1 Propositional Logic

We are going to introduce formal languages in which one can regiment mathematical and philosophical reasoning, without the distracting complexities and vagaries of natural language. In this chapter, we begin with the language of propositional logic, the logic of the connectives $\neg$, $\rightarrow$, $\land$, $\lor$, etc. This language is woefully inadequate for any serious applications, but it is a useful prototype to introduce general ideas and techniques that we'll also use for more powerful languages.

1.1 Syntax

When talking about language, we must distinguish the language that is being talked about, the *object language*, from the *meta-language* in which the talking takes place. Throughout these notes, the meta-language will be English, with added technical vocabulary that will be introduced as we go along. Our first object language is the language of propositional logic, or $\mathfrak{L}_0$.

The *primitive symbols* of $\mathfrak{L}_0$ are:

- a non-empty (and countable) set of *sentence letters*,
- the *connectives* '$\neg$' and '$\rightarrow$', and
- the parentheses, '(' and ')'.

The sentence letters are classified as *non-logical* symbols; the other expressions are *logical*. (The point of this classification will become clear in section 1.3.)

Definition 1.1

A *sentence* of $\mathfrak{L}_0$ is a finite string of symbols, built up according to the following formation rules:

(i) Every sentence letter is a sentence.
(ii) If a string A is a sentence, then so is $\neg A$.
(iii) If strings A and B are sentences, then so is $(A \rightarrow B)$.

In definition 1.1, I use 'A' and 'B' as metalinguistic variables for strings of symbols in the object language. Throughout these notes, I will often use capital letters from the beginning of the alphabet for sentences in the object language. I'll sometimes use the lowercase letters 'p' and 'q' to denote arbitrary sentence letters. I haven't said what the sentence letters of $\mathfrak{L}_0$ look like. It doesn't matter (as long as none of them has any other primitive $\mathfrak{L}_0$-symbols as a part, which I hereby stipulate). Strictly speaking, $\mathfrak{L}_0$ is therefore not a single language, but a family of languages, with different stocks of sentence letters.

Unless stated otherwise, metalinguistic variables are to be understood as universally quantified. Condition (ii) in definition 1.1, for example, doesn't talk about a particular string A, which I failed to specify. Rather, it says that *for all strings* A, if A is a sentence then $\neg A$ is a sentence. By '$\neg A$', I mean the string that obtained by putting '$\neg$' in front of whatever string 'A' picks out. Similarly for '$(A \to B)$' and other such cases.

We introduce '$\land$', '$\lor$', and '$\leftrightarrow$' as metalinguistic abbreviations. That is, if A and B are $\mathfrak{L}_0$-sentences, we write

- $(A \land B)$ for $\neg(A \to \neg B)$;
- $(A \lor B)$ for $(\neg A \to B)$;
- $(A \leftrightarrow B)$ for $\neg((A \to B) \to \neg(B \to A))$.

We could have added '$\land$', '$\lor$', and '$\leftrightarrow$' as primitive symbols; you'll soon understand why we didn't. At any rate, you should remember that nothing is lost by restricting the primitive connectives to '$\neg$' and '$\to$': in classical propositional logic, all connectives can be defined in terms of these two.

It is convenient to also have a zero-ary connective $\top$ that is always true, and a dual $\bot$ that is always false. We introduce these as further metalinguistic abbreviations. Where p is an arbitrary sentence letter (say, the first in some alphabetical order), we write

- $\top$ for $p \to p$;
- $\bot$ for $\neg(p \to p)$.

Where no ambiguity threatens, I'll often omit parentheses and quotation marks. For example, I might write $A \to B$ instead of '$(A \to B)$'.

Exercise 1.1 Why did I say that no sentence letter of $\mathfrak{L}_0$ must have any other primitive $\mathfrak{L}_0$-symbol as a part? What could go wrong otherwise?

Suppose we want to show that every $\mathfrak{L}_0$-sentence has some property. The standard method for doing this is called *proof by induction on complexity*.

Proofs by induction on complexity are based on the fact that every $\mathfrak{L}_0$-sentence is built up from sentence letters by finitely many applications of the formation rules in definition 1.1. The *complexity* of a sentence is the number of applications of these rules. A sentence letter has complexity 0; $\neg p$ has complexity 1; $(p \to \neg q)$ has complexity 2; and so on. To show that every $\mathfrak{L}_0$-sentence has some property, it suffices to show two things:

(i) Every sentence of complexity 0 has the property.

(ii) *If* every sentence of complexity below n has the property *then* so does every sentence of complexity n.

Step (i) is called the *base case* of the proof; step (ii) the *inductive step*. The antecedent of (ii), that every sentence of complexity below n has the property, is called the *induction hypothesis*.

As an example, let's prove that every $\mathfrak{L}_0$-sentence has an even number of parentheses.

Proposition 1.1

Every $\mathfrak{L}_0$-sentence has an even number of parentheses.

Proof by induction on complexity.

Base case. We need to show that every sentence letter has an even number of parentheses. A sentence letter has zero parentheses. Zero is even.

Inductive step. We need to show that *if* some sentences have an even number of parentheses *then* so does every sentence generated from these sentences by a single application of a formation rule from definition 1.1. We need to consider two cases, because there are two formation rules.

First, we need to show that if A has an even number of parentheses, then so does $\neg A$. This is true because $\neg A$ has the same number of parentheses as A. Second, we need to show that if A and B have an even number of parentheses, then so does $(A \to B)$. This is true because $(A \to B)$ has two more parentheses than A and B together, and the sum of two even numbers plus two is always even. □

We'll use this method again and again, not just when we talk about sentences of $\mathfrak{L}_0$. Whenever a set of objects is generated from some base objects by finitely many applications of some operations, we can use the method to show that all of the objects have some property.

By the way: Now you can see why it is useful to have only two primitive connectives. If we had '$\land$', '$\lor$', and '$\leftrightarrow$' as well, we would have to check three more cases in every proof by induction on complexity.

Exercise 1.2 Prove by induction on complexity that in every $\mathfrak{L}_0$-sentence, the number of occurrences of sentence letters exceeds the number of occurrences of '$\to$' by 1. (There are two *occurrences* of 'p' in '$p \to p$'.)

Exercise 1.3 Prove by induction on n that for every natural number n, $0 + 1 + \ldots + n = n(n+1)/2$. That is, first show that the claim holds for $n = 0$; then show that if it holds for some n then it also holds for $n + 1$.

1.2 The propositional calculus

In this section, I'll explain how one can reason in $\mathfrak{L}_0$. We'll start with the ancient idea that to prove a statement means to derive it from some axioms. On this conception, a *proof system* consists of some axioms and some rules for deriving new sentences from old ones.

The first complete proof system for propositional logic, of this kind, was proposed by Gottlob Frege in 1879. I'll present a slightly simplified version, due to Jan Łukasiewicz and John von Neumann. We have three *axiom schemas*, meaning that every instance of these schemas is an axiom:

A1 $A \to (B \to A)$

A2 $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

A3 $(\neg A \to \neg B) \to (B \to A)$

An *instance* of an axiom schema is a sentence obtained by uniformly replacing the metalinguistic variables 'A', 'B', and 'C' by object-language sentences. For example, '$p \to ((p \to p) \to p)$' is an instance of A1, with 'A' replaced by 'p' and 'B' by '$(p \to p)$'.

The only rule of inference is *modus ponens*, which in this context is also known as *detachment*:

MP From A and $A \to B$ one may infer B.

I'll call this proof system *the propositional calculus*, although it is really only one of many equivalent calculi, all of which could be given that name.

Definition 1.2

A *proof* in the propositional calculus is a finite sequence of $\mathfrak{L}_0$-sentences $A_1, A_2, \dots A_n$, each of which is either an instance of A1–A3 or follows from earlier sentences in the sequence by MP. A *proof of A* is a proof whose last sentence is A.

We write '$\vdash_0 A$' (in the meta-language) to express that there is a proof of A in the propositional calculus.

Here is a proof of $p \to p$, showing that $\vdash_0 p \to p$:

1.	$p \to ((p \to p) \to p)$	Instance of A1
2.	$(p \to ((p \to p) \to p)) \to ((p \to (p \to p)) \to (p \to p))$	Instance of A2
3.	$(p \to (p \to p)) \to (p \to p)$	From 1, 2 by MP
4.	$p \to (p \to p)$	Instance of A1
5.	$p \to p$	From 3, 4 by MP

The result can be generalized. If we replace the sentence letter p by any sentence A throughout the proof, we still get a proof that conforms to definition 2.2. So we've effectively shown that $\vdash_0 A \to A$ for any sentence A.

Proof systems like our propositional calculus are called *axiomatic calculi* or *Hilbert-style calculi*. As the example illustrates, they tend to be difficult to use. They are also unnatural in that they focus on establishing logical truths. More often than not, when we turn to logic, we're interested in *consequence*: we want to know whether a certain conclusion follows from some premises, where these premises aren't logical truths. In a strict axiomatic calculus, any question about consequence must be reformulated as a question about logical truth: to test whether B is a logical consequence of $A_1, \dots, A_n$, one would check whether $(A_1 \land \dots \land A_n) \to B$ is a logical truth.

We can, however, also extend our calculus to handle deductions from non-logical premises.

Definition 1.3

A *deduction* of an $\mathfrak{L}_0$-sentence A from a set Γ ("gamma") of $\mathfrak{L}_0$-sentences in the propositional calculus is a finite sequence of sentences $A_1, A_2, \dots A_n$, with $A_n = A$, each of which is either an instance of A1–A3, an element of Γ, or follows from previous sentences by MP.

We write '$\Gamma \vdash_0 A$' to express that there is a deduction of A from Γ. For example, '$\{p, q\} \vdash_0 p$' is a sentence in our metalanguage saying that p is deducible from p and q. We usually omit the set braces and simply write '$p, q \vdash_0 q$'.

The following *structural principles* about the $\vdash_0$ relation immediately follow from definition 1.3.

Id $\quad A \vdash_0 A$.
Mon $\quad$ If $\Gamma \vdash_0 A$ then $\Gamma, B \vdash_0 A$.
Cut $\quad$ If $\Gamma \vdash_0 A$ and $\Delta, A \vdash_0 B$ then $\Gamma, \Delta \vdash_0 B$.

Here, 'Id' stands for 'Identity' and 'Mon' for 'Monotonicity'. As usual, 'A' and 'B' range over arbitrary $\mathfrak{L}_0$-sentences; 'Γ' and 'Δ' ('delta') range over arbitrary sets of $\mathfrak{L}_0$-sentences; 'Γ, B' is shorthand for '$\Gamma \cup \{B\}$', the union of Γ and $\{B\}$. (The union of two sets is the set that contains all and only the elements that are in either of the two sets.)

The Id principle says that for any sentence A, there is a deduction of A from A. Why is this true? By definition 1.3, a deduction of A from A would be a sequence of sentences ending in A, each of which is either an instance of A1–A3, the sentence A itself, or follows from previous sentences by MP. The sequence consisting of the single sentence A meets these conditions. (As do many longer sequences. But one sequence is enough to show that $A \vdash_0 A$.)

Exercise 1.4 Display another deduction of A from A, with at least two sentences.

Exercise 1.5 Explain why Mon and Cut similarly follow from definition 1.3, without invoking A1–A3 or MP.

A proof (in the sense of definition 2.2) is a special case of a deduction (in the sense of definition 1.3), with an empty set of premises Γ. The following theorem shows that every deduction can be converted into a proof.

Theorem 1.1: The Deduction Theorem (DT)

If $\Gamma, A \vdash_0 B$ then $\Gamma \vdash_0 A \to B$.

We'll prove this in a moment. First I want to explain how it converts deductions into proofs. Suppose there is a deduction of B from Γ. Being finite, this deduction can use only finitely many premises from Γ. Call them $A_1, \dots, A_n$. So we have

$$A_1, \dots, A_n \vdash_0 B.$$

By the Deduction Theorem, we can infer that

$$A_1, \dots, A_{n-1} \vdash_0 A_n \to B.$$

Applying the theorem again, we get

$$A_1, \dots, A_{n-2} \vdash_0 A_{n-1} \to (A_n \to B).$$

Continuing in this way, we can move all the premises to the right, until we get

$$\vdash_0 A_1 \to (A_2 \to (\dots (A_n \to B) \dots)).$$

To prove the Deduction Theorem, we use induction on the lines in a proof. We assume that there is a deduction $B_1, B_2, \dots, B_n$ of B from $\Gamma \cup \{A\}$. We then show that $\Gamma \vdash_0 A \to B_k$ for each $k = 1, 2, \dots, n$. Since $B_n = B$, this will show that $\Gamma \vdash_0 A \to B$. To show that $\Gamma \vdash_0 A \to B_k$ for each $k = 1, 2, \dots, n$, it suffices to show that

(i) $\Gamma \vdash_0 A \to B_1$, and

(ii) *if* $\Gamma \vdash_0 A \to B_i$ for all $0 < i < k$, *then* $\Gamma \vdash_0 A \to B_k$.

In fact, (ii) alone is sufficient. To see this, consider the case $k = 1$. Since there is no i with $0 < i < 1$, the *if*-part of (ii) is vacuously true. If we can show (ii), we can therefore also show that $\Gamma \vdash_0 A \to B_1$.

Proof of the Deduction Theorem.

Let $B_1, B_2, \dots, B_n$ be a deduction of B from $\Gamma \cup \{A\}$. We shall prove by induction on k that $\Gamma \vdash_0 A \to B_k$ for all $k = 1, 2, \dots, n$: We will show that *if* $\Gamma \vdash_0 A \to B_i$ for all $0 < i < k$, *then* $\Gamma \vdash_0 A \to B_k$. We distinguish three cases, corresponding to the ways in which B_k can appear in the deduction, according to definition 1.3.

Case 1. B_k is an axiom. By A1, $B_k \to (A \to B_k)$ is also an axiom. By MP, we can derive $A \to B_k$. So $\vdash_0 A \to B_k$, and so $\Gamma \vdash_0 A \to B_k$ by Mon.

Case 2. B_k is an element of $\Gamma \cup \{A\}$. We need to consider two subcases.

Subcase 2a. B_k is in Γ. Then $\Gamma \vdash_0 B_k$ by Id and Mon. As in case 1, we also have $\vdash_0 B_k \to (A \to B_k)$ by A1, so we get $\Gamma \vdash_0 A \to B_k$ by Mon and MP.

Subcase 2b. B_k is A. Then $A \to B_k$ is $A \to A$. We've just proved above that $\vdash_0 A \to A$. By Mon, we have $\Gamma \vdash_0 A \to A$.

Case 3. B_k follows by MP from two previous lines B_i and $B_i \to B_k$ in the deduction. By induction hypothesis, one can deduce $A \to B_i$ and $A \to (B_i \to B_k)$ from Γ. Axiom A2 gives us

$$(A \to (B_i \to B_k)) \to ((A \to B_i) \to (A \to B_k)).$$

With this, the deduction of $A \to (B_i \to B_k)$ and $A \to B_i$ from Γ can be extended by two applications of MP to a deduction of $A \to B_k$. □

The proof of the Deduction Theorem requires axioms A1 and A2. If we're interested in what can and can't be deduced in the propositional calculus, we never need to invoke A1 and A2 again. If needed, they can be recovered from DT and MP. Here is how we can recover A1 ($A \to (B \to A)$):

1.	$A \vdash_0 A$	(Id)
2.	$A, B \vdash_0 A$	(1, Mon)
3.	$A \vdash_0 B \to A$	(2, DT)
4.	$\vdash_0 A \to (B \to A)$	(3, DT)

Note that this is not a proof in the propositional calculus. It is a metalinguistic argument showing that a certain proof in the propositional calculus exists. Let me go through the steps.

Line 1 says that A is deducible from A. This is the Id principle for deductions. I've explained above why it holds. Line 2 says that A is deducible from A and B. This follows from line 1 by the general fact (called Monotonicity) that if A is deducible from some premises then A is also deducible from these premises together with any further premises. In the present case, it's easy to see directly why the claim on line 2 holds: the sentence A qualifies a deduction of A from A and B. Line 3 now applies the Deduction Theorem. It claims there is a deduction of $B \to A$ from A. It's not immediately obvious what this deduction looks like, and we don't need to know: since we've proved the Deduction Theorem, we can be sure that such a deduction exists. In the same way, line 4 infers that there is a proof of $A \to (B \to A)$. Of course, we knew that all along: if A and B are arbitrary sentences, $A \to (B \to A)$ is an axiom (A1), so it can be proved in one line, by simply writing it down. The point of the argument is that we didn't need to invoke A1: all A1 instances are provable in *any* calculus that satisfies the structural rules and DT.

Exercise 1.6 Show in the same way that A2 can be derived from DT and MP. That is, show from the structural rules, DT, and MP that $\vdash_0 (A \to (B \to C)) \to ((A \to B) \to (A \to C))$.

Exercise 1.7 Is the converse of the DT true as well? How is it related to MP?

Now for some facts about negation.

Theorem 1.2: Ex Falso Quodlibet (EFQ)

If $\Gamma \vdash_0 A$ and $\Gamma \vdash_0 \neg A$, then $\Gamma \vdash_0 B$

Proof.

1. $\Gamma \vdash_0 A$ (Assumption)
2. $\Gamma \vdash_0 \neg A$ (Assumption)
3. $\Gamma, \neg B \vdash_0 \neg A$ (2, Mon)
4. $\Gamma \vdash_0 \neg B \to \neg A$ (3, DT)
5. $\vdash_0 (\neg B \to \neg A) \to (A \to B)$ (A3)
6. $\Gamma \vdash_0 A \to B$ (4, 5, MP)
7. $\Gamma \vdash_0 B$ (1, 6, MP) □

Theorem 1.3: Double Negation Elimination (DNE)

If $\Gamma \vdash_0 \neg\neg A$ then $\Gamma \vdash_0 A$.

Proof.

1. $\Gamma \vdash_0 \neg\neg A$ (Assumption)
2. $\Gamma, \neg A \vdash_0 \neg\neg A$ (1, Mon)

3. $\Gamma, \neg A \vdash_0 \neg A$ (Id)
4. $\Gamma, \neg A \vdash_0 A$ (2, 3, EFQ)
5. $\Gamma \vdash_0 \neg A \to A$ (4, DT)
6. $\Gamma, \neg A \vdash_0 \neg(\neg A \to A)$ (3, 4, EFQ)
7. $\Gamma \vdash_0 \neg A \to \neg(\neg A \to A)$ (6, DT)
8. $\vdash_0 (\neg A \to \neg(\neg A \to A)) \to ((\neg A \to A) \to A)$ (A3 with $B = (\neg A \to A)$)
9. $\Gamma \vdash_0 (\neg A \to A) \to A$ (7, 8, MP)
10. $\Gamma \vdash_0 A$ (5, 9, MP) □

Theorem 1.4: Reductio Ad Absurdum (RAA)

If $\Gamma, A \vdash_0 B$ and $\Gamma, A \vdash_0 \neg B$, then $\Gamma \vdash_0 \neg A$.

Proof.

1. $\Gamma, A \vdash_0 B$ (Assumption)
2. $\Gamma, A \vdash_0 \neg B$ (Assumption)
3. $\Gamma, A \vdash_0 \neg A$ (1, 2, EFQ)
4. $\Gamma, \neg\neg A \vdash_0 \neg\neg A$ (Id, Mon)
5. $\Gamma, \neg\neg A \vdash_0 A$ (4, DNE)
6. $\Gamma, \neg\neg A \vdash_0 \neg A$ (3, 5, Cut)
7. $\Gamma \vdash_0 \neg\neg A \to \neg A$ (6, DT)
8. $\Gamma, \neg\neg A \vdash_0 \neg(\neg\neg A \to \neg A)$ (5, 6, EFQ)
9. $\Gamma \vdash_0 \neg\neg A \to \neg(\neg\neg A \to \neg A)$ (8, DT)
10. $\Gamma \vdash_0 (\neg\neg A \to \neg(\neg\neg A \to \neg A)) \to ((\neg\neg A \to \neg A) \to \neg A)$ (A3)
11. $\Gamma \vdash_0 (\neg\neg A \to \neg A) \to \neg A$ (9, 10, MP)
12. $\Gamma \vdash_0 \neg A$ (7, 11, MP) □

We needed A3 in the derivation of these facts. As in the case of A1 and A2, we won't need A3 any more, now that we have EFQ, DNE, and RAA. The relation $\vdash_0$ is fully characterized by the structural rules Id, Mon, Cut, together with MP, DT, EFQ, DNE, and RAA.

We could have used different axioms, or a different combination of axioms and inference rules to obtain the same result. Frege's original calculus, for example, has six axioms and an additional rule of substitution. But it is equivalent to the calculus I've introduced, since it determines the same relation $\vdash_0$,

Exercise 1.8 Show that $\Gamma \vdash_0 \bot$ iff there is a sentence A for which $\Gamma \vdash_0 A$ and $\Gamma \vdash_0 \neg A$.

Exercise 1.9 Show: (a) $\neg A \vdash_0 A \to B$. (b) $B \vdash_0 A \to B$. (c) $A \to \neg A \vdash_0 \neg A$;

Exercise 1.10 Show, by first expanding the definition of $\land$, that $\Gamma \vdash_0 A \land B$ iff both $\Gamma \vdash_0 A$ and $\Gamma \vdash_0 B$.

We can also design different *types* of proof systems that are equivalent to the propositional calculus. For example, you may have noticed that our metalinguistic proofs, using Id, Mon, Cut, MP, DT, etc., are generally simpler than proofs in our official calculus. We can turn these proofs into their own calculus. Each line of a proof, in this calculus, is a *sequent* $\Gamma \vdash_0 A$. There are no axioms. Instead, we have the rules Id, Mon, Cut, MP, etc. to operate on sequents. To show that A follows from Γ, one tries to derive the sequent $\Gamma \vdash_0 A$.

Now, I've introduced '$\Gamma \vdash_0 A$' to mean 'there is a deduction of A from Γ in the propositional calculus'. We don't want the lines in our new calculus to refer to deductions in another calculus. So we should replace '$\vdash_0$' by a different symbol. The standard choice is'$\Rightarrow$'. Also, it turns out that we can drop Mon and Cut in favour of a slightly strengthened form of Id:

Id$^+$ $\Gamma, A \Rightarrow A$

We can also drop EFQ, as it is derivable from RAA. The remaining rules of our new calculus are:

MP From $\Gamma \Rightarrow A$ and $\Gamma \Rightarrow A \to B$, infer $\Gamma \Rightarrow B$.
DT From $\Gamma, A \Rightarrow B$, infer $\Gamma \Rightarrow A \to B$.
RAA From $\Gamma, A \Rightarrow B$ and $\Gamma, A \Rightarrow \neg B$, infer $\Gamma \Rightarrow \neg A$.
DNE From $\Gamma \Rightarrow \neg\neg A$, infer $\Gamma \Rightarrow A$.

This is a stripped-down version of the *sequent calculus* invented by Gerhard Gentzen in the 1930s. It determines the same proof relation as our propositional calculus: a sequence $\Gamma \Rightarrow A$ is provable in the sequent calculus iff there is a deduction of A from Γ in the propositional calculus.

Here is a simple schematic proof in the sequent calculus to show that $A \to B$ and $B \to C$ together entail $A \to C$.

1. $A{\to}B,\ B{\to}C,\ A \Rightarrow A{\to}B$ Id$^+$
2. $A{\to}B,\ B{\to}C,\ A \Rightarrow B{\to}C$ Id$^+$
3. $A{\to}B,\ B{\to}C,\ A \Rightarrow A$ Id$^+$
4. $A{\to}B,\ B{\to}C,\ A \Rightarrow B$ 1, 3, MP
5. $A{\to}B,\ B{\to}C,\ A \Rightarrow C$ 2, 4, MP
6. $A{\to}B,\ B{\to}C \Rightarrow A{\to}C$ 5, DT

When writing out proofs like this, one often needs to repeat the same sentences on the left of '⇒' again and again. So-called *natural deduction* calculi introduce shortcuts to avoid these repetitions, dropping the '⇒' symbol and using lines or boxes to indicate the sentences to its left. You may have encountered such a calculus in your intro logic course. If so, you may want to write down a natural-deduction proof of the above entailment and compare it with the sequent-calculus proof. (Can you see how the two are related?)

You may also have come across *tableau calculi* or *tree proof* calculi. These are, in effect, upside-down sequent proofs (of a slightly different type) in which all sentences are pushed to the left of the arrow.

All these calculi are much easier to use than our propositional calculus. On the flip side, proofs in the propositional calculus are easier to *describe* than proofs in the other calculi: a proof in our calculus is simply a list of $\mathfrak{L}_0$-sentences, each of which is either an instance of A1–A3 or follows from earlier sentences by MP. This makes it easier to prove metatheorems about what is or is not provable in the calculus. Since all the calculi are equivalent, and we're mostly interested in metatheorems, we'll take the propositional calculus to be the official calculus of classical propositional logic.

We'll focus on classical logic in this course. But it is worth mentioning that there are also *non-classical* logics for $\mathfrak{L}_0$. These always drop one or more of the principles Id, Mon, Cut, MP, DT, EFQ, DNE, and RAA, and sometimes replace them by other principles. For example, *intuitionistic logic* drops DNE. This has the possibly attractive consequence that the rules for negation become self-contained in the sense that they don't allow proving any negation-free sentences that can't already be proved without them.

Exercise 1.11 Give a sequent calculus proof of *Peirce's Law*: $((p \to q) \to p) \to p$. The proof requires DNE, although the sentence doesn't contain any negation symbol.

1.3 Semantics

You may have noticed that I have introduced $\mathfrak{L}_0$ without saying anything about what the expressions of the language mean. Introductory logic texts often suggest that '$\neg$' and '$\to$' have roughly the same meaning as 'not' and 'if …then' in English. But we haven't built this tenuous connection to English into the formal language. In this section, we're going to study a more rigorous theory of meaning for $\mathfrak{L}_0$.

The status of this kind of theory is controversial. Some hold that the meaning of a logical expression is given by the rules for reasoning with the expression, which we've already described. This approach to meaning is sometimes called *inferential role semantics*. (*Semantics* is the study of meaning.)

The kind of theory we're about to develop instead belongs to the tradition of *truth-conditional semantics*. The guiding idea of truth-conditional semantics is that the meaning of a sentence can be given by stating what (typically non-linguistic) conditions must be satisfied for the sentence to be true. The German sentence 'Schnee ist weiss', for example, is true iff snow is white, and arguably this information captures the core of its meaning. On the truth-conditional approach, the meaning of sub-sentential expressions like 'weiss' or '$\neg$' is determined by their contribution to the truth-conditions of sentences in which they occur.

If we apply this approach to $\mathfrak{L}_0$, we first need to assign truth-conditions to the sentence letters. To a first approximation, this might look as follows:

p: snow is white.
q: grass is purple.
…

Here I give the truth-conditions by using English sentences. This is not ideal, because English sentences may not have fully precise and determinate truth-conditions. (It isn't clear what, exactly, must be the case for 'snow is white' to be true.) Fortunately, we'll see in a moment that we don't need to worry about this problem because we won't really need to assign a meaning to the sentence letters after all.

Moving on, we need to explain how the logical operators contribute to the truth-conditions of sentences in which they occur. This is the important part. We do this inductively, as follows:

(i) $\neg A$ is true iff A is not true.
(ii) $A \to B$ is true iff A is not true or B is true.

To see what this is saying, let's pretend that I managed to assign precise truth-conditions to the sentence letter p. We thereby know in what kinds of scenarios p is true and in what kinds of scenarios it is false. The above statement about $\neg$ now tells us $\neg p$ is true in precisely those scenarios in which p is not true. In general, it tells us how to determine the conditions under which $\neg A$ is true based on the conditions under which A is true. Similarly, the statement about $\to$ tells us how to determine the conditions under which $A \to B$ is true based on the conditions under which A and B are true.

The truth-conditional conception of meaning is useful in logic because it ties in with a natural conception of entailment. Intuitively, some premises entail a conclusion iff the truth of the premises guarantee the truth of the conclusion; that is, there is no conceivable scenario in which the premises are true while the conclusion is false. If that's right then knowledge of truth-conditions is exactly what we need if we want to know whether some premises entail some conclusion.

In fact, logic is about a particular type of entailment. Suppose we give the following truth-conditions to p and q:

p: Snow is white.
q: Snow is purple.

Then p entails $\neg q$: there is no scenario in which snow is white and also purple. The inference from p to $\neg q$ is valid, but it is not *logically* valid. That's because it depends on the meaning of the non-logical expression p and q. Logic abstracts away from the interpretation of non-logical expressions. Some premises *logically entail* a conclusion iff there's no conceivable scenario in which the premises are true and conclusion false, on any interpretation of the non-logical expressions.

Here we need a distinction between "logical" and "non-logical" expressions. This is best seen as a matter of choice. In *epistemic logic*, for example, a regimented version of 'it is known that' counts as logical. Since propositional logic is the logic of the Boolean connectives, '$\neg$' and '$\to$' here count as logical; the sentence letters are non-logical.

We now have this preliminary account of logical entailment:

> Some premises Γ logically entail a sentence A iff every scenario and interpretation of the sentence letters that makes the sentences in Γ true also makes A true.

We can render this simpler and more precise. Think of what you need to know about a pair of a scenario S and an interpretation I of the sentence letters in order to determine whether an arbitrary $\mathfrak{L}_0$-sentence – say, $\neg p$ – is true. I could tell you that p means that snow is purple, and that the scenario is one in which snow is red. You could then figure out that $\neg p$ is true (relative to S and I), using the interpretation rule for negation and the information I gave you. But you don't need all that information. It would be enough if I merely told you that p means something that isn't true in S. By the interpretation rule for negation, you could infer that $\neg p$ is true in S under I. Generalizing, all the information we need about a pair of a scenario S and an interpretation I to determine whether an arbitrary $\mathfrak{L}_0$-sentence is true in S under I is which sentence letters are true and which are false in S under I. This means that instead of quantifying over scenarios and interpretations, we can simply quantify over assignments of truth-values to the sentence letters. Such assignments are often called 'interpretations', but I find this misleading. We'll call them 'models'.

Definition 1.4: Model

A *model* for $\mathfrak{L}_0$ is an assignment σ ("sigma") of truth-values to the sentence letters of $\mathfrak{L}_0$.

That is, a model is a function σ that assigns to each sentence letter p one of the two truth values, which I'll label 'T' and 'F'. We use standard function notation here, using '$\sigma(p) = T$' to express that σ assigns the value T to p and '$\sigma(p) = F$' to express that σ assigns F to p.

