5 Computability

In the next three chapters, we take a look at computability theory: the study of what can and what can't be computed by a mechanical algorithm. This will allow us to show that there is no algorithm for deciding whether a first-order sentence is valid. It will also provide a basis for proving Gödel's incompleteness theorems.

5.1 The Entscheidungsproblem

Suppose you wonder whether a certain first-order sentence is (logically) valid. You might try to construct a proof of the sentence in the first-order calculus. By the soundness of the calculus, such a proof would establish that the sentence is valid. By the completeness of the calculus, there is a proof for any valid sentence. But how can you find such a proof? How do you know where to start? Is there a general algorithm for finding a proof – a recipe that you can follow mechanically, without relying on insight or intuition, that is guaranteed to find a proof if there is one?

There is. A proof is a finite sequence of sentences. We can go through all these sequences, one by one, until we find a proof of the target sentence.

Let me spell out this algorithm in more detail. I assume that we're dealing with a countable first-order language (although this isn't essential for the algorithm). We begin by assigning to each symbol in the language a natural number that represents its position in some fixed "alphabetical" order. I'll call this the *code number* of the symbol.

For each natural number n, there are only finitely many strings with length n, made up of symbols whose code number is at most n. The algorithm goes through all these strings, for increasing values of n. In the first stage, we generate all strings of length 1 made of symbols whose code number is at most 1. (There is only one such string.) In the second stage, we generate all strings of length 2 made of symbols whose code number is at most 2. And so on.

Whenever we have generated a string, we check if it is a proof of the target sentence. That is, we check if the generated string divides into sentences (separated by, say, a comma) in such a way that (i) each sentence is either an instance of A1–A7 or follows

from previous sentences by MP or Gen, and (ii) the last sentence is the target sentence. This is a simple, mechanical task.

If a sentence has a proof, this algorithm will eventually find it. (Needless to say, the algorithm is terribly inefficient. There are much better algorithms. I've implemented one that runs in your web browser: see www.umsu.de/trees/. But efficiency is not our current concern.)

What if a sentence doesn't have a proof, because it isn't valid? Then the algorithm I've described will run forever. It will search through longer and longer strings of symbols, and never find a proof.

So we don't yet have an algorithm for deciding whether a sentence is valid. We have, in effect, an algorithm that outputs 'yes' whenever the sentence to which it is applied is valid; but it doesn't output 'no' when the sentence is invalid. Instead, the algorithm then runs forever. Can we do better? Can we find an algorithm that always outputs either 'yes' or 'no', depending on whether the input sentence is valid or not? This is David Hilbert's *Entscheidungsproblem* ("decision problem"), raised in Hilbert and Ackermann's monograph *Grundzüge der Theoretischen Logik* in 1928.

Suppose, for a moment, that we had such an algorithm. More generally, suppose we had an algorithm for deciding whether a first-order sentence is entailed by a given set of axioms. If we then had a complete axiomatization of some mathematical area, all questions in that area could be answered mechanically. In 1928, it seemed plausible that all areas of mathematics could be completely axiomatized, so that all truths about them could be derived from the relevant axioms. With an algorithm for deciding validity and entailment, we would then have a mechanical algorithm for answering all mathematical questions. In principle, although perhaps not in practice, all of mathematics would reduce to simple mechanical calculation. No insight or intuition or brilliance would be required any more. This vision was articulated by Leibniz in the 17th century. In 1928, it seemed within reach.

So, is there an algorithm for deciding whether any given first-order sentence is valid? The answer was established by Alonzo Church and Alan Turing in 1936: no. First-order logic is, as we say, *undecidable*.

How could one prove this? It is obviously not enough to show that this or that algorithm doesn't do the job. One needs to prove that no algorithm does the job. This requires developing a precise and general concept of an algorithm. Hilbert's Entscheidungsproblem thereby led to the development of computability theory: the study of what can and what can't be computed by a mechanical algorithm.

Exercise 5.1 Explain why the following problems are all equivalent: (a) decide whether a first-order sentence is valid, (b) decide whether a sentence is provable in the first-order calculus, (c) decide whether a first-order sentence is satisfiable (true in some model), (d) decide whether a first-order sentence is consistent (one can't derive a contradiction from it in the first-order calculus).

Exercise 5.2 If a sentence isn't valid, it has a countermodel – a model in which it is false. Why can't we solve the Entscheidungsproblem by simultaneously searching for a proof and a countermodel? (The countermodel search would systematically look through all models and check if the target sentence is true or false, by going through the recursive definition of truth in a model.)

