_1 requirement iff $c \,?\vdash \exists x \mathcal{R}(x)$ within the meaning of Definition 4.9.

(3) $c \,?\vdash \exists x \mathcal{R}(x)$ where $\mathcal{R}(x)$ is a Π^0_k requirement for $k \geqslant 2$ iff there is a $d \leqslant c$ and an $n \in \mathbb{N}$ such that $d \,?\vdash \mathcal{R}(n)$.

(4) $c \,?\vdash \mathcal{R}$ where $\mathcal{R}(x)$ is a Π^0_k requirement iff $c \,?\nvdash \neg\mathcal{R}$. ♢

We now extend Proposition 4.10 and Proposition 4.11, first by showing that the forcing question relation is Σ^0_n for Σ^0_n requirements:

Proposition 4.14. Let c be a condition of Jockusch-Soare forcing or Sacks forcing. Let $\mathcal{R}$ be a Σ^0_n (resp. Π^0_n) requirement. The relation $c \,?\vdash \mathcal{R}$ is Σ^0_n (resp. Π^0_n). ⋆

Proof. The case where $\mathcal{R}$ is a Σ^0_1 requirement has been dealt with in Proposition 4.3 and Proposition 4.5. The case where $\mathcal{R}$ is a Σ^0_2 requirement was dealt with in Proposition 4.10.

Suppose that $\mathcal{R}$ is a Σ^0_{n+1} requirement equal to $\exists x \mathcal{Q}(x)$ where $\mathcal{Q}(x)$ is a Π^0_k requirement for $k \geqslant 2$. Then, $c \,?\vdash \exists x \mathcal{Q}(x)$ iff there exists a forcing condition d and an integer n such that $d \leqslant c$ and such that $d \,?\vdash \mathcal{Q}(n)$. The predicate $d \leqslant c$ is Π^0_2 (in the case of Jockusch-Soare forcing and Sacks forcing) and the predicate $d \,?\vdash \mathcal{Q}(n)$ is by Π^0_k induction for $k \geqslant 2$. It follows that the predicate $c \,?\vdash \exists x \mathcal{Q}(x)$ is Σ^0_{k+1}.

Finally if $\mathcal{R}$ is a Π^0_n requirement. Then, $\mathcal{P} \,?\vdash \mathcal{R}$ iff $\mathcal{P} \,?\nvdash \neg\mathcal{R}$ which is Π^0_n by induction hypothesis. ■

We now show the extension of Proposition 4.11: if $c \,?\!\vdash \mathcal{R}$ then we will be able to find an extension $d \leqslant c$ which will force $\mathcal{R}$, but be careful, it is here semantic forcing and not syntactic forcing.

Proposition 4.15. Let c be a condition of the Jockusch-Soare or Sacks forcing. Let $\mathcal{R}$ be an arithmetic requirement. If $c \,?\!\vdash \mathcal{R}$ then there exists $d \leqslant c$ such that $d \Vdash \mathcal{R}$. ⋆

Proof. If $\mathcal{R}$ is a Σ^0_1 or Π^0_1 requirement then the result is trivial by definition. If $\mathcal{R}$ is a Σ^0_2 or Π^0_2 requirement then it is Proposition 4.11.

Suppose $\mathcal{R} = \exists x \mathcal{S}(x)$ where $\mathcal{S}(x)$ is a Π^0_k requirement for $k \geqslant 2$. Suppose that $c \,?\!\vdash \exists x \mathcal{S}(x)$. By definition, there is an extension $d \leqslant c$ and an integer $n \in \mathbb{N}$ such that $d \,?\!\vdash \mathcal{S}(n)$. By induction hypothesis, there exists an $e \leqslant d$ such that $e \Vdash \mathcal{S}(n)$. In particular $e \Vdash \exists n\, \mathcal{S}(n)$.

Suppose $\mathcal{R} = \forall x \mathcal{S}(x)$ where $\mathcal{S}(x)$ is a Σ^0_k requirement for $k \geqslant 2$. Suppose that $c \,?\!\vdash \forall x \mathcal{S}(x)$. Then, $c \,?\!\nvdash \exists x \neg \mathcal{S}(x)$. By definition, for any extension $d \leqslant c$ and any $n \in \mathbb{N}$, $d \,?\!\nvdash \neg \mathcal{S}(n)$. Let us show that for all n, the set $D_n = \{d : d \Vdash \mathcal{S}(n)\}$ is dense below c. Let $n \in \mathbb{N}$ and $d \leqslant c$. By definition as $d \,?\!\nvdash \neg \mathcal{S}(n)$ then $d \,?\!\vdash \mathcal{S}(n)$. By induction hypothesis, as $d \,?\!\vdash \mathcal{S}(n)$, then there exists an extension $e \leqslant d$ such that $e \Vdash \mathcal{S}(n)$. The set D_n is therefore dense below c. It follows from Exercise 2.4 that $c \Vdash \mathcal{S}(n)$ for all n, therefore $c \Vdash \forall x \mathcal{S}(x)$, in other words $c \Vdash \mathcal{R}$. ■

This forcing question will allow us a fine control of what computes arbitrary iterations of the Turing jump. This is the subject of the next section.

4.3. Forcing question

Let us try to abstract a little from what has been done so far, in order to study the notion of forcing question, independently of the forcing we are working with.

Definition 4.16. Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing. A *forcing question* is a relation $?\!\vdash$ between conditions and requirements, such that for any condition $c \in \mathbb{P}$ and any arithmetic requirement $\mathcal{R}$

(1) If $c \,?\!\vdash \mathcal{R}$, then there exists an extension $d \leqslant c$ such that d forces $\mathcal{R}$

(2) $c \,?\!\vdash \mathcal{R}$ or $c \,?\!\vdash \neg\mathcal{R}$. ◊

It follows from (1) and (2) that if $c \,?\!\nvdash \mathcal{R}$, then there exists an extension $d \leqslant c$ such that d forces $\neg\mathcal{R}$. Any forcing relation $\Vdash$ induces a forcing question $?\!\vdash$ by defining $c \,?\!\vdash \mathcal{R}$ for a Σ^0_n requirement $\mathcal{R}$ if there exists an extension $d \leqslant c$ such that $d \Vdash \mathcal{R}$ and $c \,?\!\vdash \mathcal{R}$ for a Π^0_n requirement if $c \,?\!\nvdash \neg\mathcal{R}$.

If the forcing notion is computable, as is the case for Cohen forcing, the complexity of the forcing question for Σ^0_n requirements inherits from the complexity of the forcing relation for the same requirements.

Exercise 4.17. Let $(\mathbb{P}, \leqslant)$ be a Cantor forcing, and $\Vdash$ a forcing relation. Show that the relation defined by $c \mathrel{?{\vdash}} \mathcal{R}$ if there exists an extension $d \leqslant c$ such that $d \Vdash \mathcal{R}$ is a forcing question. ◇

4.3.1. Preservation of the arithmetic hierarchy

A certain number of weakness properties of sufficiently generic sets for Cantor forcings depend on the existence of a forcing question with good definitional properties. In what follows, we fix a Cantor forcing $(\mathbb{P}, \leqslant)$ as well as a forcing question $?{\vdash}$. The following Proposition 4.19 is the first example, and generalizes Theorem 10-5.7.

Definition 4.18. A forcing question $?{\vdash}$ *preserves the arithmetical hierarchy* if for any condition c and any Σ^0_n requirement $\mathcal{R}$, the relation $c \mathrel{?{\vdash}} \mathcal{R}$ is Σ^0_n uniformly in $\mathcal{R}$. ◇

Proposition 4.19. *Let $?{\vdash}$ be a forcing question which preserves the arithmetic hierarchy. Then, for any $n \geqslant 1$, for any set A which is not Σ^0_n, and for any sufficiently generic set G, A is not $\Sigma^0_n(G)$.* ⋆

Proof. For all $e \in \mathbb{N}$, let

$$D_e = \{c \in \mathbb{P} : (\exists m \notin A\ c \Vdash m \in W_e^{G^{(n-1)}}) \text{ or } (\exists m \in A\ c \Vdash m \notin W_e^{G^{(n-1)}})\}$$

Let us show that D_e is a dense set in $(\mathbb{P}, \leqslant)$. Let $c \in \mathbb{P}$ and let

$$U = \{m \in \mathbb{N} : c \mathrel{?{\vdash}} m \in W_e^{G^{(n-1)}}\}$$

According to Lemma 10-5.6, the requirement $m \in W^{G^{(n-1)}}$ is Σ^0_n uniformly in m, so since the forcing question preserves the arithmetic hierarchy, the set U is Σ^0_n. The set A not being Σ^0_n, then the symmetric difference $U \Delta A = (U \setminus A) \cup (A \setminus U)$ is not empty. Let $m \in U \Delta A$.

Case 1: $m \in U \setminus A$. Then, by definition of the forcing question, there exists an extension $d \leqslant c$ such that $d \Vdash m \in W_e^{G^{(n-1)}}$. In particular, $d \in D_e$.

Case 2: $m \in A \setminus U$. Then, still by the definition of the forcing question, there exists an extension $d \leqslant c$ such that $d \Vdash m \notin W_e^{G^{(n-1)}}$. Again, $d \in D_e$.

In all cases, there is an extension of d in D_e, so the set D_e is dense. Let F be a sufficiently generic filter for $(\mathbb{P}, \leqslant)$. By density of D_e, we can assume that F intersects D_e for all $e \in \mathbb{N}$. Let $G = \dot{F}$. By definition of the forcing relation, for all $e \in \mathbb{N}$, either $m \in W_e^{G^{(n-1)}}$ for an $m \notin A$, or $m \notin W_e^{G^{(n-1)}}$ for an $m \in A$, so A is not $\Sigma^0_n(G)$. ■

Corollary 4.20
Let $?\vdash$ be a forcing question which preserves the arithmetic hierarchy. Then, for any $n \geqslant 0$ and any set A not $\emptyset^{(n)}$-computable, for any sufficiently generic set G, A is not $G^{(n)}$-computable.

Proof. As A is not $\emptyset^{(n)}$-computable, it is not Δ^0_{n+1}. Either A is not Σ^0_{n+1}, or $\overline{A}$ is not Σ^0_{n+1}. By Proposition 4.19, for any set G sufficiently generic for $(\mathbb{P}, \leqslant)$, A is not $\Sigma^0_{n+1}(G)$ in the first case, and $\overline{A}$ is not $\Sigma^0_{n+1}(G)$ in the second case. In any case, A is not $\Delta^0_{n+1}(G)$ and therefore not $G^{(n)}$-computable. ∎

Corollary 4.21
Let $n \geqslant 0$ and $A \in 2^{\mathbb{N}}$ be a non $\emptyset^{(n)}$-computable set. If G is sufficiently generic for Jockusch-Soare forcing or for Sacks forcing, A is not $G^{(n)}$-computable.

Proof. According to Proposition 4.14, these two forcing notions preserve the arithmetic hierarchy. ∎

4.3.2. Preservation of hyperimmunity

Recall that a function $f : \mathbb{N} \to \mathbb{N}$ is hyperimmune relative to X if it is not dominated by any X-computable function (see Section 7-4). No function is hyperimmune relative to all Turing degrees, starting with the Turing degree of the function itself. However, if a function is hyperimmune, it is also hyperimmune relative to any computably dominated degree. We can therefore consider that the computably dominated degrees "preserve" the hyperimmunities of all functions simultaneously. We will now study to what extent sufficiently generic sets for Cantor forcings preserve hyperimmunities.

We saw in Section 10-3.1 that any weakly 1-generic set X was of hyperimmune degree. More precisely, the principal function p_X of X which to n associates the n-th element of X is hyperimmune. The sets which are sufficiently generic for Cohen forcing therefore do not preserve all the hyperimmunities simultaneously. However, it is possible to ensure that they preserve hyperimmunity from any fixed hyperimmune function.

Definition 4.22. A forcing question $?\vdash$ is *compact* if for any $c \in \mathbb{P}$, any arithmetic requirement $\mathcal{R}(x)$, if $c \mathrel{?\vdash} \exists x \mathcal{R}(x)$, then there exists a finite set $U \subseteq \mathbb{N}$ such that $c \mathrel{?\vdash} \exists x \in U\ \mathcal{R}(x)$. ♢

The compactness of a forcing question is sufficient to ensure preservation of hyperimmunity:

Proposition 4.23. Let $?\vdash$ be a compact forcing question which preserves the arithmetic hierarchy. Then, for any $n \geqslant 0$ and any hyperimmune function $f : \mathbb{N} \to \mathbb{N}$ relative to $\emptyset^{(n)}$, for any set G sufficiently generic for $(\mathbb{P}, \leqslant)$, f is hyperimmune relative to $G^{(n)}$. ⋆

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be a hyperimmune function relative to $\emptyset^{(n)}$. For all $e \in \mathbb{N}$, let

$$D_e = \{c \in \mathbb{P} : (\exists m\ c \Vdash \Phi_e(G^{(n)}, m)\uparrow) \text{ or } (\exists m\ c \Vdash \Phi_e(G^{(n)}, m)\downarrow < f(m))\}$$

Let us show that D_e is a dense set in $(\mathbb{P}, \leqslant)$. Let $c \in \mathbb{P}$ and let $g : \mathbb{N} \to \mathbb{N}$, the partial function which for all m, searches for a finite set $U \subseteq \mathbb{N}$ such that $c\,?\vdash \Phi_e(G^{(n)}, m)\downarrow\ \in U$. If such a set U is found, $g(m) = 1 + \max U$, otherwise $g(m)$ is not defined. Knowing that the forcing question preserves the arithmetic hierarchy, g is partial $\emptyset^{(n)}$-computable. Two cases arise:

Case 1: There is an m such that $g(m)$ is not defined. Then, by compactness of the forcing question, $c\,?\nvdash \Phi_e(G^{(n)}, m)\downarrow$, so there exists a $d \leqslant c$ such that $d \Vdash \Phi_e(G^{(n)}, m)\uparrow$. In particular, $d \in D_e$.

Case 2: The function g is total $\emptyset^{(n)}$-computable. By hyperimmunity of f relative to $\emptyset^{(n)}$, there is an $m \in \mathbb{N}$ such that $g(m) \leqslant f(m)$. In particular, $c\,?\vdash \Phi_e(G^{(n)}, m)\downarrow\ \in U$ for a finite set U such that $\max U < f(m)$. By definition of a forcing question, there is an extension $d \leqslant c$ such that $d \Vdash \Phi_e(G^{(n)}, m)\downarrow\ \in U$, therefore $\Phi_e(G^{(n)}, m)\downarrow < f(m)$. It follows that $d \in D_e$.

In all cases, there is an extension of d in D_e, so the set D_e is dense. Let F be a sufficiently generic filter for $(\mathbb{P}, \leqslant)$. By density of D_e, we can assume that F intersects D_e for all $e \in \mathbb{N}$. Let $G = \dot{F}$. By definition of the forcing relation, for all $e \in \mathbb{N}$, either $\Phi_e(G^{(n)})$ is a partial function, or $\Phi_e(G^{(n)}, m)\downarrow < f(m)$ for an $m \in \mathbb{N}$, therefore f is hyperimmune relative to $G^{(n)}$. ■

The canonical forcing question of Cohen forcing is compact and preserves the arithmetic hierarchy, which implies that any set sufficiently generic for Cohen forcing preserves the hyperimmunity of any previously fixed function. It can be shown that the same goes for the forcings of Jockusch-Soare and Sacks.

4.3.3. Preservation of non PA degrees

We will end the study of the properties of the forcing questions with a criterion for not computing a PA degree.

Definition 4.24. A forcing question $?\vdash$ is Π-*merging* if for any $c \in \mathbb{P}$, any pair of Π^0_n requirements $\mathcal{R}_0$, $\mathcal{R}_1$ such that $c \mathrel{?\vdash} \mathcal{R}_0$ and $c \mathrel{?\vdash} \mathcal{R}_1$, there is an extension $d \leqslant c$ which simultaneously forces $\mathcal{R}_0$ and $\mathcal{R}_1$. $\diamondsuit$

Proposition 4.25. Let $?\vdash$ be a Π-merging forcing question which preserves the arithmetic hierarchy. Then, for any $n \geqslant 0$ for any set G sufficiently generic for $(\mathbb{P}, \leqslant)$, $G^{(n)}$ is not of PA degree relative to $\emptyset^{(n)}$. $\star$

Proof. According to Theorem 8-6.2, a Turing degree is PA iff it computes a $\{0,1\}$-valued DNC function. In what follows, we will assume that $\Phi_0, \Phi_1, \dots$ is an enumeration of all $\{0,1\}$-valued Turing functionals. For all $e \in \mathbb{N}$, let

$$D_e = \left\{ c \in \mathbb{P} : \begin{array}{ll} & (\exists m\ c \Vdash \Phi_e(G^{(n)}, m)\uparrow) \\ \text{or} & (\exists m\ c \Vdash \Phi_e(G^{(n)}, m)\downarrow = \Phi_e(\emptyset^{(n)}, m)) \end{array} \right\}$$

Let us show that D_e is a dense set in $(\mathbb{P}, \leqslant)$. Let $c \in \mathbb{P}$ and let $g : \mathbb{N} \to \mathbb{N}$, the partial function which for any m, searches for an integer $v \in \{0,1\}$ such that $c \mathrel{?\vdash} \Phi_e(G^{(n)}, m)\downarrow = v$. If such a v is found, $g(m) = v$, otherwise $g(m)$ is not defined. Knowing that the forcing question preserves the arithmetic hierarchy, g is partial $\emptyset^{(n)}$-computable. Two cases arise:

Case 1: There is an m such that $g(m)$ is not defined. Then, for all $v \in \{0,1\}$, $c \mathrel{?\nvdash} \Phi_e(G^{(n)}, m)\downarrow = v$, in other words $c \mathrel{?\vdash} \neg(\Phi_e(G^{(n)}, m)\downarrow = v)$. As the relation is Π-mergeable, there exists a $d \leqslant c$ such that d force at the same time $\neg(\Phi_e(G^{(n)}, m)\downarrow = 0)$ and $\neg(\Phi_e(G^{(n)}, m)\downarrow = 1)$, however the functional being $\{0,1\}$-valued, d therefore forces $\Phi_e(G^{(n)}, m)\uparrow$. In particular, $d \in D_e$.

Case 2: The function g is total $\emptyset^{(n)}$-computable. Knowing that no degree is PA relative to itself, there exists an integer $m \in \mathbb{N}$ such that $g(m) = \Phi_m(\emptyset^{(n)}, m)$. In particular, $c \mathrel{?\vdash} \Phi_e(G^{(n)}, m)\downarrow = g(m)$, so by definition of a forcing question, there exists an extension $d \leqslant c$ such that $d \Vdash \Phi_e(G^{(n)}, m)\downarrow = g(m) = \Phi_m(\emptyset^{(n)}, m)$. It follows that $d \in D_e$.

In all cases, there is an extension of d in D_e, so the set D_e is dense. Let F be a sufficiently generic filter for $(\mathbb{P}, \leqslant)$. By density of D_e, we can assume that F intersects D_e for all $e \in \mathbb{N}$. Let $G = \dot{F}$. By definition of the forcing relation, for any $e \in \mathbb{N}$, either $m \mapsto \Phi_e(G^{(n)}, m)$ is a partial function, or $\Phi_e(G^{(n)}, m)\downarrow = \Phi_m(\emptyset^{(n)}, m)$ for an $m \in \mathbb{N}$, therefore $G^{(n)}$ is not of PA degree relative to $\emptyset^{(n)}$. ∎

The forcing question for Cohen forcing is Π-merging, just like the forcing question for computable Sacks forcing. On the other hand, there is no Π-

merging forcing question for Jockusch-Soare forcing, because there exist non-empty Π^0_1 classes containing only sets of PA degree.

Exercise 4.26. A forcing question $?\vdash$ is Π-ω-*merging* if for any $c \in \mathbb{P}$, any sequence of Π^0_n requirements $\mathcal{R}_0, \mathcal{R}_1, \dots$ such that $c \,?\!\vdash \mathcal{R}_i$ for all $i \in \mathbb{N}$, there exists an extension $d \leqslant c$ which simultaneously forces $\mathcal{R}_i$ for all i. Show that if $?\vdash$ is a Π-ω-merging forcing question which preserves the arithmetic hierarchy, then for any set G sufficiently generic and all n, its iterated jump $G^{(n)}$ is not of DNC degree relative to $\emptyset^{(n)}$. $\diamond$

Chapter 12

Quest for natural degrees

The beginnings of computability theory enabled to observe that all computable enumerable sets originating from natural problems were either computable or as powerful as the halting problem, moreover via a many-one reduction (see Section 5-4). This led Post to ask the following question in 1944:

Question (Post's problem [189]). Are there non-computable c.e. degrees which are strictly weaker than the halting problem? $\star$

Post's problem remained open for almost a decade, before being solved in the affirmative by Muchnik [168] and Friedberg [66] via the priority method, which we will see in Section 13-3. Post's problem has since seen many other different resolutions, not necessarily using the priority method. We can cite for example the construction of a non-computable K-trivial c.e. set that we will see with Theorem 16-4.5. However, all these constructions are based on a complex argument allowing the ad-hoc construction of an "artificial" set having the desired properties, and the only undecidable "natural" decision problems known to date are reduced to the halting problem[1]. Conversely, the halting problem seems to arise naturally all over the place. The question then arose of the properties that give it this naturalness.

[1] This statement should be taken with caution, however, and will be attenuated in the last section of this chapter.

1. Three emblematic undecidable problems

Before tackling Post's problem directly, let us see three emblematic examples of undecidable c.e. decision problems, all many-one equivalent to the halting problem.

Post correspondence problem

We start with a problem defined by Post himself in 1946, which is called the Post correspondence problem, and which should not be confused with "Post's problem" which refers to the above question.

Given two finite lists of strings $\sigma_0, \ldots, \sigma_n \in 2^{<\mathbb{N}}$ and $\tau_0, \ldots, \tau_n \in 2^{<\mathbb{N}}$, is there a sequence of indices $(i_k)_{k \leqslant K}$ —possibly with repetition— such that the concatenations $\sigma_{i_0}\sigma_{i_1} \ldots \sigma_{i_K}$ and $\tau_{i_0}\tau_{i_1} \ldots \tau_{i_K}$ form the same string?

The question may seem simple on the surface, and one might even think at first that it is easy to create an algorithm to solve it. After all, the number of strings involved is finite. On further reflection, the problem shouldn't appear so obvious, and for good reason: although it may seem surprising, it is an undecidable question, and as difficult as whether a computer program halts or not.

To show it, Post finds a nifty way to encode the computation of a Turing machine via instances of the correspondence problem. Thus, an instance for which a correspondence exists will correspond to a computation which halts, and an instance for which no correspondence exists will correspond to an infinite computation.

Domino problem

In 1961, Hao Wang imagines the following problem: given a finite set of *tiles*, that is to say of squares having a color on each of their sides, the objective is to make a tiling of the plan using only tiles in our finite set, requiring that two neighbour tiles share the same color on their common side. There are of course sets of tiles for which such a tiling of the plane is possible, and others for which it is not.