Next, we specify how an assignment of truth-values to the sentence letters determines an assignment of truth-values to all sentences of $\mathfrak{L}_0$, in accordance with our above interpretation rules for '$\neg$' and '$\rightarrow$'. We write '$\sigma \Vdash A$' (read: "σ satisfies A") to mean that A is true in the model σ, and '$\sigma \nVdash A$' to mean that A is not true in σ. The satisfaction relation $\Vdash$ is defined as follows:

Definition 1.5

Let σ be a model for $\mathfrak{L}_0$. For any sentence letter p and sentences A and B:

(i) $\sigma \Vdash p$ iff $\sigma(p) = T$.

(ii) $\sigma \Vdash \neg A$ iff $\sigma \nVdash A$.

(iii) $\sigma \Vdash A \to B$ iff $\sigma \nVdash A$ or $\sigma \Vdash B$.

For example, if $\sigma(p) = T$ and $\sigma(q) = F$, then $\sigma \Vdash p \to (q \to \neg q)$, as you can confirm by working through definition 1.5.

We can now define logical entailment, as already announced:

Definition 1.6

Sentences Γ *(logically) entail* a sentence A (for short, $\Gamma \models A$) iff every model that satisfies every sentence in Γ also satisfies A.

We allow Γ to be infinite, and to be empty. If something is (logically) entailed by the empty set of premises, it is called *(logically) valid*. Since our topic is logic, I'll drop 'logically' when talking about validity and entailment from now on.

Definition 1.7

A is *valid* (for short, $\models A$) iff every model satisfies A.

Exercise 1.12 Explain why Γ entails A according to the earlier, informal definition iff $\Gamma \models A$ (as defined by definition 1.6).

Exercise 1.13 Which of these claims are true? (a) $\models \top$, (b) $\models \bot$, (c) $\top \models \bot$, (d) $\bot \models \top$.

Exercise 1.14 Show that all instances of A1–A3 are valid.

Exercise 1.15 I have introduced five arrow-like symbols: $\rightarrow$, $\vdash_0$, $\Rightarrow$, $\Vdash$, and $\models$. Explain what each of them means and to which language it belongs. (For the record: we will never use $\Rightarrow$ again.)

1.4 Soundness and Completeness

We have explored two perspectives on logic. The first was *proof-theoretic*: we studied proofs and deductions, defined as arrangements of symbols conforming to certain rules, without any extrinsic concern for what the symbols might mean. We then turned to a *model-theoretic* perspective, defining notions of validity and entailment in purely semantic terms.

Ideally, we'd like the two perspectives to harmonize: a sentence should be provable iff it is valid. More generally, we should have $\Gamma \vdash_0 A$ iff $\Gamma \models A$. In this section, we will show that this is indeed the case.

We have two directions to check. We first show that if $\Gamma \vdash_0 A$ then $\Gamma \models A$. This shows that the propositional calculus, and all the calculi equivalent to it, are *sound* with respect to the model-theoretic conception of entailment: anything that can be deduced from some premises in the calculus is entailed by the premises.

Afterwards, we'll show the converse, that if $\Gamma \models A$ then $\Gamma \vdash_0 A$. This shows that the calculus is *complete*: whenever something is entailed by some premises, it can be deduced from the premises.

The soundness proof is straightforward.

Theorem 1.5: Soundness of the propositional calculus

If $\Gamma \vdash_0 A$, then $\Gamma \models A$.

Suppose $\Gamma \vdash_0 A$. This means that there is a sequence $A_1, A_2, \ldots, A_n$ such that $A_n = A$ and each A_k in the sequence is either an axiom, a member of Γ, or follows from previous sentences by MP. We show by induction on k that $\Gamma \models A_k$. The theorem follows by taking $k = n$.

Case 1. A_k is an axiom. Then $\Gamma \models A_k$ by exercise 1.14.

Case 2. A_k is a member of Γ. Then $\Gamma \models A_k$ holds trivially.

Case 3. A_k follows from previous sentences A_i and $A_i \rightarrow A_k$ by MP. By induction hypothesis, $\Gamma \models A_i$ and $\Gamma \models A_i \rightarrow A_k$. It follows by clause (iii) of definition 1.5 that

$\Gamma \models A_k$. □

Completeness is harder. The first completeness proof for a propositional calculus was given by Paul Bernays in 1918. We're going to use a different technique, due to Leon Henkin (1949), that works for a wide range of logics. We'll use it again in chapter 3 to prove completeness for first-order logic.

Before we start, we need to define two key concepts. Let Γ be a set of $\mathfrak{L}_0$-sentences. We'll say that Γ is *consistent* if one can't deduce a contradiction from it: there is no sentence A such that $\Gamma \vdash_0 A$ and $\Gamma \vdash_0 \neg A$; equivalently, by exercise 1.8: $\Gamma \nvdash_0 \bot$. We say that Γ is *satisfiable* if there is some model that satisfies every sentence in Γ.

The following lemmas allow us to reformulate completeness in terms of consistency and satisfiability.

Lemma 1.1

$\Gamma \cup \{\neg A\}$ is satisfiable iff $\Gamma \not\models A$.

Proof. Immediate from definitions 1.5 and 1.6. □

Lemma 1.2

$\Gamma \nvdash_0 A$ iff $\Gamma \cup \{\neg A\}$ is consistent.

Proof. Suppose $\Gamma \cup \{\neg A\}$ is inconsistent. Then $\Gamma \vdash_0 \neg\neg A$ by RAA and so $\Gamma \vdash_0 A$ by DNE. Contraposing, this means that if $\Gamma \nvdash_0 A$ then $\Gamma \cup \{\neg A\}$ is consistent. Conversely, suppose $\Gamma \vdash_0 A$. Then $\Gamma, \neg A \vdash_0 A$ by Mon and $\Gamma, \neg A \vdash_0 \neg A$ by Mon and Id. So $\Gamma \cup \{\neg A\}$ is inconsistent. □

Now, completeness requires that whenever $\Gamma \models A$ then $\Gamma \vdash_0 A$. Equivalently, by contraposition: Whenever $\Gamma \nvdash_0 A$ then $\Gamma \not\models A$. By lemma 1.2, $\Gamma \nvdash_0 A$ iff $\Gamma \cup \{\neg A\}$ is consistent. By lemma 1.1, $\Gamma \not\models A$ iff $\Gamma \cup \{\neg A\}$ is satisfiable. So what remains to be shown to establish completeness is this: *Every consistent set of sentences is satisfiable.*

We are going to prove this in two steps. First, we show that every consistent set can be extended to a maximal consistent set. A set is *maximal consistent* if it is consistent and contains either A or $\neg A$, for each $\mathfrak{L}_0$-sentence A. Then we show that every maximal consistent set is satisfied by a model that makes true all and only the sentence letters in the set.

En route to the first step, we start with an easy observation.

Lemma 1.3

If Γ is consistent, then for any sentence A, either $\Gamma \cup \{A\}$ or $\Gamma \cup \{\neg A\}$ is consistent.

Proof. Suppose for reductio that $\Gamma \cup \{A\}$ is inconsistent, and so is $\Gamma \cup \{\neg A\}$. By RAA, it follows from the first assumption that $\Gamma \vdash_0 \neg A$, and from the second that $\Gamma \vdash_0 \neg\neg A$. So Γ is inconsistent. □

Now the first step:

Lemma 1.4: Lindenbaum's Lemma

Every consistent set is a subset of some maximal consistent set.

Let Γ_0 be some consistent set of sentences. Let $S_1, S_2, \ldots$ be a list of all $\mathfrak{L}_0$-sentences (in some arbitrary order). For every number $i \geq 0$, define

$$\Gamma_{i+1} = \begin{cases} \Gamma_i \cup \{S_i\} & \text{if } \Gamma_i \cup \{S_i\} \text{ is consistent} \\ \Gamma_i \cup \{\neg S_i\} & \text{otherwise.} \end{cases}$$

This gives us an infinite list of sets $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$. We show by induction that each set in the list is consistent.

Base case. Γ_0 is consistent by assumption.

Inductive step. We assume that some set Γ_i in the list is consistent, and show that Γ_{i+1} is consistent. By lemma 1.3, either $\Gamma_i \cup \{S_i\}$ or $\Gamma_i \cup \{\neg S_i\}$ is consistent. If $\Gamma_i \cup \{S_i\}$ is consistent, then Γ_{i+1} is $\Gamma_i \cup \{S_i\}$ (by construction), so Γ_{i+1} is consistent. If $\Gamma_i \cup \{S_i\}$ is not consistent, then Γ_{i+1} is $\Gamma_i \cup \{\neg S_i\}$, so again Γ_{i+1} is consistent.

So all of $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$ are consistent. Now let Γ be the set of sentences that occur in at least one of the sets $\Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3 \ldots$. (That is, let Γ be the union of $\Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3, \ldots$.) Evidently, Γ is maximal and Γ_0 is a subset of Γ. It remains to show that Γ is consistent.

Suppose not (for reductio). Then there are sentences $A_1, \ldots, A_n$ in Γ from which $\bot$ is deducible. All of these sentences have to occur somewhere on the list $S_1, S_2, \ldots$. Let S_j be the first sentence from $S_1, S_2, \ldots$ that occurs after all the $A_1, \ldots, A_n$. Since all $A_1, \ldots, A_n$ are in Γ, they have to be in Γ_j. So Γ_j is inconsistent. But we've seen that all of $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$ are consistent. □

For the second step, we also need a preliminary observation:

Lemma 1.5

If Γ is maximal consistent and $\Gamma \vdash_0 A$, then $A \in \Gamma$.

Proof. If $A \notin \Gamma$, then $\neg A \in \Gamma$ by maximality. We then have $\Gamma \vdash_0 A$ and $\Gamma \vdash_0 \neg A$, contradicting consistency. □

Here comes step 2.

Lemma 1.6: Truth Lemma

Every maximal consistent set Γ is satisfied by the model σ_Γ that assigns T to every sentence letter in Γ and F to every other sentence letter.

Proof. We show that for every $\mathfrak{L}_0$-sentence A, $\sigma_\Gamma \Vdash A$ iff $A \in \Gamma$. The proof is by induction on the complexity of A.

Base case: A is a sentence letter. Then the claim directly follows from the construction of σ_Γ.

Inductive step. We consider the two cases for complex sentences. Assume first that A is $\neg B$, for some sentence B. We have to show that $\sigma_\Gamma \Vdash \neg B$ iff $\neg B \in \Gamma$. Left to right: Assume $\sigma_\Gamma \Vdash \neg B$. Then $\sigma_\Gamma \nVdash B$ by definition 1.5. By induction hypothesis, it follows that $B \notin \Gamma$. Then $\neg B \in \Gamma$ because Γ is maximal. Right to left: Assume $\neg B \in \Gamma$. Then $B \notin \Gamma$ because Γ is consistent. By induction hypothesis, it follows that $\sigma_\Gamma \nVdash B$. So $\sigma_\Gamma \Vdash \neg B$ by definition 1.5.

Now assume that A is $B \to C$, for some sentences B and C. We show that $\sigma_\Gamma \Vdash B \to C$ iff $B \to C \in \Gamma$. Left to right: Assume $\sigma_\Gamma \Vdash B \to C$. Then $\sigma_\Gamma \nVdash B$ or $\sigma_\Gamma \Vdash C$ by definition 1.5. If $\sigma_\Gamma \nVdash B$ then $B \notin \Gamma$ by induction hypothesis; so $\neg B \in \Gamma$ by maximality and so $B \to C \in \Gamma$ by lemma 1.5 and exercise 1.9(a). If $\sigma_\Gamma \Vdash C$ then $C \in \Gamma$ by induction hypothesis, and so $B \to C \in \Gamma$ by lemma 1.5 and exercise 1.9(b). Right to left: Assume $B \to C \in \Gamma$. Assume first that B is also in Γ. Then $C \in \Gamma$ by lemma 1.5 and MP. By induction hypothesis, $\sigma_\Gamma \Vdash C$, and so $\sigma_\Gamma \Vdash B \to C$ by definition 1.5. Assume, alternatively, that B is not in Γ. By induction hypothesis, then $\sigma_\Gamma \nVdash B$, and again $\sigma_\Gamma \Vdash B \to C$ by definition 1.5. □

Let's put the pieces together:

Theorem 1.6: Completeness of the propositional calculus

If $\Gamma \models A$ then $\Gamma \vdash_0 A$.

Proof by contraposition. Assume $\Gamma \nvdash_0 A$. Then $\Gamma \cup \{\neg A\}$ is consistent by lemma 1.2. By lemma 1.4, $\Gamma \cup \{\neg A\}$ is contained in a maximal consistent set Γ^+. By lemma 1.6, there is a model σ_{Γ^+} that satisfies Γ^+ and hence $\Gamma \cup \{\neg A\}$. So $\Gamma \not\models A$. □

Exercise 1.16 Suppose we have two proof systems $\vdash_1$ and $\vdash_2$ such that whenever $\Gamma \vdash_1 A$ then $\Gamma \vdash_2 A$. Does the soundness of one system imply the soundness of the other? If so, in which direction? How about completeness?

Exercise 1.17 Someone might worry that the propositional calculus is inconsistent in the sense that it allows proving $\bot$ (from no premises). Can you allay this worry?

Exercise 1.18 Show that if we add any further axiom schema to A1–A3 that is not already provable in the propositional calculus, then we get an inconsistent calculus. (This means that the calculus is *Post-complete*, after Emil Post, who first proved the present fact in 1921.)
Hint: By the completeness theorem, any schema that isn't provable in the calculus has invalid instances. Can you see why a schema with invalid instances must have inconsistent instances?

2 First-Order Predicate Logic

In this chapter, we'll review the syntax and semantics of first-order predicate logic. In contrast to propositional logic, most (some would say: all) mathematical reasoning can be formalized in first-order logic.

2.1 Syntax

I'll begin with a basic version of first-order logic, without function symbols and identity; these will be added in section 2.4. For now, the *primitive symbols* of a first-order language $\mathfrak{L}_1$ therefore fall into the following categories (whose members must not be part of one another):

- a countably infinite set of *(individual) variables*,
- a countably infinite set of *(individual) constants*,
- for each natural number n, a set of *n-ary predicate symbols*,
- the *connectives* '$\neg$' and '$\to$',
- the *universal quantifier symbol* '$\forall$',
- the parentheses '(' and ')'.

The individual constants and variables constitute the *singular terms* of $\mathfrak{L}_1$. Intuitively, their function is to pick out an object, which might be a person, a number, a set, or anything else. Predicate symbols are used to attribute properties or relations to these objects. For example, we might have individual constants 'a' and 'b' for Athens and Berlin and a binary predicate 'R' for *being west of*. 'Rab' would then state that Athens is west of Berlin, and 'Rba' that Berlin is west of Athens. The predicate symbol always comes first.

From *atomic sentences* like 'Rab' or 'Fa', we can form complex sentences in the familiar way with the help of '$\neg$', '$\to$', and the parentheses: $\neg Rab$, $(Rab \to Fa)$, $\neg(Rab \to Fa)$, etc.

The real power and complexity of first-order logic comes from its quantificational apparatus. The quantifier symbol '$\forall$' allows making general claims about all objects –

where by ‘all’ I mean all objects in the intended domain of discourse. In a formal theory of arithmetic, for example, the intended domain of discourse would consist of the natural numbers 0, 1, 2, 3, etc. It would not include Athens. In this context, ‘$\forall xFx$’ would state that every natural number has the property expressed by ‘F’.

Some practice is required to become familiar with the use of ‘$\forall$’, as it has no direct analog in natural language. The closest translation of ‘$\forall xFx$’ in English is something like

> Everything is such that it is F.

This can obviously be simplified to ‘Everything is F’; but in that sentence, ‘everything’ combines directly with a predicate (‘is F’), whereas ‘$\forall x$’ combines with an expression of sentential type, ‘Fx’. The variable ‘x’ works much like the pronoun ‘it’ in English. Overt variables are sometimes used in English when quantifiers are nested:

> For every number x there is a number y greater than x such that every number greater than y is greater than x.

This can be easily expressed in first-order logic:

$$\forall x \exists y(Gyx \land \forall z(Gzy \to Gzx)).$$

Definition 2.1

A *formula* of a basic first-order language $\mathfrak{L}_1$ is a finite string built up according to the following formation rules:

(i) If P is an n-ary predicate symbol of $\mathfrak{L}_1$ and $t_1, \ldots, t_n$ are singular terms of $\mathfrak{L}_1$ then $Pt_1 \ldots t_n$ is a formula.
(ii) If A is an $\mathfrak{L}_1$-formula, then so is $\neg A$.
(iii) If A and B are $\mathfrak{L}_1$-formulas, then so is $(A \to B)$.
(iv) If x is a variable and A is a formula of $\mathfrak{L}_1$ then $\forall xA$ is a formula.

Here, ‘P’, ‘t_1’, ‘t_n’, ‘A’, ‘B’, ‘x’ are metalinguistic variables standing for expressions in $\mathfrak{L}_1$. I haven’t specified what the predicate symbols, individual constants, and variables of the object language look like.

As in the case of propositional logic, we introduce some shortcuts in the metalanguage, writing

- $(A \land B)$ for $\neg(A \to \neg B)$;
- $(A \lor B)$ for $(\neg A \to B)$;
- $(A \leftrightarrow B)$ for $\neg((A \to B) \to \neg(B \to A))$.
- $\top$ for $A \to A$;
- $\bot$ for $\neg(A \to A)$;
- $\exists xA$ for $\neg\forall x\neg A$.

The last of these is new. We'll omit parentheses and quotation marks when no ambiguity threatens.

Definition 2.1 allows for formulas like these:

$$Rax$$
$$Fa \to Gx$$

If, as before, we interpret 'a' as denoting Athens and 'R' as being west of, 'Rax' could be read as 'Athens is west of x'. But variables, unlike constants, don't pick out a definite object. Their only function is to construct quantified statements. '$\forall xRax$' would say that Athens is west of everything, but 'Rax' doesn't really say anything. It is neither true nor false.

Formulas like 'Rax' and '$Fa \to Gx$' that contain a variable without a matching quantifier are called "open". Formulas without such variables are "closed". Only closed formulas make a genuine claim about the intended domain of discourse.

Let's make this distinction more precise. A *quantifier* consists of the symbol '$\forall$' followed by a variable. That variable is said to be *bound* by the quantifier. Next, define a *subformula* of a formula as any part of the formula that is itself a formula. For example, Fx is a subformula of $\forall xFx$. The *scope* of an occurrence of a quantifier $\forall x$ in a formula is the shortest subformula that contains the occurrence. So the scope of $\forall x$ in $Fa \lor \forall x(Fx \to Gy)$ is $\forall x(Fx \to Gy)$. An occurrence of a variable in a formula is *bound* if it lies in the scope of an occurrence of a quantifier that binds it. An occurrence of a variable that isn't bound is *free*. A variable is *free in* a formula if it has at least one free occurrence in it. A formula in which some variable is free is *open*. A formula that isn't open is *closed*. A *sentence* is a closed formula.

Exercise 2.1 Given an example of a formula in which the variable x is free even though not all occurrences of x in the formula are free.

Exercise 2.2 Assume that 'F' is a 1-ary (= *monadic*) predicate symbol, 'a' is a constant, and 'x' a variable. Which of the following are formulas? Which are sentences? Mark the scope of each quantifier.
(a) $Fa \to \forall xFx$ (b) $\forall xFx \to Fx$ (c) $\forall xFa$ (d) $\forall x(Fa \to \forall x\neg(Fx \to Fa))$

2.2 The first-order predicate calculus

Frege's *Begriffsschrift* from 1879 contains a complete proof system for first-order logic. However, the formal language of the *Begriffsschrift* is not a first-order language, as it allows quantifying into predicate position. In Frege's language, one can say not only things like '$\forall x\, Fx$', but also '$\forall X\, Xa$', which (roughly) means that a has every property. Quantifiers that bind predicate-type expressions are called *second-order*, and the resulting logic is called *second-order logic*. We'll take a closer look at second-order logics in chapter **??**.

A complete calculus for pure first-order logic was first presented by David Hilbert and Wilhelm Ackermann in 1928. I give a slightly simplified version of their calculus, which I'll call *the first-order predicate calculus*.

Like the propositional calculus from the previous chapter, the first-order calculus consists of some axioms and inference rules. In fact, we'll take over all the axioms and rules of the propositional calculus. All instances of A1-A3 are axioms, and Modus Ponens (MP) is a rule of our new calculus. To these, we add some principles for dealing with quantifiers. Let's think about what we need.

A common inference pattern in first-order logic is "universal instantiation": having shown that *every* object has some property, we infer that a particular object c has that property. To state this precisely, we need some notation for substitution. If A is a formula, x a variable, and c an individual constant, I write '$A(x/c)$' for the formula obtained from A by replacing all free occurrences of x in A with c. For example, '$Fx(x/a)$' denotes the formula 'Fa', but '$\forall xFx(x/a)$' denotes '$\forall xFx$' rather than the nonsensical '$\forall aFa$'. In informal contexts, I'll often write '$A(x)$' to indicate that A is a formula in which the variable x occurs freely; '$A(c)$' is then shorthand for '$A(x/c)$'.

Exercise 2.3 Let A be $\forall x(Fx \to Gy) \to \forall yFy$. What is $A(y/b)$?

We can now formulate a rule of universal instantiation: if A is a formula, x a variable and c a constant, one may infer $A(x/c)$ from $\forall xA$. We won't actually add this as a new rule, however, because we can just as well add a corresponding axiom schema:

A4 $\forall xA \to A(x/c)$

Given A4, we can use MP to reason from $\forall xA$ to $A(x/c)$.

We introduce a genuine rule – called *(Universal) Generalization* – for the inference in the opposite direction, from a particular case to a general claim:

Gen From A one may infer $\forall xA(c/x)$.

Here, '(c/x)' expresses the inverse of '(x/c)': $A(c/x)$ is the formula obtained from A by replacing all occurrences of c in A with x, except for any occurrences that would let x become bound. (Just as $A(x/c)$ only replaces free occurrences of x, we want $A(c/x)$ to only create free occurrences of x.)

The Gen rule requires explanation. Do we really want to infer $\forall xFx$ from Fa? The inference clearly isn't valid: it's easy to imagine cases where a particular object a is F, but other objects are not F. But remember that each line in a strictly Hilbert-style axiomatic proof is either an axiom or follows from an axiom by an inference rule. None of our axioms or rules will allow making specific claims about any particular object that one couldn't equally make about all other objects. Fa won't be provable. The only provable sentences involving individual constants will be logical truths like $Fa \to Fa$. And here, the inference to $\forall x(Fx \to Fx)$ is safe.

To get a complete calculus, we need one more axiom schema:

A5 $\forall x(A \to B) \to (A \to \forall xB)$, if x is not free in A

To see the point of this, suppose that in the course of a proof we have established the following claims, for some A and $B(x)$, where x isn't free in A:

$\forall x(A \to B(x))$
A

If we had the rule of universal instantiation, we could deduce $A \to B(c)$ from the first line, then use MP to infer $B(c)$ and finally infer $\forall xB(x)$ by Gen. This reasoning can't be replicated after we've replace universal instantiation by the axiom schema A4. So we add A5, which allows inferring $\forall xB(x)$ by two applications of Modus Ponens.

Here's a summary of our axioms and rules:

A1	$A \to (B \to A)$
A2	$(A \to (B \to C)) \to ((A \to B) \to (A \to C))$
A3	$(\neg A \to \neg B) \to (B \to A)$
A4	$\forall x A \to A(x/c)$
A5	$\forall x(A \to B) \to (A \to \forall x B)$, if x is not free in A
MP	From A and $A \to B$ one may infer B.
Gen	From A one may infer $\forall x A(c/x)$.

Definition 2.2: Proof

A *proof* of a sentence A in the first-order calculus is a finite sequence of sentences $A_1, A_2, \ldots A_n$ with $A_n = A$, such that each A_i is either an instance of A1-A5 or follows from earlier sentences in the sequence by MP or Gen.

I'll use '$\vdash A$' to express that A is provable in the first-order calculus.

As in the case of propositional logic, it is convenient to generalize our Hilbert-style proof system to allow for deductions from premises. In this case, we have to restrict the use of Gen: we don't want to infer $\forall x F x$ from Fa.

Definition 2.3

If A is a first-order sentence and Γ a set of first-order sentences, a *deduction* of A from Γ in the first-order calculus is a finite sequence of sentences $A_1, A_2, \ldots A_n$, with $A_n = A$, such that each A_i is either an instance of A1–A5, an element of Γ, or follows from previous sentences by MP or Gen, but without applying Gen to an individual constant c that occurs in one of the sentences in the sequence that are elements of Γ.

We write '$\Gamma \vdash A$' to express that there is a deduction of A from Γ. Let's investigate what this relation looks like.

The structural principles Id, Mon, and Cut from the previous chapter hold for every axiomatic calculus. So we have

Id	$A \vdash A$.
Mon	If $\Gamma \vdash A$ then $\Gamma, B \vdash A$.
Cut	If $\Gamma \vdash A$ and $\Delta, A \vdash B$ then $\Gamma, \Delta \vdash B$.

(As in the previous chapter, we generally omit set brackets on the left-hand side of '$\vdash$', and write a comma to indicate unions: '$\Gamma, B \vdash A$' is shorthand for '$\Gamma \cup \{B\} \vdash A$'.)

The Deduction Theorem also still holds, but this isn't automatic.

Theorem 2.1: The Deduction Theorem (DT)

If $\Gamma, A \vdash B$ then $\Gamma \vdash A \to B$.

Proof. Let $B_1, B_2, \dots, B_n$ be a deduction of B from $\Gamma \cup \{A\}$. We prove by strong induction on k that $\Gamma \vdash A \to B_k$ for all $k = 1, 2, \dots, n$. That is, we show that *if* $\Gamma \vdash A \to B_i$ for all $i < k$, *then* $\Gamma \vdash A \to B_k$.

We need to distinguish four cases, corresponding to the ways in which B_k can appear in the deduction: as an axiom, as an element of $\Gamma \cup \{A\}$, from an application of MP, or from an application of Gen. The proof for the first three cases is exactly as in the proof of the Deduction Theorem for the propositional calculus. It remains to check the case of Gen.

Assume B_k follows from B_i by an application of Gen. So B_k is of the form $\forall x B_i(c/x)$, and c doesn't occur in Γ or A. By induction hypothesis, there is a deduction of $A \to B_i$ from Γ. As c doesn't occur in Γ, we can apply Gen, getting a deduction of $\forall x(A \to B_i)(c/x)$. Since c doesn't occur in A, this formula can also be written as $\forall x(A \to B_i(c/x))$. A5 gives us $\forall x(A \to B_i(c/x)) \to (A \to \forall x B_i(c/x))$. By MP, we therefore get a deduction of $A \to \forall x B_i(c/x)$ from Γ. □

You may remember that we don't need to invoke A1 and A2 any more once we have DT and MP. Similarly, once we have DT, MP, and Gen, we no longer need A5, as any instance of it can be derived. Here is how.

Assume, as in the statement of A5, that x is not free in A. Let c be a constant that doesn't occur in A or B. Then:

1.	$\forall x(A \to B), A \vdash \forall x(A \to B)$	(Id, Mon)
2.	$\vdash \forall x(A \to B) \to (A \to B(x/c))$	(A4, x not free in A)
3.	$\forall x(A \to B), A \vdash A \to B(x/c)$	(MP, 1, 2)
4.	$\forall x(A \to B), A \vdash A$	(Id, Mon)
5.	$\forall x(A \to B), A \vdash B(x/c)$	(MP, 3, 4)
6.	$\forall x(A \to B), A \vdash \forall x B$	(Gen, 5, $B(x/c)(c/x) = B$)
7.	$\forall x(A \to B) \vdash A \to \forall x B$	(DT, 6)

8. $\vdash \forall x(A \to B) \to (A \to \forall xB)$ (DT, 7)

From A4 and MP, we get the rule of universal instantiation:

Theorem 2.2: Universal Instantiation (UI)

If $\Gamma \vdash \forall xA$ then $\Gamma \vdash A(x/c)$.

Proof. Assume $\Gamma \vdash \forall xA$. By A4, $\vdash \forall xA \to A(x/c)$. So by MP, $\Gamma \vdash A(x/c)$. □

From this and DT, we can derive any instance of A4. So we won't need to invoke A4 any more.

The derivations of EFQ, DNE, and RAA from the previous chapter all go through as before, and make any appeal to A3 unnecessary.

In fact, we know from the completeness theorem for propositional logic that all truth-functional tautologies are provable from A1–A3 and MP. A *truth-functional tautology* is a sentence that is true on every truth-value assignment to atomic sentences. For example, $Fa \to Fa$ is a truth-functional tautology, and so is $\neg\neg\forall xFx \to \forall xFx$.

Theorem 2.3: Tautologies (Taut)

$\vdash A$ whenever A is a truth-functional tautology.

Proof. Consider the propositional language $\mathfrak{L}_0$ whose "sentence letters" are the atomic sentences of the first-order language. By theorem 1.6 (the completeness theorem for propositional logic), every sentence in this language that is true on every truth-value assignment is provable from A1–A3 and MP. □

As in the case of propositional logic, we could use the facts that we have established about $\vdash$ (Id, Mon, Cut, DT, UI, Taut, together with MP and Gen) to define a sequent calculus. From this, we could derive the kind of natural deduction or tableau calculus that you have probably learned in your intro logic course. We won't pause to explore these matters.

Exercise 2.4 Show that if $\Gamma \vdash A(x/c)$ then $\Gamma \vdash \exists xA$.

Exercise 2.5 Show that if $\Gamma \vdash \forall x(A \to B)$ then $\Gamma \vdash \forall xA \to \forall xB$.

2.3 Semantics

I've already explained informally how first-order languages are interpreted: individual constants pick out objects in the intended domain of discourse; predicate symbols express properties or relations among these objects. We'll now make this more precise.

Our semantics is inspired by the truth-conditional approach to meaning. Plausibly, we can determine the conditions under which an atomic first-order sentences is true by assigning objects to individual constants, and properties or relations to predicate symbols. For example, if we know that '*a*' picks out Athens, '*b*' Berlin, and '*R*' the property of being west of, we can determine that '*Rab*' is true in a possible scenario iff Athens is west of Berlin in that scenario.

Now remember that logic abstracts away from the meanings of non-logical expressions. Some premises logically entail a conclusion iff there is no conceivable scenario in which the premises are true and the conclusion false, *under any interpretation of the non-logical vocabulary*. The non-logical parts of a first-order language are its individual constants and predicate symbols. As in the case of propositional logic, we will define a *model* as a structure that contains just enough information about a scenario and an interpretation of the non-logical vocabulary to determine the truth-values of all sentences.