5.2 Computable functions

Let's try to get clearer about what we mean by an algorithm. In a sense, it's trivial that for every mathematical question there is an algorithm that gives the answer. Let Q be a question and A its answer. Here is an algorithm for answering Q: write down A. For example, if Q is '134 times 97?', the algorithm for answering Q is to write down '12,998'. No calculation required.

But that's not really what we mean by an algorithm. An algorithm doesn't just provide the answer to a single question. An algorithm is an instruction for finding the answer to every question of a certain type. Typically, there are infinitely many questions of that type. An algorithm for multiplication, for example, is an instruction by which one can find the answer to every 'x times y?' question. More generally, an algorithm takes inputs and produces an output. Any such algorithm computes a *function*: a function from the inputs to the outputs. So we'll understand an algorithm as a recipe or instruction for computing a function. The task of developing a precise notion of an algorithm turns into the task of developing a precise notion of *computable functions*: functions for which there is a recipe by which one can compute the function's value for any input.

The recipe must meet certain conditions. It must be precise and determinate, so that it can be followed mechanically, without relying on human judgement or insight. It must be specified in a finite way that is fixed in advance, without depending on the input. It must not invoke outside sources of information.

In school, you learned such algorithms for addition and multiplication. These functions are computable. But note that neither you nor any computer is actually able to add

or multiply arbitrarily large numbers. At some point, you'd run out of paper and energy; the computer would run out of memory. The concept of computability that we're trying to capture is *in principle computability*, setting aside practical limitations of memory, time, paper, patience, and pencils.

In the previous section, I described an algorithm for finding proofs. When given a valid sentence, the algorithm returns a proof. When given an invalid sentence, it runs forever. The algorithm therefore computes a partial function: it doesn't return an output for every input. Let's stipulate that this is the correct way of computing partial functions: if a function is undefined for a certain input, an algorithm for computing the function must run forever when given that input. (In practice, we often let algorithms return a special 'undefined' value: when asked to divide a number by zero, you wouldn't spend the rest of your life trying to compute the answer, which you know doesn't exist. Strictly speaking, you are not computing the division function, which is partial, but a modified total function that returns 'undefined' for division by zero.)

Remember that functions are individuated "extensionally" by which outputs they return for which inputs. The same function can always be presented in many ways. If a function is presented in a peculiar way, we may not know *which* algorithm computes it, but as long as there is such an algorithm, the function is computable. For example, the function on $\mathbb{N}$ given by

$$f(x) = \begin{cases} 0 \text{ if Julius Caesar liked cheese} \\ 1 \text{ otherwise} \end{cases}$$

is trivially computable.

Exercise 5.3 Show that this function is computable by specifying two algorithms, one of which is sure to compute the function.

My definition of computability still looks vague. What, exactly, are "precise and determinate" instructions that "can be followed mechanically"? This is what logicians had to figure out in the 1930s.

They came up with a number of different suggestions. Alonzo Church suggested that the computable functions (on the natural numbers, at least) are precisely the functions that are definable in his lambda-calculus. Stephen Kleene, drawing on work by Gödel and Herbrand, suggested that the computable functions are those that can be defined by a certain recursive process that we'll study in chapter 7. More convincingly, Alan Turing suggested that a function is computable iff it is computed by a certain abstract model of a

mechanical computing device, now known as a "Turing machine". We'll look at Turing machines in chapter 6.

These suggestions turned out to be equivalent, in the sense that they define the very same class of functions. Later attempts to define computability in terms of register machines, Post systems, Markov algorithms, or combinatory definability also led to the same class of functions. Moreover, nobody has ever presented a function that is computable by the informal definition I gave above but not by one of these formal definitions. There are strong reasons to think that no such function exists.

We thus have a remarkable case where a seemingly vague concept turns out not to be vague at all. The concept of a "mechanically computable" function seems to pick out precisely the functions that are, say, computable by a Turing machine or definable in the lambda calculus.

The hypothesis that our informal concept of mechanical computability coincides with these formal definitions is known as *Church's Thesis*, or as the *Church-Turing Thesis*. It is a "Thesis" rather than a theorem because it does not admit a mathematical proof. (A rigorous proof would first require a mathematically precise definition of 'mechanically computable'.)

If we want to prove that there is no algorithm for computing a certain function, we generally need to invoke the Church-Turing Thesis. Consider, for example, the function that returns 'yes' for every valid first-order sentence and 'no' for every invalid one. An algorithm for computing this function would solve the Entscheidungsproblem. In chapters 6 and **??**, we'll prove that there is no such algorithm. But all we can actually prove is that the function isn't computable in any of the formal senses mentioned above. We can prove, for example, that no Turing machine can compute the function. From this, we will infer "by the Church-Turing Thesis" that there is no algorithm for solving the Entscheidungsproblem.