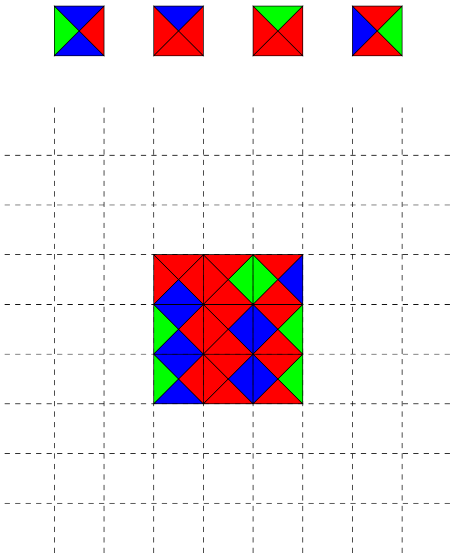


Figure 1.1: Start of a tiling of the plan using the four tiles above.

A lemma deriving from that of König applies to tilings of the plane using a finite number of tiles: if for all n there exists a tiling of $n \times n$ tiles, then there exists an infinite tiling of the plane. We then deduce that if a finite set of tiles does not allow the plane to be paved, there exists n such that no tiling of size $n \times n$ is possible. If a tiling is impossible, it suffices to look for the smallest n such that no tiling configuration of size $n \times n$ will work. In other words, the finite sets of tiles which do not allow the plane to be paved can be enumerated by an algorithm.

Wang then conjectures that the same is true for finite sets of tiles making it possible to pave the plane, which would make the problem of tiling the plane decidable: there would exist an algorithm making it possible to decide, given a finite set of tiles if the latter may or may not pave the plane.

But in 1966, Robert Berger (a student of Wang) shows that the problem of paving the plane is not decidable, by reducing to it again the halting problem for Turing machines, in the manner of Post: Berger creates a paving system allowing to "simulate" the computation being carried out on a Turing machine, an impossible paving meaning that the machine halts, and a possible paving meaning that the computation continues indefinitely.

Tenth Hilbert problem

A Diophantine equation is a polynomial equation with one or more unknowns, whose solutions are sought among integers — or possibly rational

— the coefficients being themselves also integers. For example $a^2 + b^2 = c^2$ is a Diophantine equation having many solutions like $a = 3, b = 4$ and $c = 5$. On the other hand, there are Diophantine equations having no solution. Some of them have asked for their resolution — to find the solutions or show that they do not have any — considerable efforts from many mathematicians over several centuries. The example par excellence is certainly the famous "Fermat's last theorem" which states that for any integer $n > 2$, the equation $a^n + b^n = c^n$ has no integer solutions.

Fermat states his theorem in the margin of a translation of "Arithmetic of Diophantus", in which he writes: "*On the contrary, it is impossible to divide either a cube into two cubes, or a Quadruple in two quadruple, that is in general any power greater than the square in two powers of the same degree: I have discovered a truly marvelous demonstration of this, which this margin is too narrow to contain*".

Many mathematicians have sought this wonderful demonstration for centuries without success. It is only after 357 years of efforts that the mathematician Andrew Wiles, helped by his student Richard Taylor, will provide a proof using mathematical tools obviously much more complex than those which existed at the time of Fermat.

In 1900, Hilbert places the question of solving Diophantine equations in tenth position in his famous list of 23 problems: "*Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.*"

Hilbert asks, before we have a formal definition, the existence of an *algorithm* allowing to know if any Diophantine equation admits a solution or not.

Here again, the set of Diophantine equations having a solution is computably enumerable; it suffices to search among all the potential candidates if one of them is a solution. Martin Davis, Hilary Putnam and Julia Robinson had the idea of showing that Hilbert's tenth problem is undecidable by following a daring intuition, and which turns out to be correct: a Diophantine equation is the building block of a formula of arithmetic, in this case the equality between two terms. Gödel showed that computably enumerable sets are exactly those which can be defined by Σ^0_1 formulas of arithmetic.

Would it not be possible to transform such a formula into an equivalent Σ^0_1 formula $\exists x_1 \ldots \exists x_n \; F(x_1, \ldots, x_n)$, but where F is no more than a big Diophantine equation?

Consider for example the case of the formula

$$t_1(a_1,\ldots,a_i) = q_1(b_1,\ldots,b_j) \vee t_2(x_1,\ldots,x_k) = q_2(y_1,\ldots,y_l),$$

where t_1, t_2 and q_1, q_2 are terms. Then, this formula is true in $\mathbb{N}$ iff the formula

$$t_1(a_1,\ldots,a_i) - q_1(b_1,\ldots,b_j) = 0 \vee t_2(x_1,\ldots,x_k) - q_2(y_1,\ldots,y_l) = 0$$

is true in $\mathbb{Z}$, or equivalently if the Diophantine equation

$$\big(t_1(a_1,\ldots,a_i) - q_1(b_1,\ldots,b_j)\big)\times\big(t_2(x_1,\ldots,x_k) - q_2(y_1,\ldots,y_l)\big) = 0$$

admits solutions. We easily show something similar for the connector $\wedge$, the remaining difficulty being in the deletion of existential and bounded universal quantifications. Davis, Putnam, and Robinson succeeded in removing bounded quantifications at the cost of using the exponential function in the resulting equations. The work will then be completed by Matiiassevitch, who succeeded in encoding the exponential function in Diophantine equations, leading to the following theorem.

Theorem 1.2 (MRDP theorem)
Let $A \subseteq \mathbb{N}$ be a computably enumerable set. Then, there is a Σ^0_1 formula of arithmetic $F(x) = \exists y_1,\ldots,\exists y_n\ G(x,y_1,\ldots,y_n)$ where G has no quantifier, such that $x \in A$ iff $\mathbb{N} \models F(x)$.

The MRDP theorem is remarkable in that it illustrates the fact that the undecidability of Peano arithmetic is concealed in the very structure of addition and multiplication of integers, without any need to resort to bounded quantifications. It provides of course an answer to the tenth problem of Hilbert: if an algorithm makes it possible to know if a Diophantine equation has integer solutions, then one can create an algorithm computing the halting problem.

2. Natural Turing degrees

Let's go back to our original question: what is special about the halting problem, so that all natural c.e. decision problems are equivalent to it? What makes a Turing degree natural? Steel [226] suggests an answer: a natural Turing degree should be definable, and its definition should be relativizable to any degree. For example, the halting problem $\emptyset'$, initially defined as a particular set, namely $\{e : \Phi_e(e)\downarrow\}$, is relativized to any set X, by considering the set $X' = \{e : \Phi_e(X,e)\downarrow\}$. As we saw in Section 4-6, this is a notion on Turing degrees, in the sense that if $X \equiv_T Y$, then $X' \equiv_T Y'$.

2.1. Sacks question

Steel's idea echoes an old question from Sacks [196]: is there a solution to Post's problem that is invariant on Turing degrees? We will say that W is a *c.e. operator* if W corresponds to a Turing functional which, uniformly in a set X, enumerates a set which we will denote by W^X. Sacks asks if there is a c.e. operator W such that $X <_T W^X <_T X'$ for all X and such that $X_0 \equiv_T X_1$ implies $W^{X_0} \equiv_T W^{X_1}$ for all X_0, X_1.

By working on it a little, the reader will be able to see that the priority method used in Theorem 13-3.1 to construct a c.e. set Y such that $0 <_T Y <_T \emptyset'$, can be relativized to any oracle X to obtain a set X-c.e. Y such that $X <_T Y <_T X'$. On the other hand, it is much more uncertain that this relativization is invariant in the Turing degrees, and in fact, it is not. The first result in this direction was obtained by Lachlan, who gave a negative answer to Sacks' question, in the particular case where invariance is expected to be uniform, i.e., that we ask for the existence of functions h_1, h_2 such that if $\Phi_{a_1}(X_1) = X_2$ and $\Phi_{a_2}(X_2) = X_1$, then $\Phi_{h_1(a_1)}(W^{X_1}) = W^{X_2}$ and $\Phi_{h_2(a_2)}(W^{X_2}) = W^{X_1}$. Note that Lachlan does not require that the functions h_1, h_2 are computable, but simply that they exist.

In the case where the operator is invariant in the Turing degrees, the notation $W(\mathbf{a})$ for a Turing degree $\mathbf{a}$ has a meaning: it is the Turing degree obtained by applying to W any set in the degree $\mathbf{a}$. Lachlan actually shows the following: for any c.e. operator uniformly invariant such that $W(\mathbf{d}) \geqslant \mathbf{d}$ for any degree $\mathbf{d}$, then there exists a degree $\mathbf{a}$ such that for any degree $\mathbf{d} \geqslant \mathbf{a}$ we have $W(\mathbf{d}) = \mathbf{d}$, or then such that for any degree $\mathbf{d} \geqslant \mathbf{a}$ we have $W(\mathbf{d}) = \mathbf{d}'$. In the first case we will say that W coincides with the identity *on a cone*, and in the second that W coincides with the Turing jump *on a cone*. The expression *on a cone* then signifying on a cone in the Turing degrees, that is to say on all the degrees greater than $\mathbf{a}$ for a certain $\mathbf{a}$, the degree $\mathbf{a}$ being *the base of the cone*. Lachlan therefore obtains the following result.

Theorem 2.1 (Lachlan [136])
Let W be a c.e. operator uniformly invariant such that $W(\mathbf{d}) \geqslant \mathbf{d}$ for any degree $\mathbf{d}$. Then, W is the jump operator on a cone or W is the identity operator on a cone.

2.2. Sacks question for any degree

We have so far talked about natural c.e. degrees, arguing that $\mathbf{0}$ and $\mathbf{0}'$ are the only ones. There are, however, many natural decision problems that are strictly more powerful than the halting problem, and which are of course not c.e. We can then push the question of naturality to any degree. Let's see some canonical examples first:

1. P: The problem of determining if a polynomial of $\mathbb{Z}[x, y_0, y_1, y_2, \dots]$ has solutions for any sufficiently large element x.
2. T: The problem of knowing if a statement of arithmetic is true in $\mathbb{N}$.
3. WF: The problem of knowing if a computable tree of $\mathbb{N}^{<\mathbb{N}}$ has infinite paths.

The problem P is Σ^0_2 and and $\emptyset'' \leqslant_m P$. The problem T is such that $\emptyset^{(n)} \leqslant_m T$ for all $n \in \mathbb{N}$ uniformly in n. It is moreover many-one computable with the oracle $\emptyset^{(\omega)} = \oplus_n \emptyset^{(n)}$ and therefore corresponds to the first transfinite level in the hierarchy of Turing jumps (we will see this formally in Part IV). The WF problem is even more complex, and corresponds to a higher transfinite Turing jump that can be noted $\emptyset^{(\omega_1^{ck})}$ and which is commonly called *hyperjump* (we will also see this formally in Part IV).

Note that the iterations of the Turing jump, just like the simple Turing jump, are also relativized to any oracle invariantly on Turing degrees: for example if $X \equiv_T Y$ then $X^{(3)} \equiv_T Y^{(3)}$. The iterated jump operators are therefore also natural, and one can find for each of them decision problems which correspond to them. At the same time, we can iterate Sacks' question: the solutions to Post's problem are not the only constructions of exotic degrees in computability theory, and this question of the invariance of the operators can be extended to any construction. Consider for example the construction of a computably dominated set of Theorem 7-5.6 or that of Theorem 8-4.5. In both cases, the construction requires $\emptyset''$. One verifies easily with one or the other construction that one can create a $\emptyset''$-computable operator W such that for all X the set W^X verifies $X <_T W^X$ and is such that W^X is computably dominated relative to X. Can such an operator be invariant in the Turing degrees? If this is not the case, can we get one that can be computed in $\emptyset'''$ or in a more powerful oracle?

2.3. Martin's conjecture

Inspired by these questions, and perhaps by Lachlan's result on the c.e. operators, Martin then proposes a rather daring conjecture which essentially says: the jump operator and its iterations, are the only definable and invariant operators in Turing degrees. More formally, the conjecture has two distinct parts:

Conjecture 2.2 (Martin [1] p. 281). Let $f : 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ be a Borel function invariant in the Turing degrees. Then, we can see f as a function on the Turing degrees, in which case we have:

I. Either f is constant on a cone, or f is increasing on a cone.

II. If f is strictly increasing on a cone, then it corresponds to the Turing jump or to one of its iterations (possibly transfinite). ⋆

What do we mean by Borel function? Before answering it, let us note that if we go completely out of the framework of computability theory, we can perfectly define functions f invariant in Turing degrees such that $\mathbf{a} < f(\mathbf{a}) < \mathbf{a}'$ for all $\mathbf{a}$, simply using the axiom of choice in set theory. The objective of restricting the conjecture to Borel functions f essentially aims to prohibit the use of this axiom, and the conjecture is generally presented without the restriction for f to be Borel, and with additional assumptions on the axioms of set theory, essentially meaning: "for any function f that can be defined without using the axiom of choice".

Martin's conjecture essentially tells us that the only natural and non-constant operators in the Turing degrees are the identity operator, the jump operator and its iterations. If this conjecture is still open to this day, much progress has been made, by adding to it, as Lachlan did, the condition of uniformity in the invariance of the functions f considered. First, Steel [226] showed II of Martin's conjecture, for uniformly invariant functions, thus generalising the Lachlan result considerably. Then, still for uniformly invariant functions, Slaman and Steel [213] have shown I, and also managed to get rid, via a very clever proof, of uniformity for the case of functions f such that $f(\mathbf{a}) < \mathbf{a}$ for any degree $\mathbf{a}$ on a cone: these functions are necessarily constant on a cone.

To date, not much is known about the conjecture in the general case, and it is also also valid for the potentially much simpler question of Sacks, which also remains open in the case of non-uniformity. This story of non-uniformity has however something to instill doubt: if we basically seek to construct functions invariant in the Turing degrees, we always obtain a uniform invariance. This finding led Steel to make the following conjecture:

Conjecture 2.3 (Steel [226]). If a Borel function is invariant in the Turing degrees, then it is uniformly invariant. ⋆

If Steel's conjecture is true, then it will show Martin's conjecture.

3. Mass problems

Faced with Post's problem, we saw a first negative response, namely that under naturality assumptions, there are no c.e. sets of intermediary degree. There is another approach, complementary to the first, which consists in saying that if the iterations of the Turing jump are the only natural degrees, it is because of the too restrictive nature of a Turing degree: there are computational powers which do not correspond to Turing degrees taken individually, but to classes of degrees.

Consider for example the completions of Peano arithmetic. We have seen with Theorem 9-3.10 a proof that any complete and consistent theory which extends Peano arithmetic is non-computable, but we did not do it by showing that the halting problem could be reduced to such a theory, and for good reason: it is not always the case. There exist Δ^0_2 PA degrees not computing the halting problem. This is not incompatible with Martin's conjecture, in the sense that if one seeks to define a very specific natural extension of Peano arithmetic which is complete and consistent, one will find one which computes the halting problem or more. This is the case for example of the set of true formulas in $\mathbb{N}$.

The PA degrees form a *natural* computational power as a class: for any computable infinite binary tree, there exists a procedure which takes a completion of Peano arithmetic as input, and computes a path of the tree in return. Here we go beyond the naturality of the degrees, to consider instead the naturality of the classes of degrees. As we have seen in this book, there is a wide variety of classes of degrees, all defined in a very natural way, and which do not correspond to iterations of the halting set: the high, hyperimmune, PA degrees ... The notion of class of degrees is generally approached via the principle of *mass problems*. These go back to Kolmogorov [123], who speaks informally of them as a formalization of Brouwer's principles of intuitionist logic, before being rigorously defined by Medvedev [159] and Muchnik [170]:

Definition 3.1. A *mass problem* $\mathcal{P} \subseteq 2^{\mathbb{N}}$ is seen as the set of its possible solutions, identified, via an appropriate encoding, to elements of $2^{\mathbb{N}}$. $\diamondsuit$

For example, a problem will be solvable if it contains a computable element. Problems are studied in particular through the balance of power they maintain with one another. Muchnik suggests the following approach:

Definition 3.2 (Muchnik reduction). A mass problem $\mathcal{P}$ is *Muchnik reducible* to a mass problem $\mathcal{Q}$, in which case one notes $\mathcal{P} \leqslant_w \mathcal{Q}$ if any solution to $\mathcal{Q}$ allows to compute a solution to $\mathcal{P}$. $\diamondsuit$

For example, any non-empty Π^0_1 class reduces in Muchnik's sense to the problem consisting of complete and consistent extensions of PA. Medvedev proposes a more restrictive definition asking for uniformity:

Definition 3.3 (Medvedev reduction). A mass problem $\mathcal{P}$ is *Medvedev* *reducible* to mass problem $\mathcal{Q}$, in which case write $\mathcal{P} \leqslant_s \mathcal{Q}$ if there exists a functional Φ such that $\Phi(X) \in \mathcal{P}$ for all $X \in \mathcal{Q}$. $\diamondsuit$

Any non-empty Π^0_1 class is also reduced in Medvedev's sense to the class of PA sets. The two concepts do not, however, coincide in the general case. Jockusch [109] for example showed that the class of DNC_2 functions was reduced in the sense of Muchnik to that of DNC_3 functions, but not in the sense of Medvedev.

The equivalence classes of the relations $\equiv_w$ and $\equiv_s$ (whose definitions result from $\leqslant_w$ and $\leqslant_s$) are respectively called *Muchnik degrees* and *Medvedev degrees*, the structure of which has been extensively studied. There is a natural embedding of the Turing degrees towards the Muchnik and Medvedev degrees, by associating with a Turing degree $\deg_T(X)$ the Muchnik or Medvedev degree of the problem $\{X\}$. This embedding respects the semi-lattice structure. We can therefore consider the Muchnik and Medvedev degrees as an extension of the Turing degrees. In particular, we can define $\mathbf{0}_w$ and $\mathbf{0}'_w$, the Muchnik degrees of $\{\emptyset\}$ and $\{\emptyset'\}$, respectively. So the following proposition is in some way related to Post's original question.

Proposition 3.4. Let PA be the Muchnik degree of the PA degrees. Then

$$\mathbf{0}_w <_w \mathrm{PA} <_w \mathbf{0}'_w$$

This generalization of the Turing degrees comes at a cost: the Muchnik and Medvedev degrees are much more numerous. More precisely, the Turing degrees have the power of the continuum ($|2^{\mathbb{N}}|$) while the Muchnik and Medvedev degrees have cardinality $|2^{2^{\mathbb{N}}}|$ and have anti-chains of this size.

Chapter 13

Priority method and c.e. degrees

Among the non-computable sets, the computably enumerable sets play a particularly important role. These sets are "almost computable", in the sense that if an integer n belongs to a c.e. set A, then this information will be known in a finite time. The class of computably enumerable sets is quite natural, for several reasons.

First, the c.e. sets have several very different characterizations, which makes it a relatively *robust* class. By definition, a set is c.e. if it is the domain of a partial computable function. The non-empty c.e. sets are also precisely those which are the image of a total computable function, the function being able to be injective if the set is infinite (see Proposition 3-7.2). Equivalently, a set is c.e. if and only if it is reducible to the halting problem by a many-one reduction, or if it is Σ^0_1.

Computably enumerable sets form a *syntactic class* unlike computable sets. Indeed, the c.e. sets are precisely the Σ^0_1 sets, while the computable sets are the sets definable by both a Σ^0_1 and Π^0_1 predicate. This syntactic nature gives the c.e. sets better uniformity properties. Thus, the class of computably enumerable sets is uniformly c.e., Because it suffices to list all the partial computable functions, or in an equivalent way all the Σ^0_1 formulas. The computable sets cannot, on the other hand, be listed in a computable way (according to Theorem 7-6.2, the high degrees are exactly those allowing to list the computable sets).

Finally, and this is perhaps one of the most important arguments in favor of the naturality of c.e. sets, A number of non-decidable problems in mathematics happen to be computably enumerable. Among them, we will of course cite the halting problem, but also the set of theorems of arithmetic (see Theorem 9-3.7), or even the set of solutions of Diophantine equations (see, for this purpose, Theorem 12-1.2).

1. C.e. degrees

Being computably enumerable is a property of a set and not of a Turing degree. Indeed, we have seen that the Turing degrees are closed under complementation, or by Proposition 3-7.4, if a set and its complement are c.e., Then they are computable. In particular, the decision problem $\emptyset'$ is c.e., but is Turing-equivalent to its complement which is not. However, we have defined a notion of c.e. degree at the start of Section 7-3, definition that we repeat here:

Definition 1.1. A Turing degree is *c.e.* if it contains a computably enumerable set. ◇

We saw that the Turing degrees are not bounded, because for any degree $\mathbf{d}$, its Turing jump $\mathbf{d}'$ is strictly above. The c.e. degrees, on the other hand, are bounded by $\mathbf{0}'$.

Proposition 1.2. The c.e. degrees have for maximum degree $\mathbf{0}'$. ⋆

Proof. By Post's theorem (see Proposition 5-4.3), a set is c.e. if and only if it is many-one reducible to $\emptyset'$. The many-one reductions being special cases of Turing reductions, any c.e. degree is Turing-reducible to $\emptyset'$. ■

The c.e. degrees forming a subset of the Turing degrees, questions about Turing degrees also apply to computably enumerable degrees. Are they linearly ordered? Do they form a well-founded order? And before all this, are there c.e. degrees other than $\mathbf{0}$, the Turing degree of computable sets, and $\mathbf{0}'$, the Turing degree of the halting problem?

The c.e. degrees are notoriously difficult to handle, due to the computability constraint of their enumeration. The finite extension method is no longer suitable, and it will be necessary to appeal to very elaborate techniques to prove results similar to those obtained in the Turing degrees.

2. Permitting method

The permitting method allows to compute a c.e. set A from another c.e. set B. It is based on the notion of *computation function* seen at Section 4 -7. Recall that, given a c.e. approximation $(A_s)_{s\in\mathbb{N}}$ of a c.e. set A (or more generally for any Δ^0_2 approximation), the computation function associated with this approximation is the function $c_A : \mathbb{N} \to \mathbb{N}$ which to n associates the smallest time $s > n$ such that $A_s\restriction_n = A\restriction_n$. In particular, this function can be computed in A. Moreover, by Theorem 4-7.9, any dominating function c_A recomputes A.

Proposition 2.1 (Permitting method). Let A and B be c.e. sets of c.e. approximations $(A_s)_{s\in\mathbb{N}}$ and $(B_s)_{s\in\mathbb{N}}$, respectively. If there exists a B-computable function $f:\mathbb{N}\to\mathbb{N}$ such that for all $n,s\in\mathbb{N}$

$$A_{s+1}\restriction_n \neq A_s\restriction_n \quad\Rightarrow\quad B_{s+1}\restriction_{f(n)}\neq B_s\restriction_{f(n)},$$

then $A\leqslant_T B$. ⋆

Proof. For all n, $c_A(n)\leqslant c_B(f(n))$, but $c_B\oplus f\leqslant_T B$, so B computes a function dominating c_A. By Theorem 4-7.9, B computes A. ■

Most of the time, the function f will be computable and increasing, or even the identity function. Informally, the approximation of A is only allowed to add an element to A if at the same time, B adds an element smaller than $f(x)$. In other words, the set A waits for permission from B to add elements, which gives this method its name. The permitting method is often combined with other techniques, like the priority method, which we will see in the next section. The permitting method does not lose in generality, in the sense that we can prove the following reversal in the case where the set B is infinite.

