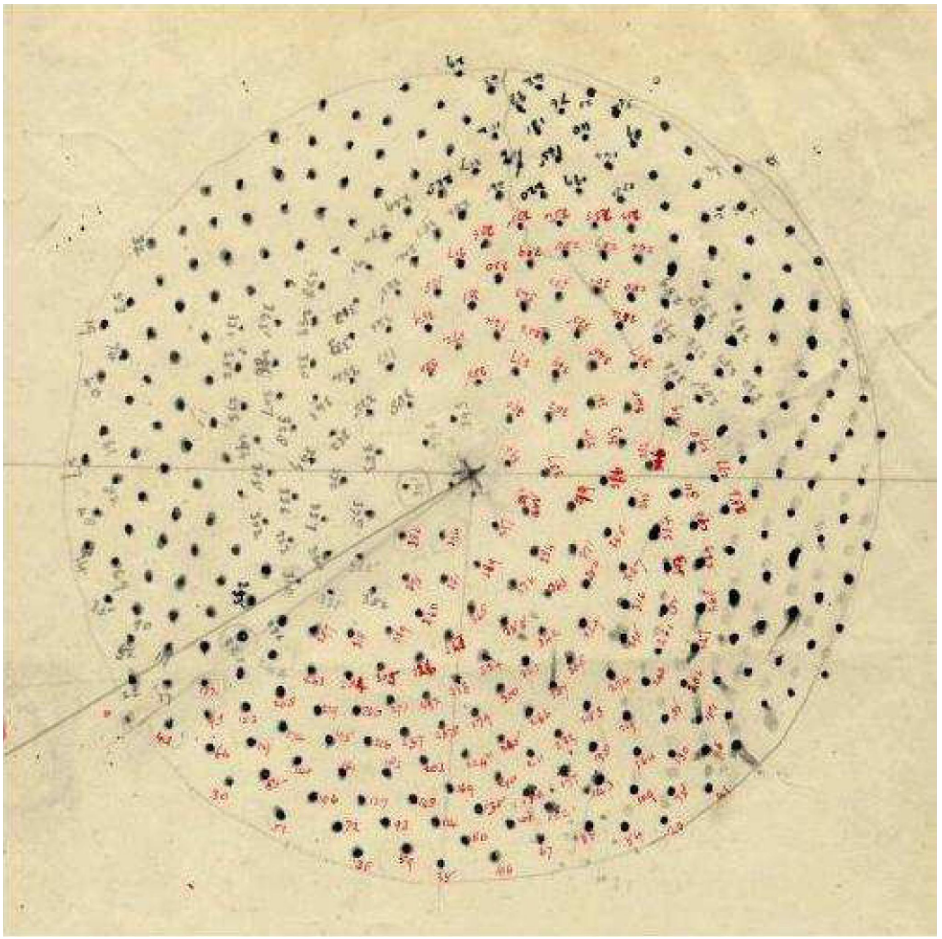
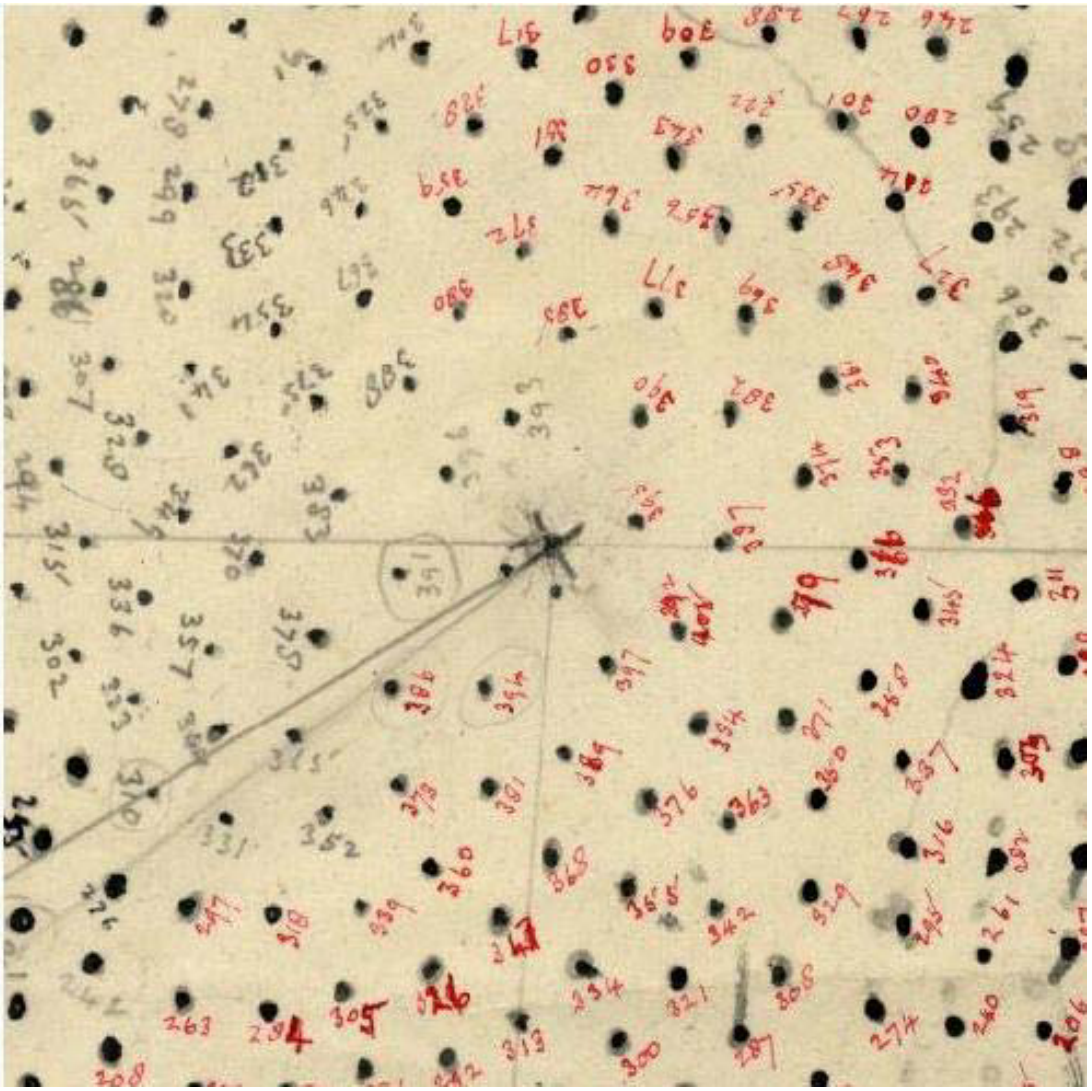
Syllabus Computability Theory



Sebastiaan A. Terwijn

Institute for Discrete Mathematics and Geometry
Technical University of Vienna
Wiedner Hauptstrasse 8–10/E104
A-1040 Vienna, Austria
terwijn@logic.at

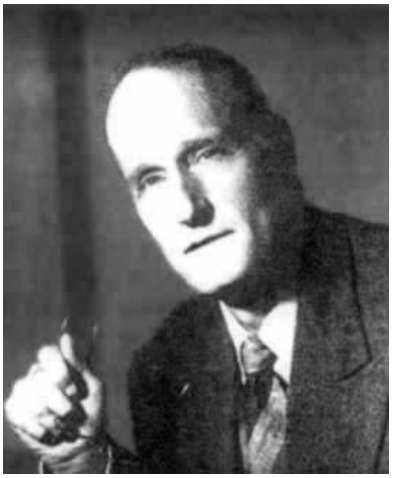


version: 2022
Cover picture and above close-up:
Sunflower drawing by Alan Turing,

Emil Post (1897–1954)

Alonzo Church (1903–1995)

Kurt Gödel (1906–1978)



Stephen Cole Kleene
(1909–1994)

Alan Turing (1912–1954)

Contents

Chapter 1

Introduction

In this syllabus we discuss the basic concepts of computability theory, also called recursion theory. There are various views as to what computability theory *is*. Odifreddi [18, 19] defines it very broadly as the study of functions of natural numbers. Another view is to see the subject as the study of *definability* (Slaman), thus stressing the connections with set theory. But the most commonly held view is to see it as the study of computability of functions of natural numbers. The restriction to functions of natural numbers is, certainly at the start of our studies, a very mild one since, as we will see, the natural numbers possess an enormous coding power so that many objects can be presented by them in one way or another. It is also possible to undertake the study of computability on more general domains, leading to the subject of higher computability theory (Sacks [25]), but we will not treat this in this syllabus.

In the choice of material for this syllabus we have made no effort to be original, but instead we have tried to present a small core of common and important notions and results of the field that have become standard over the course of time. The part we present by no means includes all the standard material, and we refer the reader to the "further reading" section for much broader perspectives. The advantage of our narrow view is that we can safely say that everything we do here is important.

As a prerequisite for reading this syllabus only a basic familiarity with mathematical language and formalisms is required. In particular our prospective student will need some familiarity with first-order logic. There will also be an occasional reference to cardinalities and set theory, but a student not familiar with this can simply skip these points.

The outline of the syllabus is as follows. In Chapter 2 we introduce the basic concepts of computability theory, such as the formal notion of algorithm, recursive function, and Turing machine. We show that the various formalizations of the informal notion of algorithm all give rise to the *same* notion of computable function. This beautiful fact is one of the cornerstones of computability theory. In Chapter 3 we discuss the basic properties of the computable and computably enumerable (c.e.) sets, and venture some first steps into the uncomputable. In Chapters 4 and 5 we introduce various measures to study unsolvability: In Chapter 4 we study the arithmetical hierarchy and m-degrees, and in Chapter 5 we study relative computability and Turing degrees. In Chapter 6 we introduce the priority method, and use it to answer an important question about the Turing degrees of c.e. sets. Finally, in Chapter 7 we give some applications of computability theory. In particular we give a proof of Gödel's incompleteness theorem.

1.1 Preliminaries

Our notation is standard and follows the textbooks [18, 28]. ω is the set of natural numbers $\{0, 1, 2, 3 \ldots\}$. By *set* we usually mean a subset of ω. The Cantor space 2^ω is the set of all subsets of ω. We often identify sets with their characteristic functions: For $A \subseteq \omega$ we often identify A with the function $\chi_A : \omega \to \{0, 1\}$ defined by

$$\chi_A(n) = \begin{cases} 1 & \text{if } n \in A \\ 0 & \text{otherwise.} \end{cases}$$

We will often simply write $A(n)$ instead of $\chi_A(n)$. The complement $\omega - A$ is denoted by $\overline{A}$. By a *(total) function* we will always mean a function from ω^n to ω, for some n. We use the letters f, g, h to denote total functions. The composition of two functions f and g is denoted by $f \circ g$. A *partial function* is a function that may not be defined on some arguments. By $\varphi(n)\downarrow$ we denote that φ is defined on n, and by $\varphi(n)\uparrow$ that it is undefined. The *domain* of a partial function φ is the set $\mathrm{dom}(\varphi) = \{x : \varphi(x)\downarrow\}$, and the *range* of φ is the set $\mathrm{rng}(\varphi) = \{y : (\exists x)[\varphi(x) = y]\}$. We use the letters φ and ψ, to denote partial functions. For such partial functions, $\varphi(x) = \psi(x)$ will mean that either both values are undefined, or both are defined and equal. For notational convenience we will often abbreviate a finite number of arguments $x_1, \ldots, x_n$ by $\vec{x}$. We will make use of the λ-notation for functions: $\lambda x_1 \ldots x_n . f(x_1, \ldots, x_n)$ denotes the function mapping $\vec{x}$ to $f(\vec{x})$. The set of all finite binary strings is denoted by $2^{<\omega}$. We use σ and τ for finite strings. The length of σ is denoted by $|\sigma|$, and the concatenation of σ and τ is denoted by $\sigma\hat{}\tau$. That τ is a subsequence (also called *initial segment*) of σ is denoted by $\tau \sqsubseteq \sigma$. For a set A, $A{\restriction}n$ denotes the finite string $A(0)\hat{}A(1)\hat{}\ldots\hat{}A(n-1)$ consisting of the first n bits of A.

At the end of every chapter are the exercises for that chapter. Some exercises are marked with a * to indicate that they are more challenging.

Chapter 2

Basic concepts

2.1 Algorithms

An algorithm is a finite procedure or a finite set of rules to solve a problem in a step by step fashion. The name algorithm derives from the name of the 9th century mathematician al-Khwarizmi and his book "Al-Khwarizmi Concerning the Hindu Art of Reckoning", which was translated into Latin as "Algoritmi de numero Indorum." For example, the recipe for baking a cake is an algorithm, where the "problem" is the task of making a cake. However, the word algorithm is mostly used in more formal contexts. The procedure for doing long divisions is an example of an algorithm that everyone learns in school. Also, every computer program in principle constitutes an algorithm, since it consists of a finite set of rules determining what to do at every step.

For many centuries there did not seem a reason to formally define the notion of algorithm. It was a notion where the adage "you recognize it when you see one" applied. This became different at the turn of the 20th century when problems such as the following were addressed:

- (The "Entscheidungsproblem",) Given a logical formula (from the first-order predicate logic), decide whether it is a tautology (i.e. valid under all interpretations) or not. The problem occurs Hilbert and Ackermann [9], but its origins can be traced back to Leibniz [14].
- (Hilbert's Tenth Problem) Given a *Diophantine equation*, that is, a polynomial in several variables with integer coefficients, decide whether it has a solution in the integers. This problem comes from a famous list of problems posed by Hilbert at the end of the 19th century [8].

A positive solution to these problems would consist of an algorithm that solves them. But what if such an algorithm does not exist? To settle the above problems in the negative, i.e. show that there exists no algorithm that solves them, one first has to say precisely *what an algorithm is*. That is, we need a formal definition of the informal notion of algorithm. This is the topic of this first chapter.

There are many ways to formalize the notion of algorithm. Quite remarkably, *all of these lead to the same formal notion*! This gives us a firm basis for a mathematical theory of computation: computability theory. Below, we give two of the most famous formalizations: recursive functions and Turing computable functions. We then proceed by proving that these classes of functions coincide. As we have said, there are many other formalizations. We refer the reader to Odifreddi [18] for further examples.

Returning to the above two problems: With the help of computability theory it was indeed proven that both problems are unsolvable, i.e. that there exist no algorithms for their solution. For the Entscheidungsproblem this was proved independently by Church [2] and Turing [30] (see Theorem 7.1.6), and for Hilbert's Tenth Problem by Matijasevich [16], building on work of Davis, Putnam, and Robinson.

2.2 Recursion

The approach to computability using the notion of *recursion* that we consider in this section is the reason that computability theory is also commonly known under the name *recursion theory*. This approach was historically the first and is most famous for its application in Gödel's celebrated incompleteness results [7], cf. Section 7.1. In fact, although the statements of these results do not mention computability, Gödel invented some of the basic notions of computability theory in order to prove his results.

2.2.1 The primitive recursive functions

Recursion is a method to define new function values from previously defined function values. Consider for example the famous Fibonacci sequence

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34\ldots$$

which was discovered by Fibonacci in 1202. Every term in the sequence is obtained by taking the sum of the two previous terms: $F(n+2) = F(n+1)+F(n)$. To get the sequence started we define $F(0) = 1$ and $F(1) = 1$. This sequence occurs in many forms in nature, for example in the phyllotaxis of seeds in sunflowers (see Turing's drawing on the cover). Counting in clockwise direction we can count 21 spirals, and counting counterclockwise yields 34 spirals. In this and in many other cases (e.g. in pineapples or daisies) the numbers of these two kinds of spirals are always two consecutive Fibonacci numbers. (Cf. also Exercises 2.5.1 and 2.5.2.) In general, a function f is defined by recursion from given functions g and h if an initial value $f(0)$ is defined (using g) and for every n, $f(n+1)$ is defined (using h) in terms of previous values. The idea is that if we can compute g and h, then we can also compute f. Thus recursion is a way to define new computable functions from given ones.

Definition 2.2.1 (Primitive recursion, Dedekind [3]) A function f is defined from functions g and h by *primitive recursion* if

$$\begin{aligned} f(\vec{x}, 0) &= g(\vec{x}) \\ f(\vec{x}, n+1) &= h(f(\vec{x}, n), \vec{x}, n). \end{aligned}$$

Here $\vec{x} = x_1, \ldots, x_k$ for some k. We also allow $k = 0$, in which case $g(\vec{x})$ is to be interpreted as a constant. In this definition, $f(n+1)$ is defined in terms of the previous value $f(n)$. A more general form of recursion, where $f(n+1)$ is defined in terms of $f(0), \ldots, f(n)$ is treated in Exercise 2.5.9.

Definition 2.2.2 (Primitive recursive functions, Gödel [7]) The class of *primitive recursive functions* is the smallest class of functions

1. containing the initial functions

$$\begin{aligned} 0 \quad & \text{(the 0-ary function constant zero)} \\ S = \lambda x.x+1 \quad & \text{(successor function)} \\ \pi_n^i = \lambda x_1, \ldots, x_n.x_i \quad & 1 \leqslant i \leqslant n, \text{ (projection functions)} \end{aligned}$$

2. closed under composition, i.e. the schema that defines

$$f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_n(\vec{x}))$$

   from the given functions h and g_i.

3. closed under primitive recursion.

For example, Grassmann gave in 1861 the following recursive definitions for the usual integer addition $+$ and multiplication $\cdot$:

$$\begin{aligned} x + 0 &= x \\ x + S(y) &= S(x + y), \end{aligned}$$

$$\begin{aligned} x \cdot 0 &= 0 \\ x \cdot S(y) &= (x \cdot y) + x. \end{aligned}$$

Note that the first recursion reduces the definition of $+$ to the initial function S, and that the second reduces multiplication to addition. For a precise proof that $+$ and $\cdot$ are primitive recursive see Exercise 2.5.3.

2.2.2 The recursive functions

Note that all primitive recursive functions are total functions. We will now give an informal argument showing that it is essential for our theory to also consider partial functions.

In any formalization of the notion of algorithm, an algorithm will be defined syntactically by a finite set of symbols. In this sense every algorithmically computable function will be a finite object. Now since the theory will require us to recognize and effectively manipulate these objects, it is natural that there should also be an effective way to *list* all computable functions by a listing of their algorithms: There should be a computable function f such that $f(n)$ is a code for the n-th algorithm, computing the n-th computable function f_n. Now suppose that we would only consider total functions. Then the function $F = \lambda n.f_n(n) + 1$ would also be a computable function, the m-th in the list, say. But then

$$F(m) = f_m(m) \neq f_m(m) + 1 = F(m),$$

a contradiction. Note that the contradiction arises since we assumed that the value $f_m(m)$ is always defined. This assumption is dropped if we also include partial functions in our theory.

The above discussion suggests that the class of primitive recursive functions is not big enough to capture the intuitive notion of computable function. Hence we add a new operator to our basic tool kit: the μ-operator. For a predicate R,

$$(\mu n)\big[R(\vec{x}, n)\big]$$

denotes the least number n such that $R(\vec{x}, n)$ holds. If such n does not exist then $(\mu n)[R(\vec{x}, n)]$ is undefined. Thus the μ operator allows us to *search* for certain values. Since such a search can be unbounded, the μ-operator can take us out of the domain of total functions.

Definition 2.2.3 (Recursive functions, Kleene [11]) The class of *partial recursive functions* is the smallest class of functions

1. containing the initial functions
2. closed under composition and primitive recursion[1]
3. closed under μ-recursion, i.e. the schema that defines the partial function

$$\varphi(\vec{x}) = (\mu y)\big[(\forall z \leqslant y)[\psi(\vec{x}, z)\downarrow] \wedge \psi(\vec{x}, y) = 0\big]$$

   from a given partial function ψ.

A *(general) recursive function* is a partial recursive function that happens to be total.

Note that in the schema of μ-recursion the μ-operator searches for a smallest y such that $\psi(\vec{x}, y) = 0$, but that it cannot "jump" over undefined values of ψ because of the first clause. This corresponds with the intuition that the search for such a y is performed by discarding values z for which we can see that $\psi(\vec{x}, z) \neq 0$. Also, as we will later see, it is undecidable whether a computable function converges, so dropping the clause would give us a class which is too large. (See also Exercise 2.5.18.) In the following, we will always use the μ-operator in the appropriate sense.