Exercise 5.4 We could avoid having to appeal to the Church-Turing Thesis by *defining* 'mechanically computable' as, say, 'definable in the lambda calculus'. Why would this be a bad idea? (You don't need to know anything about the lambda calculus to answer the question.)

Besides these *unavoidable* appeals to the Church-Turing Thesis, we will also occasionally make *avoidable* or *lazy* appeals to the Thesis. If a particular function is obviously computable, we sometimes won't bother proving that it is computable in any of the formal senses. For example, we might say that "by the Church-Turing Thesis", the multiplica-

tion function (which is obviously computable) is computable by a Turing machine. This appeal to the Church-Turing Thesis is avoidable because we could actually prove that there is a Turing machine that computes the function. (I will, incidentally, display such a machine in chapter 6.) But this would be tedious, and we can save the effort by relying on the overwhelming evidence in favour of the Church-Turing Thesis.

Exercise 5.5 Explain (informally) why, if there is an algorithm for computing two one-place functions f and g then there is also an algorithm for computing the function h given by $h(x) = f(g(x))$.

5.3 Uncomputable functions

I've mentioned – so far without proof – that the function that takes a first-order sentence as input and returns 'yes' or 'no' depending on whether the sentence is valid or not is uncomputable. Are there other uncomputable functions?

Let's think about functions that take one or more natural numbers as input and return a natural number as output. Are all such functions computable? Any example function that you might come up with (addition, multiplication, factorial, the n-th prime, etc.) is almost certainly computable. We can show, however, that there must be uncomputable functions on the natural numbers. In fact, it follows from simple cardinality considerations that *most* functions on the natural numbers are uncomputable.

How many functions are there from $\mathbb{N}$ to $\mathbb{N}$? Focus, for a start, on functions from $\mathbb{N}$ to the set $\{0, 1\}$. Every such function corresponds to a unique set of natural numbers: the set of numbers that the function maps to 1. Conversely, every set of natural numbers corresponds to a unique such function. That is, there is a bijection between the functions from $\mathbb{N}$ to $\{0, 1\}$ and the sets of natural numbers. By Cantor's theorem, there are uncountably many sets of natural numbers. So there are also uncountably many functions from $\mathbb{N}$ to $\{0, 1\}$. (One can also show that there is a bijection between the functions from $\mathbb{N}$ to $\{0, 1\}$ and the functions from $\mathbb{N}$ to $\mathbb{N}$. So the set of functions from $\mathbb{N}$ to $\mathbb{N}$ has the cardinality of $\mathcal{P}(\mathbb{N})$. But what matters is that it is uncountable.)

The set of algorithms for functions on $\mathbb{N}$, on the other hand, is countable. I've said that an algorithm must be specifiable in a finite way. So each algorithm can be given as a finite string of symbols. Moreover, we don't need uncountably many primitive symbols to define an algorithm for manipulating numbers. Since there are only countably many finite strings of symbols in a countable language, it follows that there are only countably many algorithms for functions on $\mathbb{N}$.

If the set of functions from $\mathbb{N}$ to $\mathbb{N}$ is uncountable and the set of algorithms is countable, it follows that uncountably many functions from $\mathbb{N}$ to $\mathbb{N}$ are not computable by any algorithm.

By itself, this isn't yet a serious blow to Hilbert's (and Leibniz's) dream that all of mathematics might be reduced to mechanical calculation. Most functions on $\mathbb{N}$ have no mathematical significance. They can't be defined in the language of arithmetic, or even in the language of set theory. If we can't even ask a question, we probably shouldn't worry if there is no algorithm for finding the answer.

Exercise 5.6 Explain why most functions from $\mathbb{N}$ to $\mathbb{N}$ can't be defined in the language $\mathfrak{L}_A$ of first-order arithmetic.

The finite specifiability of algorithms creates a puzzle that will lead us to a key result in computability theory, and also to a concrete example of an uncomputable function.

Let's still focus on algorithms for computing functions on $\mathbb{N}$. (We'll see in section 5.5 why this is not a serious restriction.) Any such algorithm can be written down as a finite string of symbols, in some suitable language. That language may be a restricted part of English, or a programming language like Python or JavaScript, or a special language for defining Turing machines, as will be explained in chapter 6. Any of them will do. For any sensible choice of such a language, there will be a mechanical way of checking whether a given string of symbols (in the language) specifies an algorithm. It follows that we can mechanically go through all algorithms, one by one, just as we can go through all proofs in the first-order calculus.