Proposition 2.2. Let A and B be c.e. sets such that $A\leqslant_T B$ *and* B *is infinite*. Then, there exist c.e. approximations $(A_s)_{s\in\mathbb{N}}$ and $(B_s)_{s\in\mathbb{N}}$ and a B-computable function $f:\mathbb{N}\to\mathbb{N}$ such that for all $n,s\in\mathbb{N}$

$$A_{s+1}\restriction_n \neq A_s\restriction_n \quad\Rightarrow\quad B_{s+1}\restriction_{f(n)}\neq B_s\restriction_{f(n)}$$ ⋆

Proof. Let $(A_s)_{s\in\mathbb{N}}$ and $(B_s)_{s\in\mathbb{N}}$ be c.e. approximations of A and B, respectively. Knowing that B is infinite, by accelerating its c.e. approximation, we can assume without loss of generality that $B_{s+1}\neq B_s$ for all s. Let $f:\mathbb{N}\to\mathbb{N}$ be the function which to n associates the smallest integer m such that $B_{s+1}\restriction_m\neq B_s\restriction_m$ for all $s\leqslant c_A(n)$. The function f is $c_A\oplus B$-computable, or $c_A\leqslant_T A\leqslant_T B$, so $f\leqslant_T B$. For all $n,s\in\mathbb{N}$, if $A_{s+1}\restriction_n\neq A_s\restriction_n$, then $c_A(n)\geqslant s+1$, therefore $B_{s+1}\restriction_{f(n)}\neq B_s\restriction_{f(n)}$. ■

3. Σ_1^0 priority method (finite injury)

Post asked the question in 1944 [189] whether there are computably enumerable sets which are both non-computable and Turing incomplete, that is to say which do not allow as an oracle to compute the halting problem. The question remained open for more than a decade, before being solved in the affirmative independently by Muchnik [168] and Friedberg [66], who introduced the famous *priority method*. This technique will subsequently find many applications for the study of c.e. and Δ_2^0 sets, which turn out to have a very rich structure, as witnessed by Sacks' density theorem: let X, Y be c.e. sets such that $X <_T Y$. Then, there exists a c.e. set Z such that $X <_T Z <_T Y$.

In general, the priority method serves the same purpose as the finite extension method (see Section 4-8), i.e., to construct sets satisfying properties of strength and weakness, but this time by controlling the complexity of these sets in the arithmetic hierarchy. This additional constraint is at the origin of an explosion in the complexity of constructions. Indeed, like the finite extension method, it consists of satisfying an infinity of requirements simultaneously, but while for the finite extension method, the construction is omniscient, the priority method argument must be carried out with a limited computational power. It is therefore necessary to continue building the set without having full knowledge of the situation. The construction is therefore done by trial and error, with backtracking when an error is noticed. The strategies to satisfy the requirements come into conflict, and the whole difficulty of the construction lies in ensuring that these conflicts and backtracking do not prevent the overall goal: to build a set satisfying all the requirements.

We simply start here with the easiest use of the priority method, which was the first to be introduced, to solve Post's problem. This is a *finite injury* priority method, meaning that each strategy to fulfill a requirement will only have to backtrack a finite number of times before it can achieve its goal.

Theorem 3.1 (Friedberg (1957), Muchnik (1956))
There are incomparable c.e. sets for the Turing reduction.

The statement of Friedberg and Muchnick's theorem is similar to Kleene and Post's theorem (see Proposition 4-8.1), but imposes the additional constraint on sets to be computably enumerable.

Proof. As for the theorem of Kleene and Post (see Proposition 4-8.1), we are going to construct two sets, A and B, each satisfying a strength property and a weakness property. These properties are each declined in the form of two sets of requirements: $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$:

$$\mathcal{R}_e : W_e^A \neq \overline{B} \qquad \mathcal{S}_e : W_e^B \neq \overline{A}.$$

Recall the meaning of the notation W_e^A which designates the c.e. set relative to A of code e (ie $\{n \in \mathbb{N} : \Phi_e(A, n) \downarrow\}$). The requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ ensure that $A \not\geq_T B$ because the complement of B is not c.e. in A. Symmetrically, if the requirements $(\mathcal{S}_e)_{e\in\mathbb{N}}$ are simultaneously satisfied, then $B \not\geq_T A$.

Remark

The sets A and B being c.e., it is equivalent to say that they are not computable, or that their complement is not c.e. Thus, the requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$ are without loss of generality. This formulation of requirements simplifies notations.

Since the sets A and B must be c.e., We are going to enumerate elements in A and in B during a computable process. More formally, we will define two uniformly computable sequences of finite sets $A_0 \subseteq A_1 \subseteq \dots$ and $B_0 \subseteq B_1 \subseteq \dots$ starting with $A_0 = B_0 = \emptyset$. For presentation reasons, we will omit the index s and will speak of A and B as sets under construction and evolving over time. We will use the indices when necessary to distinguish the sets at different time stages. For each $\mathcal{R}_e$ or $\mathcal{S}_e$ requirement, we will describe a process responsible for its satisfaction. The different processes will run in parallel. A process responsible for the satisfaction of a requirement $\mathcal{R}_e$ (resp. $\mathcal{S}_e$) is called *strategy for* $\mathcal{R}_e$ (resp. $\mathcal{S}_e$). In this construction, only one strategy will be needed per requirement. We will see priority arguments later which will associate several strategies with each requirement.

Satisfaction of a requirement $\mathcal{R}_e$. Here is a strategy for constructing two c.e. sets A and B satisfying a unique requirement $\mathcal{R}_e$. To gain in generality and prepare the satisfaction of several requirements, we will also assume that other processes or strategies run in parallel, and add elements to A and B over time. At any time t, the strategy for $\mathcal{R}_e$ is found in one of the following three states:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{State } \mathtt{i}\\ \text{Initial}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\text{State } \mathtt{e}\\ \text{Execution}\end{array}} \arrow[r] & \boxed{\begin{array}{c}\text{State } \mathtt{t}\\ \text{Terminated}\end{array}}
\end{tikzcd}$$

State $\mathtt{i}$. This is the initial state of the process. To get out of this state, the process goes through an initialization phase which consists in choosing a unique integer $x_{\mathcal{R}_e} \notin B$. This number exists because at any time, the sets A and B have a finite number of elements. Once $x_{\mathcal{R}_e}$ is chosen, our process sets a *restraint* on $x_{\mathcal{R}_e}$, that is to say, it forbids other processes running in parallel to add $x_{\mathcal{R}_e}$ in B. Only our process is the decision maker of the enumeration of $x_{\mathcal{R}_e}$. Note that once $x_{\mathcal{R}_e}$ is added to B, it will no longer be able to leaver it, because the set B is c.e. Once the restraint is set, the process enters State $\mathtt{e}$, called execution.

State $\mathtt{e}$. During this phase, the process executes $\Phi_e^A(x_{\mathcal{R}_e})$ until it halts. If it never halts, the process remains in this state. Note that during this phase, the set A can evolve, because other processes can add elements to A in parallel. The process must take this evolution into account, and therefore compute $\Phi_e^{A_t}(x_{\mathcal{R}_e})[t]$ at time t. If $\Phi_e^{A_t}(x_{\mathcal{R}_e})\downarrow$ at a time t, then the process poses a *restraint* on the use of this computation, that is to say on the bits of the oracle A_t used for this execution, thus preventing other processes from making a modification to it. We therefore make sure that $\Phi_e^A(x_{\mathcal{R}_e})\downarrow$, in other words that $x_{\mathcal{R}_e} \in W_e^A$. The process then adds $x_{\mathcal{R}_e}$ to B, so that $W_e^A \neq \overline{B}$, and ends up in State `t`.

State `t`. In this state, the process has completed its execution. He does leave this state.

Outcomes. Let us study the different outcomes of the strategy for $\mathcal{R}_e$. The process always goes from State `i` to State `e`, but may never reach State `t`. We therefore have two possible outcomes. Outcome `p`: it remains stuck in the execution state (Qtate `e`). In this case, $\Phi_e^A(x_{\mathcal{R}_e})\uparrow$ and $x_{\mathcal{R}_e} \notin B$, so $W_e^A \neq \overline{B}$. We say that the requirement $\mathcal{R}_e$ is *passively satisfied.* Outcome `a`: it enters the termination state (State `t`). Then, by its actions of restraints and the enumeration of $x_{\mathcal{R}_e}$ in B, it ensures that the requirement $\mathcal{R}_e$ is satisfied by the second clause of the disjunction. We then say that the requirement $\mathcal{R}_e$ is *actively satisfied.* In both cases, the requirement $\mathcal{R}_e$ is satisfied.

State vs strategy outcome

It is important to distinguish between the state of a strategy and its outcome. The state of a strategy depends on each step, and is known information at that step. The outcome of the strategy is a limit behavior that is not known in finite time. We have only seen outcomes of the form "The strategy will remain in such state after a while", but we will see other outcomes in the infinite injury priority method, such as "The strategy will go through all of its states successively."

Conflicts. Complications arise when one wants to satisfy several requirements simultaneously. Indeed, the strategy of a requirement places restreints on finite segments of A and B over time, which can conflict with the needs of other strategies. We will therefore analyze to what extent the strategies can come into conflict, whether between strategies for requirements of the same type (for example $\mathcal{R}_a$ and $\mathcal{R}_b$), or between strategies for requirements of different type (for example $\mathcal{R}_a$ and $\mathcal{S}_b$).

Satisfaction of all requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$. Generally speaking, in priority arguments, strategies for requirements of the same nature are relatively easy to satisfy simultaneously. The only possible conflicts between two requirements $\mathcal{R}_e$ and $\mathcal{R}_d$ would arise if the two strategies had chosen the same integer x (in other words $x_{\mathcal{R}_e} = x_{\mathcal{R}_d}$) during their passage of State `i` in State `e`, and one of the two, say $\mathcal{R}_e$, sees its computation $\Phi_e^A(x)$ terminate and seeks to add x in B to pass in the termination State `t`, while $\mathcal{R}_d$ is still in the executing state (State `e`). To avoid these conflicts, it suffices to associate in State `i` a different integer x for each requirement $\mathcal{R}_e$. This is possible because, B being finite during the construction, there exists an infinity of integers outside B.

Satisfaction of a requirement $\mathcal{R}_e$ and $\mathcal{S}_d$. Suppose now that we want to satisfy two requirements of opposite nature. By default, the requirement $\mathcal{S}_d$ playing a symmetrical role, the strategy to satisfy it is obtained from that for $\mathcal{R}_e$ by substituting A for B and vice versa:

- State `i`: Choose an integer $x_{\mathcal{S}_d} \notin A$ and restrain $x_{\mathcal{S}_d}$.
- State `e`: Execute $\Phi_d^B(x_{\mathcal{S}_d})$. If execution stops at time t, restrain the use of $\Phi_d^{B_t}(x_{\mathcal{S}_d})$ and add $x_{\mathcal{S}_d}$ to A, then switch to State `t`.
- State `t`: The process is complete.

A new conflict can arise between the strategy for $\mathcal{R}_e$ and that for $\mathcal{S}_d$. In State `e`, the strategy for $\mathcal{R}_e$ may see execution of $\Phi_e^A(x)$ terminate with use of s oracle bits, for $s > x_{\mathcal{S}_d}$. It then restrains $A_t \restriction_s$, preventing other processes from modifying these values. If, later, the strategy for $\mathcal{S}_d$ sees its execution of $\Phi_d^B(x_{\mathcal{S}_d})$ terminate, it will not be able to add $x_{\mathcal{S}_d}$ to A because of the previous restraint. The strategy for $\mathcal{S}_d$ is *injured* and will have to return to its State `i`, choose a new integer $x_{\mathcal{S}_d}$ large enough not to be restrained, and start the procedure again. The strategy for $\mathcal{R}_e$ being in the termination state (State `t`), it will no longer act and will therefore no longer pose a new restraint which risks injuring the strategy for $\mathcal{S}_d$.

In general, one can satisfy a finite number of requirements $\mathcal{R}_e$ and $\mathcal{S}_d$ simultaneously with the same method. As soon as a strategy restrains the use of its computation when this one passes to State `t` and ends, it injures all the strategies which have not reached State `t` yet, and which then return to State `i`. The strategies thus injured will then choose new elements free from any restraint. Note that when a strategy enters State `t`, it no longer leaves it. The injuries being caused only by the a strategy arriving in State `t`, each strategy is only injured a finite number of times, and will end up entering State `t`, or will remain stuck in State `e`.

Satisfaction of all requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$. A new problem arises when we want to satisfy an infinity of requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$. Suppose the strategy for $\mathcal{R}_e$ chooses in State `i` an integer $x_{\mathcal{R}_e}$ and starts the execution of $\Phi_e^A(x_{\mathcal{R}_e})$, but does not have time to reach the end of the execution, because the strategy for a requirement $\mathcal{S}_{d_0}$ sees its own execution terminate before $\mathcal{R}_e$, and places a restraint on $x_{\mathcal{R}_e}$ to preserve the use of its computation. The strategy for $\mathcal{R}_e$ is then injured, and goes back to State `i` and chooses a new integer $x_{\mathcal{R}_e}$ and starts the execution of $\Phi_e^A(x_{\mathcal{R}_e})$ again with its new $x_{\mathcal{R}_e}$. Before reaching the end of its computation, for lack of luck, another strategy $\mathcal{S}_{d_1}$ reaches State `t`, and again injures the strategy for $\mathcal{R}_e$, and so on infinitely often. The strategy for $\mathcal{R}_e$ will then change infinitely often integers $x_{\mathcal{R}_e}$, and the requirement $\mathcal{R}_e$ will never be satisfied.

To solve this problem, we will order the strategies. Let R_e be the strategy for $\mathcal{R}_e$ and S_e the strategy for $\mathcal{S}_e$. We order the strategies as follows:

$$R_0 > S_0 > R_1 > S_1 > R_2 > S_2 > \dots$$

considering that a strategy has lower priority than the strategies to its left, but higher priority than those to its right. For example, the strategy for $\mathcal{S}_1$ has lower priority than the strategies for $\mathcal{R}_0$ and $\mathcal{S}_0$, but has higher priority than the strategies for $\mathcal{R}_2, \mathcal{S}_2$ and so on. Thus, each strategy is "below" a finite number of strategies, and "above" the rest.

Remark

In the proof of Friedberg and Muchnik's theorem, each requirement has exactly one strategy, which makes the distinction between strategy R_e and requirement $\mathcal{R}_e$ useless. The order given on the strategies thus induces an order on the requirements. However, in the following constructs, when requirements will be assigned more than one strategy, it will be essential to give total priority order to the strategies and not to the requirements.

The golden rule that we establish is then the following.

> The restraints imposed by a strategy only apply to strategies of lower priority. Thus, a strategy can only be injured by the higher priority strategies, and can injure all lower priority strategies.

Assuming that a strategy poses only a finite number of restraints before reaching its terminal state, and ceases to act after a while if it is not injured, a simple induction shows that each strategy gets injured a finite number of times. Contrary to the satisfaction of a finite number of requirements simultaneously, it is possible that a process arrives at State `t`, but is then injured by a strategy of higher priority which will have ignored the imposed restraint. Fortunately, after a while, the higher priority strategy will stop injuring the weaker one, which will eventually stabilize.

Construction. Formally, the construction is done in stages $t = 0, 1, \dots$ and each stage is divided into sub-stages $s < t$. Initially, all strategies are in

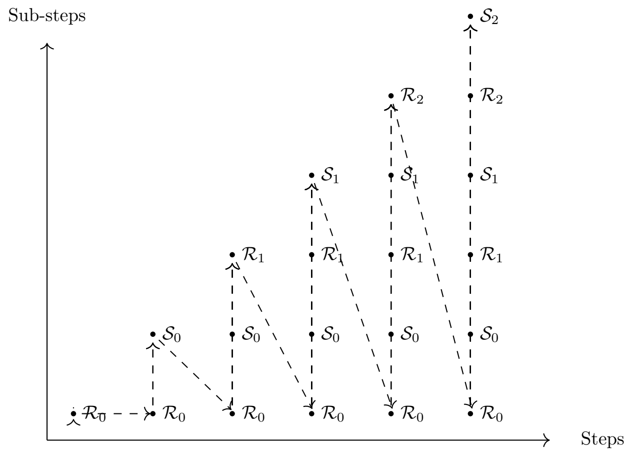
State `i`. In step $t \geqslant 0$ and sub-step $s < t$, we consider the requirement $\mathcal{R}_e$ if $s = 2e$ and $\mathcal{S}_e$ if $s = 2e + 1$. At the start of sub-step $s = 2e$, the strategy for $\mathcal{R}_e$ has 3 possible states:

(`i`) The strategy chooses an integer $x_{\mathcal{R}_e} \notin B_t$ that has not been restrained by a higher priority strategy, and restrains it. It is then found in State `e`.

(`e`) The strategy executes $\Phi_e^{A_t}(x_{\mathcal{R}_e})[t]$. If $\Phi_e^{A_t}(x_{\mathcal{R}_e})[t] \downarrow$, then it restrains all the bits used for the computation of $\Phi_s^{A_t}(x_{\mathcal{R}_e})[t]$, and injures all the lower priority strategies by returning them to State `i`. It is then found in State `t`. If $\Phi_e^{A_t}(x_{\mathcal{R}_e})[t] \uparrow$, the strategy remains in State `e`.

(`t`) The strategy does not act and remains in this state.

In the sub-step $s = 2e + 1$, we apply the same procedure for $\mathcal{S}_e$ *mutatis mutandis*, then we go to the next sub-step, until reaching t, in which case we go to step $t + 1$, and so on. This concludes the construction.



Verification: First of all, let us notice that the construction described above is indeed a computable process which enumerates two sets A and B. In particular, once an element is listed in A or B, we do not change our mind and it stays there. Note also that if a strategy for a requirement is no longer injured after a step r, then the associated requirement will be satisfied passively or actively at the end of the construction. The difficulty lies in showing that for any requirement there is indeed such an r.

Let us show by induction on $e \in \mathbb{N}$ that the strategies for the requirements $\mathcal{R}_e$ and $\mathcal{S}_e$ injure only finitely often strategies of lower priority. Suppose by induction hypothesis that there is a step t after which none of the strategies for the requirements $\mathcal{R}_d$ and $\mathcal{S}_d$ with $d < e$ injure lower priority strategies. In particular, from step t, the strategy for $\mathcal{R}_e$ will no longer be injured, and either now remains in the execution state (State `e`), or goes to after a while in the termination state (State `t`), only injuring lower priority strategies one last time. The same reasoning applies to the strategy for the requirement $\mathcal{S}_e$. So each strategy injures finitely often lower priority strategies, and each requirement will end up being satisfied, passively or actively. This concludes the proof of Theorem 3.1. ∎

Remark

The previous proof was very detailed in order to give intuitions of the priority method. We will now go more directly to the final construction for the other proofs. However, it is often useful, when trying to prove a new result using the priority method, to proceed step by step, seeking to satisfy one requirement, then several simultaneously, to end up satisfying all of them, in order to better understand their interactions.

Corollary 3.2
There exists a non-computable c.e. set A such that $A \not\geq_T \emptyset'$.

Proof. Let A and B be two c.e. sets such that neither computes the other. Then, in particular, they are both non-computable. Moreover if A could compute the halting set, it would also compute B since any c.e. set is many-one reducible to the halting problem. ∎

Before tackling more elaborate constructions based on the technique of finite injury priority methods of Friedberg and Muchnik, we see here another simple application.

Theorem 3.3
There exists a non-computable c.e. set of low degree.

Proof. We are going to construct a c.e. set A by the finite injury priority method, together with a stable total computable function $\Gamma : \mathbb{N}\times\mathbb{N} \to \{0,1\}$ satisfying the requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$:

$$\mathcal{R}_e \,:\, W_e \neq \overline{A} \qquad \mathcal{S}_e \,:\, A'(e) = \lim_t \Gamma(e,t).$$

Satisfying all the requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ ensures that the set A is not computable, while the requirements $(\mathcal{S}_e)_{e\in\mathbb{N}}$ cause A' (the halting problem relative to A) to admit a Δ^0_2 approximation, which, by the Shoenfield limit lemma (see Lemma 4-7.2), ensures that $A' \leqslant_T \emptyset'$, and therefore that A is of low degree.

Satisfaction of a requirement $\mathcal{R}_e$. Here is a strategy for satisfying a requirement $\mathcal{R}_e$, independently of other requirements. Our strategy has 3 states: initial (`i`), in execution (`e`) and terminated (`t`). The strategy takes the following steps depending on its condition.

- State `i`: Choose an integer $x_{\mathcal{R}_e} \notin A$ and restrain $x_{\mathcal{R}_e}$.
- State `e`: Execute $\Phi_e(x_{\mathcal{R}_e})$. If the execution stops at time t, add $x_{\mathcal{R}_e}$ to A, then switch to State `t`.
- State `t`: The process has completed its execution and remains in this state.

Assuming that the strategy is only injured a finite number of times, it could have two possible outcomes. Issue `p`: it will end up staying in State `e`, in which case $\Phi_e(x_{\mathcal{R}_e}) \uparrow$ and $x_{\mathcal{R}_e} \notin A$, so $W_e \neq \overline{A}$. In this case the requirement $\mathcal{R}_e$ is said to be passively satisfied. Issue `a`: The strategy will reach State `t` and stop. In this case, $\Phi_e(x_{\mathcal{R}_e}) \downarrow$, $x_{\mathcal{R}_e} \in A$, and $\mathcal{R}_e$ is said to be actively satisfied.

Satisfaction of a requirement $\mathcal{S}_e$. In step t, we will define the value of $\Gamma(x,t)$ for all $x < t$. The strategy has 2 states: in execution (`e`) and completed (`t`). Here is the procedure to follow according to each state of the strategy.

- State `e`: Execute $\Phi_e^A(e)$. While $\Phi_e^{A_t}(e)[t] \uparrow$, hold $\Gamma(e,t) = 0$. If the execution stops at time t, restrains the use of $\Phi_e^A(e)$ then go to State `t`.
- State `t`: Define $\Gamma(e,t) = 1$ for any new step t.

The two possible outcomes of the strategy are as follows. Issue `p`: it will eventually remain in execution state (State `e`), in which case $\Phi_e^A(e) \uparrow$ and $\lim_t \Gamma(e,t) = 0$. Thus, $A'(e) = 0 = \lim_t \Gamma(e,t)$. In this case the requirement $\mathcal{S}_e$ is said to be passively satisfied. Issue `a`: The strategy will reach State `t` and stop. In this case, $\Phi_e^A(e) \downarrow$ and $\lim_t \Gamma(e,t) = 1$. So $A'(e) = 1 = \lim_t \Gamma(e,t)$ and $\mathcal{S}_e$ is said to be actively satisfied.