What do you need to know about a scenario S and an interpretation I to figure out whether, say, *Rab* is true? It would obviously suffice to know (1) which objects are picked out by '*a*' and '*b*' under I, (2) which relation is expressed by '*R*' under I, and (3) whether that relation holds between those two objects in S. But you don't need all that information. It would also suffice to know (1) which objects are picked out by '*a*' and '*b*' under I, and (2) which pairs of objects in S stand in the relation expressed by '*R*' under I. For example, if I told you that '*a*' picks out Athens, '*b*' Berlin, and '*R*' expresses a relation that holds between all and only the following pairs of objects: ⟨Athens, Berlin⟩, ⟨Berlin, Paris⟩, ⟨Paris, Rome⟩, you'd know enough to figure out that '*Rab*' is true – although you don't really know what the sentence says or what the scenario is like.

Definition 2.4

A *model* $\mathfrak{M}$ of a first-order language $\mathfrak{L}_1$ consists of

(i) a non-empty set D, called the *domain* or *universe* of $\mathfrak{M}$, and

(ii) an *interpretation function I* that assigns to each individual constant of $\mathfrak{L}_1$ a member of D, and to each n-ary predicate symbol of $\mathfrak{L}_1$ a set of n-tuples from D.

An "n-tuple" is a list of n objects. A 1-tuple is simply an object. So a "set of 1-tuples from D" is a set of members of D a "set of 2-tuples from D" is a set of pairs of members of D, and so on. The set assigned to a predicate symbol is called the *extension* of the predicate symbol.

A model's domain can be arbitrarily large, but it can't be empty. That's because I've stipulated that every first-order language has infinitely many individual constants, and definition 2.4 requires that every such constant be assigned an object in the domain. This wouldn't be possible if the domain were empty. But a single object is enough because we allow that all constants pick out the same object.

It is useful to have an expression for the denotation of a non-logical symbol s in a model $\mathfrak{M}$. I'll use '$[\![s]\!]^{\mathfrak{M}}$'. That is, if $\mathfrak{M}$ is a model with interpretation function I, c is an individual constant and P a predicate symbol, then $[\![c]\!]^{\mathfrak{M}}$ is $I(c)$ and $[\![P]\!]^{\mathfrak{M}}$ is $I(P)$.

Exercise 2.6 We can mimic sentence letters by using zero-ary predicate symbols. For example, if P and Q are zero-ary predicates, then $P \to Q$ is a sentence. We might expect that a model should assign a truth-value to zero-ary predicates. How can we define the truth-values T and F to get this result out of definition 2.4? (Hint: A *0-tuple* is a list of zero objects. There is only one such list: the empty list.)

Next, we define what it takes for a sentence A to be true in a model $\mathfrak{M}$. For atomic sentences, this is easy. 'Rab', for example, is true in $\mathfrak{M}$ iff the pair of objects assigned (by $\mathfrak{M}$) to 'a' and 'b' are in the set assigned to 'R'. For negated sentences and conditionals, we can use the same clauses as in propositional logic. Quantified sentences require a little more thought.

Let $A(x)$ be some formula in which x is free. Under what conditions is $\forall x A(x)$ true in a model $\mathfrak{M}$? As a first shot, one might suggest that $\forall x A(x)$ is true iff $A(c)$ is true for every individual constant c. This is called a *substitutional interpretation* of the quantifier. It assumes that every object in the domain is picked out by some individual constant. Definition 2.4 doesn't guarantee this. It allows for models in which some objects don't have a name, just as most stars and most real numbers don't have a name in English.

What we'll say instead is that $\forall x A(x)$ is true in a model $\mathfrak{M}$ iff $A(c)$ is true in every model that is exactly like $\mathfrak{M}$ in every respect other than the interpretation of c, where c is some individual constant that doesn't already occur in $A(x)$. For example, '$\forall x Rax$' is true in $\mathfrak{M}$ iff 'Rab' is true in every model that differs from $\mathfrak{M}$ at most in the object it assigns to 'b'. By varying the interpretation of 'b', we can check whether a stands in R to every object in the domain. For definiteness, we'll say that c is the "alphabetically first" individual constant that doesn't occur in $A(x)$, assuming that the constants come with some alphabetical order.

This approach to the semantics of quantifiers goes back to Benson Mates. An equally popular alternative, due to Alfred Tarski, states that $\forall x A(x)$ is true in a model $\mathfrak{M}$ iff $A(x)$ is true for every way of assigning an individual to x. This requires defining a truth relation not just between sentences and models, but between sentences, models, and so-called "assignment functions" that assign objects to variables. The two approaches deliver the same results. I use Mates' because it requires slightly less machinery.

Definition 2.5

An $\mathfrak{L}_1$-sentence A is true in a model $\mathfrak{M}$ (for short, $\mathfrak{M} \Vdash A$) iff one of the following conditions holds.

(i) A is an atomic sentence $Pc_1 \dots c_n$ and $\langle [\![c_1]\!]^{\mathfrak{M}}, \dots, [\![c_n]\!]^{\mathfrak{M}} \rangle \in [\![P]\!]^{\mathfrak{M}}$.
(ii) A has the form $\neg B$ and $\mathfrak{M} \nVdash B$.
(iii) A has the form $(B \to C)$ and $\mathfrak{M} \nVdash B$ or $\mathfrak{M} \Vdash C$.
(iv) A has the form $\forall x B$ and $\mathfrak{M}' \Vdash B(x/c)$ for every model $\mathfrak{M}'$ that differs from $\mathfrak{M}$ at most in the object assigned to c, where c is the alphabetically first individual constant that does not occur in B.

If A is true in $\mathfrak{M}$, we also say that $\mathfrak{M}$ is *a model of* A, or that $\mathfrak{M}$ *satisfies* A. A model satisfies a set of sentences if it satisfies each sentence in the set.

Entailment and validity are defined in terms of satisfaction, as in the previous chapter.

Definition 2.6

A set of sentences Γ *entails* a sentence A (for short, $\Gamma \models A$) iff every model that satisfies Γ also satisfies A.
Sentences A and B are *equivalent* if $A \models B$ and $B \models A$.
A sentence is *valid* (for short, $\models A$) iff it is satisfied by every model.

Exercise 2.7 State the truth conditions for $\exists x A$. That is, complete this sentence: '$\exists x A$ is true in a model $\mathfrak{M}$ iff …'.

Exercise 2.8 Give a countermodel to show that $\forall x(Fx \lor Gx) \not\models \forall xFx \lor \forall xGx$.

Exercise 2.9 Show that if x is not free in B then $\forall x(A \to B)$ is equivalent to $\exists xA \to B$.

2.4 Functions and identity

Consider the sentence '$1+2 = 3$'. How could we translate this into a first-order language? We could use a three-place predicate symbol S and write '$S(1, 2, 3)$'. But this isn't ideal. It obscures the structure of the original sentence, which states an identity between $1 + 2$ and 3.

As a first step to remedy this situation, let's introduce a predicate symbol for identity. We'll use '='. So '$= ab$' states that a equals b, in the sense that a and b are the very same object. For legibility, we'll "abbreviate" this as '$a = b$'. We'll also write '$a \neq b$' for '$\neg = ab$'.

Of course, nothing in our earlier definition of first-order languages prevented us from having a predicate symbol '='. The real novelty is that we now classify '=' as a logical expression. This means that its interpretation is held fixed: in every model, '=' is interpreted as the identity relation (on the model's domain). We'll also introduce new rules for reasoning with '='. Before we get to these changes, I want to introduce the second addition to our definition of first-order languages. It allows forming complex terms like '$1 + 2$'.

Let's think about how such terms work. The expression '$1 + 2$' denotes a number: the number 3. (That's why '$1 + 2 = 3$' is true.) In general, for any numerical terms 'a' and 'b', '$a + b$' denotes a number: the sum of a and b. We can therefore understand the '+' sign as expressing a function that maps a pair of numbers to their sum. So understood, '+' is a *function symbol*. Function symbols are ubiquitous in maths. It's useful to have them in our formal languages as well.

Let me say a few general words on the concept of a function, as it will play an important role throughout these notes. A function, in the mathematical and logical sense, takes one or more objects as input and (typically) returns an object as output. An input to a function is also called an *argument* to the function; the output is called the function's *value* for that argument. The inputs and outputs are usually restricted to a certain class of objects, called the function's *domain* and *codomain*, respectively. For example, the addition function takes two numbers as input and returns a number. The square function takes a single number and returns a number. The inputs and outputs don't need to be numbers. There is an "area" function that takes a country as input and returns its area in (say) square kilometres. And there is a "mother" function that takes a person as input and returns their mother.

If a function has domain X and codomain Y, we say that it is a function *from X to Y*. If all inputs and outputs of a function belong to a set X, we say that it is a function *on* X. So the addition function and the square function are functions on the set of numbers, while the area function is a function from the set of countries to the set of numbers.

Some functions are not defined for all objects in their domain. The division function, for example, takes two numbers as input and returns a number, but it is undefined if the second input is zero. Such functions are called *partial*.

Functions are often associated with a recipe or algorithm for determining the output for a given input. There are well-known algorithms for computing sums or squares. But this isn't part of the modern concept of a function. Any mapping from inputs to outputs is a function, even if there is no recipe for determining the output.

Since functions are just mappings from inputs to outputs, they are fully determined by their values for each input. Consider, for example, the function g on the natural numbers that takes a number x as input and returns x^2 if Goldbach's conjecture is true and 0 if Goldbach's conjecture is false. Goldbach's conjecture says that every even number greater than 2 is the sum of two primes. It is not known whether the conjecture is true. So we don't know what g returns for inputs other than 0. But we know that g is either identical to the square function or to the constant function that returns 0 for every input. Both of these are trivial to compute. So we know that g is trivial to compute, even though we don't know its value for 1!

Exercise 2.10 Give an example of a function with three arguments. (That is, the function takes three objects as input.)

Let's now add function symbols to our first-order languages. These combine with singular terms to form new singular terms. For example, if 'f' is a two-place function symbol and 'a' and 'b' are individual constants, then '$f(a,b)$' is a singular term; it denotes the value of the function f for the arguments a and b. '$f(f(a,b),c)$' is another singular term; it denotes the value of f for the arguments $f(a,b)$ and c. Previously, all singular terms were just individual constants and variables, now they can be arbitrarily complex. So we need a recursive definition.

Definition 2.7

A *(singular) term* of a first-order language $\mathfrak{L}_1^=$ with functions and identity is a finite string conforming to the following formation rules.

(i) Every variable and every individual constant is a singular term.

(ii) If f is an n-ary function symbol and $t_1, \dots, t_n$ are singular terms then $f(t_1, \dots, t_n)$ is a singular term.

A singular term is *closed* if it contains no variables. Formulas and sentences are defined exactly as before.

Officially, function symbols are placed in front of their arguments, with parentheses and commas to separate the arguments. By this convention, '$(a \times b) + c$' is written '$+(\times(a, b), c)$'. For the sake of readability, we allow the more familiar infix notation as a metalinguistic "abbreviation".

Exercise 2.11 Write down a first-order sentence expressing Lagrange's Theorem, that every natural number is the sum of four squares. Use a language with individual constants '0', '1', '2', '3', ..., and function symbols '+' and '×' for addition and multiplication.

In the axiomatic calculus, we generalize A4 to allow for closed terms t where we previously had individual constants c:

A4 $\forall xA \to A(x/t)$.

'(x/t)' is the obvious extension of the substitution notation to closed terms.

We don't need any new axioms or rules for function symbols. But we have two new axiom schemas for identity:

A6 $t_1 = t_1$

A7 $t_1 = t_2 \to (A(x/t_1) \to A(x/t_2))$

Here, t_1 and t_2 are closed terms and A is a formula in which only x is free, so that all instances of the schemas are closed. A7 is often called *Leibniz' Law*. The idea is that if t_1 and t_2 are the very same object, then anything true of t_1 is also true of t_2.

Exercise 2.12 You may wonder why I didn't write A7 as $t_1 = t_2 \to (A \to A(t_1/t_2))$. In response, explain why the following sentence is an instance of A7, but not of the alternative formulation: $a = b \to (Raa \to Rab)$.

Exercise 2.13 Show: (a) if $\Gamma \vdash t = s$ then $\Gamma \vdash s = t$, (b) if $\Gamma \vdash t = s$ and $\Gamma \vdash s = r$ then $\Gamma \vdash t = r$.

We also need to adjust our semantics. The definition of a model remains the same as before, except that interpretation functions need to interpret the function symbols. We assume that all function symbols denote total functions on the model's domain.

Definition 2.8

A *model* $\mathfrak{M}$ of a first-order language $\mathfrak{L}_1^=$ with functions and identity consists of

(i) a non-empty set D and

(ii) a function I that assigns

- to each individual constant of $\mathfrak{L}_1^=$ a member of D,
- to each n-ary function symbol of $\mathfrak{L}_1^=$ an n-ary total function on D, and
- to each non-logical n-ary predicate symbol of $\mathfrak{L}_1^=$ a set of n-tuples from D.

As before, we write $[\![s]\!]^{\mathfrak{M}}$ for the denotation of a non-logical symbol s in a model $\mathfrak{M}$. It is defined as follows.

Definition 2.9

Let $\mathfrak{M}$ be a model of a first-order language $\mathfrak{L}_1^=$ and I the interpretation function of $\mathfrak{M}$.

(i) For any individual constant c, $[\![c]\!]^{\mathfrak{M}} = I(c)$.

(ii) If f is an n-ary function symbol and $t_1, \dots, t_n$ are singular terms then $[\![f(t_1, \dots, t_n)]\!]^{\mathfrak{M}} = I(f)([\![t_1]\!]^{\mathfrak{M}}, \dots, [\![t_n]\!]^{\mathfrak{M}})$.

With this, satisfaction, entailment, and validity are defined essentially as before. I'll only give the definition of satisfaction:

Definition 2.10

An $\mathfrak{L}_1^=$-sentence A is true in a model $\mathfrak{M}$ (for short, $\mathfrak{M} \Vdash A$) if one of the following conditions holds.

(i) A has the form $t_1 = t_2$ and $[\![t_1]\!]^{\mathfrak{M}} = [\![t_2]\!]^{\mathfrak{M}}$.

(ii) A is any other atomic sentence $Pt_1 \dots t_n$ and $\langle [\![t_1]\!]^{\mathfrak{M}}, \dots, [\![t_n]\!]^{\mathfrak{M}} \rangle$ is in $[\![P]\!]^{\mathfrak{M}}$.

(iii) A is of the form $\neg B$ and $\mathfrak{M} \nVdash B$.

(iv) A is of the form $(B \to C)$ and $\mathfrak{M} \nVdash B$ or $\mathfrak{M} \Vdash C$.

(v) A is of the form $\forall x B$ and $\mathfrak{M}' \Vdash B(x/c)$ for every model $\mathfrak{M}'$ that differs from $\mathfrak{M}$ at most in the object assigned to c, where c is the alphabetically first individual constant that does not occur in B.

Exercise 2.14 Explain why $\Vdash a = a$, if a is an individual constant.

Exercise 2.15 Define a model in which '1 + 1 = 2' is true and another in which it is false.

Exercise 2.16 Construct three sentences A_1, A_2, A_3 with '=' as the only predicate symbol such that (a) A_1 is true in a model $\mathfrak{M}$ iff the domain of $\mathfrak{M}$ has at least two members, (b) A_2 is true in $\mathfrak{M}$ iff the domain of $\mathfrak{M}$ has at most two members, (c) A_3 is true in $\mathfrak{M}$ iff the domain of $\mathfrak{M}$ has exactly two members.

Two final comments.

First, it would be nice if we could allow for partial functions and empty domains. The problem is that we would then have to deal with "empty" terms that don't pick out anything. (On an empty domain, every term is empty; if f denotes a partial function, $f(a)$ may be empty.) If t is empty, should we say that $t = t$ is true? What about its negation, $t \neq t$? These questions can be answered in different ways, leading to different versions of *free logic*. A free logic is simply a logic in which terms can be empty. We dodge the questions by banning empty terms.

Second comment. I've stipulated at the very start of this chapter that a first-order language must have infinitely many individual constants. This, too, is somewhat unsatisfactory. Formalized theories of arithmetic or set theory or Newtonian mechanics, for example, typically don't involve infinitely many non-logical symbols. Indeed, the standard first-order theory of sets has only one non-logical symbol: the membership predicate '$\in$'.

Why, then, did I require infinitely many individual constants? There are two reasons. The first arises in our (Mates-style) semantics of quantifiers: Clause (v) in definition 2.10 interprets $\forall xA$ in terms of $A(x/c)$, where c is a constant that doesn't occur in A. Because quantifiers can be nested without limit, this requires an unending supply of constants: we need two constants for the interpretation of $\forall x \forall yA$, three for $\forall x \forall y \forall zA$, and so on. But these constants are only used in the internal semantic machinery. Their original denotation in the model is irrelevant: it plays no role in clause (v) of definition 2.10. A similar point applies to the other reason why we need an unending supply of constants. In our first-order calculus, deriving a universal statement $\forall xB(x)$ from another universal statement $\forall xA(x)$ often requires instantiating $\forall xA(x)$ to $A(c)$, deriving $B(c)$, and then applying Gen. Here we also need an unending supply of constants to deal with nested quantifiers. But here, too, the meaning of these constants is irrelevant: they are used to denote "arbitrary" objects.

So we need a large supply of constants to play certain internal roles in the handling of quantifiers in our proof system and semantics. Conceptually, the constants we need resemble variables. They are sometimes called *eigenvariables*. We might have decided to classify them as logical. In later chapters, when I speak of the non-logical expressions of, say, formalized set theory, I will usually ignore the eigenvariables.

2.5 Soundness

We have defined two consequence relations: the proof-theoretic (syntactic) relation $\vdash$, and the model-theoretic (semantic) relation $\models$. How are they related? Can we show that $\Gamma \vdash A$ iff $\Gamma \models A$? We can. The *completeness* direction, from $\Gamma \models A$ to $\Gamma \vdash A$, is hard and will be treated in the next chapter. The *soundness* direction, from $\Gamma \vdash A$ to $\Gamma \models A$, is comparatively easy. As in the propositional case, we only have to verify that all axioms are valid and that the rules preserve validity. There is nothing terribly exciting about this proof, but let's go through it anyway.

We'll need the following lemmas.

Lemma 2.1: Coincidence Lemma

If two models $\mathfrak{M}$ and $\mathfrak{M}'$ have the same domain and agree on the interpretation of all non-logical symbols in an $\mathscr{L}_1^=$-sentence A, then $\mathfrak{M} \Vdash A$ iff $\mathfrak{M}' \Vdash A$.

Proof. The proof is a simple induction on the complexity of A. The base case is guaranteed by clauses (i) and (ii) in definition 2.10. The inductive step for $\neg$ and $\to$ is trivial. Let's look at the case where A has the form $\forall xB$.

Assume $\mathfrak{M} \nVdash \forall xB$. By clause (v) of definition 2.10, this means that $\mathfrak{M}'' \nVdash B(x/c)$ for some model $\mathfrak{M}''$ that differs from $\mathfrak{M}$ at most in the object assigned to a constant c that does not occur in B. Let $\mathfrak{M}'''$ be like $\mathfrak{M}'$ except that $[\![c]\!]^{\mathfrak{M}'''} = [\![c]\!]^{\mathfrak{M}''}$. Since $\mathfrak{M}$ and $\mathfrak{M}'$ agree on all symbols in B, and c is not in B, $\mathfrak{M}''$ and $\mathfrak{M}'''$ agree on all symbols in $B(x/c)$. So by induction hypothesis, $\mathfrak{M}''' \nVdash B(x/c)$. By clause (v) of definition 2.10, this means that $\mathfrak{M}' \nVdash \forall xB$.

We've shown that if $\mathfrak{M} \nVdash \forall xB$ then $\mathfrak{M}' \nVdash \forall xB$. The converse direction can be shown by an exactly parallel argument. So $\mathfrak{M} \Vdash \forall xB$ iff $\mathfrak{M}' \Vdash \forall xB$. □

Lemma 2.2: Extensionality Lemma

If $[\![t_1]\!]^{\mathfrak{M}} = [\![t_2]\!]^{\mathfrak{M}}$ then $\mathfrak{M} \Vdash A(x/t_1)$ iff $\mathfrak{M} \Vdash A(x/t_2)$.

Proof. The proof is by induction on complexity of A. As before, the base case is guaranteed by clauses (i) and (ii) in definition 2.10, and the inductive step for $\neg$ and $\to$ is trivial. The case where A has the form $\forall yB$ needs some work.

Assume

$$\mathfrak{M} \nVdash \forall yB(x/t_1). \tag{1}$$

We'll show that $\mathfrak{M} \nVdash \forall yB(x/t_2)$. By (1) and definition 2.10, we have

$$\mathfrak{M}^c \nVdash B(x/t_1)(y/c), \tag{2}$$

where c is the alphabetically first constant that does not occur in $B(x/t_1)$ and $\mathfrak{M}^c$ is a model that differs from $\mathfrak{M}$ at most in the interpretation of c. Let d be a constant distinct from c that does not occur in $B(x/t_1)$ or $B(x/t_2)$. let $\mathfrak{M}^{dc}$ be like $\mathfrak{M}^c$ except that $[\![d]\!]^{\mathfrak{M}^{dc}} = [\![c]\!]^{\mathfrak{M}^c}$. Since $\mathfrak{M}^{dc}$ and $\mathfrak{M}^c$ agree on all symbols in $B(x/t_1)(y/c)$, we have, by the coincidence lemma,

$$\mathfrak{M}^c \Vdash B(x/t_1)(y/c) \text{ iff } \mathfrak{M}^{dc} \Vdash B(x/t_1)(y/c). \tag{3}$$

By induction hypothesis,

$$\mathfrak{M}^{dc} \Vdash B(x/t_1)(y/c) \text{ iff } \mathfrak{M}^{dc} \Vdash B(x/t_1)(y/d). \tag{4}$$

Let $\mathfrak{M}^d$ be like $\mathfrak{M}^{dc}$ except that $[\![c]\!]^{\mathfrak{M}^d} = [\![c]\!]^{\mathfrak{M}}$. Since c does not occur in $B(x/t_1)$ and is distinct from d, $\mathfrak{M}^d$ and $\mathfrak{M}^{dc}$ agree on all symbols in $B(x/t_1)(y/d)$. So by the coincidence lemma,

$$\mathfrak{M}^{dc} \Vdash B(x/t_1)(y/d) \text{ iff } \mathfrak{M}^d \Vdash B(x/t_1)(y/d) \tag{5}$$

Since $\mathfrak{M}^d$ agrees with $\mathfrak{M}$ on the interpretation of t_1 and t_2, $[\![t_1]\!]^{\mathfrak{M}^d} = [\![t_2]\!]^{\mathfrak{M}^d}$. So by induction hypothesis,

$$\mathfrak{M}^d \Vdash B(x/t_1)(y/d) \text{ iff } \mathfrak{M}^d \Vdash B(x/t_2)(y/d). \tag{6}$$

From (2)–(6), we get

$$\mathfrak{M}^d \nVdash B(x/t_2)(y/d). \tag{7}$$

Now let e be the alphabetically first constant that does not occur in $B(x/t_2)$. Assume first that e is distinct from d. Let $\mathfrak{M}^{ed}$ be like $\mathfrak{M}^d$ except that $[\![e]\!]^{\mathfrak{M}^{ed}} = [\![d]\!]^{\mathfrak{M}^d}$. Since e doesn't occur in $B(x/t_2)(y/d)$, $\mathfrak{M}^{ed}$ and $\mathfrak{M}^d$ agree on all symbols in $B(x/t_2)(y/d)$. So by the coincidence lemma,

$$\mathfrak{M}^d \Vdash B(x/t_2)(y/d) \text{ iff } \mathfrak{M}^{ed} \Vdash B(x/t_2)(y/d). \tag{8}$$

By induction hypothesis,

$$\mathfrak{M}^{ed} \Vdash B(x/t_2)(y/d) \text{ iff } \mathfrak{M}^{ed} \Vdash B(x/t_2)(y/e). \tag{9}$$

Finally, let $\mathfrak{M}^e$ be like $\mathfrak{M}$ except that $[\![e]\!]^{\mathfrak{M}^e} = [\![e]\!]^{\mathfrak{M}^{ed}}$. By the coincidence lemma,

$$\mathfrak{M}^{ed} \Vdash B(x/t_2)(y/e) \text{ iff } \mathfrak{M}^e \Vdash B(x/t_2)(y/e). \tag{10}$$

From (7), (8), (9), and (10), we get

$$\mathfrak{M}^e \nVdash B(x/t_2)(y/e). \tag{11}$$

We assumed that e is distinct from d. If e and d are the same constant, we get (11) directly from (7). From (11) and definition 2.10, we conclude that

$$\mathfrak{M} \nVdash \forall y B(x/t_2). \tag{12}$$

We've shown that if $\mathfrak{M} \nVdash \forall y B(x/t_1)$ then $\mathfrak{M} \nVdash \forall y B(x/t_2)$. Swapping t_1 and t_2

throughout the argument, we can equally show that if $\mathfrak{M} \nVdash \forall y B(x/t_2)$ then $\mathfrak{M} \nVdash \forall y B(x/t_1)$. So $\mathfrak{M} \Vdash \forall y B(x/t_1)$ iff $\mathfrak{M} \Vdash \forall y B(x/t_2)$. □

Theorem 2.4: Soundness of the first-order calculus

If $\Gamma \vdash A$, then $\Gamma \models A$.

Proof. We first show a special case: if $\vdash A$ then $\models A$.

Assume $\vdash A$. So there is a sequence $A_1, \ldots, A_n$ with $A_n = A$ such that each A_k in the sequence is either an axiom or follows from previous sentences by MP or Gen. We show by strong induction on k that $\models A_k$.

Case 1. A_k is an instance of A1–A3. Then $\models A_k$ by exercise 1.14 and the fact that the interpretation of '$\neg$' and '$\rightarrow$' in definition 2.10 is the same as in propositional logic.

Case 2. A_k is an instance of A4: $\forall x B \rightarrow B(x/t)$. Let $\mathfrak{M}$ be any model that satisfies $\forall x B$. By definition 2.10, this means that $\mathfrak{M}' \Vdash B(x/c)$ for every model $\mathfrak{M}'$ that differs from $\mathfrak{M}$ at most in the object assigned to c, where c does not occur in B. Let $\mathfrak{M}'$ be a model of this kind with $[\![c]\!]^{\mathfrak{M}'} = [\![t]\!]^{\mathfrak{M}'}$. Since $B(x/t)$ is obtained from $B(x/c)$ by substituting t for c, it follows by the extensionality lemma that $\mathfrak{M}' \Vdash B(x/t)$. Finally, since c does not occur in $B(x/t)$, $\mathfrak{M}'$ and $\mathfrak{M}$ agree on the interpretation of all symbols in $B(x/t)$. So $\mathfrak{M} \Vdash B(x/t)$ by the coincidence lemma. This shows that any model that satisfies $\forall x B$ also satisfies $B(x/t)$. Hence every model satisfies $\forall x B \rightarrow B(x/t)$.

Case 3. A_k is an instance of A5: $\forall x(A \rightarrow B) \rightarrow (A \rightarrow \forall x B)$, where x is not free in A. Let $\mathfrak{M}$ be any model that doesn't satisfy $A \rightarrow \forall x B$. By definition 2.10, this means that $\mathfrak{M} \Vdash A$ and $\mathfrak{M}' \nVdash B(x/c)$ for some model $\mathfrak{M}'$ that differs from $\mathfrak{M}$ at most in the object assigned to some constant c that does not occur in B. Let d be the alphabetically first constant that does not occur in either A or B, and let $\mathfrak{M}''$ be like $\mathfrak{M}'$ except that $[\![d]\!]^{\mathfrak{M}''} = [\![c]\!]^{\mathfrak{M}'}$. By the extensionality lemma, $\mathfrak{M}'' \nVdash B(x/d)$. By the coincidence lemma, $\mathfrak{M}'' \Vdash A$. So $\mathfrak{M}'' \nVdash A \rightarrow B(x/d)$. Since x is not free in A, $A \rightarrow B(x/d)$ is $(A \rightarrow B)(x/d)$. So $\mathfrak{M}'' \nVdash (A \rightarrow B)(x/d)$. By definition 2.10, this means that $\mathfrak{M} \nVdash \forall x(A \rightarrow B)$. Contraposing, we've shown that any model that satisfies $\forall x(A \rightarrow B)$ satisfies $A \rightarrow \forall x B$. So every model satisfies $\forall x(A \rightarrow B) \rightarrow (A \rightarrow \forall x B)$.

Case 4. A_k is an instance of A6: $t_1 = t_1$. Then $\models A_k$ by clause (i) of definition 2.10.

Case 5. A_k is an instance of A7: $t_1 = t_2 \rightarrow (A(x/t_1) \rightarrow A(x/t_2))$. Let $\mathfrak{M}$ be any model that satisfies $t_1 = t_2$. Then $[\![t_1]\!]^{\mathfrak{M}} = [\![t_2]\!]^{\mathfrak{M}}$. By the extensionality lemma, $\mathfrak{M} \Vdash A(x/t_1)$ iff $\mathfrak{M} \Vdash A(x/t_2)$. So any model that satisfies $t_1 = t_2$ and $A(x/t_1)$ also satisfies $A(x/t_2)$.

So every model satisfies $t_1 = t_2 \to (A(x/t_1) \to A(x/t_2))$.

Case 6. A_k is obtained by MP from earlier lines A_i and $A_i \to A_k$. By induction hypothesis, A_i and $A_i \to A_k$ are valid. So A_k is valid by clause (iii) of definition 2.10.

Case 7. A_k is obtained by Gen from an earlier line A_i. So A_k has the form $\forall x A_i(c/x)$. Let $\mathfrak{M}$ be any model. By definition 2.10, we need to show that $\mathfrak{M}' \Vdash A_i(c/x)(x/d)$ for every model $\mathfrak{M}'$ that differs from $\mathfrak{M}$ at most in the object assigned to d, where d is the alphabetically first individual constant that does not occur in $A_i(c/x)$. Take any such $\mathfrak{M}'$ and d. Let $\mathfrak{M}''$ be like $\mathfrak{M}'$ except that $[\![c]\!]^{\mathfrak{M}''} = [\![d]\!]^{\mathfrak{M}'}$. By induction hypothesis, every model satisfies A_i; so $\mathfrak{M}'' \Vdash A_i$. If d is c then $\mathfrak{M}'' = \mathfrak{M}'$. Otherwise c does not occur in $A_i(c/x)(x/d)$. Either way, $\mathfrak{M}''$ and $\mathfrak{M}'$ agree on the interpretation of every symbol in $A_i(c/x)(x/d)$. By the coincidence lemma, it follows that $\mathfrak{M}' \Vdash A_i(c/x)(x/d)$.