Clearly, since the class of partial recursive functions contains partial functions, it strictly extends the class of primitive recursive functions. However, there are also *total* examples showing that the addition of the μ-operator allows us to define more total functions than just the primitive recursive ones. Examples will be easy to construct (cf. Exercise 2.5.13) as soon as we have developed the theory a bit more, in particular after we have treated the method of arithmetization. This method will also allow us to turn the informal discussion from the beginning of this section into a formal proof that the partial recursive functions can, but the total recursive functions cannot, be effectively listed.

2.3 Turing machines

In this section we consider a completely different formalization of the notion of algorithm, namely the one using Turing machines [30]. Instead of inductively building functions from previous ones, as in the definition of recursive function, this approach investigates directly what it means to perform a computation of a function, using an idealized device or machine. We imagine a tape of unbounded length, consisting of infinitely many cells, extending in a one-way infinite row from left to right say. Each cell either contains a 1 or a blank, i.e. nothing. We imagine further that we manipulate the symbols on the tape in a completely prescribed way, according to a set of rules or a *program*. The basic operations that we can perform while scanning a cell are to change its contents and to move either left or right. The action that we perform may depend on the contents of the cell and on the current *state* of the

[1] Here is is understood that if a function value in the composition or the recursion is undefined, the resulting function is also undefined.

program. The actions we thus perform may or may not terminate. In the former case we say that we have completed a computation, and may consider the contents of the tape to be its outcome. To model the intuition that our actions should be of limited complexity we require that there can be only finitely many states in the program. Instead of performing the computation ourselves we can also imagine a device containing a program to do it. We thus arrive at the picture of a Turing machine depicted in Figure 2.1.

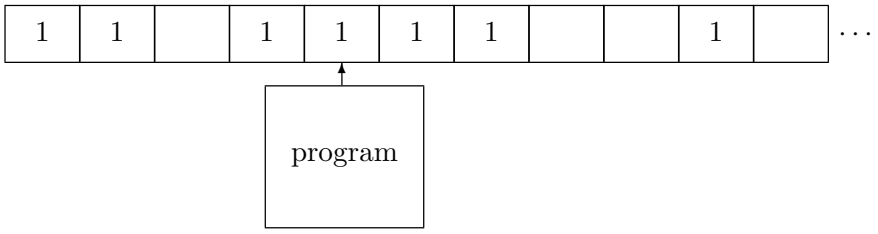


Figure 2.1: A Turing machine

Formally, the instructions in the program of a Turing machine are quintuples

$$(q_i,\, x,\, q_j,\, y,\, X),$$

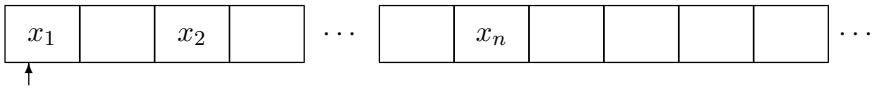
where q_i and q_j are from a finite set Q of *states*, x and y are either 1 or B (for "blank"), and $X \in \{L, R\}$. The interpretation of such an instruction is that if the machine is in state q_i and scanning a cell with the contents x, then this contents is changed to y, the next machine state is q_j, and the tape head moves one step in the direction X, where L stands for "left" and R for "right". A Turing machine program is a set of instructions saying what to do in any of the possible situations the machine may be in. Thus we formally define a *Turing machine* to be a function

$$M : Q \times \{1, B\} \to Q \times \{1, B\} \times \{L, R\}.$$

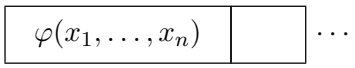
By convention, if the machine is scanning the leftmost cell and receives the instruction to move left the computation is undefined. Furthermore, the set Q contains two distinguished states: An *initial state* q_0 and a *final state* q_f. The interpretation is that the machine starts in state q_0 and halts whenever it reaches the final state q_f.

We can think of Turing machines as idealized computers, with unlimited resources such as time and memory. Notice that what is called a machine here would usually be called a program, and that the "hardware" is represented here by the tape and the reading head. A Turing machine is a finite set of instructions that can be depicted as a finite state diagram. It is working on an infinite tape that we can think of as an unbounded external memory. To model existing computers in a more detailed way one can add to the Turing machine model the boundedness of computation time and space. The resulting theory is a younger branch of computability theory called *complexity theory*. We shall not touch the many intriguing results and questions from this area here, but instead refer the reader to Odifreddi [19] for more information and references.

Definition 2.3.1 An n-ary partial function φ is *partial computable* if there is a Turing machine M such that for any $x_1, \ldots, x_n$ and any initial configuration



where by a cell with x_i we actually mean x_i+1 consecutive cells containing a 1 (this is called the *tally* representation of x_i), all other cells on the tape are blank, and the tape head is scanning the leftmost cell, then by execution the program of the machine (i.e. iteratively applying its defining function and starting in state q_0) it halts after finitely many steps in the halting state q_f with the following tape configuration



where the tape head is scanning any cell, $\varphi(x_1, \ldots, x_n)$ is again in tally representation, and we do not care about the contents of the tape cells after the blank cell following $\varphi(x_1, \ldots, x_n)$. (These cells may contain junk left over from the computation.) Furthermore, for all $x_1, \ldots, x_n$ such that $\varphi(x_1, \ldots, x_n)\uparrow$ the machine M does not halt.

A function is *computable* if it is partial computable and total. A set A is computable if its characteristic function χ_A is computable.

It should be noted that this definition is very robust and allows for many inessential variations. For example, one often sees presentations where the tape is infinite both to the left and to the right, or where the tape head is allowed to stay in position after carrying out an instruction, or where there is more than one tape, etc. Also, one could consider larger alphabets than just $\{1, B\}$. With respect to this last point: After seeing the coding techniques of Section 2.4 the reader will readily see that any finite alphabet can be coded with two symbols, so that there is no loss in confining ourselves to this alphabet.

The reader should play around a bit to see what can be done on a Turing machine. A modular approach will show that simple functions are Turing machine computable, and from these one can construct more complicated functions, much as in the way the recursive functions were defined.

Example 2.3.2 We give a program for the constant 0 function. It suffices to, whatever input is written on the tape, print a 1 (the tally representation of 0) in the leftmost cell and a blank in the second, and then halt. Since by convention the machine starts from the leftmost cell the following program achieves this:

q_0	1	q_1	1	R	move one step to the right
q_1	1	q_f	B	L	if you see a 1 replace it by a blank and halt
q_1	B	q_f	B	L	if you see a blank just halt.

Notice that we have only listed instructions that are actually used. Formally the program should also give values for the other combinations of states and cell contents, but since these are irrelevant we have left them out. □

Example 2.3.3 We give a program for the successor function. It simply scans the input to the right until it reaches a blank. This blank is replaced by a 1 and then the machine halts:

q_0	1	q_0	1	R	move to the right as long as you see ones
q_0	B	q_f	1	L	if you encounter a blank replace it by a 1 and halt.

□

Example 2.3.4 We give an example of a routine that copies the (single) input. Such routines are useful as building blocks for larger programs.

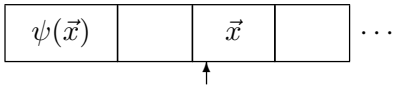
q_0	1	q_1	B	R	change initial 1 into B, move right
q_1	1	q_1	1	R	keep moving right until
q_1	B	q_2	B	R	the next blank,
q_2	1	q_2	1	R	keep moving right
q_2	B	q_3	1	L	until second blank, change this into 1, move left
q_3	1	q_3	1	L	move back
q_3	B	q_4	B	L	to previous blank
q_4	1	q_4	1	L	keep moving left
q_4	B	q_5	1	R	until first blank and change this into 1
q_5	B	q_f	B	R	if this was next to second blank halt
q_5	1	q_1	B	R	otherwise, change next 1 into B and repeat.

In the first instruction, the initial 1 (note that there always is an initial 1) is erased so that later we can recognize the beginning of the tape without falling off. (Recall the convention on page 7 about moving left when on the first tape cell.) □

The next result reveals the power of Turing machines:

Theorem 2.3.5 (Turing [30]) *Every partial recursive function is partial computable.*

Proof. The proof is by induction on the definition of the recursive function. It suffices to show that the initial functions are computable, and that composition, primitive recursion, and μ-recursion can all be simulated on a Turing machine. To facilitate the induction we prove a slightly stronger statement, namely that every partial computable function ψ can be computed in a *tidy* way, namely such that on every halting computation $\psi(\vec{x})\downarrow$ the machine halts in the configuration



with $\psi(\vec{x})$ and $\vec{x}$ in tally notation and the tape head scanning the leftmost cell of $\vec{x}$. The advantage of tidy computations is that they preserve the input and leave the tape head at a canonical place, which facilitates the merging of programs.

We leave it to the reader (Exercise 2.5.7) to write tidy programs for the initial functions. (Note that these will be much larger than the programs from Examples 2.3.2 and 2.3.3!) Note that the routine from Example 2.3.4 is a tidy machine for the projection function $\pi^1_1 = \lambda x.x$.

Below we will give informal descriptions of the cases of composition, primitive recursion, and μ-recursion, and leave it to the reader to complete the proof.

Composition: Suppose that φ is defined by $\varphi(\vec{x}) = \chi(\psi_1(\vec{x}), \ldots, \psi_n(\vec{x}))$. By induction hypothesis χ and the ψ_i can be computed by tidy machines. Given input $\vec{x}$, apply a tidy machine for ψ_1. This gives an output that allows us to immediately apply a tidy machine for ψ_2, and so on until ψ_n. (Some care has to be taken to address the possibility of computations running off the left end of the tape. This can be solved by introducing a special marker.) This gives as output $z_1, \ldots, z_n, \vec{x}$, where $z_i = \psi_i(\vec{x})$. Preserve the $\vec{x}$ by swapping it with the $z_1, \ldots, z_n$ and move the tape head to the leftmost z_1 for the final application of the tidy machine for χ. This gives $\vec{x}, \chi(z_1, \ldots, z_n), z_1, \ldots, z_n$. Finally remove the z_i's and swap the $\vec{x}$ to output $\chi(z_1, \ldots, z_n), \vec{x}$.

Primitive recursion: Suppose

$$\begin{aligned}\varphi(\vec{x},0) &= \psi(\vec{x})\\ \varphi(\vec{x},n+1) &= \chi(\varphi(\vec{x},n),\vec{x},n).\end{aligned}$$

Given input $\vec{x},n$ we use n as a counter to count down to 0. On input $\vec{x},n$ we modify the input to $n,\vec{x},m$, where m is 0 (everything in tally representation). Apply a tidy machine for ψ to obtain the configuration $n,\psi(\vec{x}),\vec{x},0$. Check if $n>0$. If $n=0$ then erase and shift the appropriate cells to output $\psi(\vec{x}),\vec{x},0$ and halt. If $n>0$ then decrease n by one tally. Move the tape head in the right position to apply a tidy machine for χ to obtain the output $n,\chi(\varphi(\vec{x},m),\vec{x},m),\varphi(\vec{x},m),\vec{x},m$. Increase this last m by one tally. Check if $n>0$. If $n=0$ then m has become equal to the original value of n, so we can erase the current n and $\varphi(\vec{x},m)$ and shift the rest to the left to output $\chi(\varphi(\vec{x},m),\vec{x},m),\vec{x},m$ and halt. If $n>0$, erase $\varphi(\vec{x},m)$ and repeat the above procedure.

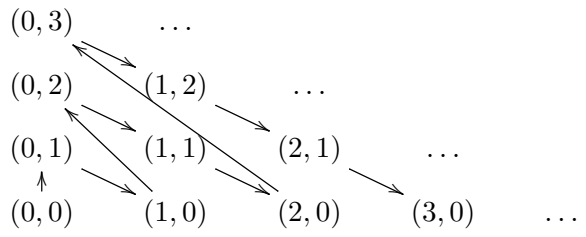
μ-Recursion: Suppose that $\varphi(\vec{x})=\mu y(\psi(\vec{x},y)=0)$. On input $\vec{x}$ extend the input to $\vec{x},n$, where initially $n=0$. Apply a tidy machine for ψ to obtain the configuration $\psi(\vec{x},n),\vec{x},n$. Test if $\psi(\vec{x},n)=0$. If yes, output $\vec{x},n$. If no, erase $\psi(\vec{x},n)$, move $\vec{x},n$ to the far left of the tape, increase n by one tally, and repeat. □

2.4 Arithmetization

Arithmetization is a method of coding a language into the language of arithmetic, so that statements from the original language are transformed into statements about the natural numbers. This method was already envisaged by Leibniz [14], but again it was first put to use by Gödel [7]. We will use arithmetization to prove a normal form theorem for partial recursive functions and to prove the converse of Theorem 2.3.5.

2.4.1 Coding functions

We start by giving a coding of the plane ω^2. We can do this by *dovetailing*:



This counting of the pairs is implemented by the *pairing function* $\lambda x,y.\langle x,y\rangle$ from ω^2 to ω defined by

$$\langle x,y\rangle = \tfrac{1}{2}\big((x+y+1)^2-(x+y+1)\big)+x = \tfrac{1}{2}\big((x+y)^2+3x+y\big).$$

Next we want a general coding mechanism for arbitrary finite sequences of numbers $x_1,\ldots,x_n$. For this we could make use of the above pairing function, but instead we choose to use a coding using the prime numbers $p_1,p_2,\ldots$ (cf. Exercise 2.5.4). We define the code number for the sequence $x_1,\ldots,x_n$ as

$$\langle x_1,\ldots,x_n\rangle = p_1^{x_1+1}\cdot p_2^{x_2+1}\cdots p_n^{x_n+1}.$$

The code thus defined is not surjective, but we do not care about that. We give the empty sequence code 1. In order to manipulate sequences we define several auxiliary functions:

- The function $exp(x, p_i)$ giving the exponent of p_i in the decomposition of x. (Cf. Exercise 2.5.5.)
- $(x)_i = exp(x, p_i) \dot{-} 1$, which gives the i-th element x_i of the sequence x for $i \geqslant 1$. ($\dot{-}$ is the truncated subtraction defined in Exercise 2.5.4.) Note that this is defined even if x is not the code of a sequence.
- The length of the sequence x can be defined as

$$lth(x) = \max\big\{n \leqslant x : \forall m \leqslant n.\ exp(x, p_m) > 0\big\}.$$

- $Seq(x) \iff x > 0 \wedge (\forall i \leqslant x)\big[exp(x, p_i) > 0 \to i \leqslant lth(x)\big]$. Every x with $Seq(x)$ is a code of a sequence.

2.4.2 The normal form theorem

We now give a first example of arithmetization, namely the coding of the partial recursive functions into arithmetic. This will yield several useful theorems about the recursive functions.

Theorem 2.4.1 (Normal Form Theorem, Kleene [11]) *There exists a primitive recursive function U and primitive recursive relations T_n, $n \geqslant 1$, such that for every n-ary partial recursive function φ there exists a number e (called a* code *of φ) such that*

$$\varphi(x_1, \ldots, x_n) = U\big(\mu y T_n(e, x_1, \ldots, x_n, y)\big).$$

Proof. The idea is to code computations by numbers, using the coding of sequences from the previous section. The predicate $T_n(e, x_1, \ldots, x_n, y)$ will then say that y is a computation of the function with code e on the arguments $x_1, \ldots, x_n$. The search $\mu y T_n(e, x_1, \ldots, x_n, y)$ will then give us one such a computation, and U will extract its outcome. Conceptually the coding is not difficult, but it is a lot of work to carry it out completely. It is a healthy exercise though, which should be done at least once.
Coding of the partial recursive functions. We inductively associate numbers with partial recursive functions, following their definitions, as follows:

- $\langle 0 \rangle$ corresponds to 0
- $\langle 1 \rangle$ to S
- $\langle 2, n, i \rangle$ to π^i_n
- $\langle 3, a_1, \ldots, a_n, b \rangle$ to $\varphi(\vec{x}) = \chi(\psi_1(\vec{x}), \ldots, \psi_n(\vec{x}))$, where $a_1, \ldots, a_n$ and b correspond to $\psi_1, \ldots, \psi_n$ and χ, respectively.
- $\langle 4, a, b \rangle$ to $\varphi(\vec{x}, y)$ defined with primitive recursion from ψ and χ, where a and b correspond to ψ and χ.
- $\langle 5, a \rangle$ to $\varphi(\vec{x}) = \mu y(\psi(\vec{x}, y) = 0)$, where a corresponds to ψ.

Computation trees. We describe computations by trees as follows. Every node in the tree will consist of one computation step. We build the tree from the leaves to the root.

- At the leaves we write computations performed by the initial functions. So these steps are of the form 0, $\varphi(x) = x + 1$, or $\varphi(x_1, \ldots, x_n) = x_i$.

- Composition: If $\varphi(\vec{x}) = \chi(\psi_1(\vec{x}), \ldots, \psi_n(\vec{x}))$ then the node $\varphi(\vec{x}) = z$ has the $n+1$ predecessors

$$\psi_1(\vec{x}) = z_1, \ldots, \psi_n(\vec{x}) = z_n,\ \chi(z_1, \ldots, z_n) = z.$$

- Primitive recursion: If $\varphi(\vec{x}, y)$ is defined with primitive recursion from ψ and χ, then $\varphi(\vec{x}, 0) = z$ has predecessor $\psi(\vec{x}) = z$ and $\varphi(\vec{x}, y+1) = z$ has the two predecessors $\varphi(\vec{x}, y) = z_1$ and $\chi(z_1, \vec{x}, y) = z$.

- μ-recursion: If $\varphi(\vec{x}) = \mu y(\psi(\vec{x}, y) = 0)$ then the computation step $\varphi(\vec{x}) = z$ has the predecessors

$$\psi(\vec{x}, 0) = t_0, \ldots, \psi(\vec{x}, z-1) = t_{z-1},\ \psi(\vec{x}, z) = 0$$

  where the t_i are all different from 0.

Coding of computation trees. The nodes of a computation tree are of the form $\varphi(\vec{x}) = z$. We code these by the numbers $\langle e, \langle x_1, \ldots, x_n\rangle, z\rangle$ where e is a code for φ. We proceed by assigning codes to trees in an inductive way. Each tree T consists of a vertex v with a finite number (possibly zero) of predecessor trees T_i, $i = 1 \ldots m$. If the subtrees T_i have been assigned the codes $\widehat{T_i}$, we give T the code $\widehat{T} = \langle v, \widehat{T_1}, \ldots, \widehat{T_m}\rangle$.
The property of being a computation tree is primitive recursive. We show that there is a primitive recursive predicate $T(y)$ saying that y is a code of a computation tree. We use the decoding function $(x)_i$ from Section 2.4.1. For notational convenience, we write $(x)_{i,j,k}$ instead of $(((x)_i)_j)_k$. Suppose that y is a code of a computation tree. Then $y = \langle v, \widehat{T_1}, \ldots, \widehat{T_m}\rangle$, hence

$$\begin{aligned}
(y)_1 &= \langle e, \langle x_1, \ldots, x_n\rangle, z\rangle \\
(y)_{1,1} &= e \\
(y)_{1,2} &= \langle x_1, \ldots, x_n\rangle \\
(y)_{1,3} &= z \\
(y)_{i+1} &= \widehat{T_i} \\
(y)_{i+1,1} &= \text{code of the vertex of } \widehat{T_i}.
\end{aligned}$$

We will express $T(y)$ in terms of a number of smaller building blocks. First we let

$$A(y) \iff \mathit{Seq}(y) \wedge \mathit{Seq}((y)_1) \wedge \mathit{lth}((y)_1) = 3 \wedge \mathit{Seq}((y)_{1,1}) \wedge \mathit{Seq}((y)_{1,2}).$$

Next we define primitive recursive predicates B, C, D, and E corresponding to the cases of initial functions, composition, primitive recursion, and μ-recursion. We only write out the case of B and leave the others as Exercise 2.5.8. For the initial functions, there are three possibilities for $v = (y)_1$:

$$\begin{gathered}
\langle\langle 0\rangle, 0\rangle \\
\langle\langle 1\rangle, \langle x\rangle, x+1\rangle \\
\langle\langle 2, n, i\rangle, \langle x_1, \ldots, x_n\rangle, x_i\rangle.
\end{gathered}$$

We thus let

$$\begin{aligned}
B(y) \iff\ & \mathit{lth}(y) = 1 \wedge \\
& \Big[\big[(y)_{1,1} = \langle 0\rangle \wedge (y)_{1,2} = 0\big] \vee \\
& \big[(y)_{1,1} = \langle 1\rangle \wedge \mathit{lth}((y)_{1,2}) = 1 \wedge (y)_{1,3} = (y)_{1,2,1} + 1\big] \vee \\
& \big[\mathit{lth}((y)_{1,1}) = 3 \wedge (y)_{1,1,1} = 2 \wedge (y)_{1,1,2} = \mathit{lth}((y)_{1,2}) \wedge \\
& \ 1 \leqslant (y)_{1,1,3} \leqslant (y)_{1,1,2} \wedge (y)_{1,3} = ((y)_{1,2})_{(y)_{1,1,3}}\big]\Big].
\end{aligned}$$

Similarly, we can define $C(y)$ to be the primitive recursive predicate expressing that y is a code of a tree of which the vertex v is a computation defined by composing the computations in the predecessor trees. So $lth((y)_{1,1}) \geqslant 3$, $(y)_{1,1,1} = 3$, and so forth. We can let the predicate $D(y)$ expresses the same for the case of primitive recursion, and the predicate $E(y)$ for the case of μ-recursion (cf. Exercise 2.5.8).