Now consider the following algorithm – I'll call it the *antidiagonal algorithm*. For any input number n, the antidiagonal algorithm generates the list of all algorithms (for functions on $\mathbb{N}$) up to the n-th entry: $A_1, A_2, \ldots, A_n$. It then runs the n-th algorithm A_n with input n and returns the output plus 1.

Think of the algorithms and their outputs arranged in a table:

Algorithm	0	1	2	3	⋯
A1	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	⋯
A2	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	⋯
A3	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	⋯
⋮	⋮	⋮	⋮	⋮	

$x_{1,0}$ is the output of algorithm $A1$ for input 0, $x_{2,3}$ is the output of algorithm $A2$ for input 3, and so on. The antidiagonal algorithm takes an input n, goes to the n-th row, then computes the value $x_{n,n}$ in the n-th column of that row, and returns this value plus 1.

This algorithm evidently computes a function on $\mathbb{N}$: it takes a number as input and returns a number as output. So it must be somewhere on the list of algorithms $A_1, A_2, A_3, \ldots$. Suppose it is the n-th algorithm on the list, for some n. What is the output of the algorithm for input n? By construction, the algorithm returns the output of the n-th algorithm for input n plus 1. But it *is* the n-th algorithm. So the output of the antidiagonal algorithm for input n is the output of the antidiagonal algorithm for input n plus 1. This is a contradiction.

What went wrong? The argument is a reductio, but what does it refute? You will have noticed that the argument closely resembles Cantor's proof that the set of sets of natural numbers is uncountable. Does it show that the set of algorithms (for functions on $\mathbb{N}$) is uncountable after all?

No. The set of algorithms really is countable. But it's true that the antidiagonal algorithm can't be on the list of algorithms. It's not on the list because it isn't a well-defined algorithm. Can you see why?

The problem is that we've allowed for algorithms that may run forever on certain inputs. Suppose some algorithm A_n on the list of algorithms runs forever when given input n. Then we can't add 1 to the output of A_n for input n, because there is no such output: $x_{n,n}$ is undefined. My definition of the antidiagonal algorithm assumed that each algorithm A_n returns an output for input n, which need not be the case.

Let's fix this bug. Let's change the antidiagonal algorithm to work as follows. Given any input n, we run the n-th algorithm on input n, as before. *If that algorithm returns an output* $x_{n,n}$, we return $x_{n,n} + 1$. But if the n-th algorithm doesn't return anything for input n, we return 0.

This *revised antidiagonal algorithm* doesn't assume that algorithms always return an output. But the above argument still goes through: the revised algorithm can't be on the list of algorithms. It is still not a genuine algorithm. Why not?

Think about how we might implement the algorithm. We get a number n as input. It's not hard to enumerate the first n algorithms. Having identified the n-th algorithm, we now want to run the n-th algorithm on input n. But what do we do if this runs forever? If we simply wait for the output, our implementation will also run forever. It won't return 0, as required. To implement the revised antidiagonal algorithm, we therefore need to implement a subroutine to check whether a given algorithm halts on a given input. If such a subroutine exists, we can implement the revised antidiagonal algorithm: when given input n, we can use the subroutine to check if the n-th algorithm halts on input n;

if no, we output 0; if yes, we run the n-th algorithm until it returns an output, then we return that output plus 1.

It's not obvious, however, whether we can find an algorithm for checking whether a given algorithm halts on a given input. In fact, there is no such algorithm. We know this *because otherwise the revised antidiagonal algorithm could be implemented*: it would be a genuine algorithm. It would be on the list of algorithms. And we know that this leads to a contradiction.

By this curious line of reasoning, we've established the following key result in computability theory: *There is no general algorithm for checking whether a given algorithm halts on a given input.*

As promised above, we also get a concrete example of an uncomputable function on $\mathbb{N}$. Fix some "alphabetical" order on the algorithms for functions on $\mathbb{N}$. Given any such ordering $A_1, A_2, \ldots$, we can define an antidiagonal function d by

$$d(n) = \begin{cases} 0 & \text{if } A_n \text{ runs forever on input } n \\ x + 1 & \text{if } A_n \text{ returns } x \text{ on input } n. \end{cases}$$

This is the function that the revised antidiagonal algorithm was supposed to compute. The *function* exists, but the algorithm doesn't: the function d is uncomputable.

Exercise 5.7 Explain why there is no mechanical way to enumerate the *total* functions on $\mathbb{N}$.

