5 Computability

In the next three chapters, we'll survey 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. To this end, 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'll assume that we're dealing with a countable first-order language (although the algorithm can easily be adapted to uncountable languages). 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 then only finitely many strings with length up to 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 string divides into sentences (separated by, say, a comma) in such a way that (1) each sentence is either an instance of an axiom schema (A1–A7) or follows from previous sentences by MP or

Gen, and (2) the last sentence is the target sentence. This is a simple, mechanical task. Whenever a sentence has a proof, this algorithm will eventually find it.

(Needless to say, the algorithm is terribly inefficient. There are much more efficient algorithms. I've implemented one that runs in your web browser: see 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. We would then also have an algorithm for deciding whether any given sentence is entailed by some axioms (assuming that one can mechanically recognize whether a sentence is among the 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, 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? Alonzo Church and Alan Turing found the answer 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 an algorithm for deciding whether a sentence is valid would also decide whether a sentence is entailed by a given set of axioms, assuming that one can mechanically recognize whether a sentence is among the axioms.

Exercise 5.2 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 sentence is satisfiable (true in some model), (d) decide whether a sentence is consistent (one can't derive a contradiction from it).

Exercise 5.3 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 '12998'. 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 to find the answer to every question of a certain type, which typically has infinitely many instances. 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 value for any input.

The recipe must be of a certain kind. 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 or randomization.

We are interested in computability in principle, setting aside limitations of time, paper, patience, and pencils. The multiplication function, for example, is computable: there are mechanical algorithms for multiplying numbers, even though, in practice, nobody (no human and no computer) can multiply arbitrarily large numbers.

We can allow for algorithms that may run forever, like the algorithm for finding proofs that I described above. Such an algorithm computes a partial function: it doesn't return an output for every input. Let's stipulate that this is the correct way for 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, we would say that 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 5.4 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 can be specified by defining some

obviously computable base functions and operations for obtaining new functions from old functions that preserve computability (see chapter **??**). More convincingly, Alan Turing suggested that a function is computable iff it is computed by a "Turing machine", an abstract model of a mechanical computing device (see chapter **??**). These suggestions were immediately recognized as equivalent, in the sense that they define the same class of functions. The same is true for later attempts to define computability in terms of register machines, Post systems, Markov algorithms, combinatorial definability, and so on: all such definitions turn out to define 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, and there are plausible arguments 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 allow for 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 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. We'll show 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 5.5 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?

Besides these *unavoidable* appeals to the Church-Turing Thesis, we will also 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 multiplication 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 **??**.) But it would often be tedious to do so, and we can save the effort by relying on the overwhelming evidence in favour of the Church-Turing Thesis.

Exercise 5.6 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 the continuum. But what matters is that it is uncountable.)

The set of algorithms that operate 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 operating 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 5.7 We've only considered functions on the natural numbers. How does the picture change if we also consider functions on the real numbers, or from arbitrary sets to arbitrary sets in the cumulative hierarchy?

The finite specifiability of algorithms leads to a puzzle. For every computable function on the natural numbers, there is an algorithm that computes it. This algorithm can be written down as a finite string of symbols, in some fixed language. This means that we can mechanically go through all computable functions on $\mathbb{N}$ just as we can go through all proofs in the first-order calculus: we can define an algorithm for enumerating all finite strings in the language, one by one, and checking for each of them whether it specifies an algorithm.

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 one: $A_1, A_2, \ldots, A_n$. It then runs the n-th algorithm A_n with input n and returns the output $A_n(n)$ 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.

The antidiagonal algorithm takes numbers as input and returns a number as output: it computes a function on $\mathbb{N}$. So it must be somewhere in the list of algorithms for those functions. Suppose it is in row n, for some n. By definition, $x_{n,n}$ is the output of the algorithm for input n. But the algorithm is defined so that it returns $x_{n,n} + 1$ for input n. This is a contradiction: an algorithm can't return two different values for the same input.

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 key is that we've allowed for algorithms that may run forever on certain inputs. Suppose that the n-th algorithm A_n runs forever on input n. Then we can't add 1 to the output of A_n on input n, because there is no such output: $x_{n,n}$ is undefined. My definition of the antidiagonal algorithm assumes 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 on input n, we return 0.

The 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 use it to check if the n-th algorithm halts on input n and output 0 if the answer is negative. If the subroutine says 'yes', we run the algorithm until it returns an output, and 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. So it would be on the list of algorithms, at some row n. And we know that it can't be on that list.

By this curious line of reasoning, we've established an important result: *There is no algorithm for checking whether a given algorithm halts on a given input.*

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 $A_1, A_2, A_3, \ldots$, we can define the antidiagonal function d by

$$d(n) = \begin{cases} 0 & \text{if } A_n \text{ runs forever on input } n \\ x + 1 & \text{if } A_n \text{ returns } x \text{ on input } n. \end{cases} \tag{5.1}$$

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

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. We can generalize this concept. In contemporary terminology, a *decision problem* is the problem 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 $\mathfrak{L}$-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.