Having thus treated all cases we can define inductively

$$\begin{aligned} T(y) \iff\ & A(y) \wedge \big[B(y) \vee C(y) \vee D(y) \vee E(y)\big] \wedge \\ & \big[lth(y) > 1 \to (\forall i)_{2 \leqslant i \leqslant lth(y)} T((y)_i)\big]. \end{aligned}$$

The predicate T is primitive recursive because it is defined in terms of primitive recursive functions applied to previous values only. (Note that always $(y)_i < y$.) That is, we have used the course-of-values recursion scheme of Exercise 2.5.9.

The definition of T_n and U. Finally, we define for each $n \geqslant 1$

$$T_n(e, x_1, \ldots, x_n, y) \iff T(y) \wedge (y)_{1,1} = e \wedge (y)_{1,2} = \langle x_1, \ldots, x_n \rangle$$

and

$$U(y) = (y)_{1,3}.$$

Obviously, T_n and U are primitive recursive. Now let φ be any n-ary partial recursive function. Then φ has a code, say e. For every $x_1, \ldots, x_n$, $\varphi(x_1, \ldots, x_n)\!\downarrow$ if and only if there is a computation tree for $\varphi(x_1, \ldots, x_n)$. The value $\varphi(x_1, \ldots, x_n)$ is extracted by U. So we have

$$\varphi(x_1, \ldots, x_n) = U\big(\mu y T_n(e, x_1, \ldots, x_n, y)\big).$$

for every $x_1, \ldots, x_n$. □

Theorem 2.4.1 shows in particular that every partial recursive function can be defined with *one* application of the μ-operator only.

2.4.3 The basic equivalence and Church's thesis

Theorem 2.4.2 (Turing [30]) *Every partial computable function is partial recursive.*

Proof. Again this is proved by arithmetization. Instead of coding the computations of partial recursive functions, as in the proof of Theorem 2.4.1, we now have to code computations of Turing machines. Again, this can be done using primitive recursive coding functions. After seeing the proof of Theorem 2.4.1, the reader should have an idea of how to proceed. As before, a primitive recursive predicate T_n and an extraction function U can be defined, with a similar meaning. E.g. $T_n(e, x_1, \ldots, x_n, y)$ will say that y codes a computation of the Turing machine with code e on inputs $x_1, \ldots, x_n$. We leave it to the reader to carry this out. □

Combining Theorems 2.3.5 and 2.4.2 we obtain the following basic equivalence:

Theorem 2.4.3 *A function is partial computable if and only if it is partial recursive.*

After seeing this equivalence, and emphasizing again that there are many other equivalent formalizations, we can from now on be more informal in our descriptions of algorithms, since the precise formalization is usually irrelevant. Other formalizations

of the informal notion of algorithm include (cf. Odifreddi [18] for some of these) *finite definability*, *representability in certain formal systems* (cf. Theorem 7.1.4), *being computable by a while-program*, *λ-definability*, *computability on register machines*, and *formal grammars*. All of these can be proved to be equivalent by the method of arithmetization. These equivalences are usually taken as indication that the informal notion of algorithm is thus rightly formalized by any of these formalisms.

Church's Thesis *Every algorithmically computable (in the informal sense) function is computable.*

Note that Church's Thesis is not a theorem, since it is not a formal statement, and hence it cannot be formally proved. Rather, it is a statement expressing a certain confidence in the right set-up of our theory.

Theorem 2.4.3 allows us to use the words recursive and (Turing) computable interchangeably from now on. Following modern usage we will mostly talk about computable functions.

Definition 2.4.4 Let U and T_n be as in the Normal Form Theorem 2.4.1.

1. φ_e^n, also denoted by $\{e\}^n$,is the *e-th partial computable n-ary function*:

$$\varphi_e^n(x_1, \ldots, x_n) = U\big(\mu y\, T_n(e, x_1, \ldots, x_n, y)\big).$$

   For the arity $n = 1$ we usually delete the n and simply write φ_e and $\{e\}$. We also often suppress n altogether if the arity of the functions is clear from the context.

2. $\varphi_{e,s}^n$, also denoted by $\{e\}_s^n$, is the *s-step finite approximation* of φ_e^n:

$$\varphi_{e,s}^n(x_1, \ldots, x_n) = \begin{cases} \varphi_e^n(x_1, \ldots, x_n) & \text{if } (\exists y < s)\big[T_n(e, x_1, \ldots, x_n, y)\big] \\ \uparrow & \text{otherwise.} \end{cases}$$

Recall the informal discussion at the beginning of Section 2.2.2. We can now make the statements made there more precise.

Theorem 2.4.5 (The Enumeration Theorem) *The partial computable functions can be effectively enumerated in the following sense:*

1. *For every n and e, φ_e^n is a partial computable n-ary function.*
2. *For every partial computable n-ary function φ there is a code e such that $\varphi = \varphi_e^n$.*
3. *For every n there exists a* universal *partial computable function, that is, a partial computable $n+1$-ary function φ such that for all e and $\vec{x}$,*

$$\varphi(e, \vec{x}) = \varphi_e^n(\vec{x}).$$

Proof. This follows from the Normal Form Theorem 2.4.1: We can simply define $\varphi(e, \vec{x}) = U\big(\mu y\, T_n(e, \vec{x}, y)\big)$. □

The Enumeration Theorem reveals an aspect that is very important to our subject, namely the double role that numbers play. On the one hand, a number can act as an input for a function, while on the other hand a number can also be a code of a function. Thus the levels of functions and function inputs are blurred, which opens up the possibility of self-reference: A number can be applied to itself. This will have profound consequences later on.

Lemma 2.4.6 (Padding Lemma) *Given a code e of a partial computable function φ, we can effectively enumerate infinitely many codes for φ.*

Proof. We can add "dummy rules" to the program described by e that do not change the function behaviour but that increase the size of the code. □

The next theorem provides a way to shift a number of function arguments into the program.

Theorem 2.4.7 (The S-m-n Theorem) *For every m and n there is an $m+1$-ary primitive recursive injective function S_n^m such that*

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

Proof. We use the arithmetization of the partial recursive functions from the proof of Theorem 2.4.1. Note that there exists a primitive recursive function f that, given c, outputs a code of $\lambda y_1 \ldots y_n.c$. (Cf. Exercise 2.5.4). Now we can simply define

$$S_n^m(e, x_1, \ldots, x_m) = \big\langle 3, f(x_1), \ldots, f(x_m), \langle 2, n, 1\rangle, \ldots, \langle 2, n, n\rangle, e\big\rangle.$$

From the coding in the proof of Theorem 2.4.1 we see that this is a code of the composition of the function $\lambda x_1 \ldots x_m y_1 \ldots y_n.\{e\}(x_1, \ldots, x_m, y_1, \ldots, y_n)$ with the functions $\lambda y_1 \ldots y_n.x_i$, for $i = 1 \ldots m$, and $\pi_n^j = \lambda y_1 \ldots y_n.y_j$, for $j = 1 \ldots n$. Clearly, this composition is a function of $y_1, \ldots, y_n$ and it computes $\{e\}(x_1, \ldots, x_m, y_1, \ldots, y_n)$. □

2.4.4 Canonical coding of finite sets

It will also often be useful to refer to finite sets in a canonical way. To this end we define a canonical coding of the finite sets as follows. If $A = \{x_1, \ldots, x_n\}$ we give A the code $e = 2^{x_1} + \ldots + 2^{x_n}$. We write $A = D_e$ and call e the *canonical code* of A. By convention we let $D_0 = \emptyset$.

Note that the coding of finite sets is different from the coding of finite sequences from Section 2.4.1. The main difference is that in a sequence the order and the multiplicity of the elements matter, and in a set they do not. Also note that if we write a canonical code e as a binary number then we can interpret it as the characteristic string of D_e. Thus our coding of finite sets is consistent with our convention to identify sets with characteristic strings, see Section 1.1.

2.5 Exercises

Exercise 2.5.1 Fibonacci discovered the sequence named after him by considering the following problem. Suppose that a pair of newborn rabbits takes one month to mature, and that every mature pair of rabbits every month produces another pair. Starting with one pair of newborn rabbits, how many pairs are there after n months?

Exercise 2.5.2 Let $F(n)$ be the n-th Fibonacci number. Let

$$G(n) = \left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}.$$

1. Express $G(n)$ in terms of the solutions to the equation $x^2 = x + 1$.

2. Use item 1 to show that $\frac{\sqrt{5}}{5}G(n) = F(n)$.

Exercise 2.5.3 Show precisely, using Definition 2.2.2, that addition $+$ and multiplication $\cdot$ are primitive recursive.

Exercise 2.5.4 Show that the following functions are all primitive recursive:

1. Constant functions: For any arity n and any constant $c \in \omega$, the function $\lambda x_1 \dots x_n.c$.
2. The signum function
$$sg(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0. \end{cases}$$
3. Case distinction:
$$cases(x, y, z) = \begin{cases} x & \text{if } z = 0, \\ y & \text{if } z > 0. \end{cases}$$
4. Truncated subtraction: $x \dot{-} y$, which is 0 if $y \geqslant x$ and $x - y$ otherwise. Hint: First define the predecessor function
$$\begin{aligned} pd(0) &= 0 \\ pd(x+1) &= x. \end{aligned}$$
5. The characteristic function $\chi_=$ of the equality relation.
6. Bounded sums: Given a primitive recursive function f, define $\sum_{y \leqslant z} f(\vec{x}, y)$.
7. Bounded products: Given f primitive recursive, define $\prod_{y \leqslant z} f(\vec{x}, y)$.
8. Bounded search: Given a primitive recursive function f, define the function $\mu y \leqslant z\big(f(\vec{x}, y) = 0\big)$ that returns the smallest $y \leqslant z$ such that $f(\vec{x}, y) = 0$, and that returns 0 if such y does not exist.
9. Bounded quantification: Given a primitive recursive relation R, the relations $P(x) = \exists y \leqslant x\ R(y)$ and $Q(x) = \forall y \leqslant x\ R(y)$ are also primitive recursive.
10. Division: The relation $x|y$, x divides y.
11. Prime numbers: The function $n \mapsto p_n$, where p_n is the n-th prime number, starting with $p_1 = 2$.

Exercise 2.5.5 Show that the function $exp(y, k)$ from page 11 giving the exponent of k in the decomposition of y is primitive recursive.

Exercise 2.5.6 (Concatenation of strings.) Show that there is a primitive recursive operation F such that for all sequences $\sigma = \langle x_1, \dots, x_m \rangle$ and $\tau = \langle y_1, \dots, y_n \rangle$,
$$F(\sigma, \tau) = \sigma^\frown\tau = \langle x_1, \dots, x_m, y_1, \dots, y_n \rangle.$$

Exercise 2.5.7 Complete the proof of Theorem 2.3.5 by filling in the details. In particular:

1. Give a tidy program for the successor function S.
2. Give a tidy program for the projection function π^i_n.
3. Given tidy programs for the functions χ, ψ_1, and ψ_2, give a tidy program for $\lambda x.\chi(\psi_1(x), \psi_2(x))$.
4. Give a tidy program for $\varphi(x, y)$ defined with primitive recursion from the functions ψ and χ, given tidy programs for the latter two functions.
5. Given a tidy program for ψ, give a tidy program for $\varphi(x) = \mu y(\psi(x, y) = 0)$.

Exercise 2.5.8 Complete the proof of Theorem 2.4.1 by filling in the definitions of the primitive recursive predicates C, D, and E on page 12.

Exercise 2.5.9 The course-of-values scheme of recursion is

$$\begin{aligned} f(\vec{x}, 0) &= g(\vec{x}) \\ f(\vec{x}, n+1) &= h\big(\langle f(\vec{x}, 0), \ldots, f(\vec{x}, n)\rangle, \vec{x}, n\big). \end{aligned}$$

Show, using the coding of sequences, that if g and h are primitive recursive, then so is f.

Exercise 2.5.10 (Simultaneous recursion) Let f_1 and f_2 be defined as follows.

$$\begin{aligned} f_1(0) &= g_1(0) & f_2(0) &= g_2(0) \\ f_1(n+1) &= h_1\big(f_1(n), f_2(n), n\big) & f_2(n+1) &= h_2\big(f_1(n), f_2(n), n\big). \end{aligned}$$

Show that if g_1, g_2, h_1, h_2 are all are primitive recursive, then so are f_1 and f_2. (Hint: Use coding of pairs to make one function out of f_1 and f_2.)

Exercise 2.5.11 (Péter) Show that there is a computable function that enumerates all primitive recursive functions. That is, construct a computable function $\lambda e, x.f(e, x)$ such that for every e the function $\lambda x.f(e, x)$ is primitive recursive, and conversely every primitive recursive function is equal to $\lambda x.f(e, x)$ for some e.

Exercise 2.5.12 Show that there does not exist a universal primitive recursive function, that is, there is no primitive recursive function $f(e, x)$ that enumerates all primitive recursive functions as in Exercise 2.5.11.

Exercise 2.5.13 Show that there is a (total) computable function that is not primitive recursive. (Hint: Use Exercise 2.5.11 and the discussion at the beginning of Section 2.2.2.)

Exercise 2.5.14 The busy beaver function $bb(n)$ is defined as follows. Given n, consider all Turing machines with $\leqslant n$ states that are defined on input 0. Define $bb(n)$ to be the maximum of all their outputs. Show that bb is not computable.

Exercise 2.5.15 Let g be a partial computable function, and let R be a computable predicate. Show that the function

$$\psi(x) = \begin{cases} g(x) & \text{if } \exists y\ R(x, y), \\ \uparrow & \text{otherwise.} \end{cases}$$

is partial computable. (This form of definition will be used throughout this syllabus.)

Exercise 2.5.16 Prove or disprove: If f and g are not computable then also the function $\lambda n.f(n) + g(n)$ is not computable.

Exercise 2.5.17 Show that there is a computable (even primitive recursive) function f such that for all n and x,

$$\{f(n)\}(x) = \begin{cases} 0 & \text{if } x \in D_n, \\ \uparrow & \text{otherwise.} \end{cases}$$

Exercise 2.5.18 Assume the existence of a noncomputable c.e. set A (an example of which will be provided in the next chapter). Define

$$\psi(x, y) = 0 \iff (y = 0 \wedge x \in A) \vee y = 1$$

and let $f(x) = (\mu y)\big[\psi(x, y) = 0\big]$. Show that

1. ψ is partial computable.
2. $f(x) = 0 \iff x \in A$.
3. f is not partial computable.

This exercise shows that we cannot simplify the scheme of μ-recursion in Definition 2.2.3 to $\varphi(x) = (\mu y)\big[\psi(x, y) = 0\big]$.

Chapter 3

Computable and computably enumerable sets

3.1 Diagonalization

The method of diagonalization was invented by Cantor to prove that the set of real numbers (or equivalently, the Cantor space 2^ω) is uncountable. The method is of fundamental importance to all of mathematical logic. A diagonal argument often runs as follows: Given a (countable) list of objects A_i, $i \in \omega$, we want to construct a certain object A not in the list. This object A is constructed in stages, where at stage i it is ensured that $A \neq A_i$. In Cantor's example the A_i are a list of elements of 2^ω, and at stage i the i-th element of A is defined to be $1 - A_i(i)$, so as to make A different from A_i. The set A is called a *diagonal set* because it is defined using the diagonal of the infinite matrix $\{A_i(j)\}_{i,j\in\omega}$ The argument shows that no countable list can include all elements of 2^ω.

From Theorem 2.4.1 we see in particular that the set of computable functions is countable since they can be coded with a countable set of codes. So from the uncountability of 2^ω we immediately infer that *there are uncountably many noncomputable functions*. The informal argument showing the need for partial functions at the beginning of Section 2.2.2 actually constituted our first example of the diagonalization method. The function $F = \lambda n.f_n(n)+1$ defined there is another example of an object obtained by diagonalization. Notice that diagonal objects like F have a self-referential flavour since the n here simultaneously refers to the object level (namely its role as the argument) and the meta-level (namely the position of the n-th object in the list). Recall that we already encountered self-reference on page 14 when we discussed the double role of numbers in computability theory. In this chapter we will develop the basic theory of the computable and the computably enumerable sets. One of our first results (Theorem 3.3.1) will exhibit both kinds of self-reference that we have just seen.

3.2 Computably enumerable sets

A computably enumerable set is a set of which the elements can be effectively listed:

Definition 3.2.1 A set A is *computably enumerable (c.e.)* if there is a computable function f such that $A = \mathrm{rng}(f)$, where $\mathrm{rng}(f) = \{y : (\exists x)[f(x) = y]\}$ is the range of f. For reasons of uniformity we also want to call the empty set $\emptyset$ computably enumerable.

C.e. sets are the effective analogue of Cantor's countable sets. They can also be seen as the analogue for sets of the notion of partial computable function. (Just as the notion of computable set is the analogue of the notion of computable function.) Part of the interest in c.e. sets is their abundance in mathematics and computer science. A few important examples of c.e. sets are:

- The set of all theorems (coded as natural numbers) of any formal system with a computable set of axioms.
- The set of Diophantine equations that have an integer solution.
- The set of words that can be produced with a given formal grammar (as in formal language theory).

Many other examples come from computer science. The most famous one is the following:

Definition 3.2.2 The *halting problem* is the set

$$H = \big\{ \langle x, y \rangle : \varphi_x(y)\!\downarrow \big\}.$$

The *diagonal halting problem* is the set

$$K = \big\{ x : \varphi_x(x)\!\downarrow \big\}.$$

The halting problem is the set representing the problem of deciding whether an arbitrary computation will yield an output, given a code of the function and the input. We check that the sets H and K are indeed computably enumerable: Suppose that $m \in H$ is a fixed element. Then H is the range of the computable function

$$f(n) = \begin{cases} \langle (n)_0, (n)_1 \rangle & \text{if } \varphi_{(n)_0,(n)_2}((n)_1)\!\downarrow \\ m & \text{otherwise.} \end{cases}$$

Thus, f on input $n = \langle x, y, s \rangle$ checks whether the computation $\varphi_x(y)$ converges in s steps, and if so it outputs $\langle x, y \rangle$. We use m to make f total. Since for every convergent computation $\varphi_x(y)$ there is an s such that $\varphi_{x,s}(y)\!\downarrow$, every such pair $\langle x, y \rangle$ will appear in the range of f.[1] That the set K is c.e. follows in a similar way.

Before we continue our discussion we make a number of general observations. First we give a number of equivalent ways to define c.e. sets:

Theorem 3.2.3 *For any set A the following are equivalent:*

(i) *A is c.e.*

(ii) *$A = \mathrm{rng}(\varphi)$ for some partial computable function φ.*

(iii) *$A = \mathrm{dom}(\varphi) = \big\{ x : \varphi(x)\!\downarrow \big\}$ for some partial computable function φ.*

(iv) *$A = \big\{ x : (\exists y)[R(x, y)] \big\}$ for some computable binary predicate R.*

[1] Note that f is total: Even if the input n is not a sequence number or not one of length 3, $f(n)$ is defined because the function $(x)_i$ is defined for all x and i, cf. Exercise 2.5.5.