Exercise 5.8 Show that every total non-increasing function on $\mathbb{N}$ is computable. A function f is non-increasing if, for all x, $f(x) \geq f(x + 1)$.

5.4 Decidable and semidecidable sets

Hilbert's Entscheidungsproblem is the problem of deciding, for any first-order sentence, whether it is valid or not. We can generalize this concept. In contemporary terminology, a *decision problem* is a task of deciding, for any object of a certain type, whether it has or lacks a certain property. In the case of the Entscheidungsproblem, the objects are first-order sentences and the property of interest is validity. Another decision problem is to decide for any natural number whether it is prime, or for any graph whether it can be coloured with three colours. There are infinitely many decision problems.

A *solution* to a decision problem is an algorithm that takes an object of the relevant type as input and returns either 'yes' or 'no', depending on whether the object has the property or not.

We have to clarify what, exactly, this means. Consider the property of being a spouse of Julius Caesar. Is there an algorithm for deciding whether a given person has this property? In one sense, yes, in another, no. I said that an algorithm must not invoke outside sources of information. One needs empirical information to decide whether a given person is a spouse of Julius Caesar. In this sense, there is no algorithm for deciding the property. On the other hand, consider the algorithm that returns 'yes' for Cornelia, Pompeia, and Calpurnia, who were, in fact, Caesar's spouses, and 'no' for everyone else. This algorithm correctly decides for any given person whether they are a spouse of Caesar, without invoking outside sources of information. But the algorithm only works contingently: it only works in worlds like ours, where Caesar had exactly these three spouses.

Let's say that an algorithm decides a property *extensionally* if it correctly classifies every object in the actual world, even if it misclassifies objects in other possible worlds. To decide a property extensionally is really to decide whether an object belongs to a certain set: to the property's extension. Henceforth, when I talk about deciding properties, I always mean deciding them extensionally (although the present complication won't often arise, because we'll mostly be concerned with mathematical properties whose extension doesn't vary from world to world).

In computability theory, it is common to speak directly of deciding sets. An algorithm *decides a set* if it returns 'yes' for every object in the set and 'no' for every other object. A set is *decidable* if there is an algorithm that decides it.

Decidability is closely related to computability. An algorithm for deciding a set computes a function that takes objects of a relevant type as input and outputs 'yes' for objects in the set and 'no' for objects not in the set. This function is called the *characteristic function* of the set. Officially, the outputs are often taken to be 1 and 0 rather than 'yes' and 'no'. That is, the characteristic function f_S of a set S is defined by

$$f_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S. \end{cases}$$

The connection between decidability and computability can now be stated as follows: a set is decidable iff its characteristic function is computable.

Exercise 5.9 Explain why every finite set is decidable.

Exercise 5.10 Are there undecidable sets? Explain briefly.

Exercise 5.11 Show that if a set S of natural numbers is decidable, then so is its complement $\bar{S}$, i.e., the set of natural numbers not in S.

We can generalize the concept of decidability to relations. An n-ary relation R is (extensionally) decidable if there is an algorithm that outputs 'yes' for every n-tuple of objects that stand in the relation and 'no' for every n-tuple of objects that don't.

An important example of a decidable relation is the relation that holds between a sequence $A_1, \ldots, A_n$ of first-order sentences and a first-order sentence B iff $A_1, \ldots, A_n$ is a proof of B in the first-order calculus. I relied on the decidability of this relation when I described the algorithm for finding proofs in section 5.1: I noted that there is a mechanical algorithm for checking whether a given sequence of sentences $A_1, \ldots, A_n$ is a proof of a sentence B in the first-order calculus. We only need to check that each sentence in $A_1, \ldots, A_n$ is either an axiom or follows from previous sentences by MP or Gen, and that the last sentence A_n is the target sentence B. The decidability of the proof relation is not an accidental feature of our calculus. It is a critical property of proof systems in general. In any acceptable proof system, there should be a mechanical procedure by which, say, a student or computer can check (or *verify*) that a purported proof is really a proof of the target sentence. No brilliance or ingenuity should be required for this task.