This completes the induction. We've shown that if $\vdash A$ then $\models A$. Now assume $\Gamma \vdash A$. That is, there is a deduction A from Γ. This deduction can involve only finitely many sentences $A_1, \ldots, A_n$ from Γ. So we also have $A_1, \ldots, A_n \vdash A$, By the deduction theorem, it follows that $\vdash A_1 \to (\ldots (A_n \to A) \ldots)$. From what we've just shown, we can infer that $\models A_1 \to (\ldots (A_n \to A) \ldots)$. It is easy to see that $\Gamma \models A \to B$ iff $\Gamma, A \models B$. So we have $A_1, \ldots, A_n \models A$ and thereby $\Gamma \models A$. □

Exercise 2.17 Soundness implies consistency: it is impossible to prove $\bot$ in the first-order calculus. Recall that a calculus is Post-complete if any addition of a new axiom schema (that is not already provable in the calculus) makes the calculus inconsistent. Can you outline a proof showing that the first-order calculus is not Post-complete?

3 Completeness

In this chapter, we're going to meet some important results about the powers and limitations of first-order logic. Our starting point is the completeness theorem, which shows that whenever a first-order sentence is entailed by some premises then it is derivable from these premises in the first-order calculus. We'll see that this positive result is tightly connected to some negative results: the compactness theorem and the Löwenheim-Skolem theorems. These theorems concern the *size* of models, by which we mean the size of their domain. To fully appreciate their implications, I'll begin with some background about the sizes of sets, which will be needed in later chapters anyway.

3.1 Cardinalities

{ Athens } is a set with one member: the city Athens. We say that { Athens } has size 1. { Athens, Berlin } has size 2. { Athens, Berlin, Cairo } has size 3. And so on. It seems straightforward. But now consider the set $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ of natural numbers. What is its size?

The official set-theoretic word for the size of a set is *cardinality*. So { Athens, Berlin, Cairo } has cardinality 3. Finite sets have finite cardinalities, which are natural numbers. But infinite sets also have a cardinality. These aren't natural numbers, but numbers of a more general sort, called *cardinals*. The infinite cardinals are also known as the *alephs*, because they are conventionally written using the Hebrew letter '$\aleph$' ("aleph"). For example, the cardinality of $\mathbb{N}$ is called '$\aleph_0$'. This is the smallest infinite cardinal. The next larger one is $\aleph_1$, followed by $\aleph_2$, and so on.

How do we determine the cardinality of an infinite set? The crucial idea goes back to Galileo and Hume, but was only fully developed by Georg Cantor in the 19th century. Following Galileo and Hume, Cantor stipulates that two sets have the same cardinality iff there is a one-to-one correspondence between their members.

To make this precise, we define the notion of a one-to-one correspondence, or bijection.

Definition 3.1

A function f from a set A to a set B is a *bijection* if it satisfies the following two conditions.

(i) For every $x, y \in A$, if $f(x) = f(y)$ then $x = y$. (Injectivity)
(ii) For every $y \in B$ there is some $x \in A$ such that $f(x) = y$. (Surjectivity)

Intuitively, a bijection pairs up each element of A with exactly one element of B, and vice versa, so that every element of either set gets a unique partner. As a shorthand, we say that sets A and B are *equinumerous* if there is a bijection from A to B. (In this case, there is always also a bijection from B to A.)

The Galileo-Hume-Cantor principle now says that *two sets have the same cardinality iff they are equinumerous*. Obviously, no finite set of numbers is equinumerous with $\mathbb{N}$. So $\mathbb{N}$ has an infinite cardinality. We define '$\aleph_0$' to name this cardinality. Using the Galileo-Hume-Cantor principle, we can determine, for any other set, whether it also has cardinality $\aleph_0$.

Consider, for example, the set of odd numbers $1, 3, 5, 7, \ldots$. The following function f is a bijection from $\mathbb{N}$ to the set of odd numbers:

$$f(n) = 2n + 1.$$

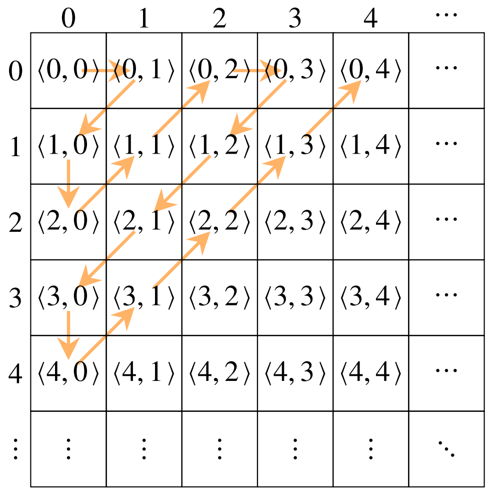
The function maps 0 to 1, 1 to 3, 2 to 5, and so on. Every natural number is mapped to a unique odd number, and no odd number is left unmapped. So the set of odd numbers also has cardinality $\aleph_0$.

Galileo found this paradoxical: how can there be as many odd numbers as natural numbers, given that the odd numbers are a proper subset of the natural numbers? Never mind, said Cantor: the resulting theory of cardinalities is consistent and mathematically fruitful, even if it may initially seem strange.

Sets that are equinumerous with $\mathbb{N}$ are also called *countably infinite* or *denumerable*. A set is *countable* if it is either finite or countably infinite. The word 'countable' alludes to the fact that a bijection between $\mathbb{N}$ and another set effectively assigns a "counter" to each member of the set. For example, the above bijection between $\mathbb{N}$ and the odd numbers assigns the counter 0 to 1, 1 to 3, 2 to 5, and so on.

Exercise 3.1 Show that (a) the set of even natural numbers and (b) the set of integers $\ldots, -2, -1, 0, 1, 2, \ldots$ are both countably infinite.

Are there any *uncountable* sets? One might suspect that the set of *pairs* of natural numbers is uncountable. But not so. Cantor's *zig-zag method* shows that there is a bijection between $\mathbb{N}$ and the set of pairs of natural numbers. We begin by arranging all pairs of natural numbers in a two-dimensional grid.



	0	1	2	3	4	$\cdots$
0	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 0,2\rangle$	$\langle 0,3\rangle$	$\langle 0,4\rangle$	$\cdots$
1	$\langle 1,0\rangle$	$\langle 1,1\rangle$	$\langle 1,2\rangle$	$\langle 1,3\rangle$	$\langle 1,4\rangle$	$\cdots$
2	$\langle 2,0\rangle$	$\langle 2,1\rangle$	$\langle 2,2\rangle$	$\langle 2,3\rangle$	$\langle 2,4\rangle$	$\cdots$
3	$\langle 3,0\rangle$	$\langle 3,1\rangle$	$\langle 3,2\rangle$	$\langle 3,3\rangle$	$\langle 3,4\rangle$	$\cdots$
4	$\langle 4,0\rangle$	$\langle 4,1\rangle$	$\langle 4,2\rangle$	$\langle 4,3\rangle$	$\langle 4,4\rangle$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

We then define a path through this grid that visits each cell exactly once. The orange arrows indicate that path. It effectively enumerates all pairs $\langle x, y\rangle$, starting with $\langle 0,0\rangle$, followed by $\langle 0,1\rangle$, $\langle 1,0\rangle$, $\langle 2,0\rangle$, $\langle 1,1\rangle$, $\langle 0,2\rangle$, and so on. The enumeration amounts to a lists of all the pairs. We get a bijection to $\mathbb{N}$ by assigning to each pair its position in the list, starting with position 0. So $\langle 0,0\rangle$ is mapped to 0, $\langle 0,1\rangle$ to 1, $\langle 1,0\rangle$ to 2, and so forth.

(We can find a formula for this bijection. As we follow the arrow, the pairs *before* any given pair $\langle x, y\rangle$ comprise all the pairs on the left-to-top diagonals before $\langle x, y\rangle$, which have 1, 2, 3, etc. pairs, plus the y pairs on the diagonal containing $\langle x, y\rangle$ itself. The total number of pairs preceding $\langle x, y\rangle$ is therefore $(1 + 2 + \ldots + (x + y)) + y$. In exercise 1.3, you showed that $1 + 2 + \ldots + (x + y) = (x + y)(x + y + 1)/2$. So the position of any pair $\langle x, y\rangle$ in the enumeration is $(x + y)(x + y + 1)/2 + y$.)

Exercise 3.2 Show that the set of ordered triples of natural numbers is countably infinite.

But it's true that not all sets are countable. Cantor established this with another powerful technique: *diagonalization*.

Theorem 3.1: Cantor's Theorem

The set of all sets of natural numbers is not countable.

Proof. Assume for reductio that the set of all sets of natural numbers is countable. This means there is a list $S_0, S_1, S_2, \ldots$ of all these sets. Now consider the set $D = \{n \in \mathbb{N} : n \notin S_n\}$ of all natural numbers n that are not in the n-th set S_n of the list. This is a set of natural numbers, so it must be somewhere in the list. That is, there is some S_k such that $D = S_k$. Now the number k is either in D or not. If k is in D then by definition of D, k is not in S_k. This is impossible, as $D = S_k$. If k is not in D, then by definition of D, k is in S_k. This is impossible, as $D = S_k$. □

We can again picture this technique with a two-dimensional grid.

	0	1	2	3	4	…
S_0	✓	✗	✗	✓	✓	⋯
S_1	✗	✗	✗	✓	✓	⋯
S_2	✗	✗	✓	✗	✗	⋯
S_3	✓	✗	✗	✓	✗	⋯
S_4	✓	✗	✗	✗	✗	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋱

Each row represents a set of natural numbers. Each column stands for a number. A checkmark indicates that the number is in the set, a cross that it is not. (So 0 is in S_0, 1 is not in S_0, 2 is not in S_0, and so on.) Cantor's method now looks at the shaded diagonal. It defines the new set D by *inverting* the diagonal, swapping crosses and checkmarks. In the picture, the inverted diagonal would begin with ✗✓✗✗✓…, meaning that D does not contain 0, does contain 1, does not contain 2 and 3, does contain 4, and so on. The so-defined set D is called the *antidiagonal* of the grid. It can't be anywhere in the list $S_0, S_1, S_2, \ldots$. So there can be no list of all sets of natural numbers.

Theorem 3.1 can be strengthened:

Theorem 3.2: Cantor's Theorem (general version)

For any set A, the set of subsets of A has strictly greater cardinality than A.

Proof. The proof proceeds essentially as before. Suppose for reductio that there is a bijection f from A to $\mathscr{P}(A)$, where $\mathscr{P}(A)$ (called the *power set* of A) is the set of all subsets of A. Define the antidiagonal set D as the set of all members X of A such that X is not in $f(X)$ – for short: $D = \{X \in A : X \notin f(X)\}$. Since D is a subset of A, it is in $\mathscr{P}(A)$. But it can't be an output of f. For suppose it is. That is, suppose $D = f(X)$ for some $X \in A$. Either X is in D or not. If X is in D, then by definition of D, X is not in $f(X)$. But $f(X) = D$, so this is impossible. If X is not in D, then by definition of D, X is in $f(X)$. Again, this is impossible. □

Cantor's theorem reveals a hierarchy of infinities. Starting with $\mathbb{N}$, we can produce larger and larger infinities by taking power sets: $\mathscr{P}(\mathbb{N})$ is larger than $\mathbb{N}$, $\mathscr{P}(\mathscr{P}(\mathbb{N}))$ is even larger, $\mathscr{P}(\mathscr{P}(\mathscr{P}(\mathbb{N})))$ is larger than that, and so on, without end. There are infinitely many infinite cardinals.

Exercise 3.3 Show that if A and B are countably infinite sets, then their union $A \cup B$ is countably infinite.

Exercise 3.4 Show that if a first-order language has countably many primitive symbols, then the set of all sentences of the language is countably infinite.

Exercise 3.5 Show that the set of real numbers is uncountable. (Hint: use the diagonalization method, assuming, for reductio, that there is a list of all decimal representations of real numbers between 0 and 1.)

3.2 Planning the completeness proof

In section 1.4, we showed that all valid sentences in propositional logic are derivable from the axiom schemata A1–A3 by Modus Ponens. This particular result wasn't easy to foresee, but it wasn't a surprise that there is *some* mechanical way of checking if a sentence of propositional logic is valid. Models of propositional logic are essentially finitary. A model is a truth-value assignment to the sentence letters. Since each sentence of propositional logic is composed of finitely many sentence letters, and there are only finitely many ways of assigning truth-values to these sentence letters, it is to be expected that there is a finitary algorithm for deciding whether all of these assignments make the sentence true.

In first-order logic, the situation is very different. A first-order model consists of an arbitrary set D together with an interpretation of the non-logical vocabulary, where every subset of D is a possible interpretation of any monadic predicate. Even if D itself is countable, this means that there are usually uncountably many models with that domain. And D doesn't have to be countable. It can have any cardinality whatsoever. The space of first-order models is enormous. The number of first-order models is vastly greater than $\aleph_0$. There is, in fact, no aleph big enough to measure the number of first-order models. It is therefore astonishing that there is a finitary, mechanical method – a proof system – by which one can find every sentence that is true across the realm of all first-order models. Astonishing, but true – as Kurt Gödel showed in 1929.

Gödel's completeness theorem (not to be confused with his *incompleteness* theorems that we'll discuss in later chapters) is noteworthy for other reasons as well. For example, it supports the hypothesis that all intuitively valid mathematical arguments can be formalized as deductions in the first-order calculus. Let's grant, for the sake of the argument, that every mathematical statement can be expressed in a first-order language. Now suppose some mathematical argument *can't* be replicated in the first-order calculus. By the completeness theorem, it follows that there is a model in which the premises are true but the conclusion false. This strongly suggests that the argument wasn't valid. We'll meet further applications of the completeness theorem later. First we need to prove it.

Our proof will follow Henkin's 1949 method, which we used in chapter 1 to prove the completeness of propositional logic. Let's recall the key steps.

We want to show that whenever a sentence A is entailed by a set of sentences Γ (meaning that every model that makes all members of Γ true also makes A true), then there is a deduction of A from Γ. For short: if $\Gamma \models A$ then $\Gamma \vdash A$. The proof is by contraposition. We assume $\Gamma \nvdash A$ and derive $\Gamma \not\models A$, as follows.

1. Assume $\Gamma \nvdash A$.
2. Infer that $\Gamma \cup \{\neg A\}$ is consistent.
3. Show that $\Gamma \cup \{\neg A\}$ can be extended to a maximal consistent set Γ^+.
4. Construct a model $\mathfrak{M}$ based on Γ^+ in which all members of Γ^+ are true.
5. Infer that $\Gamma \not\models A$.

As in chapter 1, we call a set of sentences *consistent* if no contradiction can be derived from it. That is, Γ is consistent iff there is no sentence A such that $\Gamma \vdash A$ and $\Gamma \vdash \neg A$. Equivalently (as you showed in exercise 1.8; the proof carries over from propositional logic), Γ is consistent iff $\Gamma \nvdash \bot$.

Steps 2 and 5 are easy. The following lemma establishes step 2, and is proved just as its propositional counterpart, lemma 1.2.

Lemma 3.1

$\Gamma \nvdash A$ iff $\Gamma \cup \{\neg A\}$ is consistent.

Proof. Suppose $\Gamma \cup \{\neg A\}$ is inconsistent. Then $\Gamma \vdash \neg\neg A$ by RAA and so $\Gamma \vdash A$ by DNE. Contraposing, this means that if $\Gamma \nvdash A$ then $\Gamma \cup \{\neg A\}$ is consistent. Conversely, suppose $\Gamma \vdash A$. Then $\Gamma, \neg A \vdash A$ by Mon and $\Gamma, \neg A \vdash \neg A$ by Mon and Id. So $\Gamma \cup \{\neg A\}$ is inconsistent. □

It remains to fill in steps 3 and 4: we need to show that *every consistent set of sentences is satisfiable*.

Here is the plan. Let Γ be a consistent set of sentences in some first-order language $\mathfrak{L}$ (with identity and function symbols). We'll show how one can construct a model $\mathfrak{M}$ in which all members of Γ are true. As in chapter 1, we first extend Γ to a maximal consistent set Γ^+ that will guide the construction of the model. For example, if Γ contains $Fa \lor Gb$, it's not obvious if our model should satisfy Fa or Gb or both. Γ^+ will directly contain Fa or Gb or both, so it will answer this question.

Suppose we have Fa in Γ^+. Our model $\mathfrak{M}$ must then contain an object $[\![a]\!]^{\mathfrak{M}}$ denoted by a that falls in the extension $[\![F]\!]^{\mathfrak{M}}$ of F. The nature of the object $[\![a]\!]^{\mathfrak{M}}$ is irrelevant. It proves useful to choose *the individual constant a* as the object denoted by a. Then we can say that the extension of F comprises all individual constants c for which $Fc \in \Gamma^+$. (It might seem odd to have a model whose domain consists of individual constants, and in which each constant denotes itself. But nothing in the definition of a first-order model prevents us from doing this.)

Unfortunately, there are some complications. Suppose Γ contains $\exists x Fx$ (which is short for $\neg\forall x\neg Fx$). We want all sentences in Γ to be true in our model $\mathfrak{M}$. So there must be some object in $[\![F]\!]^{\mathfrak{M}}$. But if $[\![F]\!]^{\mathfrak{M}}$ is defined as the set of individual constants c for which $Fc \in \Gamma^+$, there might be no such object, as there might be no constant c for which $Fc \in \Gamma^+$. We need to ensure that this doesn't happen. That is, we need to ensure that for each existential sentence $\exists x Fx$ in Γ^+, there is a "witnessing" sentence Fc in Γ^+. But what if Γ already contains $\neg Fc$ for every individual constant c? Then we can't add the required witness without making Γ inconsistent.

To get around this problem, we'll construct Γ^+ in an extend language $\mathfrak{L}^+$ that adds new individual constants to $\mathfrak{L}$. We can use these constants to ensure that every existential sentence in Γ has a witness.

Another complication arises from the presence of the identity symbol. If we let each individual constant denote itself, any identity statement involving different individual constants will be false. After all, no constant is identical to any other constant. But $a = b$ is consistent, and might be in Γ and therefore in Γ^+. So we'll actually let each constant denote a *set* of terms: the constant a will denote the set of closed terms t for which $a = t$ is in Γ^+.

Let's fill in the details.

3.3 The completeness proof

We'll show that every consistent set of first-order sentences is satisfiable. As we've seen above (and as we'll spell out again below), from this it is only a small step to the completeness theorem.

So let Γ be a consistent set of sentences in a first-order language $\mathfrak{L}$. Let $\mathfrak{L}^+$ be an extension of $\mathfrak{L}$ with (countably) infinitely many new individual constants. We'll extend Γ to a maximal consistent set in $\mathfrak{L}^+$. First, though, we need to confirm that Γ itself is still consistent in the extended language $\mathfrak{L}^+$. Consistency is language-relative because the language determines which instances of the axioms are available. As we switch from $\mathfrak{L}$ to $\mathfrak{L}^+$, new axioms become available. We need to confirm that these new axioms don't allow deriving a contradiction from Γ.

Lemma 3.2

If Γ is a set of $\mathfrak{L}$-sentences that is consistent within $\mathfrak{L}$, and $\mathfrak{L}^+$ extends $\mathfrak{L}$ by a set of new individual constants, then Γ is consistent within $\mathfrak{L}^+$.

Proof by contraposition. Assume that Γ is inconsistent within $\mathfrak{L}^+$. Then there is a deduction $A_1, \ldots, A_n$ of $\bot$ from Γ, where each A_i is an $\mathfrak{L}^+$-sentence. Being finite, the deduction only uses finitely many of the new constants: call them $c_1, \ldots, c_k$. The deduction also can't use more than finitely many of the old constants in $\mathfrak{L}$. Since first-order languages have infinitely many constants, we can choose k distinct constants $d_1, \ldots, d_k$ from $\mathfrak{L}$ that don't occur in the deduction. Now consider the sequence of sentences $A'_1, \ldots, A'_n$ that results from $A_1, \ldots, A_n$ by replacing each c_i by d_i. It is easy to see that

(i) if A_i is an axiom then so is A'_i;

(ii) if $A_i \in \Gamma$ then $A'_i \in \Gamma$ (sentences in Γ don't contain any new constants);

(iii) if A_i follows from $A_1, \ldots, A_{i-1}$ by MP or Gen, then A'_i follows from $A'_1, \ldots, A'_{i-1}$ by MP or Gen.

So $A'_1, \dots, A'_n$ is a deduction of $\bot$ from Γ, showing that Γ is inconsistent within $\mathfrak{L}$. □

Next, we show that Γ can be extended to a maximal consistent set Γ^+ in which every existential sentence $\exists xA$ has a witness $A(x/c)$. Such sets are called *Henkin sets*.

Definition 3.2

A set of sentences Γ in a first-order language $\mathfrak{L}$ is a *Henkin set in* $\mathfrak{L}$ if it satisfies the following conditions.

(i) Γ is consistent (within $\mathfrak{L}$).

(ii) For every $\mathfrak{L}$-sentence A, either $A \in \Gamma$ or $\neg A \in \Gamma$. (Maximality)

(iii) Whenever Γ contains a sentence of the form $\neg\forall xA$ then it also contains $\neg A(x/c)$ for some individual constant c. (Witnessing)

In the presence of the other conditions, (iii) is equivalent to the requirement that if Γ contains $\neg\forall x\neg A$ then it contains $A(x/c)$ for some c.

Exercise 3.6 Show that if Γ is a Henkin set and $\Gamma \vdash A$, then $A \in \Gamma$.

Exercise 3.7 Show that if Γ is a Henkin set (in a first-order language with identity) then for every closed term t of the language there is an individual constant c such that $t = c$ is in Γ.

Exercise 3.8 Show that if Γ is a Henkin set in some language $\mathfrak{L}$ then Γ contains $\forall xA$ iff Γ contains $A(x/c)$ for every individual constant c in $\mathfrak{L}$.

Lemma 3.3

Every consistent set of $\mathfrak{L}$-sentences Γ can be extended to a Henkin set Γ^+ in any language $\mathfrak{L}^+$ that adds infinitely many individual constants to $\mathfrak{L}$.

Proof. Let Γ be a consistent set of $\mathfrak{L}$-sentences. Let $A_1, A_2, \dots$ be a list of all $\mathfrak{L}^+$-formulas with exactly one free variable. We define a sequence of sets $\Gamma_0, \Gamma_1, \dots$ as follows:

$$\begin{aligned}\Gamma_0 &:= \Gamma\\ \Gamma_{n+1} &:= \Gamma_n \cup \{\neg\forall x A_n \to \neg A_n(x/c_n)\},\end{aligned}$$

where x is the free variable in A_n and c_n is a new $\mathfrak{L}^+$-constant that does not occur in Γ_n. (There must be some such constant because $\mathfrak{L}^+$ contains infinitely many constants that don't occur in Γ.) Let Γ' be the union $\bigcup_n \Gamma_n$ of all sets in this sequence. (That is, a sentence A is in Γ' iff it is in some Γ_n.)

We show that Γ' is consistent. Suppose not. Then there is a derivation of $\bot$ from Γ and the "Henkin sentences" $\neg\forall x A_n \to \neg A_n(x/c_n)$. This derivation can use only finitely many of the Henkin sentences. So one of the Γ_n must be inconsistent. But we can show by induction on n that each Γ_n is consistent.

The base case, for $n = 0$, holds by assumption: Γ is consistent.

For the inductive step, assume Γ_n is consistent and suppose for reductio that Γ_{n+1} is inconsistent. By RAA, we then have

$$\Gamma_n \vdash \neg(\neg\forall x A_n \to \neg A_n(x/c_n)). \tag{1}$$

It's easy to show that if $\Gamma_n \vdash \neg(A \to B)$ then $\Gamma_n \vdash A$ and $\Gamma_n \vdash \neg B$. (Both $\neg(A \to B) \to A$ and $\neg(A \to B) \to \neg B$ are truth-functional tautologies.) So (1) implies

$$\Gamma_n \vdash \neg\forall x A_n, \text{ and} \tag{2}$$

$$\Gamma_n \vdash \neg\neg A_n(x/c_n). \tag{3}$$

From (3), we get $\Gamma_n \vdash A_n(x/c_n)$ by DNE. As c_n does not occur in Γ_n, Gen yields

$$\Gamma_n \vdash \forall x A_n. \tag{4}$$

(2) and (4) show that Γ_n is inconsistent, which contradicts the induction hypothesis.

Next, we extend Γ' to a maximal consistent set Γ^+. The construction follows the proof of Lindenbaum's Lemma (lemma 1.4). Let $S_1, S_2, \ldots$ be a list of all $\mathfrak{L}^+$-sentences. Starting with Γ', we define another sequence of sets $\Gamma'_0, \Gamma'_1, \ldots$:

$$\Gamma'_0 := \Gamma'$$
$$\Gamma'_{n+1} := \begin{cases} \Gamma'_n \cup \{S_n\} & \text{if } \Gamma'_n \cup \{S_n\} \text{ is consistent,} \\ \Gamma'_n \cup \{\neg S_n\} & \text{otherwise.} \end{cases}$$

We show by induction that each Γ'_n is consistent. The *base case*, for $n = 0$, holds because Γ'_0 is Γ', which we've just shown to be consistent. For the *inductive step*, assume that a set Γ'_n in the list is consistent. By lemma 1.3 (which only depends on RAA and therefore also holds for the first-order calculus), either $\Gamma'_n \cup \{S_n\}$ or $\Gamma'_n \cup \{\neg S_n\}$ is consistent. If $\Gamma'_n \cup \{S_n\}$ is consistent, then Γ'_{n+1} is $\Gamma'_n \cup \{S_n\}$ (by construction), so Γ'_{n+1} is consistent. If $\Gamma'_n \cup \{S_n\}$ is not consistent, then Γ'_{n+1} is $\Gamma'_n \cup \{\neg S_n\}$, so again Γ'_{n+1} is consistent.

Let Γ^+ be the union $\bigcup_n \Gamma'_n$ of $\Gamma'_0, \Gamma'_1, \ldots$. Evidently, Γ is a subset of Γ^+. We show that Γ^+ is a Henkin set.

Maximality holds because for each sentence S_n, one of S_n or $\neg S_n$ is in Γ'_{n+1} and therefore in Γ^+.

To show consistency, suppose that Γ^+ is inconsistent. Then there are sentences $A_1, \ldots, A_n$ in Γ^+ from which $\bot$ is deducible. All of these sentences have to occur somewhere on the list $S_1, S_2, \ldots$. Let S_j be the first sentence from $S_1, S_2, \ldots$ that occurs after all the $A_1, \ldots, A_n$. Then all $A_1, \ldots, A_n$ are in Γ'_j. So Γ'_j is inconsistent. But we've seen that all of the Γ'_n are consistent.

It remains to show that Γ^+ has the witnessing property: whenever $\neg\forall x A$ is in Γ^+ then Γ^+ contains a corresponding sentence $\neg A(x/c)$. Let A be any formula in which x is the only free variable. By construction, Γ' contains $\neg\forall x A \to \neg A(x/c)$, for some constant c. Since Γ^+ extends Γ', it also contains this sentence. So if Γ^+ contains $\neg\forall x A$ then it contains $\neg A(x/c)$, by MP and exercise 3.6. □

Next, we show how to read off a model from a Henkin set. The model's domain will consist of sets of closed terms, so that we can stipulate that each constant c denotes the set of all terms t for which $c = t$ is in the Henkin set. Let's have a closer look at these sets.

Definition 3.3

A binary relation R on some domain D is an *equivalence relation* if it is

(i) *reflexive*: for every $x \in D$, xRx;

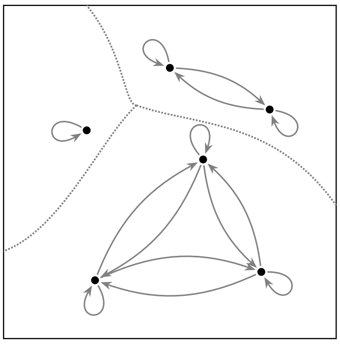
(ii) *symmetric*: for every $x, y \in D$, if xRy then yRx; and

(iii) *transitive*: for every $x, y, z \in D$, if xRy and yRz then xRz.

Lemma 3.4

If Γ is a Henkin set in a language $\mathfrak{L}$ then the relation R that holds between $\mathfrak{L}$-terms t, s iff $t = s \in \Gamma$ is an equivalence relation.

Proof. By A6, $\vdash t = t$. So $t = t \in \Gamma$ by exercise 3.6. Symmetry and transitivity follow similarly from exercise 2.13. □



An equivalence relation *partitions* the domain over which it is defined into distinct cells so that within each cell, all objects stand in the relation to one another. These cells are called *equivalence classes*. If R is an equivalence relation and x an object in the domain, we write '$[x]_R$' for the equivalence class of R that contains x. That is, $[x]_R = \{y \in D : xRy\}$. If the relation R is clear from the context, we may simply write '$[x]$'.

When we're talking about a Henkin set Γ, the relevant equivalence relation is the one defined in lemma 3.4. In what follows, '$[t]$' therefore denotes the set of all terms s for which $t = s \in \Gamma$.

Exercise 3.9 Which of these are equivalence relations on the set of natural numbers? (a) xR_1y iff $x > y$; (b) xR_2y iff $x > y$ or $y > x$; (c) xR_3y iff there are natural numbers m, n such that $x \times 2^m = y \times 2^n$.
Describe the equivalence classes of the relations that are equivalence relations.

The model $\mathfrak{M}$ that we'll construct from a Henkin set Γ will have as its domain the set of all equivalence classes $[t]$, where t is a closed term in the language of Γ. By exercise 3.7, $[t]$ always contains an individual constant. So we can also say that D is the set of all $[c]$ where c is an individual constant in the language of Γ.

We'll stipulate that the interpretation function I of $\mathfrak{M}$ assigns $[c]$ to each constant c. So we'll have $[\![c]\!]^{\mathfrak{M}} = [c]$. We'll extend this to function terms, so that $[\![f(c)]\!]^{\mathfrak{M}} = [f(c)]$. But we can't *directly* stipulate this, because interpretation functions don't assign denotations to complex terms. We have to interpret f as denoting a function on D. By definition 2.9, The denotation of $f(c)$ is the denotation of f applied to the denotation of c. Since the denotation of c is $[c]$, we want the denotation of f to be a function that returns $[f(c)]$ for input $[c]$. So we'll stipulate that for any one-place function symbol f and constant c, $I(f)$ is the function that maps $[c]$ to $[f(c)]$. Similarly for many-place function symbols.

This kind of stipulation can go wrong. Suppose $[c]$ contains two constants c and d, and $[f(c)] \neq [f(d)]$. Our stipulation would entail that $I(f)$ returns $[f(c)]$ for input $[c]$ and $[f(d)]$ for input $[d]$. But if c and d are both in $[c]$ then $[c] = [d]$. And a function can't return two different values for the same input. We must show that this problem can never arise.