Proof. (i)$\Longrightarrow$(ii). Clearly, if A is the range of a total computable function it is also the range of a partial one.
(ii)$\Longrightarrow$(iii). Suppose that $A = \mathrm{rng}(\varphi_e)$. Define the partial computable function ψ by

$$\psi(y) = \begin{cases} 0 & \text{if } \exists x \exists s\ \varphi_{e,s}(x) = y, \\ \uparrow & \text{otherwise.} \end{cases}$$

Then $\mathrm{rng}(\varphi) = \mathrm{dom}(\psi)$. Note that this form of definition is allowed by Exercise 2.5.15.
(iii)$\Longrightarrow$(iv). It suffices to note that $\mathrm{dom}(\varphi_e) = \{x : (\exists s)[\varphi_{e,s}(x)\downarrow]\}$ has the required form.
(iv)$\Longrightarrow$(i). Let A be as in the statement (iv) and suppose that $A \neq \emptyset$, say $a \in A$. Define

$$f(n) = \begin{cases} (n)_0 & \text{if } R((n)_0, (n)_1) \\ a & \text{otherwise.} \end{cases}$$

Then f is total and $A = \mathrm{rng}(f)$. □

Theorem 3.2.3 shows that we can alternatively define the c.e. sets as the domains of partial computable functions. Thus we can assign codes to c.e. sets in a canonical way, as we did for the partial computable functions:

Definition 3.2.4 We define

$$W_e = \mathrm{dom}(\varphi_e)$$

and call this the e-th c.e. set. The number e is called a *c.e. code*, or simply a *code* of W_e. Similarly we define

$$W_{e,s} = \mathrm{dom}(\varphi_{e,s}).$$

Notice that the sets $W_{e,s}$ are finite approximations to W_e and that we have $W_e = \bigcup_{s\in\omega} W_{e,s}$. Also note that by the definition of $\varphi_{e,s}$ there can be at most one element in $W_{e,s+1} - W_{e,s}$ for every s.[2] We will use these facts tacitly from now on.

Recall the definition of the halting problem $H = \{\langle x, y\rangle : \varphi_x(y)\downarrow\}$. Note that for every $\langle x, y\rangle$ we have $H(\langle x, y\rangle) = W_x(y)$. In this sense H codes all the information from all other c.e. sets. For this reason it is also called *universal*.

By definition the c.e. sets are the range of computable functions. We can also characterize the computable sets as the range of functions:

Proposition 3.2.5 *The following are equivalent:*

(i) *A is computable.*

(ii) *$A = \emptyset$ or $A = \mathrm{rng}(f)$ for some nondecreasing computable function f.*

Proof. Suppose that A is computable and nonempty. Define f by

$$f(0) = \mu n.\ n \in A$$
$$f(n+1) = \begin{cases} n+1 & \text{if } n+1 \in A, \\ f(n) & \text{otherwise.} \end{cases}$$

[2]Note that s is a bound on the whole computation of $\{e\}$, including the input, even for functions such as the constant functions that do not read the input.

Then f is computable and has range A.

Conversely, if f is computable and nondecreasing then there are two cases.
Case 1: $\mathrm{rng}(f)$ is finite. Then clearly $\mathrm{rng}(f)$ is computable.
Case 2: $\mathrm{rng}(f)$ is infinite. Then $n \in \mathrm{rng}(f)$ if and only if $n = f(m)$, where m is the smallest number such that $f(m) \geqslant n$. Such m always exists because the range of f is infinite. So also in this case $\mathrm{rng}(f)$ is computable. □

The notion of membership of an c.e. set A is asymmetric: If $n \in A$ this can be verified in finitely many steps, but if $n \notin A$ one may never find this out. The following elementary result due to Post shows that indeed c.e. sets are in a sense "half computable."

Proposition 3.2.6 *A set A is computable if and only if both A and its complement $\overline{A}$ are c.e.*

Proof. Informally: To decide whether $x \in A$, enumerate A and $\overline{A}$ until x appears in one of them. For a formal proof see Exercise 3.8.5. □

3.3 Undecidable sets

We have called a set A computable if its characteristic function χ_A is computable (Definition 2.3.1). A set A for which we are interested to know for every n whether $n \in A$ or $n \notin A$ is also called a *decision problem*, depending on the context. A decision problem A that is computable is usually called *decidable*. Note that the halting problem H is a decision problem.

Theorem 3.3.1 (Undecidability of the halting problem, Turing [30]) *H is undecidable.*

Proof. If H would be decidable then so would K since $x \in K \Leftrightarrow \langle x, x\rangle \in H$. So it suffices to show that K is undecidable. Suppose for a contradiction that K is decidable. Then also its complement $\overline{K}$ is decidable, and in particular c.e. But then $\overline{K} = W_e$ for some code e. Now

$$e \in K \iff \varphi_e(e)\!\downarrow \iff e \in W_e \iff e \in \overline{K},$$

a contradiction. □

Theorem 3.3.1 shows that *there are c.e. sets that are not computable*. The proof that K is undecidable is yet another example of the diagonalization method.

A natural property for a set of codes is to require that it is closed under functional equivalence of codes:

Definition 3.3.2 A set A is called an *index set* if

$$(\forall d, e)[d \in A \wedge \varphi_e = \varphi_d \to e \in A].$$

All sets of codes that are defined by properties of the functions rather than their codes are index sets. For example all of the following are index sets:

- $\mathrm{Fin} = \{e : W_e \text{ is finite}\}$.
- $\mathrm{Tot} = \{e : \varphi_e \text{ is total}\}$.

- $\mathrm{Comp} = \{e : W_e \text{ is computable}\}$.

All of these examples are important, and we will encounter them again in Chapter 4.

After seeing that K is undecidable it will not come as a surprise that Fin, Tot, and Comp are also undecidable, since the task to decide whether a code e is a member of any of these sets seems harder than just deciding whether it converges on a given input. But in fact, the undecidability of these sets follows from a very general result that says that essentially *all* index sets are undecidable:

Theorem 3.3.3 (Rice's Theorem) *Let A be an index set such that $A \neq \emptyset$ and $A \neq \omega$. Then A is undecidable.*

Proof. Let A be as in the statement of the theorem. Suppose that e is a code of the everywhere undefined function, and suppose that $e \in A$. Let $d \in \overline{A}$ and let f be a computable function such that

$$\varphi_{f(x)} = \begin{cases} \varphi_d & \text{if } x \in K, \\ \uparrow & \text{otherwise.} \end{cases}$$

Then

$$\begin{array}{rcccl} x \in K & \Longrightarrow & \varphi_{f(x)} = \varphi_d & \Longrightarrow & f(x) \notin A, \\ x \notin K & \Longrightarrow & \varphi_{f(x)} = \varphi_e & \Longrightarrow & f(x) \in A. \end{array}$$

Now if A would be decidable then so would K, contradicting Theorem 3.3.1. The case $e \notin A$ is completely symmetric, now choosing $d \in A$. □

By Rice's Theorem the only decidable examples of index sets are $\emptyset$ and ω. So we see that undecidability is indeed ubiquitous in computer science.

By Proposition 3.2.6 a set A is computable if and only if both A and $\overline{A}$ are c.e. We now consider disjoint sets of pairs of c.e. sets in general:

Definition 3.3.4 A disjoint pair of c.e. sets A and B is called *computably separable* if there is a computable set C such that $A \subseteq C$ and $B \subseteq \overline{C}$. A and B are called *computably inseparable* if they are not computably separable.

Note that the two halves of a computably inseparable pair are necessarily noncomputable. So the next theorem is a strong form of the existence of noncomputable c.e. sets.

Theorem 3.3.5 *There exists a pair of computably inseparable c.e. sets.*

Proof. Define

$$\begin{aligned} A &= \{x : \varphi_x(x)\!\downarrow = 0\}, \\ B &= \{x : \varphi_x(x)\!\downarrow = 1\}. \end{aligned}$$

Suppose that there is a computable set C separating A and B. Let e be a code of the characteristic function of C: $\varphi_e = \chi_C$. Then we obtain a contradiction when we wonder whether $e \in C$:

$$\begin{array}{rcccccl} e \in C & \Longrightarrow & \varphi_e(e) = 1 & \Longrightarrow & e \in B & \Longrightarrow & e \notin C, \\ e \notin C & \Longrightarrow & \varphi_e(e) = 0 & \Longrightarrow & e \in A & \Longrightarrow & e \in C. \end{array}$$ □

Computably inseparable pairs actually do occur in "nature". E.g. it can be shown that for formal systems $\mathcal{F}$ that are strong enough (at least as expressive as arithmetic) the sets of provably true and provably false formulas are computably inseparable. (The proof consists of formalizing the above argument in $\mathcal{F}$.)

3.4 Uniformity

In Theorem 3.2.3 we proved that various ways of defining c.e. sets are equivalent. But in fact some of these equivalences hold in a stronger form than was actually stated, as can be seen by looking at the proof. Consider for example the direction (ii)$\Longrightarrow$(iii). The function ψ there was defined in a computable way from the given function φ, which means that there is a computable function h such that, if e is a given code for φ, $h(e)$ is a code of ψ. We will say that the implication (ii)$\Longrightarrow$(iii) holds *uniformly*, or *uniform in codes*. In fact one can show that the equivalences of (ii), (iii), and (iv) are all uniform in codes (cf. Exercise 3.8.1).

With respect to the implication (iv)$\Longrightarrow$(i), note that in the proof we distinguished between the cases $A = \emptyset$ and $A \neq \emptyset$. Now the set

$$\{e : W_e = \emptyset\}$$

is an index set, and hence by Theorem 3.3.3 undecidable. So we cannot effectively discriminate between the two different cases. However we have uniformity in the following form:

Proposition 3.4.1 *There is a computable function f such that for every e,*

$$W_e \neq \emptyset \Longrightarrow \varphi_{f(e)} \text{ total } \wedge \operatorname{rng}(\varphi_{f(e)}) = W_e.$$

Proof. Let d be a code such that

$$\varphi_d(e,s) = \begin{cases} m & \text{if } m \in W_{e,s+1} - W_{e,s}, \\ a & \text{if } W_{e,s+1} - W_{e,s} = \emptyset \wedge a \in W_{e,u}, \text{ where } u = \mu t.W_{e,t} \neq \emptyset, \\ \uparrow & \text{otherwise.} \end{cases}$$

Define $f(e) = S^1_1(d,e)$. Then f is total since S^1_1 is, and when W_e is nonempty, $\varphi_{f(e)}$ is total with range W_e. □

As an example of a result that does not hold uniformly we consider the following. Note that there are two ways to present a finite set: We can present it by its canonical code as D_n for some n (see Section 2.4.4), i.e. provide a complete list of all its elements, or we can present it by a c.e.-code as W_e for some e, i.e. provide only an enumeration of its elements. Now we have:

Proposition 3.4.2 *We can go effectively from canonical codes to c.e. codes, but not vice versa.*

Proof. By Exercise 2.5.17 there is a computable function f such that $W_{f(n)} = D_n$ for all n, so we can pass effectively from canonical codes to c.e. codes.

Now suppose that we could effectively go from c.e. codes to canonical codes, i.e. suppose that there is a partial computable function ψ such that

$$W_e \text{ finite } \Longrightarrow \psi(e)\!\downarrow \wedge\, W_e = D_{\psi(e)}.$$

Define a computable function g such that

$$W_{g(e)} = \begin{cases} \{e\} & \text{if } e \in K, \\ \emptyset & \text{otherwise.} \end{cases}$$

Then $W_{g(e)}$ is always finite, so $\psi(g(e))\downarrow$ for every e. But then we have

$$e \in K \iff W_{g(e)} \neq \emptyset \iff D_{\psi(g(e))} \neq \emptyset.$$

But $D_n = \emptyset$ is decidable, so it follows that K is decidable, a contradiction. Hence ψ does not exist. □

In Proposition 3.7.2 we will use the recursion theorem to show that Proposition 3.2.5 also does not hold uniformly.

3.5 Many-one reducibility

When considering decision problems, or any other kind of problem for that matter, it is useful to have a means of reducing one problem to another. If one can solve a problem A, and problem B reduces to A, then we can also solve B. This also works the other way round: If we know that B is unsolvable, then so must be A. In the following we will encounter several notions of reduction. The simplest notion of reduction between decision problems is maybe the one where questions of the form "$n \in A$?" are effectively transformed into questions of the form "$f(n) \in B$?", as in the following definition. A much more general notion will be studied in Chapter 5.

Definition 3.5.1 A set A *many-one reduces* to a set B, denoted by $A \leqslant_m B$, if there is a computable function f such that

$$n \in A \iff f(n) \in B$$

for every n. The function f is called a many-one reduction, or simply an *m-reduction*, because it can be noninjective, so that it can potentially reduce many questions about membership in A to one such question about B. We say that A is *m-equivalent* to B, denoted $A \equiv_m B$, if both $A \leqslant_m B$ and $B \leqslant_m A$. Clearly the relation $\equiv_m$ is an equivalence relation, and its equivalence classes are called many-one degrees, or simply *m-degrees*.

Note that all computable sets (not equal to $\emptyset$ or ω) have the same m-degree. This is at the same time the *smallest* m-degree, since the computable sets m-reduce to any other set.

In the proof of Theorem 3.3.1 we proved that H was undecidable by proving that K is undecidable and by m-reducing K to H. Also, in the proof of Rice's Theorem 3.3.3 we showed that A was noncomputable by actually building an m-reduction from $\overline{K}$ to A. That is, we used the following observation:

Proposition 3.5.2 *If B is noncomputable and $B \leqslant_m A$ then also A is noncomputable.*

Proof. Trivial. □

Proposition 3.5.3 *A set A is c.e. if and only if $A \leqslant_m K$.*

Proof. (If) Suppose that $x \in A \Leftrightarrow f(x) \in K$ for f computable. Then $A = \{x : (\exists s)[f(x) \in K_s]\}$, where K_s is the finite s-step approximation of K. So by Theorem 3.2.3 A is c.e.

(Only if) Using the *S-m-n* theorem we define f computable such that

$$\varphi_{f(e,x)}(z) = \begin{cases} 0 & \text{if } x \in W_e, \\ \uparrow & \text{otherwise.} \end{cases}$$

Then for every e and x, $x \in W_e \Leftrightarrow \varphi_{f(e,x)}(f(e,x))\downarrow \Leftrightarrow f(e,x) \in K$. In particular $W_e \leqslant_m K$ for every e. □

An element of a class for which every element of the class reduces to it (under some given notion of reduction) is called *complete*. Proposition 3.5.3 shows that K is *m-complete* for the c.e. sets. It immediately follows that the halting set H is also m-complete. This we basically saw already on page 21 when we discussed its universality.

By counting we can see that there are many m-degrees:

Proposition 3.5.4 *There are $2^{\aleph_0}$ many m-degrees.*

Proof. Note that for every A there are only $\aleph_0$ many B such that $B \leqslant_m A$ since there are only countable many possible computable functions that can act as m-reduction. In particular every m-degree contains only $\aleph_0$ many sets. (Cf. also Exercise 3.8.10.) Since there are $2^{\aleph_0}$ many sets the proposition follows. □

3.6 Simple sets

So far we have seen two kinds of c.e. set: computable ones and m-complete ones. A natural question is whether there are any other ones, that is, we ask whether there exists a c.e. set that is neither computable nor m-complete. The existence of such sets was shown by Post, and he proved this by considering sets that have a very "thin" complement, in the sense that it is impossible to effectively produce an infinite subset of their complement. We first note the following:

Proposition 3.6.1 (Post [21]) *Every infinite c.e. set has an infinite computable subset.*

Proof. Exercise 3.8.9 □

Definition 3.6.2 A set is *immune* if it is infinite and it does not contain any infinite c.e. subset. A set A is *simple* if A is c.e. and its complement $\overline{A}$ is immune.

It is immediate from the definition that simple sets are noncomputable. They are also not complete:

Proposition 3.6.3 (Post [21]) *If a c.e. set is m-complete then it is not simple.*

Proof. First we prove that K is not simple. Notice that for every x we have that

$$W_x \subseteq \overline{K} \Longrightarrow x \in \overline{K} - W_x \tag{3.1}$$

because if $x \in W_x$ then $x \in K$, and thus $W_x \not\subseteq \overline{K}$. We can use (3.1) to generate an infinite c.e. subset of $\overline{K}$ as follows:[3] Let f be a computable function such that for all x

$$W_{f(x)} = W_x \cup \{x\}.$$

[3] We could also do this directly, but we need this proof in the second step.

Now start with a code x such that $W_x = \emptyset$ and repeatedly apply f to obtain the infinite c.e. set $V = \{x, f(x), f(f(x)), \dots, f^n(x), \dots\}$. By (3.1), $V \subseteq \overline{K}$ and V is infinite. So $\overline{K}$ contains an infinite c.e. subset, and hence is not immune.

Next we prove that if $K \leqslant_m A$, then the property that K is not simple transfers to A. Suppose that $K \leqslant_m A$ via h. Let U be a c.e. subset of $\overline{A}$. Then the inverse image $h^{-1}(U)$ is a c.e. subset of $\overline{K}$. Moreover, we can effectively obtain a code of $h^{-1}(U)$ from a code of U. Using the procedure for K above we obtain an element $x \in \overline{K} - h^{-1}(U)$, and hence $h(x) \in \overline{A} - U$. By iteration we obtain an infinite c.e. subset of $\overline{A}$. Formally: Let $g(0) = h(x)$ where $W_x = \emptyset$. Note that $g(0) \in \overline{A}$ by (3.1). Given the elements $g(0), \dots, g(n) \in \overline{A}$, let $\xi(n)$ be a code of the c.e. set $h^{-1}(\{g(0), \dots, g(n)\})$. Then by (3.1) $\xi(n) \in \overline{K} - h^{-1}(\{g(0), \dots, g(n)\})$ so we define $g(n+1) = h(\xi(n))$. Now clearly $\mathrm{rng}(g)$ is an infinite c.e. subset of $\overline{A}$, and hence A is not simple. □

Theorem 3.6.4 (Post [21]) *There exists a simple set.*

Proof. We want to build a coinfinite c.e. set A such that for every e the following requirement is satisfied:

$$R_e: \qquad W_e \text{ infinite} \Longrightarrow A \cap W_e \neq \emptyset.$$

Now we can effectively list all c.e. sets, but we cannot decide which ones are infinite by the results of Section 3.3. But given a W_e, we can just wait and see what happens with it, and enumerate an element from it when we see one that is big enough. More precisely, to keep $\overline{A}$ infinite we want to make sure that at most e elements from $\{0, \dots, 2e\}$ go into A. Since we can satisfy R_e by just enumerating one element, it suffices to allow R_e to enumerate an element x only if $x > 2e$. Thus we build the set A in stages as follows:
Stage $s = 0$. Let $A_0 = \emptyset$.
Stage $s + 1$. At this stage we are given the current approximation A_s to A. Look for the smallest $e < s$ such that $A_s \cap W_{e,s} = \emptyset$ and such that there exists an $x \in W_{e,s}$ with $x > 2e$. If such e and x are found, enumerate x into A, i.e. set $A_{s+1} = A_s \cup \{x\}$. If such e and x are not found do nothing, i.e. set $A_{s+1} = A_s$. This concludes the construction.