Construction. The construction is globally the same as that of Theorem 3.1. The strategies are ordered by decreasing priority $R_0, S_0, R_1, S_1, \ldots$ The construction is divided into stages $t = 0, 1, \ldots$ and each stage is itself divided into sub-stages $s < t$. Initially, all strategies are in State `i`. In step $t \geqslant 0$ and sub-step $s < t$, we consider the requirement $\mathcal{R}_e$ if $s = 2e$ and $\mathcal{S}_e$ if $s = 2e+1$. In sub-step $s = 2e$, we execute the strategy for $\mathcal{R}_e$ as described above depending on its state. When it reaches State `t`, all lower priority strategies are injured and either return to State `i` in the case of strategies for requirements of type $\mathcal{R}$, or in State `e` in the case of strategies for requirements of type $\mathcal{S}$. In the same way, in the sub-step $s = 2e+1$, we execute the strategy $\mathcal{S}_e$ as described above, injuring all the lower priority strategies if we reach the termination state `t`.

Verification: The set A produced is indeed c.e. because the process is computable, and does not remove any element from A once added. We easily prove by induction on $e \in \mathbb{N}$ that the strategies for requirements $\mathcal{R}_e$ and $\mathcal{S}_e$ injure only finitely often lower priority strategies. Thus, each strategy is only finitely often injured, and will end up having limit behavior. It also follows that $\lim_t \Gamma(e,t)$ exists, and by construction, is equal to $A'(e)$. This concludes the proof of Theorem 3.3. ■

4. Σ^0_2 priority method

In the previous constructions using the priority method, finite injuries are structurally ensured, that is, in the very structure of the construction, the strategies pose only a finite number of restraints when they are not injured, and are ensured by construction that they will only be injured a finite number of times. The number of injuries can even be computably bounded: In Friedberg and Muchnik's construction, the e-th process is injured at most $2^e - 1$ times.

We will now see an elaboration of the previous method, which could structurally result in an infinite number of injuries of a strategy, but construction assumptions will ensure that this never happens. Technically, this is therefore a finite injury priority method, as each strategy will only be injured a finite number of times. However, this elaboration can be seen as a degenerate version of the infinite injury priority method, shown in the next section.

Theorem 4.1 (Sacks)
For any non-computable c.e. set B, there exists a c.e. non-computable set A which does not compute B.

Proof. We are going to build a c.e. set A satisfying the following requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$:

$$\mathcal{R}_e : W_e \neq \overline{A} \qquad \mathcal{S}_e : \Phi_e^A = B \Rightarrow B \text{ is computable.}$$

Satisfying all requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ ensures that the set A is not computable, while requirements $(\mathcal{S}_e)_{e\in\mathbb{N}}$ ensure that $B \not\leq_T A$. The requirement $\mathcal{S}_e$ could also have been noted $W_e^A \neq \overline{B}$, but its current formulation better represents the form of the argument used to satisfy it.

Satisfaction of a requirement $\mathcal{R}_e$. The satisfaction of a requirement $\mathcal{R}_e$ is exactly the same as for Theorem 3.3. We recall the actions of the strategy according to its three states:

- State `i`: Choose an integer $x_{\mathcal{R}_e} \notin A$ and restrain $x_{\mathcal{R}_e}$.
- State `e`: Execute $\Phi_e(x_{\mathcal{R}_e})$. If the execution stops at time t, add $x_{\mathcal{R}_e}$ to A, then switch to State `t`.
- State `t`: The process is complete.

The two outcomes of the strategy are still `p` (passive satisfaction) and `a` (active satisfaction).

Satisfaction of a requirement $\mathcal{S}_e$. The strategy to satisfy $\mathcal{S}_e$ is more complex and less intuitive. It consists in trying to make longer and longer initial segments of Φ^A and B coincide, in a computable way, by setting each time larger and larger restraints on A to preserve the use of Φ^A. At first glance, this strategy will therefore cause infinite injury to lower priority strategies. Fortunately, the set B not being computable, the procedure will stop finding new initial segments coinciding, and will thus satisfy the requirement $\mathcal{S}_e$. More precisely, the strategy has an initial state (State `i`), and an infinity of states $(\mathtt{w}_n)_{n\in\mathbb{N}}$. During the execution of the process, we will define a computable function $\Delta : \mathbb{N} \to \{0,1\}$ supposed to coincide with the characteristic function of B. Let $(B_t)_{t\in\mathbb{N}}$ be a c.e. approximation of B.

- State `i`: Define Δ as the empty domain function, and change to State $\mathtt{w}_0$.
- State $\mathtt{w}_n$: Wait for a step $t \geqslant n$ where $\Phi_e^{A_t}[t] \upharpoonright_{n+1} = B_t \upharpoonright_{n+1}$. If this happens, restrain the use of $\Phi_e^{A_t}[t] \upharpoonright_{n+1}$, define $\Delta(n) = B_t(n)$, and switch to State $\mathtt{w}_{n+1}$.

Here, $\Phi_e^{A_t}[t] \upharpoonright_{n+1} = B_t \upharpoonright_{n+1}$ means that for all $x \leqslant n$, $\Phi_e^{A_t}(x)[t] \downarrow \in \{0,1\}$, and $\Phi_e^{A_t}(x)[t] = 1$ iff $x \in B_t$ for $x \leqslant n$. The strategy does not return to State `i` unless injured. At this time, the Δ function is also reset. However, assuming that the strategy is injured a finite number of times, the function will only be reset finitely often, and therefore will be defined in a computable way.

The strategy has an infinite number of outcomes: for all n, the outcome $\mathtt{p}_n$ consists of remaining stuck in State $\mathtt{w}_n$. If the strategy never goes out of this state, then $\Phi_e^{A_t}[t] \upharpoonright_{n+1} \neq B_t \upharpoonright_{n+1}$ for all t, hence $\Phi_e^A \neq B$, and the requirement $\mathcal{S}_e$ is satisfied because the premise of the implication is false. The strategy has a last possible outcome, of an infinite nature, which consists in going through all the states $\mathtt{w}_n$. Let us name this issue ∞. By using the hypothesis according to which B is not computable, we will now show that this outcome cannot happen.

Lemma 4.2. If the outcome ∞ occurs, then B is computable. ⋆

PROOF. Let e be the smallest integer such that the strategy for $\mathcal{S}_e$ goes through all states $(\mathtt{w}_n)_{n\in\mathbb{N}}$. By induction, all higher priority strategies are finitely injured, and will reach a limit state where they will no longer injure the strategy for $\mathcal{S}_e$. From this moment, the strategy for $\mathcal{S}_e$ will go through all the states $(\mathtt{w}_n)_{n\in\mathbb{N}}$ successively. The function Δ defined by the process is then total computable. Let us show that Δ is the characteristic function of B. Let us assume absurdly that $\Delta(n) \neq B(n)$ for an $n \in \mathbb{N}$. By definition of Δ, $\Delta(n) = B_t(n) = \Phi_e^{A_t}(n)$ for a $t \in \mathbb{N}$. Since B is c.e., This difference comes from an element that appears in B, because no element can come out of it. Thus, $\Delta(n) = B_t(n) = \Phi_e^{A_t}(n) = 0$ and $n \in B$. Let m be large enough such that $n \in B_m$. Then, at state $\mathtt{w}_m$, $\Phi_e^{A_t}(n)[t] = 0 \neq B_t(n)$ for all $t \geqslant m$, so $\Phi_e^{A_t}[t] \upharpoonright_{n+1} \neq B_t \upharpoonright_{n+1}$ for all $t \geqslant m$, and the strategy will never reach State $\mathtt{w}_{m+1}$. Contradiction. The function Δ is therefore the characteristic function of B, which proves that B is computable. ■

Construction. The overall construction is that of a standard finite injury priority method. The strategies are ordered by decreasing priority $R_0, S_0, R_1, S_1, \ldots$. The construction is divided into stages $t = 0, 1, \ldots$ and each stage is itself divided into sub-stages $s < t$. Initially, all strategies are in State $\mathtt{i}$. In step $t \geqslant 0$ and sub-step $s < t$, we consider the requirement $\mathcal{R}_e$ if $s = 2e$ and $\mathcal{S}_e$ if $s = 2e + 1$. In the sub-step $s = 2e$, the strategy for $\mathcal{R}_e$ is executed as described above depending on its state. When it reaches State $\mathtt{e}$, all lower priority strategies are injured and revert to State $\mathtt{i}$. Likewise, in the sub-step $s = 2e + 1$, the strategy for $\mathcal{S}_e$ is executed according to the steps described above. Each time it transitions to a next state $\mathtt{w}_{n+1}$, all lower priority strategies are injured and revert to State $\mathtt{i}$.

Verification is left to the reader. This concludes the proof of Theorem 4.1.■

Sacks preservation strategy

The strategy to satisfy $\mathcal{S}_e$ consists in not trying to actively differentiate two sets, but on the contrary, to preserve increasingly long common initial segments in a computable process, then to use the hypothesis of incomputability of one of the sets to deduce that this process should fail. This strategy is frequently found in this type of construction. It is sometimes referred to as *Sacks preservation strategy*, in honor of its author.

5. Π^0_2 priority method (infinite injury)

We are now going to approach a new elaboration of the priority method, known as the infinite injury method. As the name suggests, some strategies will act infinitely often by placing larger and larger restraints, causing infinite injury to lower priority strategies. We will therefore start to have conditional strategies, adopting different behaviors depending on the outcomes of the higher priority strategies, thus taking full advantage of the formalism of the "strategy tree" that we will see soon.

Our illustration of the infinite injury priority method concerns the existence of minimal pairs of c.e. degrees. It improves the Friedberg-Muchnik theorem by combining it with Sacks' preservation strategy.

Definition 5.1. Two non-computable degrees **a** and **b** form a *minimal pair* if their lower bound is **0**, in other words if for any set A such that $A \leqslant_T \mathbf{a}$ and $A \leqslant_T \mathbf{b}$, then A is computable. ♢

The existence of minimal pairs of c.e. degrees has been independently proven by Lachlan [133] and Yates [243].

Theorem 5.2 (Lachlan 1966, Yates 1966)
There is a minimum pair of c.e. degrees

Proof. We are going to construct two c.e. sets A and B satisfying the following requirements $(\mathcal{R}_e, \mathcal{S}_e, \mathcal{N}_e)_{e\in\mathbb{N}}$:

$$\mathcal{R}_e : W_e \neq \overline{A} \qquad \mathcal{S}_e : W_e \neq \overline{B} \qquad \mathcal{N}_e : \Phi_e^A = \Phi_e^B \Rightarrow \Phi_e^A \text{ is computable.}$$

The requirements $(\mathcal{R}_e)_{e\in\mathbb{N}}$ and $(\mathcal{S}_e)_{e\in\mathbb{N}}$ ensure that A and B are not computable. The $(\mathcal{N}_e)_{e\in\mathbb{N}}$ requirements force the lower bound of the degrees of A and B to be **0**.

Posner's trick

At first glance, to impose that the lower bound $\deg_T A$ and $\deg_T B$ is **0**, one would expect to have to satisfy requirements of the form $(\mathcal{N}_{i,j})_{i,j\in\mathbb{N}}$ with

$$\mathcal{N}_{i,j} : \Phi_i^A = \Phi_j^B \Rightarrow \Phi_i^A \text{ is computable.}$$

However, if $\Phi_i^A = \Phi_j^B$, then it is possible to create a new functional Φ_e which will hardcode an integer n such that $A(n) \neq B(n)$, and execute Φ_i

or Φ_j in function of the value of its oracle at position n. Thus, $\Phi_e^A = \Phi_i^A$ and $\Phi_e^B = \Phi_j^B$. This trick, due to Posner, makes it possible to simplify the notations by using a single index.

Satisfaction of a requirement $\mathcal{R}_e$ or $\mathcal{S}_e$. The satisfaction of a requirement $\mathcal{R}_e$ or $\mathcal{S}_e$ is exactly the same as for Theorem 3.3. We recall, in the case of $\mathcal{R}_e$, the actions of the strategy according to its three states:

- State `i`: Choose an integer $x_{\mathcal{R}_e} \notin A$ and restrain $x_{\mathcal{R}_e}$.
- State `e`: Run $\Phi_e(x_{\mathcal{R}_e})$. If the execution stops at time t, add $x_{\mathcal{R}_e}$ to A, then enter State `t`.
- State `t`: The process is complete.

So far, we have considered that this strategy has two outcomes, depending on the state in which it stabilizes. These two outcomes are of the same finite nature, in that they pose only a finite number of restraints when they are injured finitely often. We will therefore consider them as a single finitary outcome `f`.

Satisfaction of a requirement $\mathcal{N}_e$ The satisfaction of a requirement $\mathcal{N}_e$ will follow Sacks' preservation strategy to preserve increasingly long common initial segments to Φ_e^A and Φ_e^B. However, unlike Theorem 4.1, the process will not necessarily fail after a finite number of steps, because nothing in the assumptions prevents this equality. We are therefore in a case where the infinite outcome will be able to occur, with increasingly long restraints, resulting in an infinite injury. As in the case of Theorem 4.1, the strategy has an initial state `i`, and an infinity of states $(\mathtt{w}_n)_{n\in\mathbb{N}}$. In what follows, we will call *use* of $\Phi_e^{A_t}[t]\restriction_{n+1}$ the maximum of uses of $\{\Phi_e^{A_t}(x)[t] : x \leqslant n\}$. During the execution of the strategy, we will define a computable function $\Delta : \mathbb{N} \to \{0,1\}$ such that if Φ_e^A and Φ_e^B are total and equal, then they are both equal to Δ.

- State `i`: Define Δ as the empty domain function, and go to State $\mathtt{w}_0$.
- State $\mathtt{w}_n$: Wait for a step $t \geqslant n$ where $\Phi_e^{A_t}[t]\restriction_{n+1} = \Phi_e^{B_t}[t]\restriction_{n+1}$. If this happens, release its previous restraint, and place a restraint on the use of $\Phi_e^{A_t}[t]\restriction_{n+1}$ if n is even, and on the use of $\Phi_e^{B_t}[t]\restriction_{n+1}$ if n is odd. Next, define $\Delta(n) = \Phi_e^{A_t}(n)[t]$, and go to State $\mathtt{w}_{n+1}$.

The strategy has two possible outcomes. Issue `f` (finitary): it gets stuck in state $\mathtt{w}_n$ for a given n. In this case, either Φ_e^A or Φ_e^B is partial, or $\Phi_e^A \neq \Phi_e^B$. Issue ∞ (infinitary): the strategy goes through all states $(\mathtt{w}_n)_{n\in\mathbb{N}}$. In this case, the two sets coincide, and infinitely often, the restraint changes sides.

We still have to prove that $\Delta = \Phi_e^A = \Phi_e^B$ to deduce that this set is computable. The underlying idea of the proof is very simple, but it is a bit cumbersome to formalize. We will therefore illustrate it with a figure (see Figure 5.3) before formally proving the result through Lemma 5.4. Whatever the outcome, the requirement $\mathcal{N}_e$ is therefore satisfied.

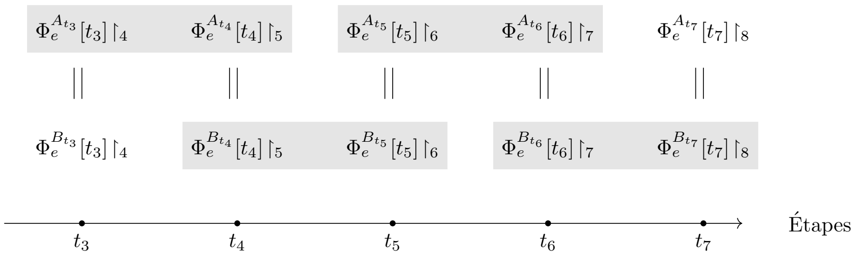

$\Phi_e^{A_{t_3}}[t_3]\restriction_4$
$\Phi_e^{A_{t_4}}[t_4]\restriction_5$
$\Phi_e^{A_{t_5}}[t_5]\restriction_6$
$\Phi_e^{A_{t_6}}[t_6]\restriction_7$
$\Phi_e^{A_{t_7}}[t_7]\restriction_8$
$\Phi_e^{B_{t_3}}[t_3]\restriction_4$
$\Phi_e^{B_{t_4}}[t_4]\restriction_5$
$\Phi_e^{B_{t_5}}[t_5]\restriction_6$
$\Phi_e^{B_{t_6}}[t_6]\restriction_7$
$\Phi_e^{B_{t_7}}[t_7]\restriction_8$
Étapes
t_3
t_4
t_5
t_6
t_7

Figure 5.3: Let t_n be the step at which we reach State $\mathtt{w}_{n+1}$. The grey rectangle between step t_3 and step t_4 means that the use of $\Phi_e^{A_{t_3}}[t_3]\restriction_4$ is restraint, hence is preserved up to step t_4. Thus, $\Phi_e^{A_{t_3}}[t_3]\restriction_4 = \Phi_e^{A_{t_4}}[t_4]\restriction_4$. At each step t_n, the initial segments of length $n+1$ of both functionals coincide. Thus, $\Phi_e^{A_{t_4}}[t_4]\restriction_5 = \Phi_e^{B_{t_4}}[t_4]\restriction_5$. We therefore have $\Phi_e^{B_{t_3}}[t_3]\restriction_4 = \Phi_e^{A_{t_3}}[t_3]\restriction_4 = \Phi_e^{A_{t_4}}[t_4]\restriction_4 = \Phi_e^{B_{t_4}}[t_4]\restriction_4$. Even if the use of $\Phi_e^{B_{t_3}}[t_3]\restriction_4$ can differ from that of $\Phi_e^{B_{t_4}}[t_4]\restriction_4$, since no restraint is posed on B between steps t_3 and t_4, the first 4 output values of the functional are preserver.

Remark

The choice of the restrained side (A if the strategy goes from a state $\mathtt{w}_n$ to $\mathtt{w}_{n+1}$ with n even, and B if n is odd) does not intervene in the proof of the validity of a strategy $\mathcal{N}_e$ independently of the others. We could just as well always have kept the same side, or even restrained both sides, which would have considerably simplified the proof of validity. However, this alternation of sides becomes necessary when one seeks to satisfy a requirement of type $\mathcal{R}$ or $\mathcal{S}$ under a strategy for $\mathcal{N}_e$, as we will see hereafter.

Lemma 5.4. If the outcome ∞ happens, then $\Delta = \Phi_e^A = \Phi_e^B$. ⋆

Proof. Let $P(n,s)$ be the proposition "Either (1) $\Delta\restriction_{n+1} = \Phi_e^{A_s}[s]\restriction_{n+1}$ with a restraint on its use, or (2) $\Delta\restriction_{n+1} = \Phi_e^{B_s}[s]\restriction_{n+1}$ with a restraint on its use." For all n, let t_n be the step at which the strategy changes to state $\mathtt{w}_{n+1}$. We will show by induction on n and s that for all $n \geqslant 0$ and $s \geqslant t_n$, the proposition $P(n,s)$ is true. By convention, $t_{-1} = 0$ and for all $s \geqslant t_{-1}$, $P(-1,s)$ is true.

Let $n \geqslant 0$. Let us show that if for all $s \geqslant t_{n-1}$, $P(n-1,s)$ is true, then $P(n,t_n)$ is true. In step t_n, $\Delta(n) = \Phi_e^{A_{t_n}}(n)[t_n] = \Phi_e^{B_{t_n}}(n)[t_n]$, and $\Phi_e^{A_{t_n}}[t_n] \upharpoonright_{n+1} = \Phi_e^{B_{t_n}}[t_n] \upharpoonright_{n+1}$. As $t_n \geqslant t_{n-1}$, by induction hypothesis, $P(n-1,t_n)$ is true, so either $\Delta \upharpoonright_n = \Phi_e^{A_{t_n}}[t_n] \upharpoonright_n$ or $\Delta \upharpoonright_n = \Phi_e^{B_{t_n}}[t_n] \upharpoonright_n$. So $\Delta \upharpoonright_{n+1} = \Phi_e^{A_{t_n}}[t_n] \upharpoonright_{n+1} = \Phi_e^{B_{t_n}}[t_n] \upharpoonright_{n+1}$. At this step, the strategy restrains $\Phi_e^{A_{t_n}}[t_n] \upharpoonright_{n+1}$ or $\Phi_e^{B_{t_n}}[t_n] \upharpoonright_{n+1}$, so $P(n,t_n)$ is true.

Let $s > t_n$. Let us show that if $P(n,s-1)$ is true, then $P(n,s)$ is true. If the strategy does not change state at step s, then it keeps its restraint, and by the use property, $P(n,s)$ remains true. If the strategy changes to a $\mathtt{w}_{p+1}$ state, then by definition, $\Phi_e^{A_s}[s] \upharpoonright_{p+1} = \Phi_e^{B_s}[s] \upharpoonright_{p+1}$, or $p \geqslant n$, so $\Phi_e^{A_s}[s] \upharpoonright_{n+1} = \Phi_e^{B_s}[s] \upharpoonright_{n+1}$. By $P(n,s-1)$, either $\Delta \upharpoonright_{n+1} = \Phi_e^{A_{s-1}}[s-1] \upharpoonright_{n+1}$ with a restraint on its use, or (2) $\Delta \upharpoonright_{n+1} = \Phi_e^{B_{s-1}}(n)[s-1] \upharpoonright_{n+1}$ with a restraint on its use. By the restraint to step $s-1$ and the use property, either $\Delta \upharpoonright_{n+1} = \Phi_e^{A_s}[s] \upharpoonright_{n+1}$, or $\Delta \upharpoonright_{n+1} = \Phi_e^{B_s}[s] \upharpoonright_{n+1}$. It follows that $\Delta \upharpoonright_{n+1} = \Phi_e^{A_s}[s] \upharpoonright_{n+1} = \Phi_e^{B_s}[s] \upharpoonright_{n+1}$. The strategy restrains the use of $\Phi_e^{A_s}[s] \upharpoonright_{p+1}$ or $\Phi_e^{B_s}[s] \upharpoonright_{p+1}$, so $P(n,s)$ is true. This ends the proof of the lemma. ■

Note that unlike Theorem 4.1, the infinite outcome will really occur with the strategy for $\mathcal{N}_e$. It therefore does not combine that well with the strategies for $\mathcal{R}_d$ and $\mathcal{S}_d$. Indeed, when this outcome occurs, the strategy for $\mathcal{N}_e$ will impose increasingly long restraints, causing infinite injury. We will therefore have to adapt the construction to allow the other requirements to be satisfied.

Execution step

It is not necessary to execute the strategies for $\mathcal{R}_e$, $\mathcal{S}_e$ and $\mathcal{N}_e$ at each step to satisfy their respective constraints. It suffices to perform them each for an infinite number of steps, while maintaining their restraints during the intermediate steps.

Let us take the example of the strategy for $\mathcal{R}_e$. If it is only executed at times $t_0 < t_1 < \dots$, it may be that it "misses" the first step t between t_0 and t_1 where $\Phi_e(x_{\mathcal{R}_e})[t]$ stops, which prevents it from entering State $\mathtt{t}$ at this step. However, in step t_1, $\Phi_e(x_{\mathcal{R}_e})[t_1]$ will also stop, and the transition to State $\mathtt{t}$ will still take place. The limit behavior of the strategy for $\mathcal{R}_e$ therefore does not depend on the choice of steps.