Consider now the *halting relation* that holds between an algorithm and an input (say, a number) iff the algorithm halts when given that input. We know that this relation is not decidable: there is no algorithm for deciding whether a given algorithm halts on a given input. On the other hand, there is an algorithm for listing all algorithms that halt on a given input n. We know that we can mechanically enumerate all algorithms, in some order $A_1, A_2, A_3, \ldots$. For each number $i = 1, 2, 3, \ldots$, we can therefore take the first i algorithms $A_1, \ldots, A_i$ and apply them to input n, letting them run for i steps. (Every algorithm can be divided into steps; it doesn't matter how exactly these are defined.) That is, we start by running A_1 on n for a single step. Then we run A_1 and A_2 on n for two steps (after one another, say). Then we run A_1, A_2, and A_3 on n for three steps, and so on. Whenever an algorithm returns an output, we add it to the list of algorithms that halt on input n. This way, every algorithm that halts on input n will eventually be listed.

So, even though there is no algorithm for deciding whether an arbitrary algorithm halts on input n, there is an algorithm for listing all and only the algorithms that do halt on input n. The property of halting on input n is not decidable, but it is *semidecidable*.

In general, a property is (extensionally) *semidecidable* if there is a mechanical procedure for listing all objects that have the property. Equivalently, there is an algorithm that outputs 'yes' for every object that has the property and never outputs 'yes' for an object that doesn't have the property.

Semidecidable properties are also called *computably enumerable*, or *recursively enumerable*, or just *r.e.* I'll mostly use the term 'computably enumerable'.

As with decidability, the concept of semidecidability, or computable enumerability, can be generalized to relations and to sets. A set is computably enumerable if there is an algorithm for listing all its elements.

Exercise 5.12 Explain why the set of valid first-order sentences is computably enumerable.

Exercise 5.13 Explain why every decidable set is computably enumerable.

The following propositions state some easy connections between decidability and computable enumerability.

Proposition 5.1: (Kleene's Theorem)

If a set and its complement are both computably enumerable then the set is decidable.

Proof. Let S be a set such that both S and its complement $\bar{S}$ are computably enumerable: there are mechanical procedures for listing the elements of S and of $\bar{S}$. We can use these to define an algorithm for deciding S: Given any object x, we run the two procedures in alternation, listing the first element of S, then the first element of $\bar{S}$, then the second element of S, then the second element of $\bar{S}$, and so on. At some stage, we must find x in either of the two lists. If it shows up in the list of elements of S, we return 'yes'. If it shows up in the list of elements of $\bar{S}$, we return 'no'. □

Proposition 5.2

If R is a decidable (binary) relation on $\mathbb{N}$, then the set of all y such that $\exists x R(x, y)$ is computably enumerable.

(By '$\exists x R(x, y)$' I mean 'there is a number x such that R holds between x and y'. I occasionally use expressions from first-order logic in the meta-language when it is convenient.)

Proof. Here is an algorithm for listing all y such that $\exists x R(x, y)$. At step 1, compute whether $R(0, 0)$ holds. At step 2, compute $R(0, 1)$ and $R(1, 0)$. In general, at each step k, compute $R(x, y)$ for all $x, y < k$. Whenever $R(x, y)$ holds, output y. This algorithm will eventually list every y such that $\exists x R(x, y)$. (It will list some y more than once. That's allowed; we could avoid it by keeping track of which y have already been listed.) □

Proposition 5.2 has a converse:

Proposition 5.3

If a set S is computably enumerable then there is a computable relation R such that $x \in S$ iff $\exists y R(x, y)$.

Proof. Assume that S is computably enumerable: there is an algorithm that lists all and only the elements of S. Let R be the relation that holds between x and y iff the algorithm has produced x among the first y items. Then $x \in S$ iff $\exists y R(x, y)$. Moreover, R is computable: given any x and y, simply run the enumerate-S algorithm for y steps; if x shows up in the list, return 'yes', otherwise return 'no'. □

Exercise 5.14 Show that if two relations R and S are computably enumerable then so is their conjunction, i.e., the relation that holds between x and y iff both $R(x, y)$ and $S(x, y)$.

Exercise 5.15 Let K be the set of algorithms that halt when given themselves as input. Is this set decidable? Is it computably enumerable?

Exercise 5.16 Let N be the set of algorithms that don't halt when given themselves as input. Is this set decidable? Is it computably enumerable?

Let's connect these concepts to the study of first-order theories from the previous chapter.

Remember that a formal theory is a (deductively closed) set of sentences. Typically, a theory is presented by giving a set of axioms. We say that a theory T is *(computably) axiomatizable* if there is a decidable set of axioms that generates the theory, so that T contains all and only the sentences that are provable from those axioms. For theories like Q and PA and ZFC, this is obviously the case: there is an algorithm for checking whether any given sentence is among the axioms of these theories.

We can also directly apply the concept of decidability to theories: a theory is decidable if there is an algorithm by which one can check, for any sentence, whether it is in the theory or not.