Now the constructed set $A = \bigcup_{s \in \omega} A_s$ is coinfinite because only half of the elements smaller than $2e$ can be enumerated into it. Moreover, if W_e is infinite it contains an $x > 2e$, which means that some element from W_e is enumerated into A at some stage. So the requirement R_e is satisfied for every e. □

The proof of Theorem 3.6.4 exhibits several features that are typical of arguments in computability theory. First notice that it is a diagonalization argument: The overall requirement that the set should neither be computable nor complete is broken up into infinitely many subrequirements R_e. We then construct in infinitely many stages a set A that satisfies all of these requirements. An important difference with the examples from Section 3.1 like Cantor's argument is that here we cannot take care of requirement R_e at stage e, because *the construction has to be effective* in order to make A c.e. Thus, instead of trying to handle the requirements in order we take a more dynamic approach, and just wait until we see that a possible action can be taken. Second, there is a mild but decided tension between the requirements R_e that want to put elements *into* A, and are thus of a positive nature, and the overall requirement that A should be coinfinite, which tries to keep elements *out of* A, and

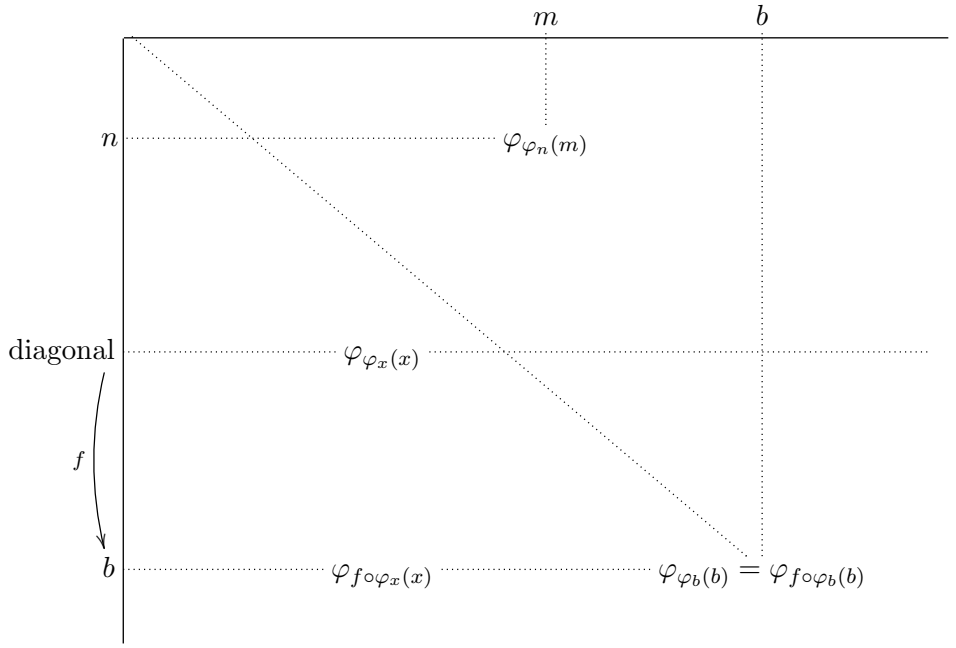


Figure 3.1: The matrix of the functions $\varphi_{\varphi_n(m)}$

hence is negative in nature. This conflict in requirements was easily solved in this case by restraining the R_e from enumerating numbers that are too small. In general the conflicts between requirements in a diagonalization argument may be of a more serious nature, giving rise to more and more complicated methods to resolve them.

Combining the results of this section we obtain:

Corollary 3.6.5 *There exists a c.e. set that is neither computable nor m-complete.*

In Section 7.3 we will see a natural example of such a c.e. set of intermediate m-degree.

3.7 The recursion theorem

The following theorem is also sometimes called the fixed-point theorem. It has a strong self-referential flavour, and the short and deceptively simple proof does not seem to explain much of the mystery.

Theorem 3.7.1 (The Recursion Theorem, Kleene [12]) *For every computable function f there exists an e (called a* fixed-point *of f) such that $\varphi_{f(e)} = \varphi_e$.*

Proof. The function $\lambda x, z.\, \varphi_{f\circ\varphi_x(x)}(z)$ is partial computable by the Enumeration Theorem 2.4.5.3, so it has a code c. By the S-m-n theorem let b be a code such that $\varphi_b(x) = S^1_1(c, x)$. Note that φ_b is total because S^1_1 is primitive recursive. Then we have

$$\varphi_{\varphi_b(x)}(z) = \varphi_c(x, z) = \varphi_{f\circ\varphi_x(x)}(z),$$

so $\varphi_b(b)$ is a fixed-point of f. □

To better understand the proof of Theorem 3.7.1 we offer the viewpoint of Owings [20], who suggested to look at the proof as a diagonal argument that fails. The

form of the theorem suggests that we look at functions of the form $\varphi_{\varphi_n(m)}$, put in a two-dimensional matrix as in figure 3.1. Here we introduce the convention that $\varphi_{\varphi_n(m)}$ denotes the totally undefined (empty) function if $\varphi_n(m)\uparrow$. Now consider the diagonal $\varphi_{\varphi_x(x)}$ of this matrix. Contrary to the usual situation in diagonal arguments, this diagonal, as a function of x, is itself again a row of the matrix! Furthermore, every function f defines a mapping of the rows, mapping the row $\varphi_{\varphi_x(y)}$ to $\varphi_{f\circ\varphi_x(y)}$. Since all this is effective there is a code b such that

$$\varphi_{\varphi_b(x)} = \varphi_{f\circ\varphi_x(x)}.$$

Taking $x = b$ we see that f has the fixed-point $\varphi_b(b)$.

The self-referential flavour of Theorem 3.7.1 lies in the fact that it *can be used to define a function in terms of its own code.* Namely it is often used in the following form: One defines an object like a partial computable function or a c.e. set A using an arbitrary code e. If the definition of A depends on e in an effective way, i.e. if there is a computable function f such that $A = W_{f(e)}$, then by Theorem 3.7.1 we may conclude that there is an e for which $A = W_e$. This justifies writing things like: "Let e be a code such that $W_e = A$" where e itself occurs in the definition of A, as long as A depends effectively on e.

A consequence of Theorem 3.7.1 is the existence of programs that print themselves when they are run. Formally, by the *S-m-n* Theorem 2.4.7 let f be a computable function such that for every x, $\varphi_{f(e)}(x) = e$. By Theorem 3.7.1 there is a fixed point e for which holds that $\varphi_e(x) = e$ for every x, i.e. the program e prints itself on any input.

As another paradoxical example, Theorem 3.7.1 implies the existence of an e such that $\varphi_e(x) = \varphi_e(x) + 1$ for every x. But of course the way out of this paradoxical situation is that φ_e can be everywhere undefined, which reminds us of the importance of the fact that we are working with partial functions.

In the next proposition we apply the recursion theorem to show that Proposition 3.2.5 does not hold uniformly.

Proposition 3.7.2 *There does not exist a partial computable function ψ such that for every e, if φ_e is total and nondecreasing then $\psi(e)\downarrow$ and $\varphi_{\psi(e)} = \chi_{\mathrm{rng}(\varphi_e)}$.*

Proof. Suppose that such ψ does exist, and let ψ_s denote the s-step approximation of ψ. By the recursion theorem, let e be a code such that

$$\{e\}(s) = \begin{cases} 1 & \text{if } \psi_s(e)\downarrow \text{ and } \varphi_{\psi(e),s}(1)\downarrow = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then $\{e\}$ is total and nondecreasing, so $\psi(e)$ must be defined and $\varphi_{\psi(e)}$ is total. Now either $\varphi_{\psi(e)}(1) = 1$, in which case $\mathrm{rng}(\{e\})$ contains only 0, or $\varphi_{\psi(e)}(1) = 0$, in which case $1 \in \mathrm{rng}(\{e\})$. In both cases $\varphi_{\psi(e)}(1) \neq \chi_{\mathrm{rng}(\{e\})}(1)$. □

3.8 Exercises

Exercise 3.8.1 Prove that the equivalences of (ii), (iii), and (iv) in Theorem 3.2.3 are all uniform in codes.

Exercise 3.8.2 Prove that the c.e. sets are closed under

(i) intersections,

(ii) bounded quantification,

(iii) images of partial computable functions.

Exercise 3.8.3 (Uniformization) A set $A \subseteq \omega^2$ is *single-valued* if

$$\forall x, y, z\big((x, y) \in A \wedge (x, z) \in A \to y = z\big).$$

Define $\mathrm{dom}(A) = \{x : \exists y\ (x, y) \in A\}$. Prove that for every c.e. $A \subseteq \omega^2$ there is a single-valued c.e. $B \subseteq A$ such that $\mathrm{dom}(A) = \mathrm{dom}(B)$.

Exercise 3.8.4 (Reduction property) Show that for any two c.e. sets A and B there are c.e. sets $\hat{A} \subseteq A$ and $\hat{B} \subseteq B$ such that $\hat{A} \cap \hat{B} = \emptyset$ and $\hat{A} \cup \hat{B} = A \cup B$.

Exercise 3.8.5 Give a formal proof of Proposition 3.2.6.

Exercise 3.8.6 Prove the following:

(i) $\leqslant_m$ is transitive.

(ii) If $X \leqslant_m Y$ and Y is c.e. then X is c.e.

(iii) $X \leqslant_m Y$ precisely when $\overline{X} \leqslant_m \overline{Y}$.

Exercise 3.8.7 Prove that K and H have the same m-degree.

Exercise 3.8.8 The *join* $A \oplus B$ of two sets A and B is defined as

$$A \oplus B = \big\{2x : x \in A\big\} \cup \big\{2x + 1 : x \in B\big\}.$$

The set $A \oplus B$ contains precisely all the information from A and B "zipped" together. Prove that for every C, if both $A \leqslant_m C$ and $B \leqslant_m C$ then $A \oplus B \leqslant_m C$. Since clearly $A, B \leqslant_m A \oplus B$ this shows that on the m-degrees $\oplus$ gives the *least upper bound* of A and B.

Exercise 3.8.9 Prove Proposition 3.6.1. (Hint: Proposition 3.2.5.)

Exercise 3.8.10 In the proof of Proposition 3.5.4 it was shown that every m-degree contains at most $\aleph_0$ many sets. Show that also every m-degree contains at least $\aleph_0$ many sets.

Exercise 3.8.11 Give a reduction $\overline{K} \leqslant_m \{e : W_e = \emptyset\}$.

Exercise 3.8.12 Give reductions $K \leqslant_m X$ and $\overline{K} \leqslant_m X$ for every $X \in \{\mathrm{Fin}, \mathrm{Tot}, \mathrm{Comp}\}$.

Exercise 3.8.13 Show that $\overline{\mathrm{Fin}} \equiv_m \mathrm{Tot} \equiv_m \{e : \mathrm{rng}(\varphi_e) \text{ is infinite}\}$.

Exercise 3.8.14 Show that the intersection of two simple sets is again simple. Prove or disprove: The union of two simple sets is again simple.

Exercise 3.8.15 Use the recursion theorem to show that the following exist:

- An e such that $W_e = \{e\}$.

- An e such that for every x, $\varphi_e(x)\downarrow \iff \varphi_{2e}(x)\downarrow$.
- An e such that $W_e = D_e$, where D_n is the canonical listing of all finite sets from Section 2.4.4.

Exercise 3.8.16 A computable function f is *self-describing* if $e = (\mu x)[f(x) \neq 0]$ exists and $\varphi_e = f$. Show that self-describing functions exist.

Exercise 3.8.17* Show that there exists a computable function f such that for all n, $f(n) < f(n+1)$ and $W_{f(n)} = \{f(n+1)\}$.

Chapter 4

The arithmetical hierarchy

In Chapter 3 we introduced m-reducibility as a tool to study decidability. But we can also view m-degrees as a measure for the *degree of unsolvability* of sets and functions. In Section 3.6 we saw by counting that there are $2^{\aleph_0}$ many m-degrees, and also that there are more than two c.e. m-degrees. (A degree is called c.e. if it contains a c.e. set.) This suggests to classify various sets from mathematics and computer science according to their m-degree. This is precisely what we do in this chapter. It turns out that for many naturally defined sets there is moreover a nice relation with the complexity of their defining formulas. So we start out by defining a hierarchy based on this idea.

4.1 The arithmetical hierarchy

Many sets of natural numbers from mathematics and computer science can be defined by a formula in the language of arithmetic $\mathcal{L}$. Usually, this language is taken to be the language of the standard model of arithmetic $\langle \omega, S, +, \cdot \rangle$, i.e. the natural numbers with successor, plus, and times. By arithmetization we can represent all primitive recursive functions in this model (cf. Theorem 7.1.4), so there is no harm in extending the language $\mathcal{L}$ with symbols for every primitive recursive predicate. Let us call this extension $\mathcal{L}^*$. We also assume that $\mathcal{L}^*$ contains constants to denote every element of ω. For ease of notation we denote the constant denoting the number n also by n. The intended model for $\mathcal{L}^*$ are the natural numbers with all the primitive recursive functions, which we denote again by ω. Thus by $\omega \models \varphi$ we denote that the formula φ is true in this model.

Definition 4.1.1 An n-ary relation R is *arithmetical* if it is definable by an arithmetical formula, i.e. if there is an $\mathcal{L}^*$-formula φ such that for all $x_1, \ldots, x_n$,

$$R(x_1, \ldots, x_n) \iff \omega \models \varphi(x_1, \ldots, x_n).$$

We can write every $\mathcal{L}^*$-formula φ in *prenex normal form*, i.e. with all quantifiers in front (the *prefix* of φ) and quantifying over a propositional combination of computable predicates (the *matrix* of φ). Now the idea is that we distinguish defining formulas by their quantifier complexity. First note that two consecutive quantifiers of the same kind can be contracted into one, using coding of pairs. E.g. if φ has the form

$$\ldots \exists x \exists y \ldots P(\ldots x \ldots y \ldots)$$

then φ can be transformed into the equivalent formula

$$\ldots \exists z \ldots P(\ldots (z)_0 \ldots (z)_1 \ldots).$$

The same holds for consecutive blocks of universal quantifiers. So we will measure quantifier complexity by only counting *alternations* of quantifiers.

Definition 4.1.2 (Kleene) We inductively define a hierarchy as follows.

- $\Sigma^0_0 = \Pi^0_0 =$ the class of all computable relations.
- Σ^0_{n+1} is the class of all relations definable by a formula of the form $\exists x R(x, \vec{y})$ where R is a Π^0_n-predicate.
- Π^0_{n+1} is the class of all relations definable by a formula of the form $\forall x R(x, \vec{y})$ where R is a Σ^0_n-predicate.
- $\Delta^0_n = \Sigma^0_n \cap \Pi^0_n$ for every n.

So Σ^0_n ($n \geqslant 1$) is the class of relations definable by an $\mathcal{L}^*$-formula in prenex normal form with n quantifier alternations, and starting with an existential quantifier. Idem for Π^0_n, but now starting with a universal quantifier. It follows from Proposition 3.2.6 that Δ^0_1 consists precisely of all the computable relations, and from Theorem 3.2.3 that Σ^0_1 consists of the c.e. relations. The superscript 0 in the notation for the classes in the arithmetical hierarchy refers to the fact that they are defined using first-order formulas, that is formulas with quantification only over numbers. There are indeed higher order versions defined using more complex formulas, but we will not treat these here. For example, the classes of the *analytical hierarchy*, denoted with a superscript 1, are defined using second-oder quantification, where quantifiers are also allowed to run over sets of natural numbers.

Proposition 4.1.3 (Closure properties)

1. *$R \in \Sigma^0_n$ if and only if $\overline{R} \in \Pi^0_n$.*
2. *Σ^0_n, Π^0_n, and Δ^0_n are closed under intersections, unions, and bounded quantification.*
3. *Δ^0_n is closed under taking complements.*
4. *For $n \geqslant 1$, Σ^0_n is closed under existential quantification, and Π^0_n is closed under universal quantification.*
5. *Σ^0_n and Π^0_n are closed downwards under $\leqslant_m$, i.e. if $A \leqslant_m B$ and $B \in \Sigma^0_n$ then $A \in \Sigma^0_n$, and similarly for Π^0_n.*

Proof. See Exercise 4.3.1. Item 1 follows from the standard procedure of pulling negations through quantifiers. Item 4 follows from the contraction of quantifiers discussed on page 32. □

The next theorem shows that our hierarchy does not collapse, and is strict at all levels.

Theorem 4.1.4 (Hierarchy Theorem) *For every $n \geqslant 1$ the following hold:*

1. *$\Sigma^0_n - \Pi^0_n \neq \emptyset$, and hence $\Delta^0_n \subsetneq \Sigma^0_n$.*
2. *$\Pi^0_n - \Sigma^0_n \neq \emptyset$, and hence $\Delta^0_n \subsetneq \Pi^0_n$.*

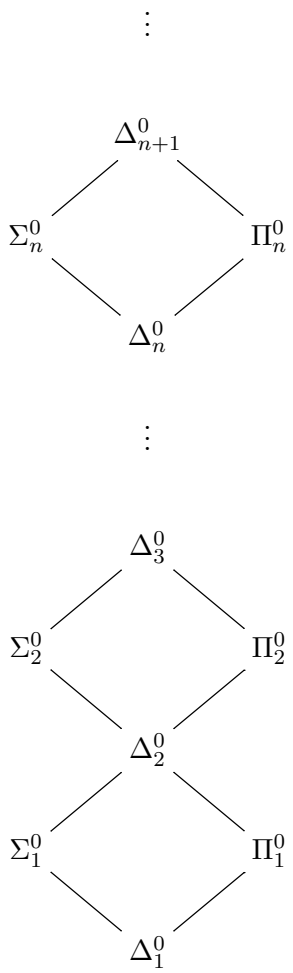


Figure 4.1: The Arithmetical Hierarchy

3. $\Sigma^0_n \cup \Pi^0_n \subsetneq \Delta^0_{n+1}$.

Proof. We define analogues K_n of the diagonal set K for every level Σ^0_n, and we imitate the proof of Theorem 3.3.1 to show that K_n is not in Π^0_n. Define

$$K_{2n} = \big\{e : \exists x_1 \forall x_2 \ldots \exists x_{2n-1} \forall s \; \varphi^{2n}_{e,s}(e, x_1, \ldots, x_{2n-1}) \uparrow \big\},$$
$$K_{2n+1} = \big\{e : \exists x_1 \forall x_2 \ldots \forall x_{2n} \exists s \; \varphi^{2n+1}_{e,s}(e, x_1, \ldots, x_{2n}) \downarrow \big\}.$$

Clearly $K_n \in \Sigma^0_n$ for every n. Also note that $K_1 = K$. Now suppose that $K_n \in \Pi^0_n$ and that n is odd. (The case of even n is similar.) Then $\overline{K}_n \in \Sigma^0_n$, so $\overline{K}_n = \big\{z : \exists x_1 \forall x_2 \ldots \exists x_n \; R(z, x_1, \ldots, x_n)\big\}$ for some computable predicate R. Now by Theorem 3.2.3 there is a code e such that for every $z, x_1, \ldots, x_{n-1}$

$$\exists x_n \; R(z, x_1, \ldots, x_n) \iff \exists s \; \varphi_{e,s}(z, x_1, \ldots, x_{n-1}) \downarrow .$$

Now we obtain a contradiction by considering $z = e$:

$$\begin{aligned} e \in \overline{K}_n &\iff \exists x_1 \forall x_2 \ldots \exists x_n \; R(e, x_1, \ldots, x_n) \\ &\iff \exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists s \; \varphi_{e,s}(z, x_1, \ldots, x_{n-1}) \downarrow \\ &\iff e \in K_n. \end{aligned}$$

Hence $K_n \notin \Pi^0_n$. Item 2 immediately follows from this, since $\overline{K}_n \in \Pi^0_n - \Sigma^0_n$.

For item 3 consider the set $K_n \oplus \overline{K}_n$, where $\oplus$ denotes the join operator defined in Exercise 3.8.8. Since both K_n and $\overline{K}_n$ are in Δ^0_{n+1}, so is their join. On the other hand, $K_n \oplus \overline{K}_n \notin \Pi^0_n$ because otherwise it would follow that $K_n \in \Pi^0_n$, as $K_n \leqslant_m K_n \oplus \overline{K}_n$ and Π^0_n is closed under $\leqslant_m$. Similarly we have $K_n \oplus \overline{K}_n \notin \Sigma^0_n$. □

For the sets K_n from the proof of Theorem 4.1.4 we can actually show more, namely that they are m-complete at their respective levels:

Proposition 4.1.5 *K_n is m-complete for Σ^0_n.*

Proof. Suppose that n is odd. (Again the case of even n is similar.) Consider any Σ^0_n set $A = \{e : \exists x_1 \forall x_2 \ldots \exists x_n \; R(e, x_1, \ldots, x_n)\}$. Define a computable function f such that

$$\varphi^n_{f(e)}(z, x_1, \ldots, x_{n-1}) = \begin{cases} 0 & \text{if } \exists x_n \; R(e, x_1, \ldots, x_n) \\ \uparrow & \text{otherwise.} \end{cases}$$

Then

$$\begin{aligned} e \in A &\iff \exists x_1 \forall x_2 \ldots \exists x_n \; R(e, x_1, \ldots, x_n) \\ &\iff \exists x_1 \forall x_2 \ldots \forall x_{n-1} \exists s \; \varphi^n_{f(e),s}(f(e), x_1, \ldots, x_{n-1}){\downarrow} \\ &\iff f(e) \in K_n. \end{aligned}$$

□

Next we consider a famous application of the arithmetical hierarchy, namely Tarski's theorem saying that arithmetical truth is itself not an arithmetical notion. Let $\varphi \mapsto \ulcorner \varphi \urcorner$ be a primitive recursive coding of $\mathcal{L}^*$-sentences, mapping sentences to numbers. $\ulcorner \varphi \urcorner$ is called the *Gödel number* of φ.