The case of the strategy for $\mathcal{N}_e$ is a bit more subtle. It may be that the default strategy for $\mathcal{N}_e$ has an infinite outcome, but that when it is executed only at times $t_0 < t_1 < \dots$, we have $\Phi_e^{A_t}[t_i] \upharpoonright_{n+1} \neq \Phi_e^{B_t}[t_i] \upharpoonright_{n+1}$, so that the strategy does not will never change to state $\mathtt{w}_{n+1}$, which is the finite outcome. Fortunately, even in this case $\mathcal{N}_e$ will be satisfied, as long as the enumeration of steps $(t_i)_{i\in\mathbb{N}}$ is computable so that the function Δ is also computable.

Satisfaction of a requirement $\mathcal{R}_d$ or $\mathcal{S}_d$ under $\mathcal{N}_e$. Suppose we want to satisfy a requirement $\mathcal{R}_d$ under the strategy for $\mathcal{N}_e$, i.e., with the strategy for $\mathcal{N}_e$ of higher priority than that for $\mathcal{R}_d$. Several solutions arise, depending on the outcome of the strategy for $\mathcal{N}_e$:

- Issue `f` (finitary). In this case, it suffices to use the standard strategy for $\mathcal{R}_e$ presented above. Indeed, the strategy for $\mathcal{N}_e$ will set a finite number of restraints, so that the strategy for $\mathcal{R}_e$ will be injured and reset a finite number of times before being satisfied. This strategy does not work if the outcome of the strategy for $\mathcal{N}_e$ is infinite (Issue ∞). Indeed, in this case, the strategy for $\mathcal{R}_e$ will be injured infinitely often, and might never satisfy $\mathcal{R}_e$.
- Issue ∞ (infinitary). Note that in this case, the restraint posed by the strategy for $\mathcal{N}_e$ will infinitely often alternate sideways, and will therefore free the other side, allowing the strategy for $\mathcal{R}_d$ to be satisfied. We will therefore only execute the strategy for $\mathcal{R}_d$ at the stages where the restraint of the strategy for $\mathcal{N}_e$ is removed from the A side. The outcome of the strategy for $\mathcal{N}_e$ being infinite, the strategy for $\mathcal{R}_d$ will be executed for an infinite number of steps, and as explained above, an infinite subset of steps is sufficient to satisfy $\mathcal{R}_d$. This strategy for $\mathcal{R}_d$ does not work, however, if the outcome of the strategy for $\mathcal{N}_e$ is finite, because it may never remove its restraint and the strategy for $\mathcal{R}_d$ therefore waits forever.

We therefore have two different strategies to satisfy $\mathcal{R}_d$ under $\mathcal{N}_e$, depending on the outcome of the strategy for $\mathcal{N}_e$. This case analysis poses a difficulty: to produce a c.e. set, The construction must be a computable process, but the outcome of $\mathcal{N}_e$ cannot be decided in a finite time. We therefore cannot know which strategy to choose for $\mathcal{R}_d$. The solution is to run the two strategies for $\mathcal{N}_e$ in parallel, each making a guess about the outcome. The one making the correct assumption will then be able to satisfy $\mathcal{R}_d$ under $\mathcal{N}_e$. We will therefore end up with a tree of strategies, induced by the successive case analysis on the outcomes of strategies of type $\mathcal{N}$. We will detail this tree structure below.

Satisfaction of a requirement $\mathcal{N}_d$ under $\mathcal{N}_e$. Similar requirements are often easy to satisfy simultaneously because their strategies generally do not conflict. In the case of the satisfaction of a requirement $\mathcal{N}_d$ under $\mathcal{N}_e$, the difficulty is not to satisfy these two requirements simultaneously, but then to leave room for a requirement of the type $\mathcal{R}$ or $\mathcal{S}$ to be satisfied under $\mathcal{N}_d$. Indeed, in the worst case, the strategies for $\mathcal{N}_e$ and $\mathcal{N}_d$ will both be infinitary, and each time the strategy for $\mathcal{N}_e$ releases its restraints on the A side, the strategy for $\mathcal{N}_d$ will set its own, so that the A side will have increasingly larger restraints at all times, not allowing strategies of type $\mathcal{R}$ to be satisfied below. The solution is to "synchronize" the strategies for $\mathcal{N}_d$ and $\mathcal{N}_e$. More precisely, the strategy for $\mathcal{N}_d$ will be defined by case analysis depending on the outcome of $\mathcal{N}_e$:

- Issue $\mathtt{f}$ (finitary). In this case, the strategy for $\mathcal{N}_d$ is the standard strategy presented above. Indeed, the restraints of the strategy for $\mathcal{N}_e$ will be finitary, and the other requirements will have the possibility of being satisfied under $\mathcal{N}_d$ by being injured finitely often by the strategy of $\mathcal{N}_e$.

- Issue ∞ (infinitary). Let's modify the strategy for $\mathcal{N}_d$, as follows: This strategy will only be executed during stages where the strategy for $\mathcal{N}_e$ changes the side of its restraints, in other words changes from state $\mathtt{w}_n$ to $\mathtt{w}_{n+1}$. This makes sure that when the strategy for $\mathcal{N}_d$ changes its restraints side, at the same step, the strategy for $\mathcal{N}_e$ will change its side restraints. Finally, we must ensure that these changes go on the same side. For this, if the strategy for $\mathcal{N}_d$ is in a state $\mathtt{w}_n$ with n an even integer, in other words if its restraints are on the B side, the strategy for $\mathcal{N}_d$ will not be executed only during the steps where the strategy for $\mathcal{N}_e$ changes from a state $\mathtt{w}_m$ to $\mathtt{w}_{m+1}$ with m even, so that both strategies put their restraints on the side A simultaneously. Likewise, if the strategy for $\mathcal{N}_d$ is in a state $\mathtt{w}_n$ with n an odd integer, it will be executed during the steps where the strategy for $\mathcal{N}_e$ changes from a state $\mathtt{w}_m$ to $\mathtt{w}_{m+1}$ with m odd.

As in the case of the satisfaction of a requirement $\mathcal{R}_d$ under a requirement $\mathcal{N}_e$, we therefore have different strategies for $\mathcal{N}_d$ for each outcome of the strategy for $\mathcal{N}_e$. We are therefore going to run two strategies in parallel, each assuming that its hypothesis is correct. We will now describe the strategy tree.

Strategy tree. The requirements are listed as follows.

$$\mathcal{N}_0, \mathcal{R}_0, \mathcal{S}_0, \mathcal{N}_1, \mathcal{R}_1, \mathcal{S}_1, \ldots$$

In the previous constructions, each requirement was assigned a unique strategy, and the priority order of the strategies followed the enumeration of requirements. We now have a tree structure of strategies based on the outcomes of the previous strategies, as follows: the requirement $\mathcal{N}_0$ has a single strategy N_ϵ (where ϵ is the empty string). The requirement $\mathcal{R}_0$

has two strategies R_∞ and $R_{\mathtt{f}}$, depending on the outcomes ∞ and $\mathtt{f}$ of the strategy $\mathcal{N}_0$. The requirement $\mathcal{S}_0$ also has two strategies $S_{\infty\mathtt{f}}$ and $S_{\mathtt{ff}}$, depending on the outcomes of the higher strategies. For example, $S_{\infty\mathtt{f}}$ must satisfy the requirement $\mathcal{S}_0$ under the assumption that the outcome of N_ϵ is ∞ and the outcome of R_∞ is $\mathtt{f}$.

In general, we can define a tree $\mathcal{T} \subseteq \{\mathtt{f}, \infty\}^{<\mathbb{N}}$ of strings in the alphabet $\{\mathtt{f}, \infty\}$, induced by the prefix relation, such that for all $\sigma \in \mathcal{T}$ and $i < |\sigma|$ such that $i \not\equiv 0 \mod 3$, $\sigma(i) = \mathtt{f}$. To each string $\sigma \in \mathcal{T}$, if $|\sigma| = 3e$ we associate a strategy N_σ for $\mathcal{N}_e$, if $|\sigma| = 3e + 1$ we associate a strategy R_σ for $\mathcal{R}_e$ and if $|\sigma| = 3e+2$, we associate a strategy S_σ for $\mathcal{S}_e$ (see Figure 5.5). To simplify the notations, we will denote by C_σ the strategy whose index is σ.

$$\begin{tikzcd}
 & & & N_\epsilon \arrow[dll, "\infty"'] \arrow[drr, "\mathtt{f}"] & & & & \mathcal{N}_0 \\
 & R_\infty \arrow[d] & & & & R_{\mathtt{f}} \arrow[d] & & \mathcal{R}_0 \\
 & S_{\infty\mathtt{f}} \arrow[d] & & & & S_{\mathtt{ff}} \arrow[d] & & \mathcal{S}_0 \\
 & N_{\infty\mathtt{ff}} \arrow[dl, "\infty"'] \arrow[dr, "\mathtt{f}"] & & & & N_{\mathtt{fff}} \arrow[dl, "\infty"'] \arrow[dr, "\mathtt{f}"] & & \mathcal{N}_1 \\
R_{\infty\mathtt{ff}\infty} \arrow[d] & & R_{\infty\mathtt{fff}} \arrow[d] & & R_{\mathtt{fff}\infty} \arrow[d] & & R_{\mathtt{ffff}} \arrow[d] & \mathcal{R}_1 \\
S_{\infty\mathtt{ff}\infty\mathtt{f}} \arrow[d] & & S_{\infty\mathtt{ffff}} \arrow[d] & & S_{\mathtt{fff}\infty\mathtt{f}} \arrow[d] & & S_{\mathtt{fffff}} \arrow[d] & \mathcal{S}_1 \\
{} \arrow[uurr, dotted] & & {} \arrow[uuuuurrr, dotted] & & {} \arrow[uurr, dotted] & & {} &
\end{tikzcd}$$

Figure 5.5: Strategy tree for the construction of a minimal pair. Full arrows indicate the tree structure, but also a partial order of priority between strategies. Dotted arrows enable to linearize the partial order to obtain a total order between strategies. For example, all the strategies on the left sub-tree of the strategy $N_{\infty\mathtt{ff}}$ are weaker than N_ϵ, R_∞, $S_{\infty\mathtt{f}}$ et $N_{\infty\mathtt{ff}}$, but are stronger than all the other strategies.

Construction. At the start of the construction, each strategy is in the initial state (State $\mathtt{i}$). At each step t, we will define a *current node*, which is a string $\sigma_t \in \mathcal{T}$ of length t, representing an approximation of the outcomes at the end of step t. More precisely, $\sigma_t(i)$ represents the approximation at step t of the outcome of the strategy $C_{\sigma_t\restriction_i}$. The current node at step t is defined by induction on the substeps $s < t$ as follows: at the start of the substep i, we assume $\sigma_t \restriction_i$ defined. We then execute the strategy $C_{\sigma_t\restriction_i}$ at time t as described above. Recall that we may not execute the strategy $C_{\sigma_t\restriction_i}$ for reasons of synchronization with the strategies for requirements of type $\mathcal{N}$. In this case, the strategy remains in its state and keeps its constraints unchanged. At the end of sub-step t, we define $\sigma_t(i) = \infty$ if $i \equiv 0 \mod 3$ and the strategy $N_{\sigma_t \restriction_i}$ has changed state at step t. In all the other cases, $\sigma_t(i) = \mathtt{f}$, because the strategies of type $\mathcal{R}$ and $\mathcal{S}$ only have the finite possible outcome, and if $i \equiv 0 \mod 3$ and $N_{\sigma_t \restriction_i}$ does not change state, we assume that its outcome is finite.

True path. We can now define the *true path* of $\mathcal{T}$, which is the path along which the outcomes are true. More precisely, the true path of $\mathcal{T}$ is the infinite sequence $P \in \Lambda^{\mathbb{N}}$ (with $\Lambda = \{\infty, f\}$ where ∞ is less than f) defined inductively on i by

$$P(i) = \liminf\{o \in \Lambda : \exists^\infty t \ (P\restriction_i)^\frown o \preceq \sigma_t\}.$$

Intuitively, the strategies along the real path are going to be the ones that make the correct assumptions about the outcomes of the previous strategies. They will be finitely injured by a higher priority strategies, and will succeed in fulfilling their requirement. We are now going to prioritize the strategies so that the strategies along the true path are finitely injured.

Priority order. Let $\Lambda = \{\infty, \mathtt{f}\}$ be the set of possible outcomes. Let us provide this set with an order of priority $(\Lambda, <_p)$ by considering that $\infty <_p \mathtt{f}$, which means that the outcome ∞ has priority over $\mathtt{f}$. This order will induce a total order $(\mathcal{T}, <_p)$ (and therefore a total order of priority on the strategies) as follows: $\sigma <_p \tau$ if $\sigma \preceq \tau$, or if i is the first position where the two strings differ, and $\sigma(i) <_p \tau(i)$. A visual example of this order is given in Figure 5.5. Note that, unlike finite injury priority methods, the strategies are generally below an infinite number of higher priority strategies. For example, the strategy $R_{\infty\mathtt{fff}}$ is below N_ϵ, R_∞, $S_{\infty\mathtt{f}}$, $N_{\infty\mathtt{ff}}$, $R_{\infty\mathtt{ff}\infty}$, $S_{\infty\mathtt{ff}\infty\mathtt{f}}, \ldots$

Remark

One would be tempted to define a priority order such that each strategy only has a finite number of higher priority strategies, for example by defining the priorities by traversing the nodes of the tree. However, there is a problem:
Suppose that the strategy N_ϵ has the outcome ∞. The strategy $R_\mathtt{f}$ making the wrong assumption according to which the outcome of N_ϵ has a finite outcome, it will be injured and reinitialized at each change of state of N_ϵ. It will therefore potentially pose an infinite number of restraints, and will therefore prevent all lower priority strategies from functioning normally. It is therefore essential that all strategies along the true path of the tree take precedence over $R_\mathtt{f}$, in order to ignore its restraints. The strategy for $R_\mathtt{f}$ must therefore be under an infinity of higher priority strategies.

Verification. Note first that by definition, the strategies along the real path P (strategies C_σ for $\sigma \prec P$) are executed at an infinite number of steps. Although a strategy is generally below an infinite number of strategies, we will show that the strategies along the real path are injured finitely often, which is the necessary condition to satisfy them. The following lemma is a property that we generally expect from an argument with Π^0_2 priority:

Lemma 5.6. Let P be the true path, and $\alpha \prec P$. Then, $\alpha \leqslant_p \sigma_t$ for a co-finite number of steps t. ⋆

Proof. By induction on the length n of α. If $n = 0$, then $\alpha = \epsilon$, and by definition, $\epsilon \leqslant_p \sigma_t$ for all t. Let $n > 0$. By induction hypothesis, there exists a threshold t_0 such that $\alpha\restriction_{n-1} \leqslant_p \sigma_t$ for all $t \geqslant t_0$. Let $S = \{t \geqslant t_0 : \sigma_t <_p \alpha\}$. Suppose absurdly that S is infinite. Note that for all $t \in S$, as $\alpha\restriction_{n-1} \leqslant_p \sigma_t$ and $\sigma_t <_p \alpha$ we necessarily have $\alpha\restriction_{n-1} \preceq \sigma_t$ with $\sigma_t(n-1) <_p \alpha(n-1)$. In other words, $\sigma_t(n-1) = \infty$ and $\alpha(n-1) = \mathtt{f}$ and $\forall t \in S$ $(\alpha\restriction_{n-1})^\frown\infty \preceq \sigma_t$. Thus, $(\alpha\restriction_{n-1})^\frown\infty \preceq P$, contradicting the hypothesis $\alpha \preceq P$. This concludes the proof of the lemma. ■

Only the strategies along σ_t are executed in step t. Thus, for any strategy C_α along the real path P, there is a threshold t_0 after which only the lower priority strategies or strategies along the real path will be executed. We can therefore prove by induction on n that the strategy $C_{P\restriction_n}$ will only be injured finitely often, and will have the outcome $P(n)$. Any requirement being represented by a strategy along the real path, the requirements will all be satisfied. This concludes the proof of Theorem 5.2. ■

Cappable degrees

We have seen two ways of creating incomparable Turing degrees. The first (see Proposition 4-8.1) consists in creating two sets simultaneously, while the second (see Proposition 4-8.2) starts from a non-computable set, and creates a second set of incomparable degree with the first. It is natural to ask whether it is possible, in the case of minimal pairs of c.e. degrees, to start from an arbitrary non-computable c.e. set, and complete it with another c.e. set to form a minimum pair. This is not the case, as proved by Yates [243] and Lachlan [133].

Definition 5.7. A c.e. degree $\mathbf{a} > \mathbf{0}$ is *cappable* if there exists a degree $\mathbf{b} > \mathbf{0}$ such that $\mathbf{a}$ and $\mathbf{b}$ form a minimal pair. Otherwise, $\mathbf{a}$ is said to be *non-cappable*. ♢

Ambos-Pies et al. [5] obtained a surprising characterization of non-cappable degrees, using promptly simple sets.

Definition 5.8. A co-infinite c.e. set A is *promptly simple* if there exists a computable enumeration $A_0 \subseteq A_1 \subseteq \dots$ and a computable function $f : \mathbb{N} \to \mathbb{N}$ such that

$$W_e \text{ infinite } \Rightarrow \exists^\infty x, s\ (x \in W_e[s] \setminus W_e[s-1] \wedge x \in A_{f(s)}).$$

In other words, a co-infinite set A is promptly simple if it is not only co-immune, but even more, this co-immunity must be achieved by making infinitely many elements enter A few time after they appear in W_e. A c.e. degree is *promptly simple* if it contains a promptly simple set.

Theorem 5.9 (Ambos-Spies, Jockusch, Shore, et Soare [5]) *The non-cappable degrees are precisely the promptly simple degrees.*

In particular, the proof showing that if a degree is not promptly simple, then it is cappable, is obtained via a variation of Theorem 5.2.

Chapter 14

Structure of the Turing degrees

The study of Turing degrees has been carried out in conjunction with that of its *structure*, as a partial order. Gerald E. Sacks is undoubtedly one of the main protagonists of this adventure.

Gerald Sacks, 1933–2019

Sacks began studying engineering at Cornell University in his youth, which he interrupted mid-term to enlist in the army for three years. It was at this time that he got his hands on a copy of *Introduction to Metamathematics* by Kleene, which fascinated him [38]. On his return to civilian life, he then oriented the rest of his studies towards mathematics. Barkley Rosser, an eminent logician who studied with Kleene and Church, agrees to take him on as a doctoral student. Sacks will become in the sixties one of the pioneers of modern computability theory. He will participate in particular as we will see in the first studies on the structure of Turing degrees, and besides his work, he will become famous for the large number of his students who will become leading logicians, among whom we can quote Harvey and Sy Friedman, Léo Harrington, Richard Shore, Théodore Slaman and Stephen Simpson. We will see that the last three members of this list took a very active part in the study of the structure of Turing degrees.

Let's pose without further ado the vocabulary we will be using.

Notation

$(\boldsymbol{\mathcal{D}}, \leqslant)$ denotes the partial order structure of Turing degrees.

We will write $\mathbf{a} \leqslant \mathbf{b}$ for two degrees $\mathbf{a}, \mathbf{b} \in \boldsymbol{\mathcal{D}}$ if $A \leqslant_T B$ for any element $A \in \mathbf{a}$ and any element $B \in \mathbf{b}$, and we will write $\mathbf{a} < \mathbf{b}$ if $\mathbf{a} \leqslant \mathbf{b}$ and $\mathbf{b} \nleqslant \mathbf{a}$. In order to be completely clear, let us recall the vocabulary of use of partial orders: given a partially ordered set A and a subset $B \subseteq A$, an *upper bound* (resp. *lower bound*) of B is an element of A greater (resp. smaller) than all the elements of B. An upper bound (resp. lower bound) of B is *minimal* (resp. *maximal*) if no other upper bound of B is smaller than it (resp. no other lower bound of B is greater than it). Finally a least upper bound (resp. greatest lower bound) of B is a *upper bound* (resp. *lower bound*) if it is smaller than any upper bound of B (resp. greater than any lower bound of B). It is easy to show that a least upper bound (resp. greatest lower bound) when it exists is unique.

1. Minimal degrees

One of the first results obtained on the structure of Turing degrees concerns the initial segments of $\boldsymbol{\mathcal{D}}$, and in particular the existence of so-called *minimal* degrees:

Definition 1.1. A Turing degree $\mathbf{d}$ is *minimal* if it is different from $\mathbf{0}$ — the computable degree— and if there is no degree $\mathbf{e}$ such that $\mathbf{0} < \mathbf{e} < \mathbf{d}$.$\diamondsuit$

In other words a set $X \in 2^{\mathbb{N}}$ is of minimal degree if it is not computable and if for any functional Φ such that $\Phi(X, n) \downarrow \in \{0, 1\}$ for all n, either the set $\{n : \Phi(X, n) \downarrow = 1\}$ is computable, or it is able to compute back X.

The proof of the existence of a minimal degree is one of the first uses of forcing in computability theory, via Sacks forcing, which we saw in Section 11 -3.

1.1. Existence

The existence of minimal degrees, due to Spector [224], was initially made by forcing with *uniform* computable f-trees, that is to say f-trees $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ such that for any size n there exists a unique string τ_n such that for any string σ of size n and any $i \in \{0, 1\}$ we have $T(\sigma i) = T(\sigma) i \tau_n$. We can also see the paths of these f-trees as being all the possibilities of completion of a set X on which an infinity of bits are not specified.

Spector's restriction is of interest for the more general study of initial segments in Turing degrees, but for minimal degrees (i.e., initial segments of size 2), Shoenfield [206] noticed that computable Sacks forcing, easier to handle, is sufficient.

Definition 1.2. Let Γ be a Turing functional. Two strings $\sigma, \tau \in 2^{<\mathbb{N}}$ form a Γ-*split* if there is an integer $n \in \mathbb{N}$ such that $\Gamma^\sigma(n)\downarrow \neq \Gamma^\tau(n)\downarrow$. An f-tree $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ is Γ-*splitting* if for all $\sigma \in 2^{<\mathbb{N}}$, the strings $T(\sigma 0)$ and $T(\sigma 1)$ form a Γ-split. An f-tree T *avoids* Γ-*splitting* if no pair of strings $\sigma, \tau \in \operatorname{Im} T$ forms a Γ-split. $\diamondsuit$

For this section, we will consider the functionals Γ as being partial functions from $2^\mathbb{N}$ to $2^\mathbb{N}$. In this particular context we will then denote by $\operatorname{dom} \Gamma$ — the domain of definition of Γ — the class $\{X \in 2^\mathbb{N} : \forall n\, \Gamma(X, n)\downarrow \in \{0, 1\}\}$. Given $X \in \operatorname{dom} \Gamma$, we will write $\Gamma(X)$ for the set $\{n \in \mathbb{N} : \Gamma(X, n)\downarrow = 1\}$, and we will speak of the totality of Γ with respect to $2^\mathbb{N}$, and not with respect to its inputs for a fixed oracle.