Consider the property of being a spouse of Julius Caesar. Is there an algorithm for deciding whether a given person has this property? In a sense, no. In another sense, yes. 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. 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 classifies every person as a spouse of Caesar or not. But it does so only contingently, 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. In computability, 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 the 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 your answer.

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 Σ ("sigma") of first-order sentences and a first-order sentence A iff Σ is a proof of A 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 deciding whether a given string of symbols is a proof in the first-order calculus. This 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.

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.

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.

The set of algorithms, on the other hand, is decidable. On any precise definition of algorithms, there is a mechanical procedure for checking whether something is an algorithm. This means that we can define a mechanical procedure for listing all algorithms, one by one, just as we can list all proofs in the first-order calculus.

From the procedure for listing all algorithms, we can define another procedure for listing all algorithms that halt on a given input n. It goes as follows. At step 1, we start with the first algorithm. We apply it to input n, but only for 1 step. At step 2, we take the first two algorithms and let them both run for 2 steps on input n. And so on. At each stage, we let the first k algorithms run for k steps on input n. If any of these algorithms returns an output, we add it to the list of algorithms that halt on input n. 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*, or *computably enumerable*. (People also often say *recursively enumerable*, or just *r.e.*.)

In general, a property is (extensionally) *computably enumerable* 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.

As with decidability, the concept of semidecidability can be generalized to relations and to sets. A set is semidecidable (or 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 semidecidable.

Exercise 5.13 Explain why every decidable set is semidecidable.

The following propositions state some easy connections between decidability and semidecidability.

Proposition 5.1: (Kleene's Theorem)

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

Proof. Let S be a set such that both S and its complement $\bar{S}$ are semidecidable. That is, 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 semidecidable.

(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. It could be easily avoided by keeping track of which y have already been listed.) □

This proposition also 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. So 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 intersection, i.e.. the relation that holds between x and y iff both $R(x, y)$ and $S(x, y)$.

Exercise 5.15 Let N be the set of algorithms that don't halt when given themselves as input. Is this set decidable? Is it computably enumerable?

Exercise 5.16 Let K be the complement of the set N from the previous exercise. That is, K is the set of algorithms that halt on themselves. Is this set decidable? Is it computably enumerable?

Exercise 5.17 Is the set of total one-place computable functions 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. To say that the set of axioms is decidable means that there is an algorithm for deciding whether a given sentence is among the axioms. For theories like Q, PA, or ZFC, this is obviously the case.

Since theories are sets, we can also apply the concept of decidability to theories: a *theory* is decidable if there is an algorithm by which one can check, for any sentence in the language 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 computably enumerable:

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. If yes, we output the last sentence in the deduction. Every sentence in T will eventually be listed. □

A theory is *complete* if for every sentence A in its language, it 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. 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 a match. 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, 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 can prove elementary facts about computability is incomplete.

Exercise 5.18 Given the undecidability of first-order logic, is the theory axiomatized by the empty set (a) computably axiomatizable? (b) decidable? (c) complete?

Exercise 5.19 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'll 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 countable (and plausibly finite) 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. As above, 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. Here is one way of achieving this, due to Gödel. It 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, 53 decomposes into $2 \times 3 \times 3 \times 3$, or $2^1 \times 3^3$. We can therefore code sequences of numbers by prime exponents: since the exponents in the prime factorization of 53 are 1 and 3, it 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 $3, 1, 2, 2$. This is coded as

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

To *decode* a number back into a string of symbols, we factor the number into primes: from 29,400, we retrieve $2^3 \times 3^1 \times 5^2 \times 7^2$, which tells us that the first character in the coded string has symbol code 3, the second has symbol code 1, and the third and forth have symbol code 2.

We've seen that (a) the inputs to any algorithm are finite strings of symbols from a countable alphabet, and (b) 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: the function that maps the code number of an input string and returns the code number of the output string.

Since there is an algorithm for coding and decoding, no generality is lost 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 5.20 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 function on $\mathbb{N}$ that this

algorithm computes, using the prime number coding?

We can now sharpen the 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 $\mathbb{N}$: 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^+.

In chapter **??**, we'll see that the standard language of arithmetic, $\mathfrak{L}_A$, allows us to construct an arithmetical expression $E(x, y, z)$ so that $E(x, y, z)$ is true of numbers x, y, z iff $\langle x, y, z \rangle \in H^+$. We can therefore also express (in $\mathfrak{L}_A$) a numerical analog of the halting relation H that holds between an algorithm a and an input i iff a halts on i: $\exists z E(x, y, z)$ is true of x and y iff x codes an algorithm that halts on the input coded by y.

We know that the halting relation H is not decidable. It follows that there can be no true, computably axiomatizable, and complete theory in $\mathfrak{L}_A$. For suppose we had such a theory. By proposition 5.5, the theory would be decidable. 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 $\exists z E(n, m, z)$ is in T.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.