Theorem 4.1.6 (Tarski [29]) *The set*

$$\{\ulcorner \varphi \urcorner : \omega \models \varphi\}$$

of true arithmetical formulas is not arithmetical.

Proof. Exercise 4.3.3. □

4.2 Computing levels in the arithmetical hierarchy

Given an arithmetically defined set it is usually easy to establish some upper bound on the level of the set in the arithmetical hierarchy, simply by writing out its defining formula in prenex normal form and counting the quantifier alternations. But of course we often want to know the exact level instead of just an upper bound. Proposition 4.1.5 provides us with a method to establish lower bounds, namely if we can show that $K_n \leqslant_m A$ or $\overline{K}_n$ then we know that A cannot occur at a lower level. We now give a number of examples, of growing complexity, illustrating this method.

Proposition 4.2.1 *The set $A = \{e : W_e = \emptyset\}$ is m-complete for Π^0_1.*

Proof. Since $e \in A \Leftrightarrow \forall x \forall s \; \varphi_{e,s}(x){\uparrow}$ we see that A is at most Π^0_1. Rice's Theorem 3.3.3 shows us that A is noncomputable, but we can actually show that A is m-complete

for Π^0_1: Let $B = \overline{W}_e$ be an arbitrary Π^0_1 set. Let f be a computable function such that

$$\varphi_{f(x)}(z) = \begin{cases} 0 & \text{if } x \in W_e, \\ \uparrow & \text{otherwise.} \end{cases}$$

Then $x \in W_e \Leftrightarrow W_{f(x)} \neq \emptyset$, so $B \leqslant_m A$ via f. Since B was arbitrary this shows the completeness of A. N.B. The reader that has done Exercise 3.8.11 could also have concluded the completeness from there. □

Proposition 4.2.2
- $\mathrm{Fin} = \{e : W_e \text{ is finite}\}$ *is m-complete for* Σ^0_2.
- $\mathrm{Tot} = \{e : \varphi_e \text{ is total}\}$ *is m-complete for* Π^0_2.

Proof. We leave it to the reader to verify that $\mathrm{Fin} \in \Sigma^0_2$ and $\mathrm{Tot} \in \Pi^0_2$ (Exercise 4.3.4). We prove the two items simultaneously by providing one reduction for both of them. Let $A = \{e : \forall x \exists y\ R(e, x, y)\}$ be any Π^0_2 set, where R is a computable predicate. Define a computable function f such that

$$\varphi_{f(e)}(z) = \begin{cases} 0 & \text{if } (\forall x < z)(\exists y)[R(e, x, y)], \\ \uparrow & \text{otherwise.} \end{cases}$$

Then

$$\begin{array}{lllll} e \in \overline{A} & \iff & \exists x \forall y\ \neg R(e, x, y) & \iff & f(e) \in \mathrm{Fin}, \\ e \in A & \iff & \forall x \exists y\ R(e, x, y) & \iff & f(e) \in \mathrm{Tot}. \end{array}$$

□

Theorem 4.2.3 (Rogers, Mostowski) $\mathrm{Comp} = \{e : W_e \text{ is computable}\}$ *is m-complete for* Σ^0_3.

Proof. That Comp is Σ^0_3 is shown in Exercise 4.3.4. Let $A = \{e : \exists x \forall y \exists z\ R(e, x, y, z)\}$ be any Σ^0_3 set. Given e we want to build a c.e. set $W_{f(e)}$ that is either cofinite (and hence computable) or simple[1] (and hence noncomputable), depending on whether $e \in A$ or not. We will enumerate $W_{f(e)}$ in an effective construction by stages. The n-th element of the complement of $W_{f(e),s}$ will be denoted by a^s_n. So at stage s we have

$$\overline{W_{f(e),s}} = \{a^s_0 < a^s_1 < a^s_2 < \dots\}$$

We think of the a^s_n as markers moving to ever higher positions. Note that $W_{f(e)}$ is coinfinite if and only if every marker eventually comes to a rest, i.e. if $a_n = \lim_{s\to\infty} a^s_n$ exists for every n. Call x *infinitary* if $\forall y \exists z\ R(e, x, y, z)$, and call x *finitary* otherwise. Every time we see evidence that x is infinitary we will enumerate a^s_x into $W_{f(e)}$. So if x really is infinitary this will make $W_{f(e)}$ cofinite. We will let $W_{f(e)}$ contain a simple set (which exists by Theorem 3.6.4). Hence $W_{f(e)}$ will be simple itself if and only if it is coinfinite. The construction is as follows.

Stage $s = 0$. We start with $W_{f(e)} = S$, where S is a simple set. (Since we cannot actually enumerate all elements of S in one stage, what we actually mean by this is that we enumerate all the elements of S into $W_{f(e)}$ over the construction.) We will maintain a length function $l(x, s)$ monitoring whether x is infinitary. Define $l(x, 0) = 0$ for every x.

Stage $s + 1$. Define

$$l(x, s) = \max\{y < s \mid (\forall y' < y)(\exists z < s)[R(e, x, y', z)]\}.$$

[1] In most textbook proofs $W_{f(e)}$ is made Turing complete, but we have not treated T-reducibility at this point yet. Moreover, just making $W_{f(e)}$ simple greatly simplifies the proof.

Let $x \leqslant s$ be minimal such that $l(x, s+1) > l(x, s)$. If such x is found enumerate a_x^s into $W_{f(e)}$, i.e. set $W_{f(e),s+1} = W_{f(e),s} \cup \{a_x^s\}$. Otherwise do nothing, i.e. set $W_{f(e),s+1} = W_{f(e),s}$. This concludes the construction.

Now to verify that $W_{f(e)}$ is what we want we consider the following cases. First suppose that all x are finitary. Then $a_n = \lim_{s\to\infty} a_n^s$ exists for every n and hence $W_{f(e)}$ is coinfinite. Since $W_{f(e)}$ also contains the simple set S it is itself simple, and hence noncomputable. Second suppose that there is an x that is infinitary. Then we already observed above that $W_{f(e)}$ is cofinite, and hence computable. Since the construction of $W_{f(e)}$ depends effectively on e the function f is computable, and is thus an m-reduction witnessing that $A \leqslant_m \text{Comp}$. Since A was arbitrary it follows that Comp is m-complete for Σ_3^0. □

The construction in the proof of Theorem 4.2.3 is actually our first example of a *priority construction*, a method that we will study in more detail in Chapter 6. As a corollary to the proof we have:

Corollary 4.2.4
- *The set* $\text{Cof} = \{e : W_e \text{ is cofinite}\}$ *is m-complete for* Σ_3^0.
- *The set* $\{e : W_e \text{ is simple}\}$ *is m-complete for* Π_3^0.

It so happens that most naturally defined arithmetical sets are m-complete for some level of the arithmetical hierarchy, so that the above method works most of the time (although the computations can get rather hairy at higher levels). However, there do exist natural examples (e.g. from the theory of randomness, cf. Proposition 7.3.3) of arithmetical sets that are not m-complete for any level, so the method does not always work.

4.3 Exercises

Exercise 4.3.1 Prove items 1 – 5 from Proposition 4.1.3.

Exercise 4.3.2 Complement Proposition 4.1.3 by showing that

(i) Π_n^0 and Δ_n^0 are not closed under $\exists$.

(ii) Σ_n^0 and Δ_n^0 are not closed under $\forall$.

(iii) Σ_n^0 and Π_n are not closed under complementation.

Exercise 4.3.3* Prove Theorem 4.1.6.

Exercise 4.3.4 Verify the following items:

1. $\text{Fin} \in \Sigma_2^0$.
2. $\text{Tot} \in \Pi_2^0$.
3. $\text{Comp} \in \Sigma_3^0$.
4. $\text{Cof} \in \Sigma_3^0$.
5. $\{e : W_e \text{ is simple}\} \in \Pi_3^0$.

Exercise 4.3.5 Show that for the notion of T-completeness defined in Chapter 5, $\{e : W_e \text{ is T-complete for } \Sigma_1^0\} \in \Sigma_4^0$.

Exercise 4.3.6 Recall the definition of self-describing functions from Exercise 3.8.16. Classify $\{e : \varphi_e \text{ is self-describing}\}$ in the arithmetical hierarchy.

Chapter 5

Relativized computation and Turing degrees

5.1 Turing reducibility

In Definition 3.5.1 we defined a notion of reduction where, in order to decide whether $n \in A$, we were allowed to ask *one* question about B, namely whether $f(x) \in B$. This can obviously be generalized to two, three, or finitely many questions about B, in any precisely prescribed way. Rather than studying such generalizations of m-reducibility here, we immediately jump to one of the most liberal notions of reduction, where questions of the form "$n \in A$?" are solved by using *any finite amount of information* from B, in any computable way. Thus, when reducing A to B, we pretend that all the information in B is given, and use this to solve A. To model that B is given we can for example add its characteristic function χ_B to the initial functions in Definition 2.2.3:

Definition 5.1.1 (Turing [31]) Let A be any set. The class of *partial A-recursive functions* is the smallest class of functions

1. containing the initial functions and χ_A,
2. closed under composition, primitive recursion, and μ-recursion.

An *A-recursive function* is a partial A-recursive function that is total.

Alternatively, we could have defined the class of A-computable functions by using an extension of the Turing machine model, where the information in A is written on an extra infinite input tape. We can again show the equivalence of this approach to Definition 5.1.1 using arithmetization, exactly as in Chapter 2. So henceforth, we shall speak about the *A-computable functions*, by which we will mean any of the equivalent concepts so obtained. All the basic results about the computable functions carry over to the A-computable functions. In particular we have a coding as in Theorem 2.4.1, and we can define:

Definition 5.1.2 The e-th partial A-computable function is denoted by φ_e^A or $\{e\}^A$.

Definition 5.1.3 (Post [22]) We say that A is *Turing reducible* to B, denoted by $A \leqslant_T B$, if A is B-computable. We say that A and B are *T-equivalent*, denoted $A \equiv_T B$ if both $A \leqslant_T B$ and $B \leqslant_T A$. The equivalence classes of $\equiv_T$ are called Turing degrees, or simply *T-degrees*.

When $A \leqslant_T B$ we will also say that A is computable *relative to* B. The set B is also sometimes called an *oracle*, since the information contained in it is given for free. The notion of relative computation is central in computability theory, since it embodies how one can use information in any computable way. Note that it is indeed more liberal than many-one reducibility, that is, we have

$$A \leqslant_m B \Longrightarrow A \leqslant_T B,$$

as the reader can easily verify. Also, the converse implication does not hold (Exercise 5.6.1).

Many notions from computability theory can be *relativized* to an arbitrary oracle A, i.e. their definitions can be generalized by considering A-computable functions instead of computable ones. For example, an A-c.e. set is a set that can be enumerated by an A-computable function. Many basic results in computability theory also relativize in the sense that they remain true when they are relativized to an arbitrary oracle. For example all of the basic notions and results from Chapter 3 relativize (cf. Exercise 5.6.3).

The following basic observation about relativized computations is fundamental, and expresses that any convergent computation, relativized or not, can use only a finite amount of information from an oracle.

Proposition 5.1.4 (Use Principle) *Let A be any oracle and suppose that $\{e\}^A_s(x)\downarrow$. Then there is a finite string $\sigma \sqsubset A$ such that $\{e\}^\sigma_s(x)\downarrow$. In particular $\{e\}^B_s(x)\downarrow$ for any $B \sqsupset \sigma$.*

Proof. Any convergent computation consists of a finite number of computation steps. Hence only a finite number of bits of A can occur in it. Note that we may also assume that $|\sigma| < s$ by Definition 2.4.4. □

Definition 5.1.5 If $\{e\}^A(x)\downarrow$ the maximum number that is used in the computation (which exists by Proposition 5.1.4) is called the *use* of the computation.

5.2 The jump operator

We can also relativize the diagonal halting set K to an arbitrary oracle A, and we denote the result by A':

$$A' = \{x : \varphi^A_x(x)\downarrow\}.$$

(A' is pronounced as "A prime".) Now it is easily verified that the proof of Theorem 3.3.1 relativizes, so that A' is not an A-computable set. Since we also have that $A \leqslant_m A'$ the operator $'$ is called the *jump operator*. Notice that $\emptyset' = K$. ($\emptyset'$ is pronounced as "zero prime".) We denote the n-th iterate of the jump of A by $A^{(n)}$. Note that $\emptyset^{(n)} \in \Sigma^0_n$ (Exercise 5.6.4).

The difference between many-one and Turing reducibility can be nicely characterized in terms of the arithmetical hierarchy:

Proposition 5.2.1 (i) $\Sigma^0_n = \{A : A \leqslant_m \emptyset^{(n)}\}$ *for every n.*

(ii) Σ^0_{n+1} *consists of all the sets that are $\emptyset^{(n)}$-c.e., that is, enumerable by a $\emptyset^{(n)}$-computable function.*

(iii) $\Delta^0_{n+1} = \{A : A \leqslant_T \emptyset^{(n)}\}$ *for every n.*

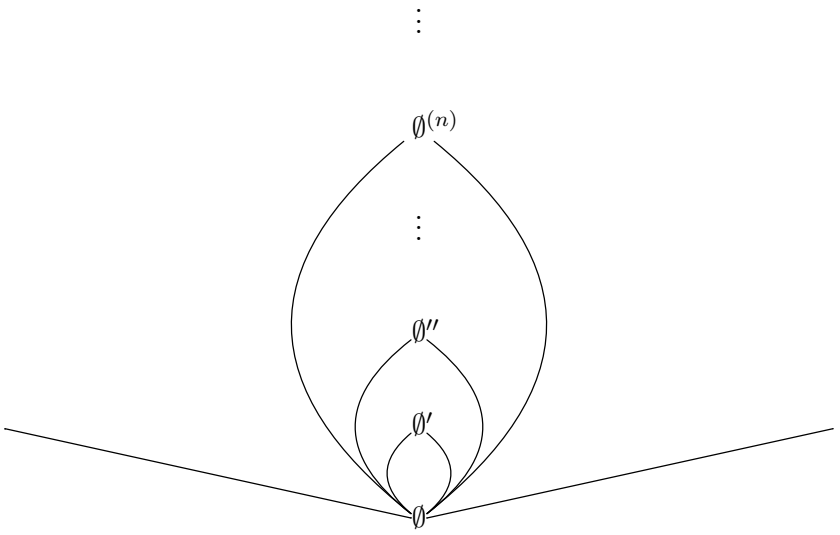


Figure 5.1: The Turing universe

Proof. We prove (i) by induction. The case $n = 1$ is Proposition 3.5.3. Now assume that $\emptyset^{(n)}$ is Σ^0_n-complete, or equivalently, that $\overline{\emptyset^{(n)}}$ is Π^0_n-complete. Then $A \in \Sigma^0_{n+1}$ if and only if A is c.e. in some Π^0_n set (by definition of Σ^0_n and the relativized version of Theorem 3.2.3), which by induction hypothesis is equivalent to A being $\overline{\emptyset^{(n)}}$-c.e., which is the same as $\emptyset^{(n)}$-c.e. By relativizing Proposition 3.5.3, we see that this is equivalent to $A \leqslant_m \emptyset^{(n+1)}$.

Note that (ii) is immediate from the proof of (i).

For (iii), we have that $A \in \Delta^0_{n+1}$ if and only if both A and $\overline{A}$ are in Σ^0_{n+1}, which by (ii) is equivalent to A and $\overline{A}$ being $\emptyset^{(n)}$-c.e. The latter is equivalent to $A \leqslant_T \emptyset^{(n)}$ by the relativized version of Proposition 3.2.6. □

From Proposition 5.2.1 we see that K is Turing complete for Δ^0_2, and that the sequence $\emptyset$, $\emptyset'$, $\emptyset''$, ..., $\emptyset^{(n)}$, ... constitutes a kind of backbone of the arithmetical hierarchy.

5.3 Limit computable sets

It often happens in mathematics and computer science that new objects are defined as limits of more basic objects. (E.g. the definition of the reals as limits of sequences of rationals, the approximation of continuous functions by polynomials, or the c.e. set W_e as the limit of $W_{e,s}$, $s \in \omega$.) Here we are interested in the complexity of objects defined as the limit of *computable* processes. Again it turns out that the arithmetical hierarchy is useful for this purpose.

Definition 5.3.1 A unary function f is *limit computable* if there is a computable binary function g such that for all n,

$$f(n) = \lim_{s \to \infty} g(n, s).$$

A set A is limit computable if its characteristic function χ_A is.

Proposition 5.3.2 (Limit Lemma, Shoenfield [26]) *A set A is limit computable if and only if $A \in \Delta^0_2$.*

Proof. (If) We use that $\Delta^0_2 = \{A : A \leqslant_T K\}$ (Proposition 5.2.1). Suppose that $A \leqslant_T K$, say $A = \{e\}^K$. Denote by K_s the s-step approximation of K, and define

$$g(x,s) = \begin{cases} \{e\}^{K_s}_s(x) & \text{if } \{e\}^{K_s}_s(x)\downarrow, \\ 0 & \text{otherwise.} \end{cases}$$

Now in general the limit $\lim_{s\to\infty}\{e\}^{K_s}_s(x)$ need not exist, since it may happen that $\{e\}^K(x)\uparrow$ but still $\{e\}^{K_s}_s(x)\downarrow$ for infinitely many s. But in this case we know that $\{e\}^K(x)\downarrow$ for every x, since it is equal to $A(x)$. So we have that $\lim_s g(x,s) = A(x)$ for every x.

(Only if) Suppose that g is a computable function such that $A(x) = \lim_s g(x,s)$ for every x. Then

$$\begin{aligned} x \in A &\iff (\exists s)(\forall t > s)[g(x,t) = 1] \\ &\iff (\forall s)(\exists t > s)[g(x,t) = 1]. \end{aligned}$$

So A is both Σ^0_2 and Π^0_2, hence Δ^0_2. □

5.4 Incomparable degrees

At this point we have seen that there are more than two c.e. m-degrees, and that in general there are uncountably many m-degrees. The same counting argument that we used for m-degrees shows that there are $2^{\aleph_0}$ many T-degrees (Exercise 5.6.6). A priori it could be possible that the T-degrees are linearly ordered. In this section we show that this is not the case. Namely, we show that there are sets A and B such that $A \not\leqslant_T B$ and $B \not\leqslant_T A$. Such sets A and B are called *Turing incomparable*, and we denote this by $A|_T B$.

Theorem 5.4.1 (Kleene and Post [13]) *There is a pair of Turing incomparable sets.*

Proof. We build a pair of incomparable sets A and B by finite extensions, that is, we define two series of finite strings σ_s and τ_s, $s \in \omega$, such that $\sigma_s \sqsubset \sigma_{s+1}$, $\tau_s \sqsubset \tau_{s+1}$, and $A = \bigcup_{s\in\omega} \sigma_s$ and $B = \bigcup_{s\in\omega} \tau_s$. The property that A and B should be incomparable can be split into infinitely many *requirements*:

$$R_{2e}: \quad A \neq \{e\}^B,$$

$$R_{2e+1}: \quad B \neq \{e\}^A.$$

We build A and B in infinitely many stages s, where at stage $s = 2e$ we take care of R_{2e} and at stage $s = 2e+1$ of R_{2e+1}. We start with $\sigma_0 = \tau_0 = \emptyset$.
Stage $s = 2e$. At this stage σ_s and τ_s are already defined, and we define σ_{s+1} and τ_{s+1}. To satisfy R_{2e} we need a witness x such that

$$A(x) \neq \{e\}^B(x). \tag{5.1}$$

Let x be the smallest number for which A is not yet defined, i.e. let $x = |\sigma_s|$. If the right hand side of (5.1) would be undefined we could take for $A(x)$ whatever we want, and if it were defined we would let $A(x)$ be different from it. First we see if there is any $\tau \sqsupseteq \tau_s$ such that $\{e\}^\tau(x)\downarrow$. If such τ does not exist we let $A(x) = 0$, i.e. $\sigma_{s+1} = \sigma_s{}^\frown 0$, and we let $\tau_{s+1} = \tau_s$. If such τ does exist, choose the minimal one (in a

fixed computable ordering of all binary strings) of length greater than x and define $\tau_{s+1} = \tau$. Further define $\sigma_{s+1} = \sigma_s{}^\frown\big(1 - \{e\}^{\tau_{s+1}}(x)\big)$.
Stage $s = 2e + 1$. Here we do exactly the same as at stage $2e$, except with A and B interchanged. This completes the construction.