A similar issue arises for the interpretation of predicates. To ensure that Ft is in Γ iff $\mathfrak{M} \Vdash Ft$, we'll stipulate that $I(F)$ is the set of all $[t]$ for which $Ft \in \Gamma$. This set isn't well-defined if there are cases where $[t]$ contains two terms t and s for which $Ft \in \Gamma$ but $Fs \notin \Gamma$.

The following lemma shows that neither problem can arise.

Lemma 3.5

If f is an n-ary function symbol, P an n-ary predicate symbol, and $s_1, \dots, s_n, t_1, \dots, t_n$ are terms such that $[s_1] = [t_1], \dots, [s_n] = [t_n]$, then

(i) $[f(s_1, \dots, s_n)] = [f(t_1, \dots, t_n)]$;

(ii) $P(s_1, \dots, s_n) \in \Gamma$ iff $P(t_1, \dots, t_n) \in \Gamma$.

Proof. Assume $[s_i] = [t_i]$ for each $i = 1, 2, \dots, n$. That is, Γ contains $s_i = t_i$, for each $i = 1, 2, \dots, n$.

(i). By A6 (and exercise 3.6), $f(s_1, \dots, s_n) = f(s_1, \dots, s_n)$ is in Γ. By n instances of A7 and MP (and exercise 3.6), it follows that $f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$ is in Γ, which entails that $[f(s_1, \dots, s_n)] = [f(t_1, \dots, t_n)]$.

(ii). Assume $P(s_1, \dots, s_n)$ is in Γ. By n instances of A7 and MP (and exercise 3.6), $P(t_1, \dots, t_n)$ is in Γ. The converse holds by the same reasoning. □

Lemma 3.5 ensures that the following definition is legitimate.

Definition 3.4

If Γ be a Henkin set in a language $\mathfrak{L}$ then the *Henkin model* $\mathfrak{M}_\Gamma$ of Γ is defined as follows.

The domain D of $\mathfrak{M}_\Gamma$ is the set $\{[c] : c \text{ is an individual constant of } \mathfrak{L}\}$.
The interpretation function I of $\mathfrak{M}_\Gamma$ maps

(i) each individual constant c to $[c]$,

(ii) each n-ary function symbol f to the function on D that maps any n objects $[t_1], \dots, [t_n]$ in D to $[f(t_1, \dots, t_n)]$,

(iii) each (non-logical) n-ary predicate symbol P to the set of all n-tuples $\langle [t_1], \dots, [t_n] \rangle$ such that $P(t_1, \dots, t_n) \in \Gamma$.

Now remember what we're trying to achieve. We want to show that every consistent set of sentences is satisfiable. We've shown in lemma 3.3 that every such set can be extended to a Henkin set. Definition 3.4 tells us how to construct a model from this Henkin set. It remains to show that all members of the Henkin set (and therefore all members of the original set) are true in this model.

We'll need the following two facts.

Lemma 3.6

If $\mathfrak{M}$ is a Henkin model and t a closed term then $[\![t]\!]^{\mathfrak{M}} = [t]$.

Proof. The proof is by induction on complexity of t. The base case is covered by clause (i) in definition 3.4. So let t be $f(t_1, \dots, t_n)$. Then

$$\begin{aligned}
[\![f(t_1, \dots, t_n)]\!]^{\mathfrak{M}} &= [\![f]\!]^{\mathfrak{M}}([\![t_1]\!]^{\mathfrak{M}}, \dots, [\![t_n]\!]^{\mathfrak{M}}) && \text{by def. 2.10} \\
&= [\![f]\!]^{\mathfrak{M}}([t_1], \dots, [t_n]) && \text{by ind. hyp.} \\
&= [f(t_1, \dots, t_n)] && \text{by def. 3.4.(iii)} \quad \square
\end{aligned}$$

Lemma 3.7

If $\mathfrak{M}$ is a Henkin model and A a formula then $\mathfrak{M} \Vdash \forall x A$ iff $\mathfrak{M} \Vdash A(x/c)$ for all individual constants c.

Proof. We first show that if $\mathfrak{M}^{d:[c]}$ is a model just like $\mathfrak{M}$ except that it assigns $[c]$ to d, and d does not occur in A, then

$$\mathfrak{M}^{d:[c]} \Vdash A(x/d) \text{ iff } \mathfrak{M} \Vdash A(x/c). \tag{1}$$

There are two cases to consider. If d is c then $\mathfrak{M}^{d:[c]}$ is $\mathfrak{M}$ and (1) holds trivially. If d is a constant other than c, then d does not occur in $A(x/c)$. So $\mathfrak{M}^{d:[c]}$ and $\mathfrak{M}$ agree on the interpretation of all symbols in $A(x/c)$ and we have $\mathfrak{M}^{d:[c]} \Vdash A(x/c)$ iff $\mathfrak{M} \Vdash A(x/c)$ by the coincidence lemma (lemma 2.1). Also, by the extensionality lemma (lemma 2.2), $\mathfrak{M}^{d:[c]} \Vdash A(x/d)$ iff $\mathfrak{M}^{d:[c]} \Vdash A(x/c)$. So (1) holds.

Now, by definition 2.10, $\mathfrak{M} \Vdash \forall xA$ iff $\mathfrak{M}' \Vdash A(x/d)$ for every model $\mathfrak{M}'$ that differs from $\mathfrak{M}$ at most in the object assigned to d, where d is the alphabetically first constant that does not occur in A. Since each such $\mathfrak{M}'$ has the same domain as $\mathfrak{M}$, the set of such $\mathfrak{M}'$ is precisely the set of variants $\mathfrak{M}^{d:[c]}$ of $\mathfrak{M}$. So $\mathfrak{M} \Vdash \forall xA$ iff $\mathfrak{M}^{d:[c]} \Vdash A(x/d)$ for every constant c, which, by (1), is the case iff $\mathfrak{M} \Vdash A(x/c)$ for every constant c. □

Lemma 3.8: Truth Lemma

If $\mathfrak{M}$ is the Henkin model of a Henkin set Γ in a language $\mathfrak{L}$, then for every $\mathfrak{L}$-sentence A, $\mathfrak{M} \Vdash A$ iff $A \in \Gamma$.

Proof by induction on complexity of A.

Base case: A is atomic. There are two subcases.

Assume that A is an identity sentence $s = t$. Then $\mathfrak{M} \Vdash s = t$ iff $[\![s]\!]^{\mathfrak{M}} = [\![t]\!]^{\mathfrak{M}}$ by definition 2.10, iff $[s] = [t]$ by lemma 3.6, iff $s = t \in \Gamma$ by definition of the equivalence classes.

Assume next that A is an atomic sentence $P(t_1, \dots, t_n)$, where P is non-logical. Then $\mathfrak{M} \Vdash P(t_1, \dots, t_n)$ iff $([\![t_1]\!]^{\mathfrak{M}}, \dots, [\![t_n]\!]^{\mathfrak{M}}) \in [\![P]\!]^{\mathfrak{M}}$ by definition 2.10, iff $([t_1], \dots, [t_n]) \in [\![P]\!]^{\mathfrak{M}}$ by lemma 3.6, iff $P(t_1, \dots, t_n) \in \Gamma$ by definition 3.4.

Inductive step: A is composed of other sentences. We have three subcases. The first two, where A is $\neg B$ or $B \to C$, are easy and go exactly as in lemma 1.6. I'll skip them.

For the final subcase, assume that A is $\forall xB$, for some formula B. We have: $\mathfrak{M} \Vdash \forall xB$ iff $\mathfrak{M} \Vdash B(x/c)$ for every constant c by lemma 3.7, iff $B(x/c) \in \Gamma$ for every constant c by induction hypothesis, iff $\forall xB \in \Gamma$ by exercise 3.8. □

With that, we have all the ingredients for the completeness proof.

Theorem 3.3: Completeness of the first-order calculus (Gödel 1929)

If $\Gamma \models A$ then $\Gamma \vdash A$.

Proof. We argue by contraposition. Assume $\Gamma \nvdash A$. Then $\Gamma \cup \{\neg A\}$ is consistent, by lemma 3.1. By lemma 3.3, $\Gamma \cup \{\neg A\}$ can be extended to a Henkin set Γ^+ in an extended language $\mathfrak{L}^+$. By the Truth Lemma, the Henkin model of Γ^+ satisfies every sentence in Γ^+, and therefore every sentence in $\Gamma \cup \{\neg A\}$. So there is a model that satisfies all sentences in Γ but doesn't satisfy A. So $\Gamma \not\models A$. □

Technically, the model that figures in this proof is not a model for the original language $\mathfrak{L}$ of Γ and A, because it also interprets the added individual constants. If this bothers you, we can define an $\mathfrak{L}$-model that falsifies $\Gamma \models A$ by restricting the interpretation function of the Henkin model to $\mathfrak{L}$-constants, without changing the domain. This is called the *reduct* of the Henkin model to $\mathfrak{L}$.

Exercise 3.10 Assume that $\Gamma_0, \Gamma_1, \ldots$ are consistent sets of sentences in a first-order language $\mathfrak{L}$, and that each Γ_i is a subset of Γ_{i+1}. Show that their union $\bigcup_i \Gamma_i$ is consistent.

3.4 Unintended Models

We've now shown that the first-order calculus is both sound (theorem 2.4) and complete (theorem 3.3): $\Gamma \models A$ iff $\Gamma \vdash A$. As Gödel pointed out, this has a curious consequence:

Theorem 3.4: Compactness of first-order logic (Gödel 1929)

If every finite subset of a set of sentences is satisfiable then the set itself is satisfiable.

Proof. Let Γ be a set of first-order sentences. Assume that Γ is not satisfiable. So $\Gamma \models \bot$. By completeness, it follows that $\Gamma \vdash \bot$: there is a deduction of $\bot$ from Γ. This deduction can only use finitely many sentences from Γ. So there is a finite subset Γ' of Γ for which $\Gamma' \vdash \bot$. By soundness, $\Gamma' \models \bot$. □

Why is this curious? Well, for one, it is easy to come up with cases where a conclusion is entailed by infinitely many premises, but not by any finite subset of those premises. For example, consider the premises 'I like the number 0', 'I like the number 1', 'I like the number 2', and so on, for all natural numbers. Together, these premises entail 'I like all natural numbers'. But no finite subset of the premises does. This doesn't contradict the compactness theorem because the inference isn't *logically* valid: it depends on the interpretation of '0', '1', '2', …, and 'natural number'. Still, it's curious that first-order logic doesn't allow any such inference to be valid. More directly, compactness implies that there is no way to fully pin down the interpretation of '0', '1', '2', etc. in a first-order language. Let's think through this more carefully.

Many branches of mathematics can be seen as studying certain *mathematical structures*. A mathematical structure consists of a set of objects together with some operations and relations on these objects. For example, arithmetic studies operations and relations on the natural numbers. A formalized, first-order theory of arithmetic will have nonlogical symbols for, say, addition ('+'), multiplication ('×'), and the less-than relation ('<'). We might add an individual constant 'n' for each number n, but for the sake of economy we can instead have a single constant '0' for 0 and another symbol 's' for the successor function that maps each number n to its successor $n+1$. Instead of '1', we can then write '$s(0)$'; '2' is '$s(s(0))$', and so on.

In *logic*, we abstract away from the meaning of the nonlogical symbols. In *arithmetic*, we don't. In arithmetic, '$1 + 2 = 3$' (that is, '$s(0) + s(s(0)) = s(s(s(0)))$') is a definite claim about the natural numbers. We say that it has an *intended interpretation*, or an *intended model*.

The intended model of arithmetic, also known as the *standard model* of arithmetic, or $\mathfrak{A}$, has as its domain the set of natural numbers $\mathbb{N}$; it interprets '0' as denoting 0, '+' as denoting the addition function +, '×' as denoting the multiplication function ×, 's' as denoting the successor function s, and '<' as denoting the less-than relation <. We can represent this as a list: $\langle \mathbb{N}, 0, +, \times, s, < \rangle$. The list represents the "structure" of the natural numbers. It identifies a set $\mathbb{N}$ and some operations and relations on that set that are picked out by the nonlogical symbols of the language. (0 counts as a zero-ary operation.)

Of course, the language of arithmetic also has unintended models. For example, we can let the domain be { Athens, Berlin } and use an interpretation function that maps '0' to Athens, '+' and '×' to the function that maps any pair of cities to Athens, 's' to the function that maps each city to itself, and '<' to the empty relation. In this model, '$s(0) + s(s(0)) = s(s(s(0)))$' is true, but so is '$s(0) = 0$'.

Exercise 3.11 Is Lagrange's Theorem (every natural number is the sum of four squares) true in this model?

We might hope that such unintended models can always be ruled out by laying down sufficiently many postulates in the formal language of arithmetic. For example, if our theory of arithmetic contains '$s(0) \neq 0$' then the above model (with Athens and Berlin) is no longer a model of the theory. So we can rule out *some* unintended models. The compactness theorem dashes any hope of ruling out *all* unintended models. Let $\mathrm{Th}(\mathfrak{A})$ be the set of all sentences true in the standard model $\mathfrak{A}$. This is called the *theory of* $\mathfrak{A}$. It contains all postulates we could possibly lay down, assuming that they should all be true in the standard model. By compactness, there is a model of $\mathrm{Th}(\mathfrak{A})$ whose domain includes junk elements that clearly aren't natural numbers.

Theorem 3.5: Non-standard Models of Arithmetic

$\mathrm{Th}(\mathfrak{A})$ has models in which some object is not reachable from 0 by finitely many applications of the successor function. (Such models are called *non-standard models* of arithmetic.)

Proof. Let $\mathfrak{L}_A^+$ be the language of arithmetic with an added individual constant c. Let $\mathrm{Th}(\mathfrak{A})^+$ be the set of sentences that extends $\mathrm{Th}(\mathfrak{A})$ by

$$c \neq 0, \quad c \neq s(0), \quad c \neq s(s(0)), \quad c \neq s(s(s(0))), \quad \ldots.$$

So $\mathrm{Th}(\mathfrak{A})^+$ says that c is different from 0, 1, 2, and so on, for all natural numbers. Every finite subset of $\mathrm{Th}(\mathfrak{A})^+$ is obviously satisfiable: it is true in the standard model $\mathfrak{A}$, interpreting c as a sufficiently large number. By the compactness theorem, it follows that $\mathrm{Th}(\mathfrak{A})^+$ is satisfiable. Any model of $\mathrm{Th}(\mathfrak{A})^+$ must have an element (denoted by c) that is not identical to any natural number: it can't be reached from 0 by finitely many applications of the successor function. The reduct of any such model to the original language of arithmetic (discarding the interpretation of c) is a model of $\mathrm{Th}(\mathfrak{A})$. □

Compactness thus reveals a deep expressive limitation of first-order logic: the structure of the natural numbers can't be captured in a first-order language.

Here is a simpler example of this kind of limitation. For each $n > 0$, it is easy to find a first-order sentence S_n that is true in all and only the models with exactly n elements. For example, $\exists x \exists y (x \neq y \land \forall z (z = x \lor z = y))$ is true in all and only models with exactly two elements. (Compare exercise 2.16.) In that sense, we can capture the intended size of a domain, as long as that size is a fixed finite number. But there is no first-order sentence that is true in all and only the models with infinitely many elements.

To see why, let S_∞ be such a sentence. Its negation $\neg S_\infty$ would be true in all and only the finite models. Now consider the set consisting of $\neg S_\infty$ together with $\neg S_1, \neg S_2, \neg S_3, \ldots$. All finite subsets of this set are satisfiable. By compactness, the whole set is satisfiable. But there is no model whose domain not infinite, and yet also has no finite size.

The following theorems show that the situation is even worse. In section 3.1, we saw that there are many levels of infinity: many infinite cardinalities. None of them can be captured in first-order logic, insofar as there is no sentence (nor even a set of sentences) that would force a model to have a particular infinite cardinality, or even to fall in any non-trivial range of infinite cardinalities.

Theorem 3.6: The (Downward) Löwenheim-Skolem Theorem

If a set of sentences in a countable first-order language has a model, then it has a countable model.

Proof. Let Γ be such a set. By lemma 3.3, Γ can be extended to a Henkin set Γ^+. By lemma 3.8, the Henkin model of Γ^+ is a model of Γ. Its domain consists of equivalence classes of closed terms in the language of Γ^+. If the language of Γ is countable, then so is the language of Γ^+. So the domain of the Henkin model is countable. □

Theorem 3.7: The Upward Löwenheim-Skolem Theorem (Tarski 1935)

If a set of sentences in a countable first-order language has an infinite model, then it has models of every infinite cardinality.

Proof sketch. The proof requires relaxing our stipulation that a first-order language must only have countably many individual constants. The completeness theorem can be proved without this assumption (although the proof becomes more complicated, as we can no longer appeal to enumerations of the sentences in the language). Compactness still follows from completeness, and we get a slightly generalized downward Löwenheim-Skolem theorem according to which every satisfiable set of sentences has a model whose cardinality is at most equal to the cardinality of the language.

Now let Γ be a set of first-order sentences in a countable language with an infinite model. From the downward theorem, we know that Γ has a countably infinite model $\mathfrak{M}$. Let κ be any cardinal greater than $\aleph_0$. Expand the language of Γ by κ new individual constants. For each pair of these constants c_i and c_j, add the sentence $c_i \neq c_j$ to Γ, thereby creating the set Γ^+. All finite subsets of Γ^+ are satisfied in $\mathfrak{M}$, with the (finitely many) new constants in the set interpreted as distinct objects in $\mathfrak{M}$. By the compactness theorem (for uncountable languages), Γ^+ has a model $\mathfrak{M}^+$. All the κ new constants denote distinct objects in this model, so $\mathfrak{M}^+$ has at least κ objects. By the generalized downward theorem, it follows that Γ^+ has a model whose cardinality is exactly κ. The reduct of that model to the language of Γ is a model of Γ with cardinality κ. □

The downward theorem was proved by Thoralf Skolem in 1920, building on an earlier proof sketch by Leopold Löwenheim. The (ill-named) upward theorem, due to Alfred Tarski, requires compactness and could only be proved after 1929. It entails, among other things, that $\text{Th}(\mathfrak{A})$ has non-standard models of every infinite cardinality.

Exercise 3.12 (a) Can you find a first-order sentence that is only true in infinite models? (Hint: let f be a function that is injective, but not surjective.) (b) Is the negation of this sentence only true in finite models?

Exercise 3.13 Consider a first-order language with a binary predicate symbol 'P' for the "parent" relation, so that Pxy means (on its intended interpretation) that x is a parent of y. We can then define the "grandparent" relation $P^2(x,y)$ as $\exists z_1(Pxz_1 \land Pz_1y)$, the "great-grandparent" relation $P^3(x,y)$ as $\exists z_1 \exists z_2(Pxz_1 \land Pz_1z_2 \land Pz_2y)$, and so on. Show that there is no way to define the "ancestor" relation (no matter what other nonlogical symbols we add to the language): there can be no formula $A(x,y)$ that is equivalent to the infinite disjunction $P(x,y) \lor P^2(x,y) \lor P^3(x,y) \lor \ldots$.

Exercise 3.14 Let $\mathfrak{A}^c$ be exactly like the standard model of arithmetic $\mathfrak{A}$, except that the number 2 is replaced by Julius Caesar: the domain of $\mathfrak{A}^c$ is $\{0, 1, \text{Caesar}, 3, 4, \ldots\}$; '$s$' denotes a function that maps 0 to 1, 1 to Caesar, Caesar to 3, ...; '+' denotes a function that maps 1 and 1 to Caesar, 1 and Caesar to 3, and so on. Is $\mathfrak{A}^c$ a model of $\text{Th}(\mathfrak{A})$? In what way is it less interesting than the non-standard models of theorem 3.5?

Exercise 3.15 Explain why every satisfiable set of first-order sentences has a model whose domain is a set of natural numbers.

4 Theories

In this chapter, we take a look at first-order theories: sets of sentences in a formal, first-order language that are assumed to describe a particular domain of objects. A theory might describe the behaviour of physical systems or moral norms, but we'll focus on mathematical theories – specifically, arithmetic and set theory.

4.1 Arithmetic

For most of history, people did maths in an informal manner, relying on a loose collection of techniques for solving specific types of problems. When giving proofs, assumptions that seemed obviously true – for example, that $0 \neq 1$ – were simply taken for granted.

In the 19th and early 20th century, mathematics was put on a more rigorous footing. Cauchy, Weierstrass, Dedekind, and others gave precise definitions of mathematical concepts (such as limits and continuity). They also formalized the exact assumptions that were needed to derive well-known results. These assumptions were collected into *axioms* for the relevant area of maths.

At the same time, more powerful mathematical theories were developed, such as the set theory of Cantor (formalized by Zermelo, Fraenkel, and others) or the type theory of Russell and Whitehead. All known branches of maths, it seemed, could be unified in such a theory, allowing for new results to be derived from the emerging connections between previously separate domains. Theorems from topology could be used to prove results in algebra.

Formally, a *theory* is a set of sentences that is closed under entailment, so that it contains everything that is entailed by it. In this chapter, we'll be concerned with theories in a formal, first-order language. We often write $\vdash_T A$ or $T \vdash A$ (rather than $A \in T$) to say that a sentence A is a member of the theory T.

This fits our earlier use of the turnstile: If A is in T, then $T \vdash A$ (by Mon and Id); conversely, if $T \vdash A$, then by the completeness of first-order logic, $T \models A$, and then A is in T because T is closed under entailment. We could write $T \models A$ instead of $T \vdash A$. Conceptually, however, theories belong to the "syntax" or "proof theory" side of logic.

A theory is simply a set of sentences. This set is usually specified by laying down some non-logical axioms. The theory then contains all and only the sentences that can be derived from these axioms. We say that a theory T is *axiomatized* by a set of sentences Γ if it contains exactly the sentences that are derivable from Γ.

Exercise 4.1 Let $\mathfrak{L}$ be some first-order language (with identity). Let T_1 be the theory axiomatized by the set of all $\mathfrak{L}$-sentences, T_2 the theory axiomatized by the empty set of sentences, and T_3 the theory axiomatized by $\{\forall x\, x \neq x\}$. Which of T_1, T_2, and T_3 are the same?

Let's take a closer look at formal theories of arithmetic. Arithmetic is the study of the natural numbers 0, 1, 2, 3, etc. We know a lot about the natural numbers. We know, for example, that $1 + 2 = 3$, that there are infinitely many primes, or that the factorial function $x!$ grows faster than any polynomial x^n. The aim of an axiomatized formal theory of arithmetic is to capture all such truths, showing exactly which assumptions (or axioms) are needed to derive which results.

An important part of the axiomatic project is to reduce the number of primitive concepts. In Section 3.4, I already mentioned that we don't need separate individual constants for each number: we can instead use a single constant '0' for the number 0 and a function symbol 's' for the successor function; the number 1 is then denoted by '$s(0)$', the number 2 by '$s(s(0))$', and so on. This is useful because it means that we don't need special axioms for each number: having defined 1, 2, and 3 as $s(0)$, $s(s(0))$, and $s(s(s(0)))$, respectively, we may hope to derive that $1 + 2 = 3$ from general assumptions about zero and the successor function. If '1', '2', and '3' were primitive symbols, it is hard to see how '$1 + 2 = 3$' could be derived from more basic principles.

In Section 3.4, I suggested that a first-order theory of arithmetic might use primitive symbols for 0, the successor function, addition, multiplication, and the less-than relation. In fact, the less-than relation can be defined in terms of the other concepts and logical expressions, since the following holds for all natural numbers x and y:

$$x < y \text{ iff } \exists z(x + s(z) = y).$$

We can therefore treat '$t_1 < t_2$', for any terms t_1 and t_2, as a metalinguistic abbreviation of '$\exists x(t_1 + s(x) = t_2)$'. (The variable x must not occurring in t_1 or t_2.)

Less obviously, we can define the concept of a prime number. Remember that a number is prime if it is greater than 1 and divisible only by 1 and itself. A number x is divisible by a number y if there is a number z such that $z \times y = x$. Thus we can express

't is prime' as:

$$s(0) < t \wedge \forall y(\exists z(z \times y = t) \to (y = s(0) \vee y = t)).$$

Exercise 4.2 Define the concepts of (a) an even number and (b) a square number.

Other concepts are harder to define. It is not obvious how one could define exponentiation x^y or the factorial $x!$ in terms of 0, s, +, and ×. We'll see in chapter 8 how it can be done. Indeed, we'll see that all computable functions and relations on the natural numbers can be defined in terms of our four primitives. That is, whenever there is an algorithm for computing a function, or for determining whether a relation holds between some numbers, then the function or relation can be defined in terms of 0, s, +, and ×.

Exercise 4.3 Can you find another primitive that we could use instead of 's'? (That is, can you find a primitive symbol φ so that $s(t)$ can be defined from 0, φ, +, and ×?)

Let's turn to the second part of the axiomatic project. Having reduced the set of primitive concepts, we need to lay down axioms that describe how the remaining concepts behave. The aim is to reduce all truths about the natural numbers to a small number of basic principles.

The first axioms we'll consider are just about 0 and s. Later, we'll add axioms for + and ×. What do we know about 0 and s? We know, for example, that every number has a successor. But we don't need to postulate this as an axiom: all function symbols in first-order logic denote total functions. What isn't guaranteed is that 's' denotes an injective function: we need to postulate that no two numbers have the same successor.

Q1 $\forall x \forall y\, (s(x) = s(y) \to x = y)$

We also know that 0 is not the successor of any number:

Q2 $\forall x\, 0 \neq s(x)$

These two axioms are already quite powerful. Let's think about what a model of them must look like. There must be at least one object, denoted by 0. There must also be an object $s(0)$. Can this be the same as 0? No: otherwise 0 would be the successor of itself, which contradicts Q2. So $s(0)$ is another object. What about $s(s(0))$? This can't be 0, by Q2. And so it can't be $s(0)$ either, by Q1: if $s(s(0)) = s(0)$, then $s(0)$ and 0 would have the same successor. So $s(s(0))$ is a third object. By iterating this reasoning, we can see that any model of Q1 and Q2 must have a chain of infinitely many objects

$$0, s(0), s(s(0)), s(s(s(0))), \ldots,$$

connected by the successor function.

Exercise 4.4 Can you find a model in which Q1 and Q2 are true, but $\forall x(s(x) \neq x)$ is false?

Exercise 4.4 shows that Q1 and Q2 don't suffice to capture all truths about 0 and s. The problem is that the two axioms don't rule out the existence of other objects, outside the chain $0, s(0), s(s(0)), \ldots$. On these other objects, the successor relation must still be injective, but it can go in a loop, or it can form a second infinite chain $a, s(a), s(s(a)), \ldots$. The following axiom rules out such additional chains, by stipulating that there is no object other than 0 that is not a successor.

Q3 $\forall x\,(x \neq 0 \to \exists y\, x = s(y))$

This doesn't help with the looping case, however. We'd like to have an axiom saying that every number can eventually be reached from 0 by repeated application of s. But there's no way to express this in first-order logic (as we proved in section 3.4). Still, we can get close by adding the following axiom schema, called the *induction schema*:

Ind $(A(0) \land \forall x\ (A(x) \to A(s(x)))) \to \forall x\, A(x)$

Here, $A(x)$ is any formula with one free variable. Think of every such formula as expressing a property. Ind then says that if some (expressible) property holds of 0, and if it is inherited from any number to its successor, then it holds of all numbers. The schema is obviously related to the method of inductive proof, where we show that all numbers have a property by showing that 0 has it and that it is inherited from any number to its successor.

Ind rules out the looping case. Consider the simplest version, where there's an object a outside the chain $0, s(0), s(s(0)), \ldots$ that is its own successor. In this model, $\forall x(s(x) \neq x)$ is false. But $\forall x(s(x) \neq x)$ follows from Q1, Q2, and Ind, as follows.

Let $A(x)$ be the formula $s(x) \neq x$. Then $A(0)$ is $s(0) \neq 0$. This is entailed by Q2. $\forall x(A(x) \rightarrow A(s(x)))$ is $\forall x(s(x) \neq x \rightarrow s(s(x)) \neq s(x))$. This is entailed by Q1. By Ind, we can derive $\forall x(s(x) \neq x)$.

Exercise 4.5 How does Ind rule out loops with two elements? That is, why isn't there a model of Q1, Q2, and Ind with two objects a and b outside the chain $0, s(0), s(s(0)), \ldots$ that are successors of each other?

Ind also rules out models with a second chain $a, s(a), s(s(a)), \ldots$. We can see this from the fact that it entails Q3:

Proposition 4.1

Ind entails Q3.

Proof. let $A(x)$ be the formula $x \neq 0 \rightarrow \exists y\, x = s(y)$. Q3 is $\forall x A(x)$. To derive this via Ind, we need to derive

(i) $A(0)$, and
(ii) $\forall x(A(x) \rightarrow A(s(x)))$.

Both of these are valid (and therefore provable) in pure first-order logic. (i) holds because $\models 0 = 0$; so the antecedent of $A(0)$ is false and $A(0)$ is true. For (ii), note that the consequent of $A(s(x))$ is $\exists y(s(x) = s(y))$, which is trivial; so $A(s(x))$ can never be false; so $A(x) \rightarrow A(s(x))$ is always true. □

Let's turn to addition and multiplication. A common way to define a function on the natural numbers is to describe how it applies to 0 and then define its value for any successor number in terms of its value for the previous number. For example, the factorial function $n!$ that maps every number n to the product $1 \times 2 \times \ldots \times n$ can be defined by the following two clauses:

(i) $0! = 1$
(ii) $s(n)! = n! \times s(n)$

This is called a definition by *(primitive) recursion*. It may at first look circular, but it is not. Take, for example, the input 2 to the factorial function. By clause (ii) of the definition, 2! is 1! × 2. To evaluate this, we need to know 1!. By clause (ii) again, 1! is 0! × 1. By the first clause, 0! is 1. Putting all this together, we have

$$2! = (1 \times 1) \times 2 = 2.$$

We can similarly define the addition function by primitive recursion on its second argument:

(i) $x + 0 = x$
(ii) $x + s(y) = s(x + y)$

These two claims are easily translated into the language of arithmetic, which gives us our next two axioms:

Q4 $\forall x(x + 0 = x)$
Q5 $\forall x \forall y(x + s(y) = s(x + y))$

The same trick works for multiplication, which we can define as repeated addition:

Q6 $\forall x(x \times 0 = 0)$
Q7 $\forall x \forall y(x \times s(y) = (x \times y) + x)$

Exercise 4.6 Explain how the primitive recursive definition of addition determines the value of 3 + 2.