The key lemma in the construction of a minimal degree says that for any functional Γ and any computable f-tree, there exists a computable sub-tree on which Γ is everywhere defined and injective, or on which Γ is a constant function, restricted to its domain of definition in the sub-f-tree.

Lemma 1.3. For every computable f-tree $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ and every Turing functional Γ, there exists a computable sub-f-tree S of T which is either Γ-splitting, or avoids Γ-splittings. $\star$

Proof. Two cases arise.

Case 1: there exists a string $\sigma \in 2^{<\mathbb{N}}$ such for all $\rho, \tau \succeq \sigma$, $T(\rho)$ and $T(\tau)$ do not form a Γ-split. Let S be the computable sub-tree defined by $S(\mu) = T(\sigma\mu)$. Then, S is sub-tree of T avoiding Γ-splittings.

Case 2: for any string $\sigma \in 2^{<\mathbb{N}}$, there are extensions $\rho, \tau \succeq \sigma$ such that $T(\rho)$ and $T(\tau)$ form a Γ-split. We then compute S as follows. We define $S(\epsilon) = T(\epsilon)$. Suppose we have computed $S(\sigma) = T(\mu)$ for strings $\sigma, \mu \in 2^{<\mathbb{N}}$. We then look for strings $\rho_0, \rho_1 \succeq \mu$ such that $T(\rho_0)$ and $T(\rho_1)$ form a Γ-split. By hypothesis, the search always succeeds. We then define $S(\sigma 0) = T(\rho_0)$ and $S(\sigma 1) = T(\rho_1)$. Note that since $T(\rho_0)$ and $T(\rho_1)$ form a Γ-split, in particular, these two strings are incomparable. Thus, S is indeed an f-tree, and $\operatorname{Im} S \subseteq \operatorname{Im} T$. By construction, S is Γ-splitting. ■

The interest of obtaining Γ-splitting or avoiding Γ-splitting f-trees is found in the following lemma.

Lemma 1.4. Let T be a computable f-tree and Γ a functional.

(1) If T avoids Γ-splittings then for all $X \in \operatorname{dom}\Gamma \cap [T]$ the element $\Gamma(X)$ is computable.

(2) If T is Γ-splitting then for all $X \in \operatorname{dom}\Gamma \cap [T]$ we have $X \leqslant_T \Gamma(X)$. $\star$

Proof. (1) Suppose that T avoids Γ-splittings. Let $X \in \operatorname{dom}\Gamma \cap [T]$. Then, to know the n-th bit of $\Gamma(X)$ it suffices to search $\sigma \in \operatorname{Im} T$ such that $\Gamma(\sigma, n) \downarrow = i$ for $i \in \{0,1\}$. The n-th bit of $\Gamma(X)$ is then equal to i.

(2) Suppose that T is Γ-splitting. Let $X \in \operatorname{dom}\Gamma \cap [T]$. Let $Y = \Gamma(X)$. To compute X from Y we proceed as follows: as $T(0)$ and $T(1)$ form a Γ-split, there exists $i \in \{0,1\}$ such that $\Gamma(T(i), n) \neq Y(n)$ for some n. We can find i in a computable way in Y. The correct prefix of X is then necessarily $T(1-i)$. Suppose we have computed a prefix $\sigma \prec X$ and a string τ such that $\sigma = T(\tau)$. As $T(\tau 0)$ and $T(\tau 1)$ form a Γ-split, there exists $i \in \{0,1\}$ such that $\Gamma(T(\tau i), n) \neq Y(n)$ for some n. We can find i in a computable way in Y. The correct prefix of X is then necessarily $T(\tau(1-i))$. By proceeding in this way, we then compute larger and larger prefixes of X from $Y = \Gamma(X)$. ■

We now have all the necessary ingredients to prove the existence of minimal degrees.

Theorem 1.5 (Spector [224])
Any sufficiently generic set for Sacks forcing is of minimal degree.

Proof. Let $(\mathbb{P}, \leqslant)$ be computable Sacks forcing. Let us note first that, according to Exercise 11-3.3, if $G \in 2^{\mathbb{N}}$ is sufficiently generic for this forcing then it is not computable.

Let Γ be a Turing functional. According to Lemma 1.3 the set of conditions $c \in \mathbb{P}$ such that c is Γ-splitting or c avoids Γ-splittings is dense. According to Lemma 1.4, in the first case, for all $G \in [c]$ the functional Γ is defined on G and $\Gamma(G) \geqslant_T G$. In the second case, according to Lemma 1.4, for all $G \in [c]$, if the functional Γ is defined on G then $\Gamma(G)$ is computable.

So if G is sufficiently generic for Sacks forcing, it is of minimal degree. ■

Corollary 1.6
There is a perfect class of sets, all of distinct minimal degrees.

Proof. We can first show that there exists a perfect tree of minimal degrees. It suffices to proceed as in the proof of Theorem 8-5.1, in order to "duplicate" the construction of a minimal degree. With the forcing formalism, let $(D_n)_{n\in\mathbb{N}}$ be a sequence of dense sets of Sacks forcing conditions, sufficient to force a set to be of minimal degree. We choose $c_\epsilon \in D_0$, then for any string σ of size n, assuming $c_\sigma \in D_n$ defined, we define $c_{\sigma 0} \leqslant c_\sigma$ and $c_{\sigma 1} \leqslant c_\sigma$ in such a way that $[c_{\sigma 0}] \cap [c_{\sigma 1}] = \emptyset$. We can also make sure that the first branching node of $c_{\sigma i}$ strictly extends the first branching node of c_σ for $i \in \{0, 1\}$. The final tree is given by the set of strings τ such that there is $X \in 2^{\mathbb{N}}$ for which $\tau \in \bigcap_{\sigma \prec X} c_\sigma$.

Once we have a perfect tree containing only minimal degrees, we can refer to Exercise 8-5.4 to extract from it a perfect subtree containing only pairwise incomparable degrees. ■

1.2. Computation of a minimal degree

A fine analysis of the level of effectiveness necessary to carry out Theorem 1.5 shows that there exists a $\emptyset''$-computable minimal degree. It is in fact possible to considerably improve this result, by noting that the use of computable f-trees is not absolutely necessary:

C.e. tree

A c.e. f-tree is a partial function $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ such that for any σ such that $T(\sigma)\downarrow$, either $T(\sigma 0)\uparrow$ and $T(\sigma 1)\uparrow$, or $T(\sigma 0)\downarrow\succeq T(\sigma)$ and $T(\sigma 1)\downarrow\succeq T(\sigma)$ with $T(\sigma 0)$ and $T(\sigma 1)$ incomparable.

The use of c.e. f-trees which are Γ-splitting or avoid Γ-splittings, allows to make constructions of minimal degrees requiring less computational power. It is then a question of effective constructions which do not fall strictly speaking any more into the scope of forcing. We list, without proving them, three important results which use this new type of tree to more easily compute sets of minimal degrees. By "more easily", one needs to understand "with little computational power", the proofs being on the contrary more complex …

Theorem 1.7 (Sacks [193])
There is a minimal degree under $\emptyset'$.

Note that according to Proposition 7-4.7, Sacks' result implies the existence of a minimal hyperimmune degree. This result was subsequently improved by Yates [244], and independently by Cooper [43]:

Theorem 1.8 (Yates [244], Cooper [43])
Let X be a non-computable c.e. set, then X computes a set G of minimal degree.

Note that a minimal degree can never be c.e.: a sophisticated use of the famous priority method which we saw in Chapter 13, makes it possible to show that any non-computable c.e. set A compute another non-computable c.e. set B which does not compute it A (see Theorem 5.1 for the result in all its generality).

Finally, Groszek and Slaman showed that any PA degree could compute a minimal degree, via the following remarkable result.

Theorem 1.9 (Groszek et Slaman [82])
Any PA degree computes a minimal degree. More precisely, there exists a non-empty Π^0_1 class whose members are either of minimal degree, or compute a non-computable c.e. degree.

Exercise 1.10. ($\star$) Show that any non-empty Π^0_1 class contains a set of the same degree as a c.e. set. Deduce that there is no non-empty Π^0_1 class containing only elements of minimal degrees. ◇

The possibility of constructing "complex" minimal degrees has also been well studied. The main result in this direction is as follows.

Theorem 1.11 (Kumabe [129])
There is a minimal degree which is also DNC.

Kumabe first showed the existence of a minimal DNC degree, in a very complex article, which was never published. The proof was then reworked by Kumabe and Lewis [129], and the presentation was subsequently simplified by Khan and Miller [114], who rewrote it under the formalism of forcing, via *Forcing with Bushy Trees*. The following exercise shows that a minimal degree on the other hand can never be DNC_2.

Exercise 1.12. ($\star$) Show that there exists a non-empty Π^0_1 class of sets $X \oplus Y$ such that X is of PA degree and Y and of PA degree relative to X. Deduce that no PA set is of minimal degree (the reader can consult Proposition 24-2.4 for an iteration of this result). ◇

1.3. Relativization: minimal cover

The construction of a minimal degree with the forcing on f-trees is relativized to any Turing degree in the following sense.

Definition 1.13. Let $\mathbf{a}$ and $\mathbf{b}$ be Turing degrees. We say that $\mathbf{b}$ is a *minimal cover* of $\mathbf{a}$ if $\mathbf{b} > \mathbf{a}$ and if there is no degree $\mathbf{c}$ such that $\mathbf{a} < \mathbf{c} < \mathbf{b}$. ♢

In other words, $\mathbf{b}$ is a minimal cover of $\mathbf{a}$ if it is a minimal element in the cone of Turing degrees strictly above $\mathbf{a}$. The relativization of Theorem 1.5 shows the following theorem.

Theorem 1.14
Any Turing degree has minimal cover.

Proof. Let $A \subseteq \mathbb{N}$ be any set. The objective is to build a set $B >_T A$ such that any set C computable by B is either A-computable, or such that $A \oplus C \geqslant_T B$. Thus, if $B \geqslant_T C >_T A$ we will have $C \geqslant_T B$.

It suffices to consider a variant of Sacks forcing, for which our f-trees are this time A-computable, and such that each of their paths computes A. We could, for example, restrict ourselves to A-computable f-trees such that A is encoded in the even bits of each path of the f-tree.

It suffices then to repeat the proof of Theorem 1.5 with this new partial order, noting the following difference: given a Γ-splitting f-tree T, for all $X \in [T]$ we now need A in order to find X starting from $\Gamma(X)$: indeed we need the knowledge of T. This is the reason why we will have $A \oplus \Gamma(X) \geqslant_T X$ and not $\Gamma(X) \geqslant_T X$. ■

Note that a minimal cover $\mathbf{b}$ of $\mathbf{a}$ does not exclude the existence of degrees $\mathbf{c} < \mathbf{b}$ incomparable with $\mathbf{a}$. This leads us to define a stronger notion of cover:

Definition 1.15. Let $\mathbf{a}$ and $\mathbf{b}$ be Turing degrees. We say that $\mathbf{b}$ is a *strong minimal cover* of $\mathbf{a}$ if $\mathbf{b} > \mathbf{a}$ and if for any Turing degree $\mathbf{c} < \mathbf{b}$, we have $\mathbf{c} \leqslant \mathbf{a}$. ♢

As noted in the proof of Theorem 1.14, the relativized version of Theorem 1.5 does not prove the existence of a strong minimal cover for any Turing degree, and for good reason: some degrees do not admit any strong minimal cover, although many will.

Ishmukhametov [99] has established an elegant characterization of the c.e. degrees admitting a strong minimal cover.

Theorem 1.16 (Ishmukhametov [99])
A c.e. set $A \subseteq \mathbb{N}$ admits a strong minimal cover iff any A-computable function $f : \mathbb{N} \to \mathbb{N}$ is bounded for n sufficiently large by the function $n \mapsto$

min $\{s \in \mathbb{N} : \emptyset'[s] \upharpoonright_n = \emptyset' \upharpoonright_n\}$.

A general characterization of the degrees admitting a strong minimal cover is for the moment unknown, although many partial results have been established (see Lewis [147]).

2. Nature of $\boldsymbol{\mathcal{D}}$

What does the partial order $(\boldsymbol{\mathcal{D}}, \leqslant)$ look like? Regarding its size first, we have seen in this book several constructions of perfect trees where each path is in a different Turing degree (see for example Exercise 7-5.8, Exercise 8-5.3 or Exercise 8-5.4). This gives us an injection of $2^{\mathbb{N}}$ into $\boldsymbol{\mathcal{D}}$. Using the axiom of choice we can choose a representative in each Turing degree, which gives an injection of $\boldsymbol{\mathcal{D}}$ into $2^{\mathbb{N}}$. The cardinality of $\boldsymbol{\mathcal{D}}$ is therefore $|2^{\mathbb{N}}|$, that of $2^{\mathbb{N}}$. Note that we cannot necessarily choose a representative in each Turing degree if we do not have the axiom of choice. The fact remains that "morally", $\boldsymbol{\mathcal{D}}$ is of cardinality $|2^{\mathbb{N}}|$.

Given an element $\mathbf{a} \in \boldsymbol{\mathcal{D}}$, the set of elements below $\mathbf{a}$ is at most countable since a set can only compute a countably many elements. On the other hand, the cardinality of the elements above $\mathbf{a}$ is that of $2^{\mathbb{N}}$: given $A \in \mathbf{a}$ we can easily create a perfect tree whose paths are all of the form $A \oplus X$ for $X \in 2^{\mathbb{N}}$, and all in different Turing degrees.

The use of the Turing join leads us to the following consideration: given two sets A, B, the set $A \oplus B$ computes both A and B, and any set computing at the same time times A and B computes $A \oplus B$. In terms of degrees, this implies that every pair of degrees degrees $\mathbf{a}, \mathbf{b}$ has a least upper bound. There is therefore a minimum of structure in this partial order, for which we introduce the following concept.

Definition 2.1. A *lattice* is a partially ordered set in which any pair of elements a, b has a least upper bound noted $a \cup b$ and a greatest lower bound noted $a \cap b$. An *upper (resp. lower) semilattice* is an ordered set for which every pair of elements has a least upper bound (resp. greatest lower bound). ♢

The paragraph preceding this definition leads to the following theorem, appearing in the founding article of the study of the structure of Turing degrees.

Theorem 2.2 (Kleene et Post [122])
$(\boldsymbol{\mathcal{D}}, \leqslant)$ is an upper semilattice of cardinality $|2^{\mathbb{N}}|$, with a smallest but no largest element, such that each element has a most countably many

elements below it, and a set of cardinality $|2^{\mathbb{N}}|$ of elements above it.

We sometimes base the vocabulary of Turing degrees on that of the sets they contain: given two degrees $\mathbf{a}, \mathbf{b}$ we say that the least upper bound $\mathbf{a} \cup \mathbf{b}$ of $\mathbf{a}$ and $\mathbf{b}$ is the *join* of $\mathbf{a}, \mathbf{b}$. Note that in an upper semilattice, any finite sequence of elements also admits a least upper bound. In particular for a finite set of degrees $\mathbf{a}_1, \dots, \mathbf{a}_n$ we will denote it $\mathbf{a}_1 \cup \dots \cup \mathbf{a}_n$ and it will be the degree of the join $A_1 \oplus \cdots \oplus A_n$ for any representatives $A_i \in \mathbf{a}_i$.

What happens for countable sets of degrees? The following theorem implies that if such a set is closed under join —i.e. $\mathbf{a} \cup \mathbf{b}$ is in our set for all $\mathbf{a}, \mathbf{b}$ in our set— and has no maximal element, then it does not have any least upper bound.

Theorem 2.3 (Sacks [196])
Any countable set of degrees closed under join and without maximal element has minimal upper bounds in quantity $|2^{\mathbb{N}}|$.

Proof sketch. First note that an upper bound of a set of integers of degrees $(\mathbf{a}_n)_{n \in \mathbb{N}}$, is also an upper bound of $\mathbf{a}_0 \leqslant \mathbf{a}_0 \cup \mathbf{a}_1 \leqslant \mathbf{a}_0 \cup \mathbf{a}_1 \cup \mathbf{a}_2 \leqslant \dots$. As $(\mathbf{a}_n)_{n \in \mathbb{N}}$ has no maximal element and is closed under join, we can therefore consider without loss of generality that our set of degrees is such that $\mathbf{a}_n < \mathbf{a}_{n+1}$. For all n, let A_n be a representative of $\mathbf{a}_n$.

It is now enough to elaborate on Sacks forcing (see Section 1) allowing to create the minimal cover of a degree. We start with an A_0-computable f-tree whose paths all compute A_0. A forcing condition will be an A_n-computable f-tree whose paths all compute A_n (for a certain n). We extend such a forcing condition $T : 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ to the f-tree Q such that $\operatorname{Im} Q \subseteq \operatorname{Im} T$ consists of the paths which encode $A_{n+1} \oplus X$ for all $X \in 2^{\mathbb{N}}$. Formally, $Q(\sigma i) = T(A_{n+1} \upharpoonright_{|\sigma i|} \oplus \sigma i)$ for any string σ and any $i \in \{0, 1\}$. As $A_{n+1} \geqslant_T A_n$, then A_{n+1} can compute T and thus find $A_{n+1} \oplus X$ from the path of Q which encodes $A_{n+1} \oplus X$ in T.

The way of doing a minimal cover does not change, and the technique described in Section 1 applies in the same way. The resulting generic set G will compute each set A_n, and will be such that anything that is computed by G and which can compute each set A_n, can also compute G.

To obtain minimal upper bound in quantity $|2^{\mathbb{N}}|$, one can build a perfect tree of minimal upper bound by subdividing the construction in two, then each substructure in two, etc., as in the proof of Theorem 8-5.1. ■

A countable set of degrees closed under join and with no maximum element therefore always has two distinct minimal upper bounds. We can therefore deduce the following corollary.

Corollary 2.4
A countable set of degrees closed under join and with no maximal element never has a least upper bound.

Note that for a sequence of sets $(A_n)_{n\in\mathbb{N}}$, the set $\bigoplus_{n\in\mathbb{N}} A_n$ (see Definition 10-3.24) is not in general a minimal upper bound, as shown by the following elegant result.

Theorem 2.5 (Enderton et Putnam [59], Sacks [198])
There exists a minimal upper bound of $(\emptyset^{(n)})_{n\in\mathbb{N}}$ whose double jump is in the same Turing degree as that of $\bigoplus_n \emptyset^{(n)}$.

Proof sketch. For a direction, it suffices to notice that the double jump of any upper bound of $(\emptyset^{(n)})_{n\in\mathbb{N}}$ allows to compute $\bigoplus_n \emptyset^{(n)}$. Let B be an upper bound and let f be a computable function such that $f(X') = X$ for any X (see Exercise 4-6.4 for more details on such a function). Using the double jump of B we look for a functional code e_1 such that $f(\Phi_{e_1}(B)) = \emptyset$, then a functional code e_2 such that $f(\Phi_{e_2}(B)) = \Phi_{e_1}(B)$, etc.

For the other direction, it suffices to see that the construction of an upper bound of $(\emptyset^{(n)})_{n\in\mathbb{N}}$ is effective using $\bigoplus_n \emptyset^{(n)}$, and forces at each step a functional Φ_e to be partial or else total on all the elements of the tree considered: one thus does not only compute the resulting generic G, but one can also determine the set of the codes of total functionals on G. The double jump of G reduces to this set (see Exercise 5-7.2). ■

To end this section, we answer the question that may have tormented the reader since the beginning of this chapter: is the structure $(\boldsymbol{\mathcal{D}}, \leqslant)$ a lattice? We will see that it is not, and for this we introduce the notion of exact pair.

Definition 2.6. The degrees $\mathbf{a}, \mathbf{b}$ form a *exact pair* for a set of degrees $C \subseteq \boldsymbol{\mathcal{D}}$ if $\mathbf{a}$ and $\mathbf{b}$ each bound all the degrees of C, and if each degree below both $\mathbf{a}$ and $\mathbf{b}$ is also bounded by a degree of C. ◇

Theorem 2.7
Any countable set of degrees C closed under join admits an exact pair.

Proof. Let $(\mathbf{a}_n)_{n\in\mathbb{N}}$ be a set of Turing degrees closed under join. Let A_n be a representative of $\mathbf{a}_n$. The idea is to construct two sets $G_0 = \bigoplus_n X_n^0$ and $G_1 = \bigoplus_n X_n^1$ such that each column X_n^i for $i \in \{0,1\}$ is equal to the set A_n, except for a finite number of bits. It is clear that such sets G_0, G_1 allow us to compute all the A_n. We must now build them via an adapted forcing in such a way that if G_0 and G_1 compute the same set, then this set is computable by $A_0 \oplus A_1 \oplus \cdots \oplus A_m$ for a certain m.

Our forcing conditions are made up of triplets (σ_0, σ_1, n) where $\sigma_0, \sigma_1 \in 2^{<\mathbb{N}}$ and $n \in \mathbb{N}$. The parameter n is used to control the possible extensions of our conditions. We have $(\sigma_0, \sigma_1, n) \succeq (\tau_0, \tau_1, m)$ for two forcing conditions if $\sigma_0 \preceq \tau_0$, if $\sigma_1 \preceq \tau_1$, if $n \leqslant m$ with the restriction that for all $\langle k, a\rangle$ such that $|\sigma_i| \leqslant \langle k, a\rangle < |\tau_i|$ for $k \leqslant n$, we must have $\tau_i(\langle k, a\rangle) = A_k(a)$. Given a set of conditions $(\sigma_0^0, \sigma_1^0, n_0) \succeq (\sigma_0^1, \sigma_1^1, n_1) \succeq (\sigma_0^2, \sigma_1^2, n_2) \succeq \ldots$, the generic set G_0 will be the limit of $\sigma_0^0 \preceq \sigma_0^1 \preceq \sigma_0^2 \preceq \ldots$ and the generic set G_1 will be the limit of $\sigma_1^0 \preceq \sigma_1^1 \preceq \sigma_1^2 \preceq \ldots$. Note that the restriction on the possible extensions guarantees that as long as the sequence $n_0 \leqslant n_1 \leqslant n_2 \leqslant \ldots$ is unbounded, each n-th column of G_i will indeed be equal to A_n except for a finite number of bits.

For any pair of functionals Φ_{e_0}, Φ_{e_1}, we will force $\Phi_{e_0}(G_0) = \Phi_{e_1}(G_1) = X$ implies $X \leqslant_T A_0 \oplus \cdots \oplus A_n$ for some n. Note that if G_0 and G_1 bound the same degree, there necessarily exist two functionals such that G_0 and G_1 compute the same set in this degree. Such a construction therefore achieves our objectives.

Let (σ_0, σ_1, n) be a forcing condition and Φ_{e_0}, Φ_{e_1} be two functionals. We are looking for two strings τ_0, τ_1 such that (τ_0, τ_1, n) is a valid extension of (σ_0, σ_1, n), and such that $\Phi_{e_0}(\tau_0, x)\downarrow \neq \Phi_{e_1}(\tau_1, x)\downarrow$ for some x. If this search is successful, then we take (τ_0, τ_1, n) as an extension of (σ_0, σ_1, n). Otherwise it means that on all x, the computations $\Phi_{e_0}(\tau_0, x)$ and $\Phi_{e_1}(\tau_1, x)$, when they halt, return the same value for any valid extension $(\tau_0, \tau_1, n) \succeq (\sigma_0, \sigma_1, n)$.