Now first note that $\bigcup_{s\in\omega} \sigma_s$ and $\bigcup_{s\in\omega} \tau_s$ are total, since at every stage they become defined on a new number x. Second, R_{2e} is satisfied at stage $2e$ by the choice of σ_{s+1} and τ_{s+1}, and similarly for R_{2e+1}. □

The finite extension method used in the proof of Theorem 5.4.1 is quite powerful and yields a wealth of information about the structure of the Turing degrees. Arguments where sets are constructed using finite extensions are called *Kleene-Post arguments*. There are interesting methodological connections with other areas of mathematics, such as Baire category theory and Cohen forcing in set theory. The proof of Theorem 5.4.1 can be extended to show much more, namely that there is a set of size $2^{\aleph_0}$ such that every two of them are mutually incomparable (Exercise 5.6.9). This shows that the Turing universe is as least as wide as it is high.

5.5 Inverting the jump

Since for the jump A' of any set A we obviously have $A' \geqslant_T \emptyset'$, the range of the jump operator is included in the cone $\{B : B \geqslant_T \emptyset'\}$. In this section we show, using a Kleene-Post argument, that the range of the jump operator is in fact equal to this cone.

Theorem 5.5.1 (Jump Inversion Theorem, Friedberg [6]) *The range of the jump operator is* $\{B : B \geqslant_T \emptyset'\}$.

Proof. We show that for every $B \geqslant_T \emptyset'$ there is an A such that $A' \equiv_T B$. So let $B \geqslant_T \emptyset'$. We define $A = \bigcup_{s\in\omega} \sigma_s$ by finite extensions, as in the proof of Theorem 5.4.1. We start with $\sigma_0 = \emptyset$.
Stage $s = 2e$. At this stage σ_s is already defined, and we define σ_{s+1}. Look if there is a $\sigma \sqsupseteq \sigma_s$ such that

$$\{e\}^\sigma(e)\!\downarrow .$$

If such σ exists, take the smallest one and define $\sigma_{s+1} = \sigma$. (Otherwise we let $\sigma_{s+1} = \sigma_s$.)
Stage $s = 2e + 1$. At this stage we code B into A by setting

$$\sigma_{s+1} = \sigma_s{}^\frown B(e).$$

This completes the construction. We verify that A is as we wanted. First note that the construction is computable in B: the even stages can be computed by $\emptyset'$, and hence by B since $\emptyset' \leqslant_T B$, and the odd stages can be computed by B directly. Second note that the construction is also $A \oplus \emptyset'$-computable, since the even stages are $\emptyset'$-computable, so given $\emptyset'$ we can for every e compute σ_{2e+1} from σ_{2e} and then with the help of A compute σ_{2e+2}, hence $B(e)$. So we have

$$B \leqslant_T A \oplus \emptyset' \leqslant_T A'.$$

The second inequality holds because both $A \leqslant_T A'$ and $\emptyset' \leqslant_T A'$. (Cf. Exercise 5.6.2.) Conversely, $A' \leqslant_T A \oplus \emptyset'$ since to decide whether $\{e\}^A(e)\!\downarrow$ we simulate the construction until at stage $2e$ we can compute whether σ exists. If so, we know we took it

so $e \in A'$, and if not then necessarily $e \notin A'$. Now we also have $A \leqslant_T B$ since the construction is B-computable, as already remarked above. Summarizing we have

$$A' \leqslant_T A \oplus \emptyset' \leqslant_T B.$$

Combining this with what we had before we see that $A' \equiv_T B$, as we wanted. □

The proofs of Theorem 5.4.1 and Theorem 5.5.1 are extensions of the diagonalization method discussed at the beginning of Section 3.1. In particular the proof of Theorem 5.5.1 contains germs of the method of *forcing*, which is yet another incarnation of the diagonalization method and that has been pushed to great heights in set theory. We will further discuss the power of diagonalization in the next chapter.

5.6 Exercises

Exercise 5.6.1 Verify the following items:

1. $A \leqslant_m B \Rightarrow A \leqslant_T B$.
2. $\overline{A} \leqslant_T A$.
3. $A \leqslant_T B \not\Rightarrow A \leqslant_m B$.

Exercise 5.6.2 Recall the join operator $\oplus$ defined in Exercise 3.8.8. Show that $\oplus$ also is a least upper bound operator in the Turing degrees, i.e. prove that for every C, if both $A \leqslant_T C$ and $B \leqslant_T C$ then $A \oplus B \leqslant_T C$.

Exercise 5.6.3

- Verify that Theorem 3.2.3 relativizes.
- Show that $A \leqslant_m A'$ and that Theorem 3.3.1 relativizes.

Exercise 5.6.4 Recall the sets K_n from the proof of Theorem 4.1.4. Show that $\emptyset^{(n)} \equiv_m K_n$ for every n.

Exercise 5.6.5 Show that if A is c.e. and $f \leqslant_T A$, then f has a computable approximation $f(n) = \lim_s g(n, s)$, and in addition there exists a modulus function $m \leqslant_T A$ such that $(\forall s \geqslant m(n))[f(n) = g(n, s)]$.

Exercise 5.6.6 Show that every T-degree is countable, and that there are $2^{\aleph_0}$ many T-degrees.

Exercise 5.6.7 Show that there is a pair of Turing incomparable sets in Δ^0_2. (Hint: Show that the construction in the proof of Theorem 5.4.1 is computable in K.)

Exercise 5.6.8 Show that the m-degrees are not linearly ordered. (Hint: Consider K and $\overline{K}$.)

Exercise 5.6.9* Show that there is a set of $2^{\aleph_0}$ many T-degrees that are pairwise incomparable. (Hint: Construct a perfect subtree of 2^ω such that any two infinite paths in it satisfy the requirements of Theorem 5.4.1.)

Exercise 5.6.10 Exercise 5.6.9 shows that the width of the Turing universe is $2^{\aleph_0}$. Show that the height of the universe is $\aleph_1$, the first uncountable cardinal.

Exercise 5.6.11* A set A is *low* if $A' \leqslant_T \emptyset'$, that is, the jump of A is as low as possible. Note that every computable set is low. Prove that there are also noncomputable low sets. (Hint: Combine jump inversion with the construction of a noncomputable set.)

Exercise 5.6.12* Let A be a c.e. set and suppose that A has a computable enumeration $\{A_s\}_{s\in\omega}$ such that for all $e \in \omega$

$$(\exists^\infty s)\big[\{e\}_s^{A_s}(e)\!\downarrow\big] \implies \{e\}^A(e)\!\downarrow .$$

Prove that then A is low. (Hint: Use the Limit Lemma.)

Exercise 5.6.13 (Continuous functionals) A *computable functional* (also called a *Turing functional*) $F : 2^\omega \to 2^\omega$ is a total function of the form $F(X)(n) = \{e\}^X(n)$, that is, the n-th bit of the output is given by the computation $\{e\}^X(n)$. Similarly, an *A-computable functional* is a total function of the form $F(X)(n) = \{e\}^{A\oplus X}(n)$.

1. Show that every A-computable functional is continuous. (Hint: Use Proposition 5.1.4.)
2. Conversely, show that for every continuous functional F there is an A such that F is A-computable. (Hint: Let A code the modulus of continuity of F.)
3. Conclude that there are $2^{\aleph_0}$ continuous functionals.

Chapter 6

The priority method

6.1 Diagonalization, again

In Section 3.1 we discussed the method of diagonalization. Afterwards we encountered the method in several of our fundamental results, such as the undecidability of the halting problem (Theorem 3.3.1), the existence of computably inseparable sets (Theorem 3.3.5), the existence of simple sets (Theorem 3.6.4), the recursion theorem (Theorem 3.7.1), and the existence of incomparable Turing degrees (Theorem 5.4.1).

In general, in a diagonal argument one tries to construct an object A with some global property P by splitting P into infinitely many subrequirements R_e. The object A is then built using a construction with infinitely many stages. For example, in Cantor's diagonalization construction A is a set with the global property that it is not on a given countable list of sets A_i. This property can be split into the infinitely many subrequirements R_i saying that $A \neq A_i$ for every i. In the construction we can simply satisfy all R_i in order by defining $A(i) = 1 - A_i(i)$.

After the proof of Theorem 3.6.4 we discussed how it is not always possible to satisfy the requirements in a diagonalization argument in order, but that due to effectivity constraints we sometimes have to adopt a more dynamic approach. In this chapter we further generalize the method in order to solve an important question about the c.e. Turing degrees.

Apart from the problem that sometimes we cannot effectively see how to satisfy a requirement it may happen that the requirements in a diagonalization construction are in conflict with each other. Note that in Cantor's argument the actions taken to satisfy every requirement do not interfere with each other. However, in Theorem 3.6.4 we already saw a conflict between requirements wanting to keep elements out of A and requirements trying to put elements into it. Also in the proof of Theorem 4.2.3 there were conflicting requirements, although at that point we did not need to make this explicit. However, for what follows it will be instructive to explicitly analyze these requirements. We use the notation from the proof. In the proof we built a c.e. set $W_{f(e)}$ that was cofinite if an infinitary x existed, and coinfinite otherwise. The requirement that $W_{f(e)}$ be coinfinite can be split up into the requirements R_x saying that the marker a_x^s comes to a rest, i.e. that $\lim_{s\to\infty} a_x^s$ is finite for every x. However, if x is infinitary then $\lim_s a_x^s = \infty$ and therefore also $\lim_s a_y^s = \infty$ for every $y > x$. That is, the action of enumerating a_x^s is in conflict with R_y. This conflict is resolved by giving the first action *priority* over the second. Thus, if x is infinitary, every requirement R_y with $y > x$ fails to be satisfied. If on the other hand all x are finitary every R_x is satisfied.

In general, the priority method is a method for resolving conflicts between re-

quirements. If in a construction an action taken to satisfy a requirement R_i conflicts with that of a requirement R_j we say that R_j is *injured*. In this chapter we will only consider constructions where the conflicts between requirements are of a mild nature, namely where every requirement is injured only finitely often. Such arguments are called *finite injury* priority arguments. The finite injury method is significantly more powerful than the Kleene-Post finite extension method from Chapter 5, and can be used to answer many questions for which the latter fails to provide an answer. In the next section we discuss the first example of such a question.

6.2 A pair of Turing incomparable c.e. sets

In Theorem 3.6.4 we showed that there are c.e. sets of intermediate m-degree. After proving this Post asked the following famous question [21]:

Post's Problem *Do there exist c.e. sets of intermediate Turing degree? That is, is there a c.e. set that is neither computable nor Turing-complete for the c.e. sets?*

In Theorem 5.4.1 we showed the existence of a pair of Turing incomparable sets. By Exercise 5.6.7 these such pairs exist also below K. If we would succeed in making these sets c.e. we would have settled Post's Problem in the affirmative. This is exactly how Friedberg [5] and Muchnik [17] solved the problem, independently of each other.

Theorem 6.2.1 (Solution to Post's Problem, Friedberg [5] and Muchnik [17]) *There exists a pair of Turing incomparable c.e. sets.*

Proof. We build a pair of incomparable c.e. sets A and B, using the same requirements as in the proof of Theorem 5.4.1: For every e we want to ensure

$$R_{2e}: \quad A \neq \{e\}^B,$$

$$R_{2e+1}: \quad B \neq \{e\}^A.$$

As before, for R_{2e} we need a witness x such that $A(x) \neq \{e\}^B(x)$. We can pick a witness x and initially define $A(x) = 0$, and wait until $\{e\}_s^{B_s}(x)$ converges for some s. If this never happens R_{2e} is automatically satisfied. If $\{e\}_s^{B_s}(x)\downarrow = 1$ we are done, and if $\{e\}_s^{B_s}(x)\downarrow = 0$ we can enumerate x into A so that $A(x) = 1$ and again R_{2e} is satisfied. The difference with Theorem 5.4.1 is that here we have no complete control over the set B. In fact, we know in advance that A and B will be noncomputable, and for a typical noncomputable set one can never know whether maybe later some small element will appear in the set (cf. Proposition 3.2.5). Having satisfied R_{2e} as above, it might happen that R_{2e+1} wants to enumerate some element into B below the use of the computation $\{e\}_s^{B_s}(x)\downarrow$, and thus spoiling it. To resolve such conflicts, we give the requirements the priority ordering

$$R_0 > R_1 > R_2 > R_3 > \dots$$

In case of a conflict we allow the highest priority requirement R_i (i.e. the one with the smallest index i) to proceed, thus possibly injuring lower priority requirements. On the other hand, if $i < j$ we will not allow the low priority requirement R_j to injure a computation of R_i. We now give the formal construction of A and B by describing their enumerations A_s and B_s, $s \in \omega$. To monitor the numbers that requirement R_e wants to keep out of A or B we will use a *restraint function* $r(e, s)$. We will say that

a requirement R_e is *initialized* at a stage s when its current witness $x_{e,s}$ is redefined to a number greater than any number previously used in the construction, and by setting the restraint $r(e,s)$ to 0. We will say that R_{2e} *requires attention* at stage s if since the last stage that R_{2e} was initialized no action for it was taken, and for the current witness $x_{2e,s}$ it holds that $\{e\}_s^{B_s}(x_{2e,s})\downarrow$. Similarly for R_{2e+1}.
Stage $s = 0$. Let $A_0 = B_0 = \emptyset$. Initialize all requirements by setting $x_{e,0} = 0$ and $r(e,0) = 0$ for every e.
Stage $s+1$. Let $i < s$ be minimal such that R_i requires attention. (If no such i exists we do nothing and proceed to the next stage.) Suppose that $i = 2e$ is even. Define

$$r(2e, s+1) = \mathrm{use}\big(\{e\}_s^{B_s}(x_{2e,s})\big)$$

and say that R_{2e} *acts*. If $\{e\}_s^{B_s}(x_{2e,s}) = 0$ define $A_{s+1}(x_{2e,s}) = 1$. Since this action may injure lower priority requirements, we initialize all R_j with $j > i$. The case where i is odd is completely symmetrical, interchanging the role of A and B. This completes the construction.

We verify that all requirements are satisfied. We prove by induction that every requirement acts only finitely often, and is only finitely often initialized. Suppose that s is a stage such that no requirement R_j, $j < i$ acts after stage s. Then R_i is never initialized after stage s, and $\lim_{t\to\infty} x_{i,t} = x_{i,s}$. If R_i never requires attention after stage s then it is automatically satisfied. If R_i requires attention at some stage $t \geqslant s$ then its restraint $r(i,t)$ is set to the use of the computation $\{e\}_t^{B_t}(x_{i,t})$ (in case $i = 2e$ is even). Since all R_j with $j > i$ are initialized at stage t, in particular their witnesses are redefined to be greater than $r(i,t)$, so no action for these requirements can change this computation. Since also no R_j with $j < i$ acts after t, the computation is preserved forever, and hence R_i is satisfied. So we see that every requirement is initialized only finitely often, and afterwards is satisfied either vacuously or by one additional action, and hence also acts only finitely often. □

6.3 Exercises

Exercise 6.3.1 Show that the c.e. m-degrees are not linearly ordered.

Exercise 6.3.2* Show that there is a low simple set A. Note that such a set provides another solution to Post's Problem. (Hint: Use the priority method to build A in stages. Use the characterization from Exercise 5.6.12 to make A low, and use the strategy from Theorem 3.6.4 to make A simple.)

Exercise 6.3.3 Show that there exists an infinite sequence $\{A_i\}_{i\in\omega}$ of c.e. sets that are pairwise incomparable, i.e. such that $A_i|_T A_j$ for every $i \neq j$.

Chapter 7

Applications

Computability theory is a standard ingredient in any course in mathematical logic. Although branches like model theory, set theory, and proof theory have developed into individual subjects of their own, the ties with computability theory remain numerous, and new ones keep on being found. Besides, with the rise of computer science it has become indispensable as a background theory for this relatively new area. Several computer science topics have stemmed from computability theory, such as complexity theory, learning theory, and formal language and automata theory. (The latter is built on yet another characterization of the computable sets, namely as those that can be generated by a *grammar*.) In many ways various of the above areas do not have a clear separation. E.g. in the view of Odifreddi [18, 19], the areas of complexity and computability theory and descriptive set theory are all the same.

In this chapter we briefly indicate three other areas where computability theory is pivotal, namely proof theory, constructivism, and randomness.

7.1 Undecidability in logic

By a *formal system* we will mean any c.e. set $\mathcal{F}$ of logical formulas. This is of course a crude version of the notion of formal system usually employed in mathematics and computer science (with proper axioms and deduction rules and the like) but it will suffice here and make what follows all the more general. By arithmetization we may treat any formal system $\mathcal{F}$ as a set of natural numbers (recall the Gödel numbering from page 35). In particular, we can speak of $\mathcal{F}$ being *decidable* or not. The requirement that $\mathcal{F}$ is c.e. corresponds to the fact that most natural formal systems have a computable set of axioms from which all theorems of the system are deduced, so that the set of deduced formulas is c.e. Such systems are also called *axiomatizable*. If a formula φ is enumerated into $\mathcal{F}$ we say that $\mathcal{F}$ *proves* φ and we call φ a *theorem* of $\mathcal{F}$, and denote this by $\mathcal{F} \vdash \varphi$. The above definitions apply to formal systems in any logical language, but as in Chapter 4 we will concentrate on formal systems in the language of arithmetic. Recall that we use $\omega \models \varphi$ to denote that an arithmetical sentence φ holds in the standard model of arithmetic.

Some essential properties of a logical system are whether what it proves is true, whether it is consistent, and whether it decides the truth of every formula:

Definition 7.1.1 Let $\mathcal{F}$ be a formal system in the language of arithmetic.

- $\mathcal{F}$ is *sound* if $\mathcal{F} \vdash \varphi$ implies that $\omega \models \varphi$.
- $\mathcal{F}$ is *consistent* if there is no formula φ such that both φ and $\neg\varphi$ belong to $\mathcal{F}$.

- $\mathcal{F}$ is *complete* if for every formula φ, either $\mathcal{F} \vdash \varphi$ or $\mathcal{F} \vdash \neg\varphi$. Otherwise $\mathcal{F}$ is called *incomplete*.

Proposition 7.1.2 *A consistent and complete formal system is decidable.*

Proof. This is a form of Proposition 3.2.6. If $\mathcal{F}$ is complete we can enumerate the theorems of $\mathcal{F}$ until either φ or $\neg\varphi$ appears. By consistency of $\mathcal{F}$ at most one can appear, and by completeness at least one will appear. Hence we can compute whether φ is a theorem of $\mathcal{F}$ or not. □

In the rest of this section we will work with *Robinson Arithmetic* $\mathcal{Q}$, which contains the constant 0, the function symbols S, $+$, and $\cdot$, and the following axioms (besides the axioms for first-order predicate logic with equality):

A1. $S(x) = S(y) \to x = y$

A2. $S(x) \neq 0$

A3. $x \neq 0 \to (\exists y)[x = S(y)]$

A4. $x + 0 = x$

A5. $x + S(y) = S(x + y)$

A6. $x \cdot 0 = 0$

A7. $x \cdot S(y) = x \cdot y + x$

Note that the last four axioms are just the recursive relations for $+$ and $\cdot$ we already encountered on page 5. It would be possible to work with systems weaker than $\mathcal{Q}$, but $\mathcal{Q}$ will be convenient for our purposes. The system $\mathcal{PA}$ of *Peano Arithmetic*, which is very often used as a basic system of arithmetic, is obtained from $\mathcal{Q}$ by adding to it the axiom scheme of induction.

A crucial notion is the power that formal systems have to represent functions:

Definition 7.1.3 A function f is *representable* in a formal system $\mathcal{F}$ if there is a formula φ such that for all x and y,

$$\begin{aligned} f(x) = y &\implies \mathcal{F} \vdash \varphi(x, y), \\ f(x) \neq y &\implies \mathcal{F} \vdash \neg\varphi(x, y). \end{aligned}$$

Here the x and y in $\varphi(x, y)$ are shorthand for the terms $S^x(0)$ and $S^y(0)$.