The theory axiomatized by Q1–Q7 is called *Robinson Arithmetic*, or Q. It will play an important role in chapter 9. The standard first-order theory of arithmetic, called *Peano Arithmetic*, or PA, replaces Q3 by Ind: its axioms are Q1, Q2, Ind, and Q4–Q7. (The theory is named after Giuseppe Peano, although Peano points out that essentially the same theory was proposed earlier by Dedekind).

Are all truths in the language of arithmetic entailed by the axioms of PA? For a while, this seemed plausible. Gödel's first *incompleteness* theorem revealed that the answer is no: there are arithmetical truths that aren't provable in PA. So PA $\neq$ Th($\mathfrak{A}$). We'll prove this in ch. 9. As we'll see, the problem can't be fixed by adding a few more axioms or axiom schemas. PA isn't just incomplete; there's a good sense in which it is *incompletable*.

Exercise 4.7 Show that the following are in PA:
(a) $\forall x\, x < s(x)$; (b) $\forall x \forall y(x < y \to 0 < y)$; (c) $\forall x \forall y(x + y = y + x)$.

Exercise 4.8 We've seen that Q1–Q3 don't rule out structures in which the successor function goes in a loop for some objects outside $0, s(0), s(s(0)), \ldots$.

(a) Show that adding Q4–Q7 doesn't help: define a model $\mathfrak{M}$ of Q1–Q7 with two objects a and b that are successors of each other.

(b) Using the definition of '<' from earlier in this section, determine whether $a < b$, $a < a$, and $0 < a$ are true in your model $\mathfrak{M}$.

(c) Is $\forall x \forall y(x+y = y+x)$ true in your model $\mathfrak{M}$? If yes, change the interpretation of + to make it false while keeping Q1–Q7 true.

4.2 Set theory

In the 19th century, set-theoretic concepts were increasingly used by mathematicians to make their theories and definitions more precise. For example, Dedekind defined the real numbers in terms of sets of rational numbers, which allowed for new, more rigorous proofs of many results in real analysis.

The concept of a set was initially not seen as belonging to a separate mathematical theory (*set theory*). Rather, it was treated as a logical concept. To speak of the set of such-and-suchs, it was assumed, is just to speak of the such-and-suchs taken together. As Georg Cantor put it in 1895: a set is 'a collection of definite, well-differentiated objects […] into a whole'. It was assumed that, as a matter of logic, whenever there are some (definite, well-differentiated) objects, there is also a set of these objects.

Dedekind had defined the real numbers in terms of sets of rational numbers. The rational numbers can, in turn be defined in terms of sets and integers, and the integers in terms of sets and natural numbers. Frege realized that one can define the natural numbers entirely in terms of sets. (See section 4.3 below for one way to do this.) Familiar properties of the natural numbers – and, by extension, of the integers, rationals, and reals – can then be derived from apparently logical properties of sets. Hence there emerged the philosophical project of *logicism*: the idea that all of maths could be reduced to logic and definitions.

This was the life project of Frege, who invented the calculus of predicate logic in order to show that all of arithmetic could be derived from purely logical axioms by simple logical rules like MP and Gen. Frege's "logical axioms" included one assumption about sets – his "axiom V". This is a second-order axiom involving the term-forming operator $\{x : A(x)\}$. We can express it as a first-order schema:

$$\text{V} \quad \{x : A(x)\} = \{x : B(x)\} \leftrightarrow \forall x(A(x) \leftrightarrow B(x)).$$

$A(x)$ and $B(x)$ are arbitrary formulas with one free variable. Axiom V says that different sets never have the very same members. This makes sense if a set of things is just those things "considered as a whole". But the use of the set operator $\{x : A(x)\}$ in an otherwise standard first-order language also implies that for any formula $A(x)$ there is a corresponding set $\{x : A(x)\}$. This is known as the *naive comprehension principle*.

Unfortunately for Frege, the naive comprehension principle is inconsistent, as Bertrand Russell pointed out to him in a letter in 1902. Consider the formula $x \notin x$, saying that x is not a member of itself. Assume that there is a set of all things to which this formula applies. Call this set R. Is R a member of itself? If it is, then by the definition of R, it is not a member of itself. If it isn't, then by the definition of R, it is a member of itself. This is a contradiction. So $x \notin x$ is a formula for which there is no corresponding set $\{x : x \notin x\}$.

There is something odd about the idea that a set might contain itself. One imagines sets as abstract ”containers", and a container can hardly contain itself. Ernst Zermelo, who had independently noticed Russell's paradox, developed this intuition into a paradox-free formal theory.

According to Zermelo, we should think of the sets as built in layers or stages. We start with things that are not sets, called *individuals* or *urelements*. At the next stage, we form all sets of these individuals. We may now have sets of rocks and cities, like $\{\text{Athens}, \text{Berlin}\}$, but we don't have any sets containing other sets. At the next stage, we form all sets whose elements are either individuals or sets of individuals. This includes all sets from the first stage, but it also includes sets like $\{\{\text{Athens}, \text{Berlin}\}, \text{Athens}\}$, with sets from the previous stage as elements. We continue in this manner. Whenever a set occurs at some stage, it can be used as an element of sets at later stages. But not otherwise: a set can only appear at a stage after all its elements have appeared. So we never get a set that contains itself. Nor do we get a set of all sets that don't contain themselves: this would be the set of all sets; such a set would contain itself, which is impossible.

Oddly, this hierarchical construction works even if there are no individuals. Starting with no individuals, we can construct one set of individuals: the empty set $\emptyset$. From this, we can form another set: $\{\emptyset\}$. And once we have $\emptyset$ and $\{\emptyset\}$, we can form $\{\emptyset, \{\emptyset\}\}$ and $\{\{\emptyset\}\}$. And off we go. For purely mathematical applications, it turns out that this *pure* hierarchy is often enough.

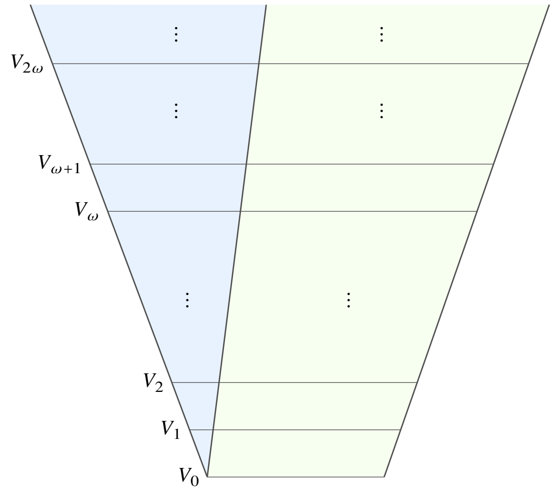
Let's make the structure of the set-theoretic hierarchy, called the *cumulative hierarchy*, or simply V, more precise. Each stage of the hierarchy is a set of sets. The first stage, V_0, is the set of individuals. In the pure hierarchy, this is the empty set:

$$V_0 = \emptyset.$$

From any stage V_k, we recursively define the next stage V_{k+1} as the set of all sets whose elements are in V_k. This is just the power set of V_k:

$$V_{k+1} = \mathscr{P}(V_k).$$

In the pure hierarchy, $V_1 = \mathscr{P}(\emptyset) = \{\emptyset\}$, $V_2 = \mathscr{P}(\{\emptyset\}) = \{\emptyset, \{\emptyset\}\}$, and so on. This yields an infinite sequence $V_0, V_1, V_2, \dots$ of ever-larger sets, all ultimately built from the empty set.



But we don't stop there. After all the stages $V_0, V_1, V_2, \dots$, there is another stage V_ω ("V omega"). V_ω contains all sets that have appeared at any earlier stage. That is, V_ω is the union of all earlier stages:

$$V_\omega = \bigcup_{k<\omega} V_k.$$

While all sets in the sequence $V_0, V_1, V_2, \dots$ are finite, V_ω has infinitely many elements.

From V_ω, we can form yet further sets by repeating the previous recipes. At stage $V_{\omega+1}$, we collect all the subsets of V_ω. (Many of these are infinite and thus didn't appear at any earlier stage.) That is, $V_{\omega+1} = \mathscr{P}(V_\omega)$. We then form $V_{\omega+2} = \mathscr{P}(V_{\omega+1})$, and so on. After all the stages $V_\omega, V_{\omega+1}, V_{\omega+2}, \dots$, there is another stage $V_{\omega+\omega}$, or $V_{\omega\cdot 2}$. It contains all sets that have appeared at any earlier stage: $V_{\omega\cdot 2} = \bigcup_{k<\omega} V_{\omega+k}$. From $V_{\omega\cdot 2}$, we construct $V_{\omega\cdot 2+1}$, $V_{\omega\cdot 2+2}$, etc. by taking power sets. Then we construct $V_{\omega\cdot 3}$ by taking the union of all earlier stages. And so on and on.

And we don't stop there. After all the stages $V_\omega, \dots, V_{\omega\cdot 2}, \dots, V_{\omega\cdot 3}, \dots$, there is another stage $V_{\omega\cdot\omega}$, or V_{ω^2}, where we take the union of all previous stages. From this, we construct further stages by taking power sets and unions. Eventually, we reach V_{ω^3}, then V_{ω^4}, etc. Then we take the union of all these stages to get V_{ω^ω}, and so on and on, through $V_{\omega^{\omega^{\omega}}}$, through stages with infinitely high towers of ω, and much, much further. The cumulative hierarchy is *vast*.

Exercise 4.9 Consider the pure hierarchy. How many sets are in V_3? How many are in V_4?

Exercise 4.10 Is the cardinality of $V_{\omega+1}$ greater than the cardinality of V_ω?

Let's now try to axiomatize this conception of sets. That is, we'll try to find a set of sentences in a suitable first-order language that describes the structure of the cumulative hierarchy. The description I just gave, with its 'and so on's and 'after all these stages' can't be directly translated into first-order logic. We have to take a more indirect approach.

The most popular axiomatization of set theory is *ZFC*, for 'Zermelo-Fraenkel set theory with the Axiom of Choice'. Its only primitive concept is the membership relation. So we have a single non-logical symbol: the binary predicate symbol '$\in$'. From this, other concepts are defined. For example, we can define the subset relation $\subseteq$ as follows:

$t_1 \subseteq t_2$ abbreviates $\forall x(x \in t_1 \to x \in t_2)$.

Let's go through the axioms of ZFC. The quantifiers are assumed to range over the pure sets. Our first axiom is known as the axiom of *extensionality*.

Z1 $\quad \forall x \forall y((\forall z(z \in x \leftrightarrow z \in y)) \to x = y)$.

This says that a set is determined by its elements: no two sets have the same elements. Unlike Frege's Axiom V, Z1 doesn't imply that for any formula $A(x)$ there is a corresponding set $\{x : A(x)\}$. Instead of this unrestricted comprehension principle, we have a more restricted principle, called the *separation axiom*. It's actually a schema:

Z2 $\quad \forall y \exists z \forall x(x \in z \leftrightarrow (x \in y \land A(x)))$

This says that for any set y and any formula $A(x)$, there is a set z that contains just those elements of y of which $A(x)$ is true. That is, provided that we already have a set y, we can use any formula to carve out a subset of y containing those elements of y of which the formula is true.

The next axiom postulates the existence of the empty set, the base level of the hierarchy.

Z3 $\exists x \forall y (y \notin x)$.

This says that there is something (a set) that has no elements. By the extensionality axiom, there is only one such thing. It's convenient to have a name for it: '$\emptyset$'. But '$\emptyset$' isn't officially part of the language. The only singular terms in the language of set theory are variables. So we can't say that '$\emptyset$' is shorthand for some more complex term in the language, in the way we could treat '3' as shorthand for 's(s(s(0)))'. What we can do instead is give a *contextual* or *syncategorematic* definition of '$\emptyset$', as follows:

$A(\emptyset)$ abbreviates $\exists x(\forall y\, y \notin x \;\wedge A(x))$.

Here, $A(x)$ is an expression with one free variable x, and $A(\emptyset)$ is that expression with '$\emptyset$' in place of x. For example, consider the expression

$$\forall x(\emptyset \subseteq x).$$

By the convention for $\emptyset$, it is shorthand for

$$\exists z(\forall y(y \notin z) \wedge \forall x(z \subseteq x)).$$

By the convention for $\subseteq$, this is in turn shorthand for

$$\exists z(\forall y(y \notin z) \wedge \forall x \forall v(v \in z \rightarrow v \in x)).$$

The same trick is needed to talk about operations on sets. To define the union operation $\cup$, for example, we need to find a formula that is true of sets x, y, and z iff z is the union of sets x and y. Such a formula is not hard to find:

$$\forall v(v \in x \vee v \in y \leftrightarrow v \in z).$$

With this, we can give a contextual definition of '$\cup$':

$A(t_1 \cup t_2)$ abbreviates $\exists x(\forall y(y \in t_1 \vee y \in t_2 \leftrightarrow y \in x) \wedge A(x))$,

where x and y do not occur in A.

We can similarly define the intersection operation $\cap$:

$A(t_1 \cap t_2)$ abbreviates $\exists x(\forall y(y \in t_1 \wedge y \in t_2 \leftrightarrow y \in x) \wedge A(x))$.

Exercise 4.11 Give contextual definitions of $\bigcup t$ and $\mathscr{P}(t)$. $\bigcup t$ is the union of all sets in t; $\mathscr{P}(t)$ is the set of all subsets of t.

The next two axioms guarantee that for every set x, there is a set $\bigcup x$ comprising all elements of elements of x, and a set $\mathscr{P}(x)$ comprising all subsets of x. Z4 is the *union axiom*, Z5 the *powerset axiom*.

Z4 $\forall x \exists u \forall y(y \in u \leftrightarrow \exists z(z \in x \wedge y \in z))$.

Z5 $\forall x \exists p \forall y(y \in p \leftrightarrow y \subseteq x)$.

Next, we have the *pairing axiom*:

Z6 $\forall x \forall y \exists z \forall v(v \in z \leftrightarrow (v = x \vee v = y))$

This says that for any sets x, y there is a set $\{x, y\}$ that contains exactly x and y. This is needed, for example, to ensure that $x \cup y$ exists whenever x and y exist: the pairing axiom gives us $\{x, y\}$, from which we get $x \cup y = \bigcup\{x, y\}$ by Z4.

The sets x and y in Z6 needn't be different. For the case where $x = y$, the axiom says that for every set x there is a set $\{x, x\} = \{x\}$ that contains exactly x. This is called the *singleton* set of x. We'll help ourselves to $\{t\}$ as a contextually defined term:

$A(\{t\})$ abbreviates $\exists x(\forall y(y \in x \leftrightarrow y = t) \wedge A(x))$.

We make use of this abbreviation in our next axiom, the *axiom of infinity*:

Z7 $\exists x(\emptyset \in x \wedge \forall y(y \in x \rightarrow y \cup \{y\} \in x))$.

Without the axiom of infinity, we couldn't guarantee the existence of any infinite set. In the next section, we'll see that the set x whose existence is guaranteed by Z7 can be understood as the set $\mathbb{N}$ of natural numbers.

Exercise 4.12 List three members of the set whose existence is guaranteed by Z7.

Next, we have the axiom of *foundation* (or *regularity*). It ensures that every set (every object in the domain) is part of the cumulative hierarchy. Consider any nonempty set x. The elements of x are other sets. If x is in the cumulative hierarchy, then its elements must have appeared at earlier stages in the construction, and there must be some stage at which the first of them appeared. Let y be one of these earliest elements. Since all elements of y appear strictly before y, it follows that none of the elements of y are elements of x. That is, every nonempty set x of sets must have an element y that is disjoint from x:

Z8 $\forall x(x \neq \emptyset \to \exists y(y \in x \land x \cap y = \emptyset))$

There are two more axioms. Next is the *axiom of replacement*, due to Abraham Fraenkel. It is motivated by the observation that the naive comprehension principle only seems to go wrong in cases where the formula $A(x)$ from which it allows defining a set $\{x : A(x)\}$ is true of every set, or of things of which there are as many as there are sets. For example, Russell's $x \notin x$ is true of all the sets. Objects of which there are as many as there are sets are sometimes said to form a *proper class*. (This concept is formalized in some extensions of ZFC, such as the von Neumann-Bernays-Gödel set theory NBG.) In essence, the axiom of replacement says that if there are no more objects of a certain kind than there are members of some set x, then these objects also form a set. After all, they can't form a proper class if there are no more of them than there are members of x, which is known to be a set.

To see how this may be used, suppose that we have a construction that defines a set s_n for each natural number. So there are as many sets $s_0, s_1, s_2, \ldots$ as there are natural numbers. If we identify the natural numbers with the elements of the set whose existence is guaranteed by Z7, we know that the natural numbers form a set. The replacement axiom allows us to conclude that the $s_0, s_1, s_2, \ldots$ also form a set. It is called 'replacement' because it allows replacing all members i of a known set by other things $f(i)$.

Let's review how we compare the sizes of infinite collections. By the standards of section 3.1, a set x is no larger than a set y iff there is an injective function from x to y: a function that maps each element of x to an element of y, without mapping different elements of x to the same element of y. In set theory, we don't have functions as separate objects. But we can simulate them by sets. Since functions are fully determined by which outputs they return for which inputs, we can identify them with sets of input-output pairs. For example, the square function on the natural numbers would be identified with the set of ordered pairs $\langle 0,0 \rangle$, $\langle 1,1 \rangle$, $\langle 2,4 \rangle$, $\langle 3,9 \rangle$, etc.

To complete this definition, we need to provide a set-theoretic surrogate for the concept of an ordered pair. An ordered pair $\langle x,y \rangle$ isn't simply the set $\{x,y\}$: we want to distinguish $\langle 2,4 \rangle$ from $\langle 4,2 \rangle$. In general, ordered pairs $\langle x_1,y_1 \rangle$ and $\langle x_2,y_2 \rangle$ are identical iff $x_1 = x_2$ and $y_1 = y_2$. Can we find set-theoretic constructs that satisfy this condition? Easy. The standard construction, due to Kazimierz Kuratowski, identifies $\langle x,y \rangle$ with the set

$\{\{x\}, \{x, y\}\}$. You can easily show that, on this definition, $\langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle$ iff $x_1 = x_2$ and $y_1 = y_2$.

Exercise 4.13 Find another set-theoretic construction of $\langle x, y \rangle$ that satisfies the identity condition for ordered pairs.

Return to the axiom of replacement. We want to use the axiom show that certain objects form a set, given that there are no more of them than there are members of some other set. Unfortunately, we can't assume in this context we have already established the existence of any functions from the objects to the other set. (If we knew that there is a set of ordered pairs $\langle x, y \rangle$ in which each of our objects figures as a first member, we could infer that the objects form a set by the axioms of union and separation: we wouldn't need replacement).

Instead of invoking functions, the replacement axiom therefore uses *formulas* to express that there is a functional relationship. First, a convenient abbreviation:

$\exists! x A(x)$ abbreviates $\exists x(A(x) \land \forall y(A(y) \to y = x))$,

$\exists! x A(x)$ says that there is exactly one x such that $A(x)$. So $\forall x \exists! y A(x, y)$ says that $A(x, y)$ expresses a functional relationship: it relates each x to exactly one y. If there is such a functional relationship between the members of some set v and some ys, there can be no more ys than there are members of v. The axiom of replacement, which is really an axiom schema, allows us to conclude that there is a set w that contains all these ys:

Z9 $\forall v((\forall x(x \in v \to \exists! y A(x, y)) \to \exists w \forall x(x \in v \leftrightarrow \exists y(y \in w \land A(x, y)))))$

Replacement is needed to ensure that the cumulative hierarchy extends beyond the finite stages $V_0, V_1, V_2, \ldots$. From Z1–Z8 (sometimes called *Zermelo set theory* or Z), we get the finite stages, but we can't prove the existence of V_ω. With Replacement, we can show that there is a set $\{V_n : n \in \mathbb{N}\}$ of all finite stages: the formula $A(x, y)$ in Z9 says that y is obtained from $\emptyset$ by x applications of the power set operation. V_ω is the union of $\{V_n : n \in \mathbb{N}\}$.

Finally, we have Zermelo's Axiom of Choice. This says that if we have a set x of non-empty sets, then there is a set y that contains exactly one element from each set in x.

Z10 $\forall x[\forall z(z \in x \to z \neq \emptyset) \to \exists y \forall z(z \in x \to \exists! v(v \in z \land v \in y))]$

Unlike the other axioms, the Axiom of Choice states that a certain set exists without describing how it can be constructed: we are not told *which* element of each set in x is in y. For this reason (as well as certain strange consequences in the theory of measures), the axiom has long been controversial. Nowadays, it is generally accepted, as many important mathematical results depend on it.

Exercise 4.14 The axioms of ZFC guarantee that for any three things a,b,c, there is a set {a,b,c}. Explain how.

Exercise 4.15 Explain why the separation axiom implies that there is no set of all sets.

4.3 Sets and numbers

The Axiom of Infinity draws attention to an infinite sequence of sets, called the *finite von Neumann ordinals*, or simply the *finite ordinals*:

- $\emptyset$
- $\{\emptyset\}$
- $\{\emptyset, \{\emptyset\}\}$
- $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$
- …

This sequence has the structure of the natural numbers. We can think of $\emptyset$ as 0, $\{\emptyset\}$ as 1, $\{\emptyset, \{\emptyset\}\}$ as 2, and so on. The successor of any number n is $n \cup \{n\}$. (Note that, conveniently, each number n in this construction has exactly n elements.)

More formally, we can use the finite ordinals to define a model of arithmetical theories like $\text{Th}(\mathfrak{N})$ and PA. Recall that a model of a theory is a structure consisting of a domain and an interpretation of the non-logical symbols in which all sentences in the theory are true. For a model of $\text{Th}(\mathfrak{N})$, we can choose as the domain the set ω of finite ordinals. The interpretation function maps the '0' symbol to $\emptyset$ and the successor symbol 's' to the function that maps each set $x \in \omega$ to $x \cup \{x\}$. The standard recursive definitions of addition and multiplication then determine the interpretation of '+' and '×'. (If n and m are in ω, $n + m$ will be the unique set in ω that has exactly $n + m$ elements, and $n \times m$ the unique set with $n \times m$ elements.) This shows that the natural number structure can be embedded in the structure of sets. The same is true for almost every mathematical structure.

There is more. Suppose we read

- '0' as an abbreviation of '$\emptyset$',
- '$s(t)$' as an abbreviation of '$t \cup \{t\}$',
- '$t_1 + t_2$' and '$t_1 \times t_2$' as abbreviations of the corresponding operations on sets,

and we restrict all quantifiers in PA to range over ω, so that '$\forall xA$' becomes '$\forall x(x \in \omega \rightarrow A)$' All axioms of PA are then provable in ZFC. We say that PA is *interpretable* in ZFC. In general, a theory T is interpretable in ZFC if there is a translation scheme of the kind I've sketched under which all sentences in T are provable in ZFC.

A wide range of mathematical theories are interpretable in ZFC. In that sense, ZFC is *at least as strong* as these other theories: whatever they can prove, ZFC can prove as well (if only under the appropriate translation scheme).

I'm not going to prove that PA is interpretable in ZFC. The proof isn't hard, but a little fiddly. To get a sense of what needs to be shown, consider the second axiom of PA:

Q2 $\forall x\, 0 \neq s(x)$

Under the above translation scheme, this turns into $\forall x(x \in \omega \rightarrow (\emptyset \neq x \cup \{x\}))$. And that's easily provable in ZFC.

Exercise 4.16 Sketch a proof of the translated Q2 axiom (from the axioms of ZFC).

Let's now have a closer look at the finite ordinals. They have some interesting properties.

For one, every member of a finite ordinal is also a subset of it. Sets of this kind are called *transitive*. That's because a transitive set z is a set such that whenever $x \in y$ and $y \in z$ then $x \in z$.

Another special property of the finite ordinals is that they are *linearly ordered by* $\in$: any two members of x are related one way or the other by $\in$. I'll say, for short, that the finite ordinals are $\in$*-ordered*.

In ZFC, the finite ordinals can be defined as the transitive and $\in$-ordered sets with finitely many elements. Now suppose we drop the finiteness condition. Let's define an *ordinal* as a transitive and $\in$-ordered set. The finite ordinals are ordinals, but they are not the only ones. For example, ω, the set of finite ordinals, is itself an ordinal. (As you can confirm, it is transitive and $\in$-ordered.) ω is an *infinite ordinal*. So is $\omega \cup \{\omega\}$: the set we get from ω by adding ω itself as an element. Following our earlier definition of the successor relation, we can see $\omega \cup \{\omega\}$ as the "successor" of ω. The successor of $\omega \cup \{\omega\}$ is $\omega \cup \{\omega\} \cup \{\omega \cup \{\omega\}\}$, and so on.

The ordinals form a *transfinite* sequence. If we identify the finite ordinals with the natural numbers, the transfinite sequence of ordinals look like this:

$$0, 1, 2, \ldots, \omega, \omega+1, \omega+2, \ldots \omega+\omega, \ldots$$

Like 0, ω is not the successor of any ordinal. Infinite ordinals that are not successors are called *limit ordinals*. The next limit ordinal after ω is $\omega+\omega$, or $\omega \cdot 2$. It is the union of all ordinals $\omega+n$, where n is a finite ordinal. The next limit ordinal after $\omega \cdot 2$ is $\omega \cdot 3$. After all the limit ordinals $\omega \cdot n$ and all their successors comes their union $\omega \cdot \omega$, or ω^2 – another limit ordinal. Much later we reach ω^ω, ω^{ω^ω}, and so on.

The ordinals extend the idea of "counting" beyond the finite. This has many mathematical applications. Above, I've used the ordinals to label stages in the cumulative hierarchy. I used limit ordinals to label stages at which we take the union of the earlier stages, and successor ordinals to label stages at which we take power sets.

Exercise 4.17 Show that ω is transitive and $\in$-ordered.

Exercise 4.18 Show from the axioms of ZFC that every ordinal has a successor.

Exercise 4.19 Is the set of all ordinals an ordinal?

The ordinals can also be used to interpret the theory of cardinals that I outlined in the previous chapter. Remember that two sets have the same cardinality iff there is a bijection between them. For finite sets, cardinalities are naturally identified with natural numbers: {Athens, Berlin, Cairo} has cardinality 3. But what kind of thing is the cardinality of an infinite set? In section 3.1, we gave them names: we called them $\aleph_0$, $\aleph_1$, etc. But I didn't say more about what these things might be.

The standard answer in contemporary set theory identifies the cardinals with certain ordinals: the cardinality of any set x is defined as *the least ordinal that is equinumerous with x*.

For finite sets, this yields the expected results. {Athens, Berlin, Cairo} is equinumerous with $\{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$, which is the ordinal 3. So the cardinality of { Athens, Berlin, Cairo } is 3. The cardinality of ω is ω. That's because ω is the least ordinal that is equinumerous with ω. Since ω is countably infinite, and we defined $\aleph_0$ as the cardinality of any countably infinite set, this means that $\omega = \aleph_0$.

Beyond $\aleph_0$, things get interesting. The cardinality of $\omega+1$ is still $\aleph_0$. Remember that ω+1 is $\omega \cup \{\omega\}$: it is ω with one extra element. If you add a single element to a countably infinite set, you always get another countably infinite set. So $\omega+1$ is equinumerous with ω. Since the cardinality of a set is the *least* ordinal equinumerous with it, the cardinality of $\omega+1$ is ω (a.k.a. $\aleph_0$).

After ω, the *ordinal numbers* and the *cardinal numbers* diverge. ω is both an ordinal and a cardinal. But $\omega + 1$ is only an ordinal. We've introduced '$\aleph_1$' to name the next cardinal after $\aleph_0$. By our current definition, $\aleph_1$ is first ordinal in the transfinite sequence of ordinals that is not equinumerous with ω. It comes surprisingly late. It's not $\omega+1$, or $\omega \cdot 2$, or ω^2, or ω^ω, or ω^{ω^ω}. All of these are equinumerous with ω. $\aleph_1$ comes much later. And yet we know, from Cantor's theorem, that there are infinitely many different cardinalities. Indeed, for every ordinal κ, there is a distinct cardinal $\aleph_\kappa$, which is itself an ordinal!

By Cantor's theorem, the cardinality of $\mathscr{P}(\omega)$ is greater than $\aleph_0$. How much greater? Cantor conjectured, but was unable to prove, that the cardinality of $\mathscr{P}(\omega)$ is $\aleph_1$. Since $\mathscr{P}(\omega)$ is equinumerous with the set of real numbers $\mathbb{R}$, which is also known as the *continuum*, Cantor's conjecture was that there is no set whose cardinality is strictly between that of the natural numbers and that of the real numbers. This became known as the *continuum hypothesis*.

In 1938, Gödel proved that the continuum hypothesis is consistent with ZFC (assuming ZFC itself is consistent): it can't be disproved from the axioms of ZFC. In 1963, Paul Cohen showed that the negation of the continuum hypothesis is also consistent with ZFC (assuming ZFC is consistent). So the continuum hypothesis can be neither proved nor disproved in ZFC.

I find this odd. Take the set of real numbers $\mathbb{R}$. We know that this set is uncountable. We can get a countable set by removing sufficiently many elements from $\mathbb{R}$. Can we also remove elements from $\mathbb{R}$ so that we get a set that's still uncountable, but smaller than $\mathbb{R}$? I would expect this simple question to have a definite answer. But it can't be answered from the standard axioms of set theory. We could, of course, add the continuum hypothesis as a further axiom. But we could equally add its negation. Neither leads to a contradiction.

Early set theorists assumed that all questions about pure sets have definite answers that can be established by an extended kind of logic. The status of the continuum hypothesis casts doubt on this picture. By now, hundreds of other statements are known that can neither be proved nor disproved in ZFC. We can investigate structures in which they hold and structures in which they fail. Perhaps there is no "true" structure of sets after all. When we describe the cumulative hierarchy, we seem to describe a unique structure. We say that $V_{\omega+2}$ contains *all subsets* of $V_{\omega+1}$. But we can't tell whether these subsets include sets with a cardinality between $\aleph_0$ and the continuum. If the concept of 'all subsets' has a definite meaning, this meaning seems impossible to pin down.

4.4 Unintended models, again

In section 4.1, we looked at non-standard models of Q: models in which all axioms of Q are true but whose structure is clearly not that of the natural numbers. I didn't emphasize it at the time, but Peano Arithmetic also has non-standard models. These are harder to construct directly. But we know that they exist, from the compactness theorem.

Theorem 4.1

There are non-standard models of Peano Arithmetic.