Note that by definition of what is a valid extension of (σ_0, σ_1, n), it is possible to enumerate them using $A_0 \oplus \cdots \oplus A_n$. If for all x, there exists a valid extension (τ_0, τ_1, n) such that $\Phi_{e_0}(\tau_0, x)\downarrow$ or $\Phi_{e_1}(\tau_1, x)\downarrow$ then we can compute via $A_0 \oplus \cdots \oplus A_n$ the unique element Z thus potentially computable by any generic G_0 via Φ_{e_0} or any generic G_1 via Φ_{e_1}. Otherwise at least one of the two computations $\Phi_{e_0}(G_0, x)$ or $\Phi_{e_1}(G_1, x)$ will be partial over a certain x.

So if G_0, G_1 are sufficiently generic for this forcing, for any functional Φ_{e_0}, Φ_{e_1} we will have $\Phi_{e_0}(G_0) = \Phi_{e_1}(G_1) = X$ implies $X \leqslant_T A_0 \oplus \cdots \oplus A_n$ for some n. It follows that G_0, G_1 is an exact pair for degrees $(\mathbf{a}_n)_{n\in\mathbb{N}}$. ■

We can now deduce that $(\boldsymbol{\mathcal{D}}, \leqslant)$ is not a lattice.

Theorem 2.8 (Kleene et Post [122])
The upper semilattice $\boldsymbol{\mathcal{D}}$ is not a lattice. There are in particular degrees $\mathbf{a}, \mathbf{b}$ which do not have a greatest lower bound.

Proof. It suffices to consider a set of degrees $\mathbf{c}_0 < \mathbf{c}_1 < \mathbf{c}_2 < \ldots$ closed under join. This set of degrees therefore admits an exact pair $\mathbf{a}, \mathbf{b}$. Such an exact pair has no greatest lower bound since any degree under both $\mathbf{a}$ and $\mathbf{b}$ is also under a degree $\mathbf{c}_n$ for some n. ∎

3. Universality of $\boldsymbol{\mathcal{D}}$

We see in this section that $(\boldsymbol{\mathcal{D}}, \leqslant)$ presents a certain universality, in the sense that *all partial orders* can *be embedded* in $\boldsymbol{\mathcal{D}}$, except those which cannot claim it for reasons of cardinality.

Definition 3.1. A partial order $(A, \leqslant)$ *embeds* into $(\boldsymbol{\mathcal{D}}, \leqslant)$ if there is an injection $f : A \to \boldsymbol{\mathcal{D}}$ such that $a \leqslant b$ iff $f(a) \leqslant f(b)$. ♢

The structure $(\boldsymbol{\mathcal{D}}, \leqslant)$ therefore contains in it all the partial orders which are not larger than it. This claim, which will be made precise, is actually a bit wrong: there is an open question on this subject, which we will mention shortly. Note that this does not necessarily inform us about the computational complexity of $\boldsymbol{\mathcal{D}}$. Consider for example the partial computable order $\leqslant_R \subseteq \mathbb{Q}^2 \times \mathbb{Q}^2$ defined by $(p_1, p_2) \leqslant_R (q_1, q_2)$ if $p_1 \leqslant q_1$ and $p_2 \leqslant q_2$ for $p_1, p_2, q_1, q_1 \in \mathbb{Q}$. It is not very difficult to show that any countable partial order embeds into $(\mathbb{Q}^2, \leqslant_R)$. The construction of an embedding is done without difficulty, by constructing the injection in a greedy way, element by element, without ever violating at each finite stage the constraints of an embedding. The structure $(\mathbb{Q}^2, \leqslant_R)$ is not computably complex, but it is sufficiently rich in terms of possibilities to contain all the partial orders.

We will use several times the existence of sets called *computably independent*.

Definition 3.2. A countable collection of sets $(X_n)_{n\in\mathbb{N}}$ is *computably independent* if $X_i \not\leqslant_T \bigoplus_{j\neq i} X_j$ for all $i \in \mathbb{N}$. ♢

The existence of computably independent sets does not present any particular difficulties, and we can refer to Exercise 10-3.25 to see that if $G = \bigoplus_n G_n$ is a 1-generic set, then the sets $(G_n)_{n\in\mathbb{N}}$ are computably independent.

3.1. Embeddings in $\boldsymbol{\mathcal{D}}$

Let us immediately see what was announced, in the form of a first theorem.

Theorem 3.3 (Sacks [196])
Any countable partial order embeds into the Turing degrees.

Proof. We saw in the introduction of this section that any countable partial order can be embedded in the structure $(\mathbb{Q}^2, \leqslant_R)$ defined by $(p_1, p_2) \leqslant_R (q_1, q_2)$ if $p_1 \leqslant q_1$ and $p_2 \leqslant q_2$.

It suffices then to show that $(\mathbb{Q}^2, \leqslant_R)$ embeds into $(\boldsymbol{\mathcal{D}}, \leqslant)$. Let $(X_n)_{n\in\mathbb{N}}$ be a sequence of computably independent sets, and let $(a_n)_{n\in\mathbb{N}}$ be an enumeration of the elements of $\mathbb{Q}^2$. The embedding f assigns to the element a_n the Turing degree of the set $\bigoplus_{a_m \leqslant_R a_n} X_m$. We can easily verify $a_n \leqslant_{\mathcal{R}} a_m$ iff $f(a_n) \leqslant f(a_m)$. ∎

Sacks subsequently sought to extend his result to larger partial orders. After all, $(\boldsymbol{\mathcal{D}}, \leqslant)$ admits $|2^{\mathbb{N}}|$ for cardinality. We cannot of course expect any partial order of cardinality $|2^{\mathbb{N}}|$ to be embedded in $(\boldsymbol{\mathcal{D}}, \leqslant)$: if an element in a partial order has an uncountable quantity of predecessors, there is no hope of constructing an embedding of this order to $\boldsymbol{\mathcal{D}}$ since each element of $\boldsymbol{\mathcal{D}}$ has only a countable quantity below it. We must therefore respect this restriction, but are there others? We need here to anticipate a little on the ordinals which will be introduced in Chapter 27, and in particular on the ordinal ω_1, the smallest uncountable infinite ordinal. Sacks obtained the following results.

Theorem 3.4 (Sacks [194])
Any partial order with one of the following properties can be embedded in Turing degrees:

1. *the order is of cardinality $|2^{\mathbb{N}}|$ and each element has a finitely many predecessors;*
2. *the order is of cardinality $|\omega_1|$ and each element has at most countably many predecessors;*
3. *the order is of cardinality $|2^{\mathbb{N}}|$, each element has at most countably many predecessors, as well as at most ω_1 successors.*

In particular, if we assume the continuum hypothesis, namely $|\omega_1| = |2^{\mathbb{N}}|$, Sacks' theorem is optimal: any partial order of cardinality $|2^{\mathbb{N}}|$, and where each element has at most a countable quantity of predecessors, can embed into the Turing degrees. But, if we do not assume the continuum hypothesis, the question is still open.

Question 3.5. Can we embed into $(\mathcal{D}, \leqslant)$ any partial order of cardinality $|2^{\mathbb{N}}|$ where each element has at most countably many predecessors? $\star$

If we do not present here the proof of Sacks, we nevertheless see two ingredients, via the notions of chain and anti-chain.

Definition 3.6. A *chain* is a linearly ordered set of Turing degrees. An *anti-chain* is a set of pairwise incomparable Turing degrees. $\diamondsuit$

Proposition 3.7 (Sacks [196]).

(1) Each countable chain can be extended in the Turing degrees. In particular, any maximal chain is of cardinality ω_1.

(2) Each anti-chain of cardinality less than $|2^{\mathbb{N}}|$ can be extended in the Turing degrees. In particular, any maximal anti-chain is of cardinality $|2^{\mathbb{N}}|$. $\star$

Proof.

(1) If the chain has a largest element, we can consider its Turing jump. Otherwise, we can consider the degree of the Turing join of a representative of each of its elements.

(2) Let D be the set of minimal degrees. By Corollary 1.6, D is of cardinality $|2^{\mathbb{N}}|$. Let C be an anti-chain of Turing degrees with cardinality less than $|2^{\mathbb{N}}|$. Each element of C has a most countably many elements below it. Thus, the downward closure of C has the same cardinality as C. There must therefore exist an element of $\mathbf{d} \in D$ which is not computed by any element of C. Since $\mathbf{d}$ is a minimal degree, it cannot bound any element of C. We deduce that $C \cup \{\mathbf{d}\}$ is an anti-chain. ∎

3.2. Extension of embeddings of $\mathcal{D}$

The notion of embedding can be considered weak, in particular because it does not say anything about the relations that the degrees maintain in the image of an embedding, with the degrees which are not in this image. One way to overcome this weakness is to consider an already existing embedding of a structure $(C, \leqslant)$ towards Turing degrees, and to try to see to what extent this embedding can extend to an extension of the partial order on C. Such a thing is of course not always possible: if elements $a_0, a_1, b \in C$ with $a_0 < b, a_1 < b$ and a_0, a_1 incomparable, are sent to degrees $\mathbf{a}_0, \mathbf{a}_1$ and $\mathbf{a}_0 \cup \mathbf{a}_1$, then such an embedding cannot be extended to any upper bound $c < b$ of a_0, a_1. For this, we introduce the notion of consistent extension.

Definition 3.8. Let C be an upper semilattice and let $D \supseteq C$. Then, D is a *consistent extension* of C if:

(1) for $a, b < d$ with $a, b \in C$ and $d \in D \setminus C$ we have $a \cup b < d$;

(2) no element of $D \setminus C$ is under an element of C.

Note that since C is an upper semilattice, $a \cup b \in C$ for all $a, b \in C$. ♢

Theorem 3.9 (Kleene et Post [122])
Let C be a finite upper semilattice and let D be a finite consistent extension of C. Then, any embedding f from $(C, \leqslant)$ into $(\boldsymbol{\mathcal{D}}, \leqslant)$ can be extended to an embedding of $(D, \leqslant)$ into $(\boldsymbol{\mathcal{D}}, \leqslant)$.

Proof. Let f be an embedding of $(C, \leqslant)$ into $(\boldsymbol{\mathcal{D}}, \leqslant)$. We denote by $\mathbf{a}$ the image of $a \in C$ by f. Let $a \in D \setminus C$ be minimal in $D \setminus C$. Since D is a consistent extension then C can be partitioned into a list of elements $(b_i)_{i \leqslant n}$ and $(c_i)_{i \leqslant m}$ such that $b_0 \cup \ldots \cup b_n < a$, such that a is incomparable with each c_i and such that $b_0 \cup \ldots \cup b_n$ is not above any c_i. It suffices then to construct a Turing degree $\mathbf{a}$ such that $\mathbf{b}_0 \cup \ldots \cup \mathbf{b}_n < \mathbf{a}$ and such that $\mathbf{a}$ is incomparable with each $\mathbf{c}_i$.

By using the fact that $\mathbf{b}_0 \cup \ldots \cup \mathbf{b}_n$ is not above any $\mathbf{c}_i$, we easily construct by finite extensions a degree $\mathbf{d}$ such that $\mathbf{d} \cup \mathbf{b}_0 \cup \ldots \cup \mathbf{b}_n$ is not above any $\mathbf{c}_i$ and such that $\mathbf{b}_0 \cup \ldots \cup \mathbf{b}_n$ is not above $\mathbf{d}$. The embedding is then extended by sending a to $\mathbf{d} \cup \mathbf{b}_0 \cup \ldots \cup \mathbf{b}_n$. By minimality of the choice of a the set $D - \{a\}$ is now a consistent extension of the closure of $C \cup \{a\}$ in the upper semilattice. We can therefore start again until each element of D is assigned. ■

We will see with Lemma 4.2 that the converse of the theorem works: it needs to be a consistent extension so that any embedding is extendible. The previous theorem can be extended:

Theorem 3.10 (Sacks [194])
Let C be a countable upper semilattice and let f be an embedding of $(C, \leqslant)$ in $(\boldsymbol{\mathcal{D}}, \leqslant)$. Let D be a consistent and countable extension of C. Then, f can be extended by an embedding of $(D, \leqslant)$ into $(\boldsymbol{\mathcal{D}}, \leqslant)$.

3.3. Initial segments of $\boldsymbol{\mathcal{D}}$

Another way to strengthen the study of possible embeddings is to consider embeddings on *initial segments* of $\boldsymbol{\mathcal{D}}$

Definition 3.11. A *initial segment* of $\boldsymbol{\mathcal{D}}$ is a downward-closed set of degrees. A *final segment* is an upward-closed set of degrees. $\diamondsuit$

An embedding on an initial segment gives us complete information on all the degrees which are below those of the image of the embedding. For example the construction of a minimal degree indicates to us that the order $a < b$ of two elements a, b can be embedded into an initial segment of $\boldsymbol{\mathcal{D}}$. The existence of a minimal degree with a strong minimal cover (see Definition 1.15) indicates to us that the order $a < b < c$ of three elements a, b, c can be embedded into an initial segment of $\boldsymbol{\mathcal{D}}$. By elaborating on the construction of minimal degrees, Lachlan and Lebeuf obtained the following remarkable result.

Theorem 3.12 (Lachlan et Lebeuf [137])
A countable partial order embeds into an initial segment of $(\boldsymbol{\mathcal{D}}, \leqslant)$ iff it is an upper semilattice with a smallest element.

The proof of Lachlan and Lebeuf is done little by little, the most difficult step being to show it for any finite upper semilattice with a smallest element. This is a non-trivial modification of the construction of minimal degrees. Consider for example the partial diamond order given by $a \leqslant b_1$, $a \leqslant b_2$, $b_1, b_2 \leqslant c$ and b_1, b_2 incomparable. We must then construct two minimal degrees $\mathbf{b}_1, \mathbf{b}_2$, such that $\mathbf{b}_1 \cup \mathbf{b}_2 = \mathbf{c}$, and such that $\mathbf{b}_1, \mathbf{b}_2$ are *the only non-computable degrees* found under $\mathbf{c}$. Such a construction is based on a forcing with computable *uniform* f-trees, as explained in Section 1.1. One can for example build with such a forcing a set $X = X_0 \oplus X_1$ such that X_0 and X_1 are incomparable and minimal, and such that everything which is computed by X is either under X_0, either under X_1, or can recompute X. The detailed proof can be consulted in [177] or [142].

Note that the theorem of Lachlan and Lebeuf gives a complete characterization of Turing ideals of the form $\boldsymbol{\mathcal{D}}(\leqslant \mathbf{a})$ for a certain degree $\mathbf{a}$ (i.e., the set of elements which are found under $\mathbf{a}$): these are the upper semilattices which are at most countable, having a smallest and a largest element.

We can push the theorem of Lachlan and Lebeuf a little further:

Theorem 3.13 (Abraham et Shore [2])
A partial order of cardinality ω_1 is isomorphic to an initial segment of $(\boldsymbol{\mathcal{D}}, \leqslant)$ iff it is an upper semilattice with a smallest element, and in which each element has a most countably many predecessors.

In particular if one makes the assumption of the continuum hypothesis, that completely characterizes exactly the possible initial segments of the partial order $(\boldsymbol{\mathcal{D}}, \leqslant)$. If we do not make the assumption of the continuum hypothesis, then things get complicated:

Theorem 3.14 (Groszek et Slaman [81])
There is a model of ZFC in which there is an uncountable partial order with a smallest element and for which each element has a finite number of predecessors, and which be embedded into an initial segment of $(\boldsymbol{\mathcal{D}}, \leqslant)$.

4. First-order theory of $\boldsymbol{\mathcal{D}}$

We now discuss the complexity of $\boldsymbol{\mathcal{D}}$. The question that interests us is the following: given a first-order statement which concerns the Turing degrees, can we decide whether the latter is satisfied or not? The language that we can use consists only of the relations $\leqslant, <$ or $=$, but it will be possible to extend this language to whatever is definable with $\leqslant, <$ or $=$. For example $\mathbf{0}$, the minimal degree, is definable as being the only degree which satisfies the formula $F(\mathbf{x}) = \forall \mathbf{y}\ \mathbf{x} \leqslant \mathbf{y}$. We can then for example express the existence of a minimal degree as follows: $\exists \mathbf{x}\ (\mathbf{0} < \mathbf{x} \wedge \forall \mathbf{y} \leqslant \mathbf{x}\ (\mathbf{y} = \mathbf{0} \vee \mathbf{y} = \mathbf{x}))$. The fact that $\mathbf{0}$ can be defined by the formula $F(\mathbf{x})$ makes it possible to get rid of it with an equivalent formula:

$$\exists \mathbf{z}\ (F(\mathbf{z}) \wedge \exists \mathbf{x}\ (\mathbf{z} < \mathbf{x} \wedge \forall \mathbf{y} \leqslant \mathbf{x}\ (\mathbf{y} = \mathbf{z} \vee \mathbf{y} = \mathbf{x}))).$$

It is of course longer, and we will therefore allow ourselves these language extensions. We will use in particular the function with two arguments $\cup$ which gives us the join of two degrees, and which is also definable by the formula

$$F(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{x} \leqslant \mathbf{z} \wedge \mathbf{y} \leqslant \mathbf{z} \wedge \forall \mathbf{a}\ ((\mathbf{x} \leqslant \mathbf{a} \wedge \mathbf{y} \leqslant \mathbf{a}) \rightarrow \mathbf{z} \leqslant \mathbf{a}).$$

We easily verify that for all $\mathbf{a}, \mathbf{b} \in \boldsymbol{\mathcal{D}}$, the degree $\mathbf{a} \cup \mathbf{b}$ is the unique degree $\mathbf{z}$ such that $F(\mathbf{a}, \mathbf{b}, \mathbf{z})$.

Now let's come back to our question: given a statement about Turing degrees, can we decide whether the latter is true or false? The question is *a priori* of great complexity if we approach it directly: a statement of the type $\exists \mathbf{a}$ amounts to a statement of the type "does there exist a set X such that". This is a second-order quantification, that is to say, relating not to integers, but to sets of integers. This kind of quantification will be discussed in detail in Part IV. The statements with quantifications on the integers being already undecidable, it is a safe bet that this is also the case for statements on Turing degrees. We can nevertheless show that if the formula is of the type $\forall\exists$ or of the type $\exists\forall$, we can then decide whether it is true or false. This constitutes our first theorem on the subject:

Theorem 4.1 (Lerman [142] (théorème 4.4) et Shore [207])
The Π^0_2 theory of $(\boldsymbol{\mathcal{D}}, \leqslant)$, i.e., the set of Π^0_2 statements true in $(\boldsymbol{\mathcal{D}}, \leqslant)$, is decidable.

To prove the previous theorem, it is essentially enough to see that the converse of Theorem 3.9 is true:

Lemma 4.2. Let C be a finite upper semilattice and let D be a finite extension of C. Then, D is a consistent extension iff any embedding f of $(C, \leqslant)$ into $(\boldsymbol{\mathcal{D}}, \leqslant)$ can be extended to an embedding of $(D, \leqslant)$ into $(\boldsymbol{\mathcal{D}}, \leqslant$). ⋆

Proof. Theorem 3.9 gives us a direction of the lemma. Suppose now that D is not a consistent extension of C. If an element of $d \in D$ is such that $a, b \leqslant d$ but $a \cup b \nleqslant d$, it suffices to consider an embedding which associates $\mathbf{a} \cup \mathbf{b}$ with $a \cup b$. It will then be impossible to extend such an embedding to D. Suppose now that a degree of D is below a degree of C. By Theorem 3.12, there exists an embedding of C on an initial segment of $\boldsymbol{\mathcal{D}}$. Here again, such an embedding cannot be extended to a degree lower than a degree of C. ■

We now have the necessary ingredient to show Theorem 4.1:

Proof of Theorem 4.1. Consider a statement of the form

$$\forall c_1, \dots, \forall c_n \ \exists d_1, \dots, d_m \ F(c_1, \dots, c_n, d_1, \dots, d_m),$$

where F is a Boolean combination of atomic formulas. Let C be the finite set of the possible models of upper semilattices generated by c_i and compatible with the conditions given by F. If C is empty, then the formula is not satisfied without even considering the part of F mentioning the d_i. Otherwise, for all the models of C, we check if this model can be completed in a way compatible with the conditions given by F, and in such a way that the d_i are a consistent extension. If this is the case the formula is true, otherwise it is false. ■

This is the limit of what is decidable. Lachlan [134] showed that the Π^0_3 theory was no longer so.

Theorem 4.3 (Schmerl (see Corollary 4.6 of [142]))
The Π^0_3 theory of $(\boldsymbol{\mathcal{D}}, \leqslant)$ is undecidable.

How complex is the theory of $(\boldsymbol{\mathcal{D}}, \leqslant)$? Can it be decided, for example, using the Turing jump, or even using the disjoint union of all the finite iterations of the Turing jump? We will see that it is not the case: the theory of $(\boldsymbol{\mathcal{D}}, \leqslant)$ is of *maximal* complexity. What do we mean by this? Consider T_2, the second-order theory of $(\mathbb{N}, \times, +, 0, 1)$, i.e., the set of second-order formulas which are true in $\mathbb{N}$. Recall that a second-order formula is of the form $\forall X\ \exists Y\ \ldots F(X, Y, \ldots)$ where the variables $X, Y, \ldots$ are sets of integers, and where F is a first-order formula, parameterized by these sets.

The theory T_2 is therefore the set of second-order formulas which are true in $\mathbb{N}$. If we have access to T_2, we can know if a formula of the first-order theory of $(\boldsymbol{\mathcal{D}}, \leqslant)$ is true: the quantifications $\exists\mathbf{a}$ and $\forall\mathbf{a}$ can be replaced by quantifications on the elements of $2^{\mathbb{N}}$ and using Theorem 9-3.4 relativized to the parameters of the second-order, which allows to transform a $\Sigma^0_n(X, Y, \ldots)$ predicate into a Σ_n formula of arithmetic, we can transform a formula of the first-order F of $(\boldsymbol{\mathcal{D}}, \leqslant)$ into an equivalent second-order formula F^* of $(\mathbb{N}, \times, +, 0, 1)$. Simpson showed that the reverse was also true: through ingenious coding, it is possible to transform a formula of second-order arithmetic into an equivalent formula of $(\boldsymbol{\mathcal{D}}, \leqslant)$.