Every decidable theory is computably axiomatizable: we can use the theory itself as the set of axioms. The converse doesn't hold: a computably axiomatizable theory need not be decidable. It will, however, always be semidecidable, as the following proposition shows.

Proposition 5.4

Every computably axiomatizable first-order theory is computably enumerable.

Proof. Let T be a computably axiomatizable first-order theory, generated by a decidable set of axioms Γ. To enumerate all sentences in T, we can go through all strings in the language of T, one by one, and check for each if it is a deduction from Γ in the first-order calculus. This is possible because membership in Γ is decidable. If we find that a string is a deduction from Γ we output the last sentence in that deduction. Every sentence in T will eventually be listed. □

Ideally, we'd like a theory of, say, arithmetic to be complete, in the sense that it contains all truths about its intended model. Since every sentence A is either true or false in the intended model, the theory would then contain either A or $\neg A$, for every sentence A in its language. This is how completeness of theories is usually defined: a theory is *complete* if, for every sentence A in its language, the theory contains either A or $\neg A$.

Proposition 5.5

Every computably axiomatizable and complete first-order theory is decidable.

Proof. Let T be a computably axiomatizable and complete first-order theory, generated by a decidable set of axioms Γ. If T is inconsistent, it is trivially decidable: every sentence is in T. Assume that T is consistent. To decide whether a sentence A is in T, we go through all strings in the language of T, and check for each if it is a deduction of either A or $\neg A$ from Γ. Since the theory is complete, we must eventually find one or the other. If we find a deduction of A, we return 'yes'; if we find a deduction of $\neg A$, we return 'no'. □

At this point, we are closing in on Gödel's incompleteness theorem. Let T be a first-order theory that can prove elementary facts about computability. Specifically, assume the language of T contains terms for algorithms and natural numbers, and allows constructing a formula $H(x, y)$ so that T can prove $H(a, n)$ iff the algorithm denoted by a halts on input n. If T were decidable, we could decide the halting relation: we could check whether an algorithm a halts on input n, by checking whether $H(a, n)$ is in T. Since the halting relation is undecidable, T must be undecidable. By proposition 5.5, it follows that any computably axiomatizable theory that "knows" elementary facts about computability is incomplete.

Exercise 5.17 Let T be the theory axiomatized by the empty set. Given the undecidability of first-order logic (which we still haven't proved), is T (a) computably axiomatizable? (b) decidable? (c) complete?

Exercise 5.18 Show that every theory with a computably enumerable set of axioms can be axiomatized by a decidable set of axioms. Hint: replace each original axiom A by a sentence of the form $A \wedge A \wedge \ldots \wedge A$. (This is known as *Craig's re-axiomatization theorem*, after William Craig, who proved it in 1953.)

5.5 Coding

Think of how you might compute 134 times 97, using pen and paper. You'd probably begin by writing down '134' and '97'. What thereby appears on the paper are not the numbers themselves, but strings of symbols that represent the numbers. '134' denotes

the number 134 in decimal notation. The same number is denoted by 'CXXXIV' in Roman numerals, or by '10000110' in binary An algorithm for multiplication operates on the chosen representation. The algorithms for addition and multiplication that you learned in school assume that the inputs are given in decimal notation.

We may assume that, in general, an algorithm operates on strings of symbols. If we want to define an algorithm for computing functions on some other kinds of object (say, numbers or graphs or cities) these objects must first be encoded as suitable strings of symbols.

We can say a little more about these strings. Since an algorithm must be finitely specifiable, it can only make use of finitely many differences in the input. It follows that the possible inputs to an algorithm must be representable as finite strings of symbols from a finite (or at most countable) alphabet. For example, the decimal representation of any number is a finite string of symbols from the alphabet '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'.

How many finite strings can be formed from a countable alphabet? Countably many. We can show this by specifying an injective function from the set of such strings to the set of natural numbers. Such a function is called a *coding function*, as it codes strings as numbers.

To define a coding function, we first assign a unique natural number to each symbol in the alphabet. How this is done depends on the alphabet. Often, the symbols come in some natural "alphabetical" order. We can then assign 1 to the first symbol, 2 to the second, and so on. Let $\#s$ be the number assigned to symbol s. I'll call $\#s$ the *symbol code* of s.

With symbol codes in hand, the task of coding sequences of symbols reduces to the task of coding sequences of natural numbers as single numbers. I'll describe a standard way of doing this, due to Gödel.