Proof. Let c be an individual constant other than 0. Let Γ be the set of sentences consisting of the axioms of PA together with all the sentences

$$c \neq 0, c \neq s(0), c \neq s(s(0)), \ldots .$$

Every finite subset of Γ is true in the standard model of arithmetic: just interpret c as a sufficiently large natural number. By the compactness theorem, Γ has a model. All axioms of PA are true in this model. But the object denoted by c (in this model) can't be a natural number: it lies outside the number sequence 0,1,2,3, etc. □

Intuitively, Peano Arithmetic doesn't "know" that there are no numbers besides 0,1,2,3, etc.: its axioms are compatible with the existence of further numbers. We know from theorem 3.5 that there's no way to add the missing information to PA, in the form of further axioms: even the set of all truths in the language of arithmetic, $\text{Th}(\mathfrak{N})$, has non-standard models.

Exercise 4.20 PA rules out structures in which the "non-standard numbers" form either a loop or a second chain $a, s(a), s(s(a)), \ldots$. What else could a non-standard model look like?

Are there also non-standard models of ZFC? Let's first clarify the standard model. A model of ZFC consists of a set D of objects and an interpretation function I that assigns some relation on D to the symbol '$\in$'. In the *intended* model, D is the set of all sets, and I maps '$\in$' to … Wait. There is no set of all sets!

In a sense, every model of ZFC is a non-standard model. For every model has a set as its domain, but there is no set of all sets. The real sets form a proper class.

The problem is that we've formalized our semantic concepts in set-theoretic terms. We've define models as set-theoretic structures. The intended interpretation of ZFC can't be formalized in this way.

You may wonder how ZFC can have models in our set-theoretic sense at all. In any set-theoretic model of ZFC, the domain is a set, but ZFC entails that there is no set of all sets. We can strengthen this puzzle. Let's take for granted that ZFC is consistent. By the completeness theorem, it follows that ZFC has a model. By the (downward) Löwenheim-Skolem theorem, it follows from this that ZFC has a *countable* model. Call that model $\mathfrak{M}$. The domain of $\mathfrak{M}$ contains only countably many objects. Yet all sentences in ZFC are true in $\mathfrak{M}$, including sentences saying that there are uncountably many things in $\mathscr{P}(\omega)$, even more in $\mathscr{P}(\mathscr{P}(\omega))$, and so on. This is known as "Skolem's Paradox".

It's not a real paradox. A set x is countable if there is an injective function from x to ω; x is uncountable if there is no such function. ZFC proves that $\mathscr{P}(\omega)$ is uncountable by proving that there is no injective function from $\mathscr{P}(\omega)$ to ω. Remember that functions are represented as sets of ordered pairs. To say that there is an injective function from x to ω is to say that there is a set f of pairs $\langle y, n \rangle$ such that for each $y \in x$ there is exactly one $n \in \omega$ for which $\langle y, n \rangle \in f$, and for each $n \in \omega$ there is at most one $y \in x$ for which $\langle y, n \rangle \in f$. ZFC proves that there is no such set f for $x = \mathscr{P}(\omega)$. In the countable model $\mathfrak{M}$, there may, in fact, be a bijection between the objects denoted by ω and $\mathscr{P}(\omega)$. That is, we may be able to construct such a bijection. But it need not be an object in the domain of $\mathfrak{M}$. If it is not, the statement that there is no bijection of the given type is *true* in $\mathfrak{M}$.

There is still a puzzle, however. It is related to the puzzle from the end of the previous section. How do we manage to latch onto the set-theoretic universe? We could program an AI to interpret the language of ZFC in a countable model. The AI would *say* that there are uncountably many sets. It would say all the right things. But its conception of sets would seem to be radically different from ours. For we can see that there are, in fact, only countably many of the things it calls ‘sets’. Given that this is possible, how can we be sure that we are not equally mistaken about the true sets? How do we know that there isn’t an outside perspective from which one can see that there only countably many of the things we call ‘sets’?

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 called 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'.) But it is not a speculative conjecture. As I'll explain in chapter 7, we *know* that it is true.

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 9, 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 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 6.) But this would be tedious, and we can save the effort by relying on 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 $\mathscr{P}(\mathbb{N})$. 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	$\cdots$
A1	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$\cdots$
A2	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$\cdots$
A3	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

$x_{1,0}$ 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

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

Proposition 5.4

Every axiomatizable first-order theory is computably enumerable.

Proof. Let T be an 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 axiomatizable and complete first-order theory is decidable.

Proof. Let T be an 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 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) 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 $A \land A \land \ldots \land A$. (This is known as *Craig's trick*, 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, 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.

6 Turing computability

In 1936, Alan Turing introduced a formal model of computation by defining a simple type of computer – now known as a *Turing machine* – that, he suggested, can implement any mechanical algorithm.

6.1 Turing machines

Let's think about what is involved in following an algorithm. An algorithm converts some input string into an output string. Let's assume that the input string is received on a piece of paper. The algorithm specifies what one should do with that string, providing step-by-step instructions to add, remove, or change symbols on the paper, until the output string is produced. At each step in the process, the algorithm clearly specifies what to do next, based on what's currently on the paper and on the current stage of the computation. When the computation is finished, the output string must be marked as such on the paper, perhaps by circling or underlining it. For definiteness, let's stipulate that the algorithm should contain instructions to erase everything else on the paper, leaving only the output.

A Turing machine is a machine that implements a process of this kind. It reads and writes symbols on a piece of paper, thereby converting an input string into an output string by following precise, step-by-step instructions.

Turing's key insight was that such a machine can be designed in a very simple way. To begin, we can assume that the paper on which the machine operates is a single strip of paper: any algorithm that requires writing symbols above or below other symbols can be reformulated as an algorithm that only requires writing symbols to the left or right of other symbols. A Turing machine therefore operates on a *tape* in which the symbols are always arranged in a single line. The tape is divided into "cells" or "squares", each of which can hold a single symbol. Since we're interested in what can be computed in principle, without worrying about practical limitations, we assume that the tape is unbounded. (If the machine reaches the end of the tape, the tape is automatically extended.)

Next, we make the steps in the computation as simple as possible. Evidently, any instruction for writing down a sequence of symbols can be broken down into a sequence of instructions for writing down a single symbol. We'll therefore assume that at each step, a Turing machine writes at most one symbol onto its tape. Similarly, we assume that a Turing machine can only read a single cell on its tape at a time. Any instruction for reading larger chunks of the tape can be broken down into instructions for reading single cells.

At each step, a Turing machine therefore operates on a single cell of its tape. We say that it has a *head* that is positioned on this cell. At each step, the machine can read the content of the current cell; it can erase that content or replace it with a different symbol, and it can move its head left or right, by one cell.

For definiteness, we assume that each step involves all these actions. That is, each step in a Turing machine computation consists of three parts:

1. Read the content of the current cell;
2. Erase the content of the current cell and either leave it blank or write a new symbol onto it;
3. Move the head one cell to the left or right.

We can make one last simplification. Every known algorithm operates on strings from a finite alphabet. We can code these strings as numbers greater than 0. Each such number *n* can be written in *unary* notation, as a sequence of *n strokes*. Since there are effective algorithms for converting the original strings into sequences of strokes and back, any algorithm that operates on the original strings can be converted into an algorithm that operates on sequences of strokes. We'll therefore assume that the only symbol available to a Turing machine is the stroke.

In sum, a Turing machine has an unbounded tape, divided into cells, each of which can hold either a stroke or be blank. The machine works in steps, in accordance with a predefined program. At each step, the machine's head is positioned at a particular cell of the tape. A step involves reading the content of that cell, replacing it with either a blank or a stroke, and moving the head one cell to the left or right.

A program for a Turing machine specifies what the machine does at each step. This generally depends on the content of the current cell. A simple program might look as follows:

Step 1. If the current cell contains a stroke, erase it and move right; if the current cell is blank, write a stroke and move left.

Step 2. If the current cell contains a stroke, leave it and move left; if the current cell is blank, write a stroke and move left.

We're not assuming that a machine that executes this program would somehow read and understand the instructions. Turing machines only read strokes or blanks on their tape. A machine that executes the above program would simply be wired to follow the two instructions, one after the other, and then stop.

To build such a machine, we would need an internal switch or counter to keep track of where it is in the computation – whether it should follow instruction 1 or instruction 2. The machine might, for example, have a switch that can be in one of two positions, "up" and "down". We could then build it in such a way that it follows instruction 1 if the switch is up and instruction 2 if the switch is down. The switch would start in the up position and flip to down after the machine has finished following the first instruction.

To allow for programs with more than two instructions, we must allow the switch to have any finite number of positions. These switch positions are called *states* of the machine, and labelled $q_0, q_1, q_2, \ldots$. It doesn't matter how they are implemented. You can think of each state as indicating a "line" in the program the machine is executing.

Exercise 6.1 What does the above machine do if it starts (a) on a tape with a single stroke under its head? (b) on an empty tape?

Many algorithms involve repeating certain steps. The algorithms you've learned for written addition and multiplication, for instance, probably go through each digit in the decimal representation of the input numbers, asking you to perform the same operations for each digit. To implement such an algorithm, a Turing machine must be able to go into the same state more than once. Here is a program for a machine of this kind.

Instruction 1. If the current cell contains a stroke, erase it and move right, then process Instruction 2; if the current cell is blank, write a stroke, move right, and halt.

Instruction 2. If the current cell contains a stroke, erase it and move right; if the current cell is blank, write a stroke and move right; either way, continue with Instruction 1.

A machine that implements this program still needs two states, q_0 and q_1. In state q_0, it follows instruction 1; in state q_1, it follows instruction 2. Each instruction effectively specifies three actions:

- what to write into the current cell (a stroke or a blank);
- whether to move left or right;
- what state to go into next.

We can introduce a compact notation for these instructions, using, for example, '$1, \mathrm{R}, q_1$' to mean that the machine should replace the content of the current cell by a stroke, move right, and go into state q_1, and '$\mathrm{B}, \mathrm{L}, q_0$' for "empty the current cell, move left, and go into state q_0". The above program can then be written as a table:

	1	B
q_0	$\mathrm{B}, \mathrm{R}, q_1$	$1, \mathrm{R}, q_2$
q_1	$\mathrm{B}, \mathrm{R}, q_0$	$1, \mathrm{R}, q_0$
q_2		

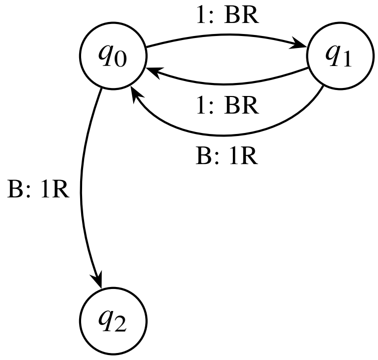
Each cell in the table holds the instruction for what to do in a given state (the row) when reading a given symbol (the column). I've added a third state, q_2, so that the directive to halt can be represented as the directive to go into a new state for which there are no instructions.

There are other ways to represent the same program. We could, for example, package it into a list of quintuples:

$$(q_0, 1, \mathrm{B}, \mathrm{R}, q_1), \quad (q_0, \mathrm{B}, 1, \mathrm{R}, q_2), \quad (q_1, 1, \mathrm{B}, \mathrm{R}, q_0), \quad (q_1, \mathrm{B}, 1, \mathrm{R}, q_0).$$

Here, $\langle q_0, 1, \mathrm{B}, \mathrm{R}, q_1 \rangle$ means that if the machine is in state q_0 and reads a stroke, then it should erase the stroke ("write a blank"), move right, and go into state q_1. Similarly for the other entries in the list.

Another popular way to represent Turing machine programs is as a flow chart. Here is the same machine again:



The nodes in the chart represent the states. Each arrow represents an instruction. '1 : BR' means: 'if you read 1, write a blank, and move right'. The new state is given by the node to which the arrow points.

Exercise 6.2 Can you figure out what this machine does if it starts at the left end of a sequence of strokes on an otherwise empty tape?

Let's do this exercise together. The machine starts in state q_0, reading the first stroke. It erases the stroke and moves right, entering state q_1. It reads the second stroke, erases it, moves right, and goes back into state q_0. It keeps alternating between q_0 and q_1 in this way, moving right and erasing strokes, until it reaches the first blank. At that point, the machine is either in state q_0 or q_1, depending on whether the original tape had an even or an odd number of strokes. If the number of strokes was even, the machine is now in state q_0; it reads the blank, prints a stroke, moves right and halts (in state q_2). If the tape originally had an odd number of strokes, the machine is in state q_1 when it reaches the first blank. It prints a stroke, moves right, and goes into q_0. It then reads another blank, prints another stroke, moves right, and halts.

The machine implements an algorithm for deciding whether the input sequence has an even or odd number of strokes. If even, the output is a single stroke; if odd, the output is two strokes.

Exercise 6.3 In computer programming, it is important to check for edge cases. Does the program correctly classify the empty input as having an even number of strokes?

Exercise 6.4 Design a Turing machine that extends any input sequence of strokes by one stroke: when starting on the left-most stroke of a sequence of *n* strokes on an otherwise blank tape, the machine eventually halts on an otherwise blank tape with a sequence of $n + 1$ strokes.

Exercise 6.5 Draw a flow chart for the machine given by the following quintuples: $(q_0, \mathrm{B}, 1, \mathrm{R}, q_1)$, $(q_0, 1, 1, \mathrm{L}, q_2)$, $(q_1, \mathrm{B}, 1, \mathrm{L}, q_0)$, $(q_1, 1, 1, \mathrm{R}, q_1)$, $(q_2, \mathrm{B}, 1, \mathrm{L}, q_1)$. Can you figure out what this machine does, when started on a blank tape?

6.2 Computing arithmetical functions

A Turing machine converts a pattern of strokes and blanks on its tape into another pattern of strokes and blanks. To compute a function that doesn't take such patterns as input or output, we must code the inputs and outputs as patterns of strokes and blanks.

Let's look at functions on the natural numbers. In the previous section, I suggested that we could code each number n as a sequence of n strokes. This works, but it has the downside that we can't distinguish between empty cells and cells that store the number 0. In this section, I'll therefore use a slightly different coding scheme, in which each number n is represented by a sequence of $n + 1$ strokes: the number 0 is coded by a single stroke, 1 by a sequence of two strokes, and so on.

We say that a Turing machine *computes a function f from $\mathbb{N}$ to $\mathbb{N}$* if, whenever it starts on the left-most stroke of a sequence of $n + 1$ strokes on an otherwise blank tape, it eventually halts on a tape with a sequence of $f(n) + 1$ strokes on an otherwise blank tape. A function f from $\mathbb{N}$ to $\mathbb{N}$ is *Turing-computable* if it is computed by some Turing machine.

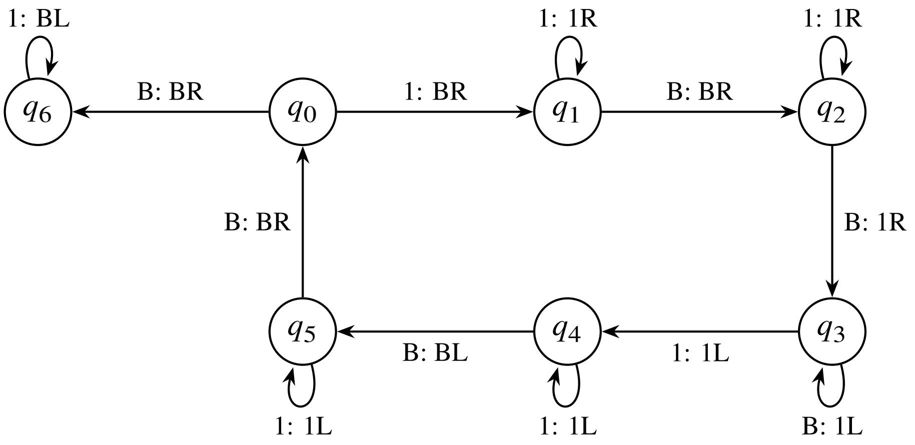
These definitions can obviously be generalized to functions with more than one argument. If a function takes n numbers as input, we stipulate that these numbers must be represented by n blocks of strokes, separated by a blank. For example, if we want a Turing machine to add the numbers 2 and 3, we would start it on a tape that looks like this:

…			1	1	1		1	1	1	1			…

Exercise 6.6 Since blanks and strokes effectively give us two symbols, one might suggest that we could code numbers in binary, so that 0 is coded as a blank (B), 1 as a stroke (1), 2 as 1B, 3 as 11, 4 as 1BB, and so on. Explain why this doesn't work.

In exercise 6.4, you designed a Turing machine that adds a single stroke to the sequence of strokes at which it starts. By our present conventions, this machine computes the successor function that takes a number as input and returns that number plus 1.

Here is a machine that computes the "times 2" function: it converts a sequence of $n + 1$ strokes into a sequence of $2n + 1$ strokes.



The output sequence is constructed to the right of the input, with a blank as a separator. The machine successively removes one stroke from the input, then moves right past the first blank after the input, then moves right past all strokes that follow the blank, prints two strokes, and returns to the left-most stroke on what remains of the input sequence, until that sequence is completely erased. At this point, the machine has created a block of $2n + 2$ strokes. It removes the left-most stroke of this block and halts.

Exercise 6.7 Design a Turing machine that computes addition: when started at the left end of a sequence of $n + 1$ strokes followed by a blank followed by $m + 1$ strokes, the machine halts on a tape with $n + m + 1$ consecutive strokes.

To implement more complex algorithms, it helps to think in terms of subroutines. Let's tackle multiplication. A Turing machine that computes multiplication would start at the left end of a block of $n + 1$ strokes, followed by a blank, followed by another block of $m + 1$ strokes, and eventually halt on a tape with $n \times m + 1$ consecutive strokes. How could we design such a machine?

We could use the first block of strokes as a counter, as in the doubling machine: we'd erase one stroke at a time from the left block; for each stroke that's erased, we add m strokes to the second block; we do this until the counter block has only two strokes left, at which point we erase these strokes and halt. (Do you understand why we'd stop when there are two strokes left in the counter?)

But how can we repeatedly add m strokes to the second block? After i iterations, the first block would have $n + 1 - i$ strokes and the second $i \times m + 1$. It's hard to extract from this the original value m. So here's a better idea: instead of directly adding m strokes to the second block, we insert *m blanks inside* the second block, after its first stroke. That is, we shift the last m strokes of the second block m squares to the right. When we're done, we fill all these blanks with strokes.

For example, consider the case of $n = 3$ and $m = 2$. The input tape is

…	1	1	1	1		1	1	1					…

The desired output is a sequence of $3 \times 2 + 1 = 7$ strokes. We begin by erasing the first stroke in the left block (the counter block). We now have

…		1	1	1		1	1	1					…

Then we shift the two last strokes in the right block two squares to the right:

…		1	1	1		1			1	1			…

Then we start over, erasing another stroke in the counter block and shifting the two strokes on the right by two more squares:

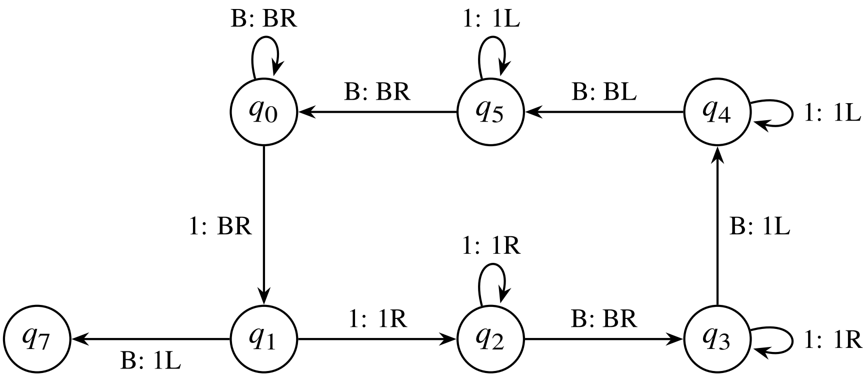
…			1	1		1			1	1			…
…			1	1		1					1	1	…

Now there are only two strokes left in the counter. We erase these two strokes and fill in all the blanks we've inserted in the right block:

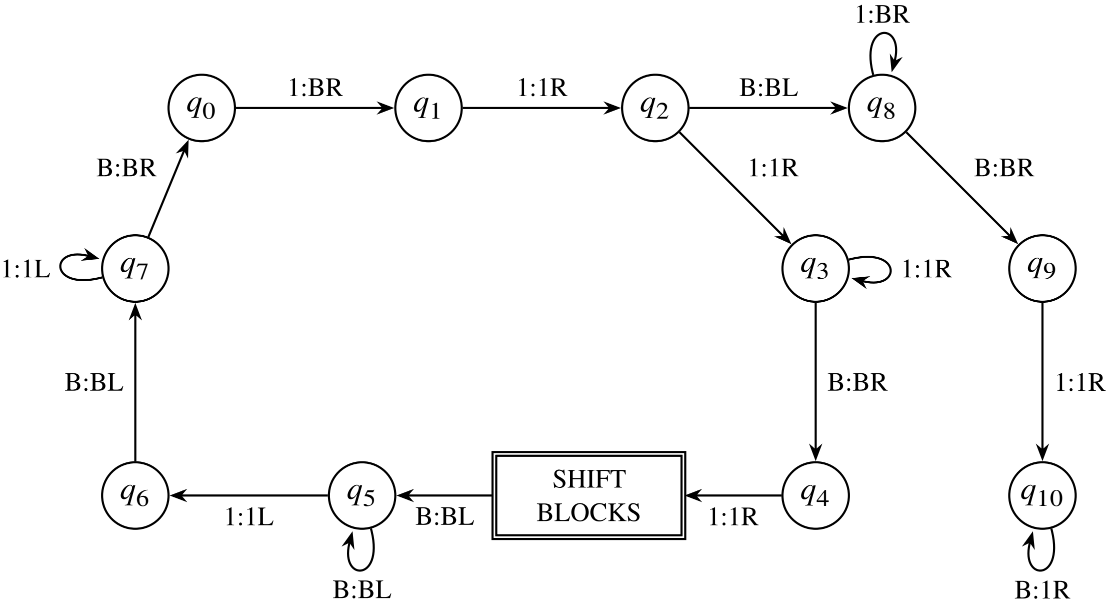
…						1	1	1	1	1	1	1	…

We have the desired output of seven strokes.

Most of this is straightforward to implement. The only slightly tricky part is the subroutine for shifting the strokes in the second block. Let's think of this as a separate task. Let's assume that the head is at the first of the m strokes that we want to shift by m squares to the right. The machine that achieves this should not change anything to the left of its starting position. Here is a machine that does the job. Let's call it 'SHIFT BLOCKS'.



We can now plug this into a multiplication machine, using the algorithm I've just described:



In q_0, this machine removes the current stroke in the counter block. It then moves right twice. If it lands on a blank, there is only one stroke left in the counter block, and the machine goes into the cleanup routine q_8–q_{10}, where it erases the remaining counter stroke and fills the blanks in the right block. Alternatively, if there are further strokes in the counter block, the machine moves right past the counter block, past the separator blank, and past the first stroke of the right block. It then calls the 'SHIFT BLOCKS' subroutine (which I've conveniently defined so that if it starts on a blank then it first moves right until it finds a stroke). After the subroutine, the machine moves back to the left end of the counter block.

It takes some practice and patience to design Turing machines that compute arithmetical functions. In the next chapter, we'll show with one very general argument that a wide range of arithmetical functions can be computed by Turing machines.

Exercise 6.8 My multiplication machine has a bug: it doesn't correctly deal with certain edge cases. Can you find the bug? Can you fix it?

Exercise 6.9 Design a Turing machine that computes the function $\max(x, y)$ that takes two numbers as input and returns the larger of the two.

6.3 Universal Turing machines

Every Turing machine computes a particular function. There is a machine for computing addition, another for multiplication, and so on. As Turing pointed out, one can also design a "universal" Turing machine that can compute *any* computable function. Such a machine takes as input an algorithm for computing a function as well as the arguments to that function. For example, if we supply the machine with an algorithm for addition and the numbers 2 and 3, it would compute the output 5. If we give it an algorithm for multiplication and the numbers 2 and 3, it would compute the output 6.

There are different ways of representing an algorithm. A natural choice, in the present context, is to use Turing machine specifications. Our universal Turing machine U will therefore take as input a specification of a Turing machine M, as well as some input I for M. Its output will be the output produced by M on input I.

Let's think about how we could build such a machine. To begin, we need to code specifications of Turing machines as patterns of strokes and blanks, so that we can feed them as input to the universal machine.

We know that every Turing machine can be represented as a list of quintuples of the form

$$\langle q_i, s, s', d, q_j \rangle,$$

where q_i and q_j are states, s and s' are tape symbols (stroke or blank), and d is a direction (left or right). We can code each of these components by a string of strokes, using (say) $i + 1$ strokes for state q_i, one stroke for the blank and for 'left', and two strokes for the stroke and for 'right'. We can then represent a quintuple by putting its component codes end to end, separated by a blank. The quintuple $\langle q_0, 1, \mathrm{B}, \mathrm{R}, q_1 \rangle$, for example, would be coded by $\boxed{1}\boxed{}\boxed{1}\boxed{1}\boxed{}\boxed{1}\boxed{}\boxed{1}\boxed{1}\boxed{}\boxed{1}\boxed{1}$. To represent an entire machine, we put all the codes for its quintuples end to end, separated by (say) three blanks. We'll call this pattern of strokes and blanks the *machine code* of the machine.

Next, we need to design a Turing machine U that can read the code of a machine M and simulate the behaviour of that machine for any input I. The input for U is the machine code of M, followed by, say, four blanks, followed by the input I for M.

While simulating M, U will divide its tape into three parts. The left part will store the code of M. The right part is a simulation of M's tape. The middle part is a working area.

MACHINE CODE	WORK AREA	SIMULATED TAPE AREA

U is going to simulate each step of running M on I. To this end, U needs to keep track of M's position on its tape. We achieve this by adding a marker for the position of M's head in the simulated tape area. To make space for the marker, we begin by inserting a blank in between any two cells of M's input, so that the original input lies in the odd-numbered squares. For example, if the original input I was

1	1		1		1	1							

then this is converted to

1		1				1				1		1	

in the simulated tape area. The even-numbered cells can now be used to mark the position of M's head. At the beginning, M's head is positioned on the first cell of its input; U marks this by putting a stroke into the first even-numbered cell of the simulated tape area:

1	1	1				1				1		1	

U also needs to keep track of M's current state. To this end, it simply stores the code of the state (a single stroke for q_0, two strokes for q_1, and so on) in the work area. Initially, U writes a single stroke there, assuming that every machine starts in q_0.

After this preparatory work, the simulation begins. It goes as follows.

Stage 1. Find the active position in the simulated tape area, by moving right until you meet the first even square with a stroke. The cell to your left holds the symbol currently scanned by M. Remember this symbol by going into distinct states depending on whether it is a blank or a stroke.

Stage 2. Either way, move left to the work area and print, to the right of the code for M's current state, a blank, followed by the code of the currently scanned symbol (1 or 11).

The machine code stored in the left part of the tape divides into quintuple blocks, each of which begins with a state code followed by a "current symbol" code. The work area therefore now contains the first two items of the quintuple that holds the instruction for what to do in the current state when reading the currently scanned symbol.

Stage 3. Move left to find the position in the machine code that matches the string in the work area, preceded by three blanks. If there's no match, the simulation is finished. In this case, erase everything but the simulated tape area, shift the content of that area to omit all the even-numbered cells, and halt. If there is a match, continue to stage 4.

Stage 4. Scan the instructions in the matched quintuple: remember the symbol to be written onto the tape and the direction to move by going into a different state depending on which symbol is to be written and in which direction to move.

Stage 5. Copy the last element of the quintuple (the new state) into the work area, erasing the previous content of the work area. Then move to the marked position in the simulated tape area, write the remembered symbol into the cell before the marker stroke. Move the marker in the remembered direction by two steps (because the simulated tape contains the marker spaces). Return to stage 1.

All this is relatively straightforward, albeit tedious, to implement. The most fiddly part is stage 3, where we need to find the position in the machine code matching the string in the work area. This requires keeping track of positions in both strings, which can be done by storing the positions in the work area. If the work area runs out of space, or the simulated machine runs off the left edge of the simulated tape area, a subroutine has to be called that moves the entire content of the simulated tape area to the right.

As described, this design is highly inefficient. But it illustrates a profound fact: a single mechanical architecture can in principle carry out any computation. All modern computers are based on this insight. You don't have to re-wire your laptop or phone whenever you want to run a new program. Instead, you load the program code (the algorithm) into memory and tell the processor to read and execute that code.

Exercise 6.10 Show that if a two-place function f is Turing-computable, then so is the one-place function g such that $g(x) = f(x, x)$.

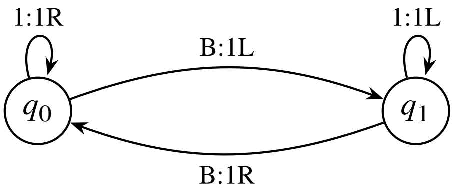
Exercise 6.11 Can the universal Turing machine simulate itself? What happens if you feed U its own machine code as input, together with some further input I for U?

Exercise 6.12 According to the Church-Turing Thesis, any effective, mechanical algorithm can be implemented by a Turing machine. Use the Church-Turing Thesis to argue that there is a universal Turing machine.

6.4 Uncomputability

In the previous chapter, I argued that the set of algorithms is computably enumerable, and inferred that there can be no algorithm that detects whether any given algorithm halts on a given input. We can now see how this plays out for Turing machines.

Every Turing machine has a finite number of states. A Turing machine can still run forever: by going into an infinite loop. Here, for example, is a machine that, when started on a consecutive string of strokes, keeps expanding that string on both ends, without ever halting. (A real computer would "crash" when running this kind of program.)



From the flow chart, it is easy to see that this machine will never halt, no matter its input. But is there a general recipe for determining whether a given Turing machine will halt, on a given input? This is the *halting problem* for Turing machines.

To be clear, the problem is not to determine, for a fixed machine M and input I, whether M will halt on I. *This* problem is trivially decidable. Rather, the problem is to find a general algorithm that decides, for any Turing machine M and any input I, whether M halts on I.

Exercise 6.13 Why is it trivially decidable whether a fixed Turing machine M halts on a fixed input I?

We can show that the halting problem can't be solved by a Turing machine. A Turing machine H that solves the halting problem would take the code of a Turing machine M and an input I for M as input and would output (say) two strokes if M halts on I and one stroke if M doesn't halt on I. We can show by a diagonal argument that such a machine H can't exist.

Theorem 6.1

The Halting Problem is undecidable by a Turing machine.

Proof. Suppose for reductio that there is a Turing machine H that decides the Halting Problem. We could then plug H into a larger machine D that takes the code for a machine M as input and halts iff M halts when given *its own code* (the code of M) as input.

This machine D would be constructed as follows. When started on the code of a machine M, it first creates a copy of the input. It then runs H on the contents of the tape, to determine if M halts on its own code. If the answer is yes, D goes into an infinite loop. If the answer is no, D halts.