Let us insist, before going further, on the *extreme* complexity of T_2. We will study in Part IV all the details of the complexity of T_2 restricted to Π^1_1 formulas, that is to say restricted to formulas within which the second-order quantifications are all universal. We will see that this theory already has a considerably high Turing degree compared to $\emptyset'$ or even all finite iterations of $\emptyset'$. This Turing degree is nevertheless well defined and it is *absolute* in the sense that the truth value of a Π^1_1 formula will be the same in the transitive models of set theory sharing the same computable ordinals (see Part IV for a formal definition). From the level of Π^1_2 complexity of the formulas, this meaning becomes more vague. The truth value of this kind of formula will however remain unchanged in all the transitive models of Set Theor,y which this time share not only the same computable ordinals, but the same countable ordinals. Subject to accepting the absoluteness of countable ordinals, the truth of the Π^1_2 formulas is also absolute. The Turing degree of the Π^1_2 level of T_2 is itself higher than all the Turing degrees discussed in this book (it is in a way the supremum of all the Π^1_1 singletons, of which we will see the definition in Section 30-4). The truth value of a Π^1_3 formula may differ between two ZFC models which share the same ordinals, and the meaning that there is to say, that such a formula is *true* or *false* vanishes here a little more. As for the Turing degree of the Π^1_3 theory of T_2, from where we are, that is to say the world of computable things, from which we observe the structure of the universe, even the best telescopes do not allow us to see it: it is simply too far away from us.

Here then is the complexity of the theory of Turing degrees! We deliver here the modern proof of Simpson's theorem, which differs from that which was originally produced, and which presents its own interest. Simpson's proof follows from the following theorem.

Theorem 4.4 (Slaman et Woodin [214])
Let $R \subseteq \boldsymbol{\mathcal{D}}^n$ for $n \in \mathbb{N}$ be a countable set of n-tuples of Turing degrees. Then, R is definable in $(\boldsymbol{\mathcal{D}}, \leqslant)$ with a finite number of parameters. Formally, there is a formula $F(x_1, \dots, x_n, y_1, \dots, y_m)$ and parameters $\mathbf{p}_1, \dots, \mathbf{p}_m \in \boldsymbol{\mathcal{D}}$ such that $(\mathbf{a}_1, \dots, \mathbf{a}_n) \in \boldsymbol{\mathcal{D}}$ iff $F(\mathbf{a}_1, \dots, \mathbf{a}_n, \mathbf{p}_1, \dots, \mathbf{p}_m)$ is true in $(\boldsymbol{\mathcal{D}}, \leqslant)$.

Let us see immediately how to use Theorem 4.4 to show Simpson's theorem: it suffices to code a standard model of arithmetics in the Turing degrees.

Theorem 4.5 (Simpson [209])
The first-order theory of the Turing degrees is many-one equivalent to that of second-order arithmetic.

Proof. We have already seen in the previous paragraphs how to transform a statement of $(\boldsymbol{\mathcal{D}}, \leqslant)$ into an equivalent statement of second-order arithmetic. Now let's see how to do the reverse.

The idea is to code a standard model of $(\mathbb{N}, +, \times, 0, 1)$ in the Turing degrees. Such a model will be coded by a finite set of parameters coding a set N of Turing degrees which represent $\mathbb{N}$, with a specific degree representing 0 and another representing 1. The relations $+$ and $\times$ are also coded by a finite set of parameters.

It is possible to create a formula of $\boldsymbol{\mathcal{D}}$ which checks whether a finite set of parameters encodes well the standard model of arithmetic: it is simply necessary to check the axioms of Robinson arithmetic (see Section 9-2.3), which are in finite number, and then check that any subset of the model has a smallest element. We can refer to Theorem 9-3.13 to see that these conditions are necessary and sufficient to verify that we are indeed dealing with the standard model of integers. The universal quantification "any subset of the model has a smallest element" can be replaced by a universal quantification on the Turing degrees used as parameters to encode subsets of N (our set of degrees which represents $\mathbb{N}$).

Given a formula F of second-order arithmetic, we can finally transform it into an equivalent formula F^* in $\boldsymbol{\mathcal{D}}$, by replacing the quantifications on the sets by quantifications on the parameters encoding these sets. The second-order formula of arithmetic will therefore be interpreted in $\boldsymbol{\mathcal{D}}$ by the formula: there are parameters encoding a standard model of arithmetic, such that F^* is verified in this model. ∎

Now let's move on to the Slaman and Woodin coding. The following lemma constitutes the difficult part of the proof. It is based on a forcing which may seem relatively simple in principle, but the execution of which is subtle and requires a lot of trickery to be completed.

Lemma 4.6 (Slaman et Woodin [214]). Any countable anti-chain in the Turing degrees is definable with three parameters. ⋆

Proof. Let $(\mathbf{a}_n)_{n\in\mathbb{N}}$ be an anti-chain in the uring degrees and let $\mathbf{b}$ be an upper bound of this anti-chain. We are going to define two degrees $\mathbf{g}_0, \mathbf{g}_1$ such that for any degree $\mathbf{y} \leqslant \mathbf{b}$ not bounding any $\mathbf{a}_i$ then $\mathbf{g}_0 \cup \mathbf{y}$ and $\mathbf{g}_1 \cup \mathbf{y}$ have a greatest lower bound and this greatest lower bound is $\mathbf{y}$. In other words, any degree both below $\mathbf{g}_0 \cup \mathbf{y}$ and below $\mathbf{g}_1 \cup \mathbf{y}$ must also be below $\mathbf{y}$. Conversely there will exist for every i a degree both below $\mathbf{g}_0 \cup \mathbf{a}_i$ and below $\mathbf{g}_1 \cup \mathbf{a}_i$ which will not be below $\mathbf{a}_i$. It follows that each $\mathbf{a}_i$ will be a minimal element satisfying the formula

$$F(\mathbf{x}) = \mathbf{x} \leqslant \mathbf{b} \ \wedge \exists \mathbf{c}\ (\mathbf{c} \nleqslant \mathbf{x} \ \wedge \ \mathbf{c} \leqslant \mathbf{g}_0 \cup \mathbf{x} \ \wedge \ \mathbf{c} \leqslant \mathbf{g}_1 \cup \mathbf{x}).$$

In particular, the degrees $\mathbf{a}_i$ will be exactly the degrees $\mathbf{x}$ satisfying the formula $F(\mathbf{x}) \wedge \forall \mathbf{y} \leqslant \mathbf{x}\ \neg F(\mathbf{y})$. The fact that the $\mathbf{a}_i$ form an anti-chain is used only to define them as minimal solutions of F, but no longer occurs thereafter.

We will use for the construction of $\mathbf{g}_0$ and $\mathbf{g}_1$ the following fact: every Turing degree contains a set X computable in any infinite subset of X. We can see it as follows: given any set Y we define X as being the set of prefixes $\sigma \prec Y$, via an encoding of the finite strings by integers.

Let B be a representative of $\mathbf{b}$. Let A_n be a representative of $\mathbf{a}_n$ computable in any of its infinite subsets. We will define two sets G_0, G_1 such that for all i, there exists $C \nleqslant_T A_i$ such that $C \leqslant_T G_0 \oplus A_i$ and $C \leqslant_T G_1 \oplus A_i$, and such that for all $Y \leqslant_T B$ and all D such that $D \leqslant_T G_0 \oplus Y$ and $D \leqslant_T G_1 \oplus Y$, then $Y \geqslant_T D$ or $Y \geqslant_T A_j$ for some j.

We proceed via a forcing which presents similarities with that of Theorem 2.7. Let $\mathbb{P}$ be the set of conditions of the form (σ_0, σ_1, n) for $\sigma_0, \sigma_1 \in 2^{<\mathbb{N}}$ with $|\sigma_0| = |\sigma_1|$ and $n \in \mathbb{N}$. The integer n is used to restrict the possible extensions, the string σ_0 is used for the first generic G_0 and the string σ_1 for the second generic G_1. As in the proof of Theorem 2.7, we can see G_0 and G_1 as being constructed by columns. The integer n indicates that the construction will henceforth be restricted on the first n columns: for a column $k \leqslant n$, if $a \notin A_k$ then there is no restriction for the bit $\langle k, a\rangle$ of the two generics. If on the other hand $a \in A_k$, then the bit $\langle k, a\rangle$ of the two generics must be identical (without necessarily being equal to $A_k(a)$). Formally we have $(\sigma_0, \sigma_1, n) \succeq (\tau_0, \tau_1, m)$ if $\sigma_0 \preceq \tau_0$, if $\sigma_1 \preceq \tau_1$, if $n \leqslant m$, and if moreover the following condition is satisfied: for all $k \leqslant n$, then for all $a \in A_k$ such that $|\sigma_i| \leqslant \langle k, a\rangle < |\tau_i|$, the values $\tau_0(\langle k, a\rangle)$ and $\tau_1(\langle k, a\rangle)$ must be the same.

Consider two sufficiently generic sets G_0, G_1 for this forcing. For each A_n and for $a \in A_n$ sufficiently large, we will have $G_0(\langle n, a\rangle) = G_1(\langle n, a\rangle)$, by definition of what is a valid extension in this forcing. In particular, the sets $X_0^n, X_1^n \subseteq A_n$ defined by $X_i^n(a) = 0$ if $a \notin A_n$ and $X_i^n(a) = G_i(\langle n, a\rangle)$ otherwise are the same except for a finite number of bits and are therefore both computed by $G_0 \oplus A_n$ and $G_1 \oplus A_n$.

Let us show that if G_0, G_1 are sufficiently generic, then no A_n can compute the sets X_0^n, X_1^n thus defined. Given a condition $\langle \sigma_0, \sigma_1, n\rangle$, we can take any extension for the side σ_0 —this then forces some bits of the extension for the other side. By considering the fact that there are necessarily infinite subsets of A_n not computable in A_n, we can necessarily find an extension $\tau_0 \succeq \sigma_0$ such that for a given functional Φ_e, $\Phi_e(A_n)$ never produces the restriction of τ_0 which will be used to make it a prefix of X_n^0— either because $\Phi_e(A_n)$ will be partial, or because it will produce a string incompatible with the prefix of X_n^0 thus forced. Since X_n^1 matches X_n^0 except over a finite number of bits, then A_n will not compute X_n^1 either. This gives us the first part of what we are trying to show: for any A_n there exists a computable set in $G_0 \oplus A_n$ and in $G_1 \oplus A_n$, but not in A_n.

It remains to show that for any $Y \leqslant_T B$ such that Y does not compute any A_n, if $G_0 \oplus Y$ and $G_1 \oplus Y$ compute the same set C, then $Y \geqslant_T C$. Let $p = (\sigma_0, \sigma_1, n)$ be a condition and let Φ_{e_0}, Φ_{e_1} be a pair of functionals. We first separate the chain σ_0 from our condition p. If there exists x and $\tau_0 \succeq \sigma_0$ such that for all $\rho_0 \succeq \tau_0$ we have $\Phi_{e_0}(Y \oplus \rho_0, x) \uparrow$, then we consider a string τ_1 such that (τ_0, τ_1, n) forms a valid extension, for which we will have forced the partiality of $\Phi_{e_0}(Y \oplus G_0)$. Suppose now that for all x and for all $\tau_0 \succeq \sigma_0$ there exists $\rho_0 \succeq \tau_0$ such that $\Phi_{e_0}(Y \oplus \rho_0, x) \downarrow$. Suppose first that there exists an extension $\tau \succeq \sigma_0$ such that for any extension $\rho_0, \rho_1 \succeq \tau$ and for all x we have $\Phi_{e_0}(Y \oplus \rho_0, x) \downarrow = a$ and $\Phi_{e_0}(Y \oplus \rho_1, x) \downarrow = b$ implies $a = b$. Then, we can only produce a unique set via $\Phi_{e_0}(Y \oplus G_0)$ for any generic G_0, and this set is then computable in Y. We therefore force $\Phi_{e_0}(Y \oplus G_0)$ to compute something which is already computable in Y. Suppose finally that for all $\tau \succeq \sigma_0$ there exists $\rho_0, \rho_1 \succeq \tau$ and x such that $\Phi_{e_0}(Y \oplus \rho_0, x) \downarrow \neq \Phi_{e_0}(Y \oplus \rho_1, x) \downarrow$. We then use the following lemma.

Lemma 4.7. For all $\tau \succeq \sigma$ there are x and two extensions $\rho_0, \rho_1 \succeq \tau$ which differ on only one bit and such that

$$\Phi_{e_0}(Y \oplus \rho_0, x) \downarrow = a \neq \Phi_{e_0}(Y \oplus \rho_1, x) \downarrow = b.$$

Proof. It suffices for that to find two extensions of τ of the same size and incompatible on a certain x. Let $i_0, \dots, i_k$ be the bits on which these extensions differ. We invert the i_0 bit in the first extension and expand it to get an a_0 value for x. If $a_0 \neq a$ we are done. Otherwise we in turn reverse i_1 in this new extension, which we extend further to obtain a value a_1, and so on. If each value $a_1 = a_2 = \dots = a_{k-1}$ then $a_{k-1} \neq b$, and our string now differs by only one bit from the one that produced b. ∎

Using the lemma, we compute with the help of Y a sequence of triples $(\tau_{0,m}, \tau_{1,m}, i_m, x_m)_{m\in\mathbb{N}}$ with $i_m < i_{m+1}$ such that each $\tau_{0,m}, \tau_{1,m}$ differ on exactly the bit i_m and are incompatible on x_n. The sequence of bits $(i_m)_{m\in\mathbb{N}}$ is an infinite Y-computable sequence. Recall that our forcing condition is of the form (σ_0, σ_1, n). If there exists i_m such that $i_m = \langle k, a\rangle$ for $k > m$, then for any extension τ' of σ_1 such that $\langle \tau_{0,m}, \tau', n\rangle$ is a valid extension, then $\langle \tau_{1,m}, \tau', n\rangle$ forms also one, because there is no constraint on the i_m bit. We can therefore find an extension of τ' of σ_1 which forces a value for x_n (assuming that we cannot force the partiality on that side), and we take the extension $\tau_{0,m}$ or $\tau_{1,m}$ of σ_0 which forces a different value. We therefore force $\Phi_{e_0}(Y \oplus G_0)$ and $\Phi_{e_0}(Y \oplus G_1)$ to be different. If at present there does not exist $i_m = \langle k, a\rangle$ for $k > m$, then by the pigeonhole principle, there must exist a certain $k \leqslant m$ for which an infinity of i_m is of the form $\langle k, a\rangle$. Also it is not possible to have $A_k(a) = 1$ for each of these i_m, because we would then have an infinite Y-computable subset of A_k, or by hypothesis all infinite subset of A_k computes A_k, and Y does not compute A_k. There must therefore exist $\tau_{0,m}, \tau_{1,m}$ and $i_m = \langle k, a\rangle$ such that $A_k(a) = 0$. Again, any extension τ' of σ_1 that is compatible with $\tau_{0,m}$ will also be compatible with $\tau_{1,m}$, because there is no constraint on the bit i_m. We can then find an extension τ' of σ_1 which forces a value on x_m — unless we can force partiality on that side — and choose an extension from $\tau_{0,m}$ and $\tau_{1,m}$ which forces another value on x_m. This concludes the proof. ∎

And there you go. The proof of the previous lemma was not without difficulty, but we are now almost out of the woods. Finally, we show that any countable subset of $\boldsymbol{\mathcal{D}}^n$ can be encoded in the Turing degrees.

Proof of Theorem 4.4. In what follows, the capitalized variables denote sets or sequences of Turing degrees. Let $R \subseteq \boldsymbol{\mathcal{D}}^n$ be a countable set of n-tuples. Let $\mathbf{b}$ be an upper bound on the set of degrees concerned by R (i.e. on the union of the projections of R on each coordinate). Let $(\mathbf{x}_i)_{i\in\mathbb{N}}$ be a list of all the degrees below $\mathbf{b}$. Note that $\mathbf{b}$ is only an upper bound and therefore that some $\mathbf{x}_i$ may not be degrees of R.

We then find an anti-chain $(\mathbf{c}_i^k)_{k \leqslant n, i \in \mathbb{N}}$ such that $B = (\mathbf{c}_i^k \cup \mathbf{x}_i)_{k \leqslant n, i \in \mathbb{N}}$ forms a set of computably independent degrees (we can easily show that such an anti-chain exists, by finite extensions). For $k \leqslant n$ fixed, let $C_k = (\mathbf{c}_i^k)_{i \in \mathbb{N}}$. We finally define

$$S = \{\mathbf{c}_{i_1}^1 \cup \mathbf{x}_{i_1} \cup \ \dots \ \cup \mathbf{c}_{i_n}^n \cup \mathbf{x}_{i_n} : (\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_n}) \in R\}.$$

Note that as B is computably independent, for an element $\mathbf{a} \in S$, there exists an n-tuple of degrees $\mathbf{b}_1, \dots, \mathbf{b}_n \in B$, unique except for the order, such that $\mathbf{a} = \mathbf{b}_1 \cup \ \dots \ \cup \mathbf{b}_n$. Moreover, the computational independence of B also guarantees that for $\mathbf{b}_1$ there exists a unique $i \leqslant n$ such that $\mathbf{b}_1 = \mathbf{c}_m^i \cup \mathbf{x}_m$ for a certain m, and this m is also unique. The same goes for $\mathbf{b}_2$, $\mathbf{b}_3$, ..., which allows to define R as follows: $(\mathbf{x}_1, \dots, \mathbf{x}_n) \in R$ if

$$\mathbf{x}_1 \leqslant \mathbf{b} \wedge \dots \wedge \mathbf{x}_n \leqslant \mathbf{b} \wedge \exists \mathbf{y}_1 \in C_1 \ \dots \exists \mathbf{y}_n \in C_n \ (\mathbf{x}_1 \cup \mathbf{y}_1) \cup \ \dots \ \cup (\mathbf{x}_n \cup \mathbf{y}_n) \in S.$$

As each C_i and as S are anti-chains, they are definable according to Lemma 4.6. Such a formula is therefore definable in the Turing degrees. ■

Slaman and Woodin used their coding technique as a starting point for a complex study of the *rigidity* of the Turing degrees. A structure is said to be *rigid* if it does not admit any automorphism other than identity, that is to say in the case of degrees, of bijections $f : \boldsymbol{\mathcal{D}} \to \boldsymbol{\mathcal{D}}$ such that $\mathbf{a} \leqslant \mathbf{b} \leftrightarrow f(\mathbf{a}) \leqslant f(\mathbf{b})$. For example, the structure $(\mathbb{R}, +\times, \leqslant)$ of the real numbers is a rigid structure. Given an automorphism $f : \mathbb{R} \to \mathbb{R}$ we must have $f(0) + f(1) = f(1)$ and therefore $f(0) = 0$, then $f(1) = f(1) \times f(1)$ and therefore $f(1) = 1$. Using $f(n+1) = f(n) + 1$ we show that f is necessarily the identity on $\mathbb{N}$, and we show the same thing on $\mathbb{Q}$ by playing with the multiplication. To finish, by using the order and the fact that $\mathbb{Q}$ is dense in $\mathbb{R}$ we then show that f is necessarily the identity on $\mathbb{R}$.

Can we do something similar in the Turing degrees or can we "swap" two degrees $\mathbf{a}$ and $\mathbf{b}$, and extend this swap consistently into an automorphism over $\boldsymbol{\mathcal{D}}$? The question remains open for the moment, even if Slaman and Woodin have considerably reduced the possibilities:

Theorem 4.8 (Slaman et Woodin [215])
Any automorphism over Turing degrees is the identity over $\boldsymbol{\mathcal{D}}(\geqslant \mathbf{0}'')$.

Slaman and Woodin use this result to show in the same article that the double jump function is definable in the Turing degrees. This result will be extended later by Shore and Slaman:

Theorem 4.9 (Shore et Slaman [208])
The Turing jump is definable in the Turing degrees, without parameters.

This result can then in turn be used to show that any automorphism over the Turing degrees is the identity over $\boldsymbol{\mathcal{D}}(\geqslant \mathbf{0}')$. The following question remains open, however.

Question 4.10. Is there an automorphism other than identity on the Turing degrees? ⋆

5. Structure of the c.e. degrees

Another important area of research in Turing degrees is to restrict their study to a well-chosen subset. In this vein, the study of the structure $(\mathcal{R}, \leqslant)$ of the c.e. degrees is certainly the most developed. We make here a quick summary of the main results concerning them.

Let's start right away with Sacks' impressive density theorem, which shows itself at the end of what is certainly one of the most complex infinite injury priority constructions:

Theorem 5.1 (Sacks (1964))
The order $(\mathcal{R}, \leqslant)$ is dense: given c.e. sets $A <_T B$, there exists a c.e. set C such that $A <_T C <_T B$.

The structure of the c.e. degrees therefore appears to be quite different from that of the Turing degrees. There are still some similarities: one thus verifies without problem that any pair of elements admits an upper bound, the set $A \oplus B$ being c.e. if A and B are both c.e. Just like in the Turing degrees, we can show — by working out on the construction of two incomparable degrees — that there exists a countable set of computably independent c.e. degrees [169]. As for the Turing degrees, we can deduce that any countable partial order embeds into the c.e. degrees, which shows, as Sacks noticed, that the Π^0_1 theory of the c.e. degrees is decidable.

Theorem 5.2 (Sacks [196])
The Π^0_1 theory of the c.e. degrees is decidable.

Given an existential formula on the c.e. degrees, it suffices to check whether it is compatible with the axioms of a partial order. If so, it is true, otherwise it is false.

As with Turing degrees, the question of embedding is not really satisfactory in itself. A more interesting question is that of lattice embedding, such that the least upper bounds are sent to the least upper bounds, and the greatest lower bounds to the greatest lower bounds. The lattice embeddings mentioned henceforth will be considered as satisfying this constraint. The existence of a minimal pair of c.e. degrees shows for example that the diamond lattice generated by incomparable $\mathbf{c}_0, \mathbf{c}_1$ (that is to say with $\mathbf{c}_0 \cap \mathbf{c}_1 < \mathbf{c}_0, \mathbf{c}_1 < \mathbf{c}_0 \cup \mathbf{c}_1$) can be embedded in this way in Turing degrees, the greatest lower bound being the degree $\mathbf{0}$.

The construction of a minimal pair of c.e. degrees (see Theorem 13-5.2) has been exploited to show much stronger results. A lattice is said to be *distributive* if $a \cap (b \cup c) = (a \cup b) \cap (a \cup c)$.

Theorem 5.3 (Thomason [232] Lachlan and Lerman)
Any countable distributive lattice embeds into the c.e. degrees

Regarding non-distributive lattices, some can embed into the c.e. degrees [135] and others can't [138]. No characterization is known to date.

We have seen that the theory of Turing degrees is of maximum complexity, the same goes for that of the c.e. degrees, which has the same complexity as the theory consisting of the true formulas of first-order arithmetic. The first result in this direction was obtained by Harrington and Shelah [87] who showed that the first-order theory of the c.e. degrees was undecidable. The proof was then simplified and improved by Harrington and Slaman, then by Nies, Shore and Slaman [175], who proved the following theorem.

Theorem 5.4 (Nies, Shore et Slaman [175])
We can effectively transform a statement F of first-order arithmetic into a statement F^* on the c.e. degrees, such that F is true in $(\mathbb{N}, +, \times, 0, 1)$ iff F^* is true in $(\mathcal{R}, \leqslant)$.

Finally, Lempp, Nies and Slaman [141] showed that the Π^0_3 theory of the c.e. degrees was already undecidable, which leaves open the following question:

Question 5.5. Is the Π^0_2 theory of the c.e. degrees decidable? ⋆