Gödel's coding scheme exploits the fact that every natural number greater than 1 has a unique prime factorization. Recall that a prime number is a number greater than 1 that only divides by 1 and itself. Every natural number greater than 1 can be uniquely decomposed into a product of prime numbers, called its *prime factors*. For example, 54 decomposes into $2 \times 3 \times 3 \times 3$, or $2^1 \times 3^3$. We can therefore code sequences of numbers (greater than 0) by prime exponents: since the exponents in the prime factorization of 54 are 1 and 3, the number 54 codes the sequence $\langle 1, 3 \rangle$. In general, a sequence of n numbers is coded as the product of the first n primes raised to the power of those numbers: the first prime raised to the power of the first number, the second prime raised to the power of the second number, and so on.

An example may help. Suppose we want to code the string 'cabb', from an alphabet

that has the symbols ‘a’, ‘b’, ‘c’, and possibly others. We first assign code numbers to ‘a’, ‘b’, and ‘c’. Let’s use 1, 2, and 3, respectively. The string ‘cabb’ thereby turns into the sequence $\langle 3, 1, 2, 2 \rangle$. This is coded as

$$2^3 \times 3^1 \times 5^2 \times 7^2 = 29,400,$$

using 3 as the exponent of the first prime, 1 as the exponent of the second, and 2 as the exponent of the third and fourth.

To *decode* a number back into a string of symbols, we use some algorithm for prime factorization. Given the input 29,400, such an algorithm would return the prime factorization $2^3 \times 3^1 \times 5^2 \times 7^2$. This tells us that the first character in the coded string has symbol code 3, the second has symbol code 1, and the third and fourth have symbol code 2. Using the symbol codes, we reconstruct the original string: ‘cabb’.

Above, I suggested that the inputs to any algorithm are finite strings of symbols from a countable alphabet. We’ve now seen that all such strings can be coded as natural numbers. This means that there’s a sense in which every algorithm computes a function on the natural numbers – viz., the function that maps the code number of any input string to the code number of the algorithm’s output string.

Since there is an algorithm for coding and decoding, this line of thought also shows that we lose no generality by focusing on algorithms for functions on $\mathbb{N}$. That’s why, in computability theory, the computable functions and relations are usually defined as functions and relations on $\mathbb{N}$. If we want an algorithm that computes a different kind of function, we know that the inputs and outputs must be representable as strings of symbols, which can be coded as natural numbers. We can therefore compute the desired function by coding the inputs as numbers, feeding the code numbers into an algorithm for computing a function on $\mathbb{N}$, and decoding the output.

Exercise 5.19 Consider an algorithm that takes a string of symbols from the alphabet {‘a’, ‘b’, ‘c’} as input and replaces the first character in the string by ‘a’ (so that it returns ‘aabb’ for ‘cabb’). Can you describe the operation on $\mathbb{N}$ that this algorithm computes, using the prime number coding?

We can now sharpen the proto-Gödelian argument for incompleteness from the end of the previous section. Consider the ternary relation H^* that holds between an algorithm a, an input i for a, and a number n iff the algorithm a halts on input i within n steps, relative to some fixed way of counting steps in the execution of algorithms. This relation is computable: given any a, i, and n, we can simply run a on input i for n steps, and return

'yes' if a has halted by then, and 'no' otherwise. Like every algorithm, this algorithm for computing H^* effectively computes a function f on $\mathbb{N}$ – viz., the function that maps the code numbers of the inputs a, i, n to the code number of the output ('yes' or 'no'), relative to some fixed coding scheme. Let H^+ be the set of triples $\langle x, y, z \rangle$ of natural numbers that f maps to the code number of 'yes'. The algorithm for computing H^* gives us an algorithm for deciding the set H^+.

As I mentioned in section 4.1, all computable functions and relations on the natural numbers can be defined in the language $\mathfrak{L}_A$ of arithmetic. (We'll prove this in chapter 8.) So there is an expression $A(x, y, z)$ in $\mathfrak{L}_A$ that holds of numbers x, y, z iff $\langle x, y, z \rangle \in H^+$. From this, we can create another expression $\exists z A(x, y, z)$ by prefixing an existential quantifier. Can you see what this says? It expresses a numerical analog of the halting relation H: $\exists z A(x, y, z)$ is true of x and y iff x codes an algorithm that halts on the input coded by y.

Now, we know that the halting relation H is not decidable. It follows that *there can be no true, computably axiomatizable, and complete theory in the language of arithmetic*. For suppose there was such a theory. By proposition 5.5, the theory would be decidable. And then we could decide the halting relation: to check whether an algorithm with code n halts on input m, we would merely have to check whether $\exists z A(n, m, z)$ is in T.