Note that this notion of representability makes sense if $\mathcal{F}$ is consistent.

The following result adds yet another characterization of the notion of computable function to our list.

Theorem 7.1.4 *Every computable function is representable in $\mathcal{Q}$. Conversely, every $\mathcal{Q}$-representable function is computable.*

Proof. Both directions can be proved using arithmetization (cf. Section 2.4). We will not prove this here. A proof can be found in Odifreddi [18, II.2.16]. That every representable function is computable is a straightforward application of arithmetization: arithmetize proofs in $\mathcal{Q}$. That every computable function is representable requires

one more idea, namely the elimination of primitive recursion using coding. This can be done as in the proof of Theorem 2.4.1, except that now we cannot use the coding functions from Section 2.4.1 since these were defined using primitive recursion. Instead we need to define different coding functions not using primitive recursion. Gödel [7] defined such a coding function β using the Chinese Remainder Theorem (cf. Odifreddi [18, I.3.6]). □

The following result is Gödel's celebrated (first) incompleteness theorem. It shows that every formal system is either too weak to contain elementary arithmetic, or incomplete.[1]

Theorem 7.1.5 (The first incompleteness theorem, Gödel [7]) *Any consistent formal system $\mathcal{F}$ extending $\mathcal{Q}$ is undecidable and incomplete.*

Proof. Let $\mathcal{F}$ be as in the theorem, and suppose that $\mathcal{F}$ is decidable. Let $\{\psi_n\}_{n\in\omega}$ be an effective enumeration of all formulas of one variable. Define the diagonal D by

$$n \in D \iff \mathcal{F} \vdash \psi_n(n). \tag{7.1}$$

Since $\mathcal{F}$ is decidable $\overline{D}$ is computable, hence by Theorem 7.1.4 it is representable. Suppose that ψ_e represents $\overline{D}$ in $\mathcal{Q}$. Since $\mathcal{F}$ extends $\mathcal{Q}$, we then have

$$\begin{aligned} n \in \overline{D} &\Longrightarrow \mathcal{Q} \vdash \psi_e(n) \quad \Longrightarrow \mathcal{F} \vdash \psi_e(n) \\ n \in D &\Longrightarrow \mathcal{Q} \vdash \neg\psi_e(n) \Longrightarrow \mathcal{F} \vdash \neg\psi_e(n), \end{aligned}$$

and since $\mathcal{F}$ is consistent we get

$$n \in \overline{D} \iff \mathcal{F} \vdash \psi_e(n).$$

Then for $n = e$ we obtain a contradiction with (7.1). This shows that $\mathcal{F}$ is undecidable. That $\mathcal{F}$ is incomplete now follows from Proposition 7.1.2. □

For consistent formal systems $\mathcal{F}$ extending $\mathcal{Q}$ Gödel also proved that a specific sentence is unprovable in $\mathcal{F}$, namely the sentence $\mathrm{Con}_{\mathcal{F}}$ expressing the consistency of $\mathcal{F}$. This result is called the *second incompleteness theorem.*

By Proposition 3.5.3 the set $\overline{K}$ is Π^0_1-complete. Let $\mathcal{F}$ be a sound formal system. Then it cannot be the case that for every $n \in \overline{K}$ the system $\mathcal{F}$ proves that $n \in \overline{K}$, since then, because the set of theorems of $\mathcal{F}$ is c.e. and $\mathcal{F}$ is sound, $\overline{K}$ would be Σ^0_1. So we see that there are (in fact infinitely many) sentences of the form $n \in \overline{K}$ that are true but that $\mathcal{F}$ cannot prove.

The next result is the negative solution to the Entscheidungsproblem mentioned on page 3:

Theorem 7.1.6 (The unsolvability of the Entscheidungsproblem, Church [2] and Turing [30]) *First-order predicate logic is undecidable.*

Proof. We use that $\mathcal{Q}$ has a finite axiomatization. Let $\bigwedge\mathcal{Q}$ be the conjunction of all axioms of $\mathcal{Q}$. Then we have for any formula φ that

$$\mathcal{Q} \vdash \varphi \iff \vdash \bigwedge\mathcal{Q} \to \varphi,$$

where the $\vdash$ on the right hand side refers to deducibility in first-order logic. Now if first-order logic would be decidable it would follow that also $\mathcal{Q}$ were decidable, contradicting Theorem 7.1.5. □

[1] Gödel originally showed this for so-called ω-consistent theories. Rosser improved this to consistent theories.

7.2 Constructivism

In mathematics one can often prove the existence of objects indirectly by showing that the assumption that they do not exist leads to a contradiction, or by deriving the existence from a disjunction $A \vee \neg A$ without knowing which case obtains. It may easily happen that one can show the existence of something in this way without having a clue about how to obtain a concrete example. In constructive mathematics one tries to prove theorems or construct objects in an explicit way. There are different schools of constructivism, according to the different meanings that can be given to the word "constructive". The school of intuitionism founded by the Dutch mathematician L. E. J. Brouwer proceeded by restricting the means of reasoning from classical logic, by for example not allowing indirect forms of reasoning as in the example above. Another important interpretation is to interpret "constructive" by "computable". The relation between these two approaches to constructivism is an interesting field of study in itself, but in this section we will stick to the latter approach and merely indicate how computability theory can be used to analyze the constructive content of theorems from classical mathematics. We will discuss only a single example, namely Königs Lemma.

Definition 7.2.1 A (binary) *tree* is a subset T of $2^{<\omega}$ that is closed under the subsequence relation $\sqsubseteq$, i.e. if $\tau \sqsubseteq \sigma$ and $\sigma \in T$ then also $\tau \in T$. A tree is *computable* simply if it is computable when coded as a set of numbers (using any computable coding of $2^{<\omega}$). A *path* in T is an infinite branch of the tree, that is a set $A \in 2^{\omega}$ such that $A\restriction n \in T$ for every n.

Theorem 7.2.2 (Königs Lemma) *Every infinite binary tree has a path.*

Proof. Let T be an infinite binary tree. We inductively "construct" a path trough T as follows. Let $\sigma_0 = \emptyset$. Given σ_n such that the set

$$\{\tau \in T : \tau \sqsupseteq \sigma_n\}$$

is infinite, at least one of $\sigma_n{}^\frown 0$, $\sigma_n{}^\frown 1$ has infinitely many extensions on T. Define $\sigma_{n+1} = \sigma_n{}^\frown 0$ if this holds for $\sigma_n{}^\frown 0$, and define $\sigma_{n+1} = \sigma_n{}^\frown 1$ otherwise. Now clearly $\bigcup_n \sigma_n$ is a path in T. □

Königs Lemma holds in more generality for finitely branching trees, i.e. trees that are not necessarily binary but in which still every node has only finitely many successors. The next result shows that in general Königs Lemma is not constructive.

Proposition 7.2.3 (Kleene) *There is an infinite computable binary tree without a computable path.*

Proof. By Exercise 7.4.3, for any disjoint pair A,B of c.e. sets there is a computable tree such that the paths in T are precisely the sets separating A and B. Taking A and B a pair of computably inseparable sets (Theorem 3.3.5) we see that T has no computable paths. □

Now we want to analyze exactly how difficult it is to compute a path in an infinite tree. The following result shows that having K as an oracle suffices.

Proposition 7.2.4 (Kreisel Basis Lemma) *Every infinite computable binary tree has a path in* Δ^0_2.

Proof. For a computable tree, the decision in the proof of Theorem 7.2.2 is computable in K. Namely, σ has only finitely many extensions on T if and only if there exists n such that all $\tau \sqsupseteq \sigma$ of length n are not in T. Thus the constructed path is also K-computable. □

Proposition 7.2.4 can be significantly improved as follows. In Exercise 5.6.11 we introduced the low sets as the sets that have the same jump as computable sets. In this sense they are close to computable sets. Note for example that a low set is not Turing complete since complete sets jump above $\emptyset''$. Now by Proposition 7.2.3 we cannot expect a computable tree to have a computable path, but the next result shows we can get quite close:

Theorem 7.2.5 (Low Basis Theorem, Jockusch and Soare [10]) *Every infinite computable binary tree has a low path.*

Proof. The proof is a magnificent example of the method of tree forcing. A discussion of this method, however, would carry us too far at this point. A proof can be found in Odifreddi [18] or Soare [28]. □

We remark that all the results of this section can be relativized by increasing the complexity of the tree, so that they are not only about computable trees but really state how difficult it is in general to compute paths in *any* kind of binary tree.

The preceding sequence of results illustrates how one often can analyze the constructive content of mathematical theorems using measures and notions from computability theory. Notice that we were not only able to conclude that Königs Lemma is not constructive, but also obtained a precise bound on how nonconstructive it is in terms of Turing degrees.

Another field where the strength of mathematical theorems is studied is the area of reverse mathematics. Although here the main tools of study are axiomatic, computability theory also plays an important role. Many other examples such as the one above can be found in the book by Simpson [27].

7.3 Randomness and Kolmogorov complexity

According to classical probability theory every string of a given length n is just as probable as another. Yet we feel that there are great qualitative differences between strings. E.g. the sequence $000\ldots0$ consisting of 100 zeros appears to us as special among all strings of length 100. This is part of the motivation for introducing a complexity measure for individual strings and reals. A general introduction to this area is Li and Vitányi [15].

Fix a universal Turing machine U. (That is, a Turing machine that can simulate all other Turing machines. Such machines exist for the same reason that universal c.e. sets exist, cf. page 21.) Given a string $\sigma \in 2^{<\omega}$, define the *Kolmogorov complexity* of σ as

$$C(\sigma) = \min\{|\tau| : U(\tau) = \sigma\}.$$

We list some basic facts of C:

- Note that C depends on the choice of the universal machine U. However the choice of U matters only an additive constant in the theory: For any two universal machines U and V and the corresponding complexity functions C_U and

C_V we have that $C_V(\sigma) \leqslant C_U(\sigma) + O(1)$ for every σ, where the constant $O(1)$ depends only on U and V. Since the choice of U does not matter much, we will henceforth simply write C instead of C_U.

- Since every string is a description of itself, we have that the complexity of any string is bounded by its length. That is, for all σ we have

$$C(\sigma) \leqslant |\sigma| + O(1).$$

Now the idea is that a *random* string is a string that lacks structure that could be used to obtain a short description of it. For example the string $000\ldots0$ consisting of 1000 zeros is quite long but is easy to describe. On the other hand, a sequence obtained by 1000 coin flips will probably not have a description that is much shorter than 1000 bits. So we define:

Definition 7.3.1 A string σ is *k-random* if $C(\sigma) \geqslant |\sigma| - k$.

Proposition 7.3.2 *For every k there exist k-random strings of any length $n > k$.*

Proof. This can be shown by simple counting: For any given length $n > k$, there are

$$\sum_{i=0}^{n-k-1} 2^i = 2^{n-k} - 1$$

programs of length $< n - k$, so there are at least $2^n - 2^{n-k} + 1$ k-random strings of length n. □

In Theorem 3.6.4 we constructed a simple set by brute force. The next result shows that there are also *natural* examples of simple sets:

Proposition 7.3.3 (Barzdin, cf. [32]) *For any k, the set of non-k-random strings is simple.*

Proof. We leave it to the reader to verify that the set of nonrandom strings is c.e. (Exercise 7.4.6). Note that its complement is infinite by Proposition 7.3.2. We have to show that the set of k-random strings is immune, i.e. does not contain any infinite c.e. subsets. Suppose that it is not immune. Then by Proposition 3.6.1 it contains an infinite computable subset, from which it follows that for every m we can compute a string $\psi(m)$ with $C(\psi(m)) \geqslant m - k$. But then $m - k \leqslant C(\psi(m)) \leqslant \log m + O(1)$, which can only be true for finitely many m, a contradiction. □

Corollary 7.3.4 *1. C is not a computable function.* (Attributed to Kolmogorov in [32].)

2. The function $m(x) = \min\{C(y) : y \geqslant x\}$ is unbounded.

3. m grows slower than any monotone unbounded computable function.

Proof. See Exercise 7.4.7. That m is unbounded is because we run out of short programs. That it grows slower than any computable function follows with an argument similar to that of Proposition 7.3.3. □

7.4 Exercises

Exercise 7.4.1 Let "$e \in$ Tot" abbreviate the formula $\forall x \exists s\ \{e\}_s(x)\downarrow$. Let $\mathcal{F}$ be a sound formal system in the language of arithmetic. Show that there are $e \in$ Tot for which $\mathcal{F} \nvdash e \in$ Tot.

Exercise 7.4.2 Prove the result of Exercise 7.4.1 by constructing such an e by direct diagonalization. (Hint: Let $\{e\}(x)\downarrow$ for every x for which $\mathcal{F}$ does not prove in x steps that $e \in$ Tot. As soon as $\mathcal{F}$ proves that $e \in$ Tot let $\{e\}$ be undefined on all further arguments.)

Exercise 7.4.3 Let A,B be any pair of disjoint c.e. sets. Show that there is a computable tree T such that the sets that are a separation of A and B are precisely the paths in T.

Exercise 7.4.4 Let $\mathcal{F}$ be a formal system. A *complete extension* of $\mathcal{F}$ is any set separating $\{\varphi : \mathcal{F} \vdash \varphi\}$ and $\{\varphi : \mathcal{F} \vdash \neg\varphi\}$.

Show that there is a complete extension of $\mathcal{F}$ of low degree. (Hint: Use the previous exercise and the low basis theorem.) For $\mathcal{F}$ a strong formal system such as $\mathcal{Q}$ or $\mathcal{PA}$ this shows that, although the set $\{\varphi : \mathcal{F} \vdash \varphi\}$ is Σ^0_1-complete, there is always a complete extension that has incomplete Turing degree.

Exercise 7.4.5 (a) Show that there is a computable tree T such that $[T]$ is equal to the set of complete and consistent extensions of $\mathcal{PA}$.

(b) Show that $[T]$ does not have any c.e. paths. (Hint: Gödel.) Note that this strengthens Proposition 7.2.3, and also shows that the low extension from Exercise 7.4.4 in general cannot be c.e.

Exercise 7.4.6 Show that for any k the set of k-random strings is Π^0_1.

Exercise 7.4.7 Verify the facts of Corollary 7.3.4. Hint: For part 3, suppose that h is a computable, monotone, and unbounded function. We have to show that $\forall^\infty x\ m(x) < h(x)$. Suppose that $\exists^\infty x\ m(x) \geqslant h(x)$. Define $F(n) = \min\{x : h(x) \geqslant n+1\}$. Show that if $m(x) \geqslant h(x) = n$ then $C(F(n)) \geqslant n$. Thus $C(F(n)) \geqslant n$ i.o. But since F is computable, we also have $C(F(n)) \leqslant \log n + O(1)$, a contradiction.

Further reading

The first textbook solely devoted to computability theory was Rogers [23]. Although it is now outdated in several respects, it is still a good read, and perfectly acceptable for use as an introduction. Another, but much shorter and advanced, early reference that was quite influential is Sacks [24]. The monumental two-volume work by Odifreddi [18, 19] is the most extensive and broadest treatment of the subject today. The number of topics discussed in this book and the ease of presentation are quite astounding. The encyclopaedic character of the book may render it less suited for an introduction, but as reference it has become indispensable. Soare's book [28] was written as a modern replacement of Rogers' book. It is an excellent reference for the more advanced topics in computability theory, with a focus on the treatment of the c.e. Turing degrees and the priority method. Sacks [25] treats computability theory on ordinals other than ω. In Chapter 7 we have already mentioned various references for applications of computability theory. An extensive collection of survey papers on computable mathematics can be found in [4]. Numerous other treatments of computability theory, often in tandem with subjects like complexity- or automata theory, can be found in any mathematics or computer science library. One can of course also always consult the original papers by the pioneers of the subject. In this case these include the ground-breaking papers by Gödel [7], Church [2], Kleene [11], Turing [30], and Post [21].

Bibliography

[1] G. Cantor, *Über eine Eigenschaft des Inbegriffs aller reellen algebraischen Zahlen*, J. Math. 77 (1874) 258–262.

[2] A. Church, *A note on the Entscheidungsproblem*, Journal of Symbolic Logic 1 (1936) 40–41.

[3] R. Dedekind, *Was sind und was sollen die Zahlen?*, Braunschweig, 1888.

[4] Yu. L. Ershov, S. S. Goncharov, A. Nerode, J.B. Remmel, and V. W. Marek (eds.), *Handbook of Recursive Mathematics*, Studies in logic and the foundations of mathematics Vol's 138 and 139, North-Holland, 1998.

[5] R. M. Friedberg, *Two recursively enumerable sets of incomparable degrees of unsolvability (solution of Post's problem, 1944)*, Proc. Nat. Acad. Sci. 43 (1957) 236–238.

[6] R. M. Friedberg, *A criterion for completeness of degrees of unsolvability*, Journal of Symbolic Logic 22(1957) 159–160.

[7] K. Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatshefte für Mathematik und Physik 38 (1931) 173–198.

[8] D. Hilbert, *Mathematische Probleme*, Proc. Int. Congr. Math. (1900) 58–114.

[9] D. Hilbert and W. Ackermann, *Grundzüge der theoretischen Logik*, Springer-Verlag, 1928.

[10] C. G. Jockusch, Jr. and R. I. Soare, Π^0_1 *classes and degrees of theories*, Transactions of the American Mathematical Society 173 (1972) 35–56.

[11] S. C. Kleene, *General recursive functions of natural numbers*, Math. Ann. 112 (1936) 727–742.

[12] S. C. Kleene, *On notations for ordinal numbers*, Journal of Symbolic Logic 3 (1938) 150–155.

[13] S. C. Kleene and E. L. Post, *The upper semi-lattice of degrees of recursive unsolvability*, Annals of Mathematics, Ser. 2, Vol. 59 (1954) 379–407.

[14] G. W. Leibniz, Dissertatio de arte combinatoria, 1666.

[15] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*, Springer-Verlag, 1993.

[16] Yu. V. Matijasevich, *Enumerable sets are Diophantine*, Dokl. Acad. Nauk. 191 (1970) 279–282.

[17] A. A. Muchnik, *On the unsolvability of the problem of reducibility in the theory of algorithms*, Dokl. Akad. Nauk SSSR, N.S. 108 (1956) 194–197 (Russian).

[18] P. G. Odifreddi, *Classical recursion theory*, Vol. 1, Studies in logic and the foundations of mathematics Vol. 125, North-Holland, 1989.

[19] P. G. Odifreddi, *Classical recursion theory*, Vol. 2, Studies in logic and the foundations of mathematics Vol. 143, North-Holland, 1999.

[20] J. C. Owings, *Diagonalization and the recursion theorem*, Notre Dame Journal of Formal Logic 14 (1973) 95–99.

[21] E. L. Post, *Recursively enumerable sets of positive integers and their decision problems*, Bull. Amer. Math. Soc. 50 (1944) 284–316.

[22] E. L. Post, *Degrees of recursive unsolvability*, Bull. Amer. Math. Soc. 54 (1948) 641–642.

[23] H. Rogers Jr., *Theory of recursive functions and effective computability*, McGraw-Hill, 1967.

[24] G. E. Sacks, *Degrees of unsolvability*, Annals of Mathematics Studies 55, Princeton University Press, 1963.

[25] G. E. Sacks, *Higher recursion theory*, Springer-Verlag, 1990.

[26] J. R. Shoenfield, *On degrees of unsolvability*, Annals of Mathematics 69 (1959) 644–653.

[27] S. G. Simpson, *Subsystems of second-order arithmetic*, Springer-Verlag, 1999

[28] R. I. Soare, *Recursively enumerable sets and degrees*, Springer-Verlag, 1987.

[29] A. Tarski, *Der Wahrheitsbegriff in den formalisierten Sprachen*, Studia Phil. 1 (1936) 261–405.

[30] A. M. Turing, *On computable numbers with an application to the Entscheidungsproblem*, Proc. London Math. Soc. 42 (1936) 230–265, corrections ibid. 43 (1937) 544–546.

[31] A. M. Turing, *Systems of logic based on ordinals*, Proc. London Math. Soc. 45 (1939) 161–228.

[32] A. K. Zvonkin and L. A. Levin, *The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms*, Russian Math. Surveys 25 (1970), 83–124.