Now we get a contradiction if we ask whether D halts on its own code: by design, D halts on the code of M iff M does not halt on its own code; so D halts on its own code iff D does not halt on its own code. It follows that D can't exist, and therefore that H can't exist. □

Exercise 6.14 Can a universal Turing machine get stuck in an infinite loop? If so, how? Could we prevent it by, say, keeping a counter of the number of simulated steps and abort the simulation if that counter exceeds some fixed limit?

The unsolvability of the halting problem can be used to show that various other functions are not Turing-computable. A neat example is the *Busy Beaver function* Σ, introduced by Tibor Rado in 1962. This function takes a number n as input and returns the largest number of strokes that can be printed by a Turing machine with n states before halting, when started on a blank tape.

For example, it is easy to see that $\Sigma(1) = 1$. Let M be any machine with just one state, q_0. When started on a blank tape, the first instruction M executes is the one for q_0 and a blank cell. The machine can either print a stroke or leave the cell blank; then it moves either left or right, to another blank cell. If the machine doesn't halt at this point, it will again be in state q_0, reading a blank cell; it will repeat the same action, moving in the same direction, without end. So the only way M can halt is by halting after the first step. The most it can print in that step is a single stroke. So the largest number of strokes that a 1-state machine can print before halting (when started on an empty tape) is 1.

A somewhat more involved argument along the same lines shows that $\Sigma(2) = 4$ and $\Sigma(3) = 6$. (In exercise 6.5, I asked you to draw the flow chart for the 3-state machine that prints 6 strokes on an empty tape and then halts.)

If the halting problem were decidable, we could easily compute the Busy Beaver function. For any input number n, we would simply enumerate all Turing machines with n states, use the halting algorithm to discard the non-halting machines, and run the remaining machines (on an empty tape) to see how many strokes they print before halting. Due to the undecidability of the halting problem, this algorithm doesn't work. In fact, there is no algorithm that computes the Busy Beaver function. More precisely, there is no Turing machine that computes the Busy Beaver function. This can be shown by showing that any machine that computes the Busy Beaver function could be used to solve the halting problem. But it can also be shown directly:

Theorem 6.2: (Rado 1962)

The Busy Beaver function is not Turing-computable.

We'll show that every Turing-computable total function f on the natural numbers is eventually overtaken by the Busy Beaver function Σ. That is, for every Turing-computable total function f on $\mathbb{N}$, there is a number k such that $\Sigma(k) > f(k)$. If Σ were Turing-computable, there would be a number k such that $\Sigma(k) > \Sigma(k)$. This is impossible. So Σ is not Turing-computable.

Let f be any Turing-computable total function from $\mathbb{N}$ to $\mathbb{N}$. Then the following function g is also Turing-computable and total:

$$g(x) = \max(f(2x), f(2x+1)) + 1.$$

To compute $g(x)$ for any x, we first create a copy of the input x at a sufficient distance from the original input. Then we use the "times 2" machine to convert the input x into $2x$, and run the machine that computes f on the resulting block of strokes. We then have $f(2x)$ on that part of the tape. Next, we use the "times 2" machine and the "add 1" machine to convert the copy of x into $2x + 1$, and run the machine that computes f on the resulting block. We now have $f(2x)$ and $f(2x+1)$ on the tape. To finish the computation of $g(x)$, we run your algorithm for computing max from exercise 6.9, add a single stroke to the result, and halt.

Let M be some such machine for computing g. If M has k states, we can define, for any input x, a machine N_x with $x + k$ states that first writes x strokes on the tape and then imitates M. (No more than x states are needed to write x strokes.)

When started on a blank tape, N_x writes $g(x)$ strokes and then halts. So there is a machine with $x + k$ states that prints $g(x)$ strokes on the empty tape and then halts. By definition of the Busy Beaver function, this means that $\Sigma(x+k) \geq g(x)$. By definition of g, both $f(2x)$ and $f(2x+1)$ are less than $g(x)$. So we have

$$\Sigma(x+k) \geq g(x) > f(2x);$$
$$\Sigma(x+k) \geq g(x) > f(2x+1).$$

But obviously, if $x \geq k$ then

$$\Sigma(2x+1) \geq \Sigma(2x) \geq \Sigma(x+k).$$

Combining these inequalities, we infer that $f(x) < \Sigma(x)$ for $x \geq 2k$. □

I mentioned above that the first few values of the Busy Beaver function are not hard to determine: $\Sigma(0) = 0$, $\Sigma(1) = 1$, $\Sigma(2) = 4$, and $\Sigma(3) = 6$. It is also known that $\Sigma(4) = 13$ and $\Sigma(5) = 4098$. As of 2025, the value of $\Sigma(6)$ is not known exactly; but it is known that there is a 6-state machine that prints $2 \uparrow\uparrow (2 \uparrow\uparrow (2 \uparrow\uparrow 10))$ strokes. So $\Sigma(6)$ is at least $2 \uparrow\uparrow (2 \uparrow\uparrow (2 \uparrow\uparrow 10))$. The up-arrow stands for repeated exponentiation: $2 \uparrow\uparrow 10$ is $2^{2^{\cdot^{\cdot^{2}}}}$ with ten twos in the tower. This number is *much* larger than, say, the number of atoms in the observable universe. $2 \uparrow\uparrow (2 \uparrow\uparrow 10)$ is a power tower of $2 \uparrow\uparrow 10$ twos. You couldn't write down all the twos in this tower even if you managed to write a '2' onto each atom in the universe. $2 \uparrow\uparrow (2 \uparrow\uparrow (2 \uparrow\uparrow 10))$ is a power tower of $2 \uparrow\uparrow (2 \uparrow\uparrow 10)$ twos. $\Sigma(7)$ is known to be at least $2 \uparrow^{11} (2 \uparrow^{11} 3)$, which I won't even try to explain. It is an incomprehensibly large number. You can inspect the machine tables for the known Busy Beaver champions at bbchallenge.org/~pascal.michel/bbc.

As Turing realised, we can also use the undecidability of the halting problem to show that Hilbert's Entscheidungsproblem is unsolvable by a Turing machine: there can be no Turing machine that decides whether any given first-order sentence is valid. The idea is that for any Turing machine M and input I, we can construct a first-order sentence $S_{M,I} \to H_{M,I}$ that is valid iff M halts on input I. The antecedent $S_{M,I}$ is a first-order description of the machine and its input; the consequent $H_{M,I}$ says that the machine halts. If we could decide whether $S_{M,I} \to H_{M,I}$ is valid (or equivalently, whether $S_{M,I}$ entails $H_{M,I}$), we could decide whether M halts on input I.

To explain what $S_{M,I}$ and $H_{M,I}$ look like, let the *configuration* of a machine M with input I at step n consist of the machine's state, the position of its head on the tape, and the tape's content at step n. $S_{M,I}$ will specify the initial configuration of M on input I, at step 0. It will also describe how the configuration changes from one step to the next, in accordance with the machine table of M. We'll need some non-logical vocabulary to spell this out.

I'll use '0' and 's' to create terms for the computation steps: '0' denotes step 0, 's(0)' step 1, and so on. For the tape positions, I use a constant 'o' ("origin") for the square at which the machine starts, and two unary function symbols 'l' and 'r' that move one square to the left and right, respectively. So '$l(l(o))$', for example, denotes the square two to the left of the starting square. I'll also introduce a predicate 'Q_i' for each state q_i of M, so that $Q_i(n)$ means that at step n the machine is in state q_i. Finally, I'll use two binary predicates '@' and '1', where $@(n,x)$ means that at step n the machine is positioned on square x, and $1(n,x)$ that at step n there is a stroke in square x.

With this vocabulary, we can express the configuration of M on input I at every step n. For example, suppose M starts in state q_0 on input 11. Then the initial configuration can be expressed as follows.

$$Q_0(0) \land @(0,o) \land 1(0,o) \land 1(0,r(o)) \land \forall y(y \neq o \land y \neq r(o) \to \neg 1(0,y)).$$

We can also express how the configuration changes from one step to the next. For example, if M has an entry $\langle q_0, 1, \mathrm{B}, \mathrm{R}, q_1 \rangle$ in its machine table, then $S_{M,I}$ would have the following conjunct:

$$\begin{aligned}&\forall x \forall y((Q_0(x) \land @(x,y) \land 1(x,y)) \to\\ &\quad (Q_1(s(x)) \land @(s(x),r(y)) \land \neg 1(s(x),y) \land \forall z(z \neq y \to (1(s(x),z) \leftrightarrow 1(x,z))))).\end{aligned}$$

This says that if at some step x the machine is in state q_0 and positioned at some square y that contains a stroke, then at step $s(x)$ the machine is in state q_1, positioned at the square to the right of y, where the square y is now blank, and all other squares have the same content as before.

$S_{M,I}$ will be a big conjunction containing, first, the initial configuration of M on input I, then all the transition rules for M, and finally some background "axioms" to fix the intended interpretation of the non-logical symbols:

T1 $\forall x\, s(x) \neq 0$

T2 $\forall x \forall y (s(x) = s(y) \rightarrow x = y)$

T3 $\forall y (r(l(y)) = y \wedge l(r(y)) = y)$

T4 $\forall x (Q_0(x) \vee Q_1(x) \vee \cdots \vee Q_n(x))$

T5 $\forall x \forall y (Q_i(x) \rightarrow \neg Q_j(x))$ for $i \neq j$

T6 $\forall x \exists y @(x, y)$

T7 $\forall x \forall y \forall z ((@(x, y) \wedge @(x, z)) \rightarrow y = z)$.

Let's turn to $H_{M,I}$. This is meant to say that M halts on input I. It is a disjunction, each disjunct of which corresponds to a state/symbol combination for which there is no entry in the machine table. For example, if there's no entry in the table for what to do in state q_1 when reading a blank, then $H_{M,I}$ will have the following as a disjunct:

$$\exists x \exists y (Q_1(x) \wedge @(x, y) \wedge \neg 1(x, y)).$$

This says that at some step x the machine is in state q_1 and positioned at a square y that is blank.

Theorem 6.3: Turing-undecidability of first-order logic (Turing 1936)

No Turing machine can decide whether any given first-order sentence is valid.

Proof sketch. Let M be any Turing machine and I any input for M. Let $S_{M,I}$ and $H_{M,I}$ be as above. We show that M halts on I iff $S_{M,I} \models H_{M,I}$. It follows that any Turing machine that solves the Entscheidungsproblem could be used to solve the halting problem.

Left to right. Suppose M halts on I after n steps. Let $\mathfrak{M}$ be any model of $S_{M,I}$. One can show by induction on n that $\mathfrak{M}$ satisfies the sentence describing the configuration of M on input I at step n. Since M halts on I at step n, it follows that $\mathfrak{M}$ satisfies $H_{M,I}$.

Right to left. Suppose M does not halt on I. We can then build a model $\mathfrak{M}$ of $S_{M,I}$ that does not satisfy $H_{M,I}$, by giving all the non-logical symbols their intended interpretation. □

I've omitted a lot of details here. Filling them in would take a few more pages. Since I'll give a full proof of Theorem 6.3, via a rather different route, in Chapter 9, I'll save us the labour.

Exercise 6.15 The proof of Theorem 6.3 shows that M halts on I iff $S_{M,I} \models H_{M,I}$. Is it also true that M *doesn't* halt on I iff $S_{M,I} \models \neg H_{M,I}$? (a) Explain why this would contradict the undecidability of the halting problem, given the completeness of first-order logic. (b) Explain informally how it can be that M doesn't halt on I, but $S_{M,I} \not\models \neg H_{M,I}$.

The proof of Theorem 6.3 doesn't just show that no Turing machine can solve the Entscheidungsproblem. It shows more concretely that any Turing machine that could solve the Entscheidungsproblem could be converted into a Turing machine that solves the halting problem: if you gave a Turing machine an "oracle" for deciding validity in predicate logic – a magical subroutine that decides validity – then that machine could solve the halting problem. In this sense, the halting problem *reduces to* the Entscheidungsproblem. Many other problems have been revealed as undecidable in this way, by showing that their solution would yield a solution to the halting problem.

Exercise 6.16 Could the oracle Turing machine just described solve the halting problem for oracle Turing machines, or only for ordinary Turing machines?

7 Recursive Functions

In this chapter, we define the class of recursive functions as the functions that can be built from certain base functions using some simple operations. We show that a wide range of functions are recursive, and that the recursive functions are exactly the Turing-computable functions.

7.1 Primitive recursive functions

In Chapter 5, I said that a function is *computable* if there are precise instructions, an *algorithm*, for determining its output for any given input, without drawing on external resources or creativity. Let's concentrate on functions with natural numbers as inputs and outputs. (I've explained in Section 5.5 why this is not a serious restriction.)

Consider the "counting on" algorithm for addition that you may have learned as a child. To compute $x + y$, you start with the number x; then you "count on" from x, adding 1 y times. The algorithm effectively reduces addition to repeated application of the successor function. Simple algorithms for multiplication similarly reduce multiplication to repeated addition.

Generalizing, algorithms for computing arithmetical functions usually invoke subroutines for computing simpler functions. These subroutines may in turn invoke other subroutines, until we reach functions that are so simple that they can be computed in a single step, without any subroutines. This suggests that the computable functions might be defined as the functions that can be built up from certain base functions using certain modes of construction. This is how Gödel defined the class of *primitive recursive* functions in his 1931 paper on the incompleteness theorems.

Following Gödel, we start with three kinds of base functions.

1. The *successor function* s returns the next larger number for any input number.
2. The *zero function* z that returns 0 for any input number.
3. For each n, i, the *projection function* π^n_i takes n numbers as input and return the i-th of them. (For example $\pi^3_2(5, 9, 2) = 9$.)

These functions are trivially computable, without needing any subroutines.

We now define two operations for constructing new functions from old ones. The first is *composition*. Given some functions f and g, we can define a new function h by applying one to the output of the other:

$$h(x) = f(g(x)).$$

If f and g are computable, then h is also computable. To compute $h(x)$, we only need to compute $g(x)$ and feed the output into f.

I've assumed that f and g both take one number as input. For the general case, assume that f is a function of m arguments, and each of $g_1, \ldots, g_m$ is a function of n arguments. We define the *composition* of f and $g_1, \ldots, g_m$ as:

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

Instead of introducing a new name 'h' for the composed function, we can also write the composition as Cn[$f, g_1, \ldots, g_m$]. For example, Cn[s, z] is the function that takes a number as input, passes it to the zero function and passes the output to the successor function: $\text{Cn}[s, z](x) = s(z(x))$. This is the constant function that always outputs 1.

Exercise 7.1 What is (a) Cn[s, s]? (b) Cn[s, Cn[s, z]]? (c) Cn[π^2_2, z, z]?

Exercise 7.2 Using Cn and the base functions, define the 1-place function that always returns 4.

Our second method for constructing functions is primitive recursion. We've met this in Section 4.1 when we talked about axioms of arithmetic. Addition, for example, can be reduced to the successor function by the following definition:

$$x + 0 = x$$
$$x + s(y) = s(x + y)$$

The definition effectively tells us how to compute $x + y$ starting from $x + 0$, then working our way up through $x + 1$, $x + 2$, and so on, until we reach $x + y$. (The first line gives us $x + 0$; the second line tells us how to get from $x + y$ to $x + s(y)$.) In imperative pseudocode, the algorithm, which resembles the counting-on strategy, could be stated as follows:

```
function add(x, y):
    let z = x
    for i from 1 to y:
        z = s(z)
    return z
```

Some more examples. First, the *factorial function* that takes a number n as input and returns the product of all numbers from 1 to n:

$$\begin{aligned} \text{fact}(0) &= 1 \\ \text{fact}(s(y)) &= s(y) \cdot \text{fact}(y) \end{aligned}$$

Next, we can define the *truncated predecessor* function that maps any positive number to its predecessor, and 0 to 0:

$$\begin{aligned} \text{pred}(0) &= 0 \\ \text{pred}(s(y)) &= y \end{aligned}$$

This is a somewhat unusual case of primitive recursion because the value of pred for $s(y)$ doesn't depend on the value of pred for y; it only depends on y itself. But we allow for that.

Using pred, we can define a *truncated subtraction* function:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} s(y) &= \text{pred}(x \dot{-} y) \end{aligned}$$

$x \dot{-} y$ is $x - y$ if $x \geq y$ and 0 otherwise.

Another useful function is the *switcheroo* function δ ("delta") that takes every positive integer to 0 and 0 to 1:

$$\begin{aligned} \delta(0) &= 1 \\ \delta(s(y)) &= 0 \end{aligned}$$

Here, the value for $s(y)$ depends on neither the value for y nor on y itself. That, too, is allowed.

Exercise 7.3 Define multiplication and exponentiation using primitive recursion.

Exercise 7.4 Use primitive recursion to define a function h that maps 0 to 0 and every positive number to 1.

Let's officially define the operation of primitive recursion. We assume that an $n+1$-place function h is defined by primitive recursion from an n-place function f and an $n+2$-place function g. The function f specifies the starting point, $h(x_1, \dots, x_n, 0)$; The function g specifies $h(x_1, \dots, x_n, s(y))$ in terms of $h(x_1, \dots, x_n, y)$. We allow g to also depend on $x_1, \dots, x_n$, and y. So our general format for primitive recursion looks like this:

$$\begin{aligned} h(x_1, \dots, x_n, 0) &= f(x_1, \dots, x_n) \\ h(x_1, \dots, x_n, s(y)) &= g(x_1, \dots, x_n, y, h(x_1, \dots, x_n, y)). \end{aligned}$$

Again, this effectively defines an algorithm for computing h:

```
function h(x1, ..., xn, y):
    let z = f(x1, ..., xn)
    for i from 0 to y-1:
        z = g(x1, ..., xn, i, z)
    return z
```

If h is defined by primitive recursion from f and g, we write $h = \text{Pr}[f, g]$. For example, addition is $\text{Pr}[\pi^1_1, \text{Cn}[s, \pi^3_3]]$. That's because

$$\begin{aligned} x + 0 &= \pi^1_1(x) = x \\ x + s(y) &= \text{Cn}[s, \pi^3_3](x, y, x + y) = s(x + y). \end{aligned}$$

Exercise 7.5 Can you express your definition of multiplication using the Pr notation?

A downside of the above format is that it doesn't directly account for our definitions of pred, fact, and δ, which are one-place functions. Recall the definition of pred:

$$\begin{aligned} \text{pred}(0) &= 0 \\ \text{pred}(s(y)) &= y \end{aligned}$$

Here there are no extra arguments $x_1, \dots, x_n$. So the function f would be a zero-place "function" that returns 0. We don't have such a function, and we can't define it using our present resources.

But there's a workaround. We can first define a two-place function dpred with a dummy first argument:

$$\begin{aligned}\mathrm{dpred}(x, 0) &= 0\\ \mathrm{dpred}(x, s(y)) &= y\end{aligned}$$

This fits our official format: $\mathrm{dpred} = \mathrm{Pr}[z, \pi^3_2]$. Since $\mathrm{pred}(y) = \mathrm{dpred}(x, y)$ for any x, we can now define pred by composition and projection from dpred:

$$\mathrm{pred} = \mathrm{Cn}[\mathrm{dpred}, \pi^1_1, \pi^1_1]$$

This trick also works for fact and δ, and any other case of "one-place primitive recursion".

We now define the class of primitive recursive functions.

Definition 7.1

A function is *primitive recursive* if it can be defined from the base functions s, z, and π^n_i by finitely many applications of composition and primitive recursion.

Proposition 7.1

Every primitive recursive function is total.

Proof. By induction on a function's construction. The base functions are evidently total. The composition of total functions is total. For primitive recursion, note that the algorithm implicitly defined by Pr always terminates after y steps when computing $\mathrm{Pr}[f, g](x_1, \ldots, x_n, y)$, returning the desired output. □

We can extend the concept of primitive recursiveness to sets and relations:

Definition 7.2

A set is primitive recursive if its characteristic function is primitive recursive. A relation is primitive recursive if its extension is primitive recursive.

Remember that the characteristic function of a set is the function that maps every member of the set to 1 and every non-member to 0. At the moment, we are interested in sets whose members are numbers or tuples of numbers. For example, the set of odd numbers is primitive recursive, as its characteristic function χ_{odd} can be defined by primitive recursion, as follows:

$$\begin{aligned} \chi_{\text{odd}}(0) &= 0 \\ \chi_{\text{odd}}(s(y)) &= \delta(\chi_{\text{odd}}(y)) \end{aligned}$$

Definition 7.2 also covers properties, because a property is a 1-place relation. A property of numbers is primitive recursive iff there is a primitive recursive function that maps every number that has the property to 1 and every other number to 0. As we've just shown, the property of being odd is primitive recursive.

An example of a two-place primitive recursive relation is the less-than-or-equal relation on $\mathbb{N}$. Its characteristic function can be defined by composition from $\dot{-}$ and δ:

$$\text{Leq}(x, y) \quad = \quad \delta(x \dot{-} y).$$

Another example is the identity relation on $\mathbb{N}$, with the following characteristic function:

$$\text{Eq}(x, y) \quad = \quad \delta((x \dot{-} y) + (y \dot{-} x)).$$

To see why this works, note that if $x = y$ then $x \dot{-} y$ and $y \dot{-} x$ are both 0; if $x \neq y$ then at least one of them is positive.

Exercise 7.6 Show that the set of even numbers is primitive recursive.

Exercise 7.7 Show that the less-than relation on $\mathbb{N}$ is primitive recursive. You may, if you want, use the functions Leq and Eq.

Exercise 7.8 Show that if a relation R is primitive recursive then so is its negation $\neg R$ (which holds of exactly those tuples that do not satisfy R).

Exercise 7.9 Show that if two relations R and S are primitive recursive then so is their conjunction $R \wedge S$ (which holds of exactly those tuples that satisfy both R and S).

7.2 Primitive recursive operations

In the last two exercises, you showed that the primitive recursive relations are closed under negation and conjunction. Since all truth-functional combinations can be built up from these two operations, it follows that the primitive recursive relations are closed under all truth-functional operations.

Proposition 7.2

If R and S are primitive recursive relations, then so are $\neg R$, $R \wedge S$, $R \vee S$, $R \to S$, and $R \leftrightarrow S$.

Proof. Immediate from exercises 7.8 and 7.9. □

I'll define three more operations for defining primitive recursive functions and relations.

First, bounded quantification. Consider the Divides relation that holds between numbers x and y iff y is a multiple of x. (For example, 3 divides 12, but 3 doesn't divide 5.) We might define this as follows:

$$\text{Divides}(x, y) \text{ iff } \exists z\, (y = x \cdot z).$$

But we don't need to quantify over all numbers z. If x and y are natural numbers, $y = x \cdot z$ can only hold if z is less than or equal to y. So we can also use a *bounded quantifier* in the definition:

$$\text{Divides}(x, y) \text{ iff } \exists z \leq y\, (y = x \cdot z).$$

This says that x divides y iff there is some number z less than or equal to y such that y equals x times z. Since multiplication and equality are primitive recursive, this relation is primitive recursive:

Proposition 7.3

If $R(x_1, \ldots, x_n, y)$ is a primitive recursive relation, then so are $\forall y \leq k\, R(x_1, \ldots, x_n, y)$ and $\exists y \leq k\, R(x_1, \ldots, x_n, y)$.

Proof. To simplify notation, I assume that R is a two-place relation $R(x, y)$. Let χ be the characteristic function of R. Define χ' by primitive recursion as follows:

$$\begin{aligned}\chi'(x,0) &= \chi(x,0)\\ \chi'(x,s(k)) &= \chi'(x,k)\cdot\chi(x,s(k))\end{aligned}$$

This function returns 1 for input x, k iff $R(x,0), R(x,1), \dots, R(x,k)$ all hold, otherwise it returns 0. So $\chi'(x,k)$ is the characteristic function of $\forall y \leq k\, R(x,y)$.

From bounded universal quantification, we obtain bounded existential quantification by truth-functional operations: $\exists y \leq k\, R(x,y)$ is equivalent to $\neg\, \forall y \leq k\, \neg R(x,y)$. □

So Divides is primitive recursive. The same is true for the property of being a prime number, as the following definition shows:

$$\text{Prime}(x) \text{ iff } 1 < x \land \forall y \leq x\, (\text{Divides}(y,x) \to (y = 1 \lor y = x)).$$

Don't get confused by the fact that this looks vaguely like a formula of first-order logic. We're not trying to define Prime in some first-order language. Everything is in the metalanguage. 'Divides' and '=' are metalinguistic names for relations on $\mathbb{N}$ that we've shown to be primitive recursive; '$\land$', '$\to$', '$\lor$', and '$\forall y \leq x$' denote operations on such relations.

Next, definition by cases. Suppose we want to define the function $\max(x,y)$ that returns the larger of two numbers x and y:

$$\max(x,y) = \begin{cases} x & \text{if } y \leq x,\\ y & \text{otherwise.}\end{cases}$$

We can use switcheroo and addition to distinguish the two cases:

$$\begin{aligned}\max(x,y) &= x\cdot\text{Leq}(y,x) + y\cdot\delta(\text{Leq}(y,x))\\ &= x\cdot\delta(y \dot{-} x) + y\cdot\delta(\delta(y \dot{-} x)).\end{aligned}$$

This trick can be generalized:

Proposition 7.4

If f and g are primitive recursive functions and R is a primitive recursive relation then the function h defined by

$$h(x_1, \dots, x_n) = \begin{cases} f(x_1, \dots, x_n) & \text{if } R(x_1, \dots, x_n), \\ g(x_1, \dots, x_n) & \text{otherwise} \end{cases}$$

is primitive recursive.

Proof. Let χ be the characteristic function of R. Then

$$f(x_1, \dots, x_n) \cdot \chi(x_1, \dots, x_n) + g(x_1, \dots, x_n) \cdot \delta(\chi(x_1, \dots, x_n))$$

is primitive recursive and defines h.

Exercise 7.10 Suppose h is defined by distinguishing three cases:

$$h(x) = \begin{cases} f(x) & \text{if } P_1(x), \\ g_1(x) & \text{if } P_2(x), \\ g_2(x) & \text{otherwise,} \end{cases}$$

where f, g_1, and g_2 are primitive recursive functions. Explain why it follows from proposition 7.4 that h is primitive recursive

The final operation I want to mention is bounded minimization. Suppose we want to define the function $\text{spf}(x)$ that returns the smallest prime number y that divides x. We can write this as follows:

$$\text{spf}(x) = \mu y\, (\text{Prime}(y) \wedge \text{Divides}(y, x)),$$

where 'μy' ("mu y") means "the least y such that …". So $\mu y\, (\text{Prime}(y) \wedge \text{Divides}(y, x))$ is the least number y such that y is prime and y divides x. That number y will never be greater than x. So we can equivalently define $\text{spf}(x)$ as the least number y less than or equal to x that is prime and divides x:

$$\text{spf}(x) = \mu y \leq x\, (\text{Prime}(y) \wedge \text{Divides}(y, x)).$$

The $\mu y \leq x$ operator expresses a bounded search. To compute $\mu y \leq x\, R(y)$, we check $R(0)$, then $R(1)$, then $R(2)$, and so on, up to $R(x)$, until we find a number y of which $R(y)$

holds; then we return that number. To ensure that the operation defines a total function, let's stipulate that $\mu y \leq x\, R(y)$ is 0 if there is no number $y \leq x$ for which $R(y)$ holds.

Exercise 7.11 Let $h(x) = \mu y{\leq}x\,(y + y = x)$. What are $h(0)$, $h(1)$, and $h(2)$?

Proposition 7.5

If $R(x_1, \dots, x_n, y)$ is a primitive recursive relation, then the function f defined by

$$f(x) = \mu y \leq x\, R(x_1, \dots, x_n, y)$$

is primitive recursive.

Proof. I'll assume that $n = 1$, to simplify the notation. Let χ be the characteristic function of $R(x, y)$. Let f be defined as follows:

$$f(x, 0) = \begin{cases} 0 & \text{if } \chi(x, 0) = 1, \\ 1 & \text{otherwise.} \end{cases}$$

$$f(x, s(k)) = \begin{cases} f(x, k) & \text{if } f(x, k) \leq k, \\ k + 1 & \text{if } \chi(x, k + 1) = 1, \\ k + 2 & \text{otherwise.} \end{cases}$$

Think of this as defining a sequence $f(x, 0), f(x, 1), f(x, 2), \dots$. At each step $k = 0, 1, 2, \dots$, $f(x, k)$ is the first y with $R(x, y)$ if we've already found one, otherwise it is $k + 1$. So $f(x, k)$ is the least number $y \leq k$ such that $R(x, y)$ holds, or $k + 1$ if there is no such number. Finally,

$$\mu y \leq x\, R(x, y) = \begin{cases} f(x, x) & \text{if } f(x, x) \leq x, \\ 0 & \text{otherwise.} \end{cases}$$

□

These operations give us a useful toolkit for defining primitive recursive functions and relations. I'll give three examples, related to coding sequences of (non-zero) numbers by prime powers, as introduced in section 5.5.

First, consider the function pri that takes a number n as input and returns the n-th prime number (counting from zero). This function is primitive recursive:

$$\begin{aligned}\mathrm{pri}(0) &= 2,\\ \mathrm{pri}(s(y)) &= \mu x \leq 2 \cdot \mathrm{pri}(y)\ (\mathrm{Prime}(x) \wedge x > \mathrm{pri}(y)).\end{aligned}$$

Here I use Chebyshev's theorem, which states that for any $n \geq 1$ there is a prime between n and $2n$.

Second, we can define a primitive recursive function $\mathrm{entry}(x, y)$ that returns the exponent of the y-th prime in the prime factorization of x:

$$\mathrm{entry}(x, y) = \mu z \leq x\ (\mathrm{Divides}(\mathrm{pri}(y)^z, x) \wedge \neg\mathrm{Divides}(\mathrm{pri}(y)^{z+1}, x)).$$

I call this 'entry' because it returns the y-th entry in the sequence coded by x when we use Gödel's scheme to code a sequence of numbers $n_1, n_2, \ldots, n_k$ as $2^{n_1} \cdot 3^{n_2} \cdots p_k^{n_k}$. For example, $\mathrm{entry}(2^3 \cdot 3^2 \cdot 5^6 \cdot 7^4, 3) = 6$.

Finally, we can define a primitive recursive function $\mathrm{len}(x)$ that returns the length of the sequence coded by x:

$$\mathrm{len}(x) = \mu y \leq x\ \forall z \leq x\ (z \geq y \rightarrow \mathrm{entry}(x, z) = 0).$$

Thus $\mathrm{len}(2^3 \cdot 3^2 \cdot 5^6 \cdot 7^4) = 4$.

7.3 Unbounded search

Any arithmetical function you can think of is almost certainly primitive recursive. But not all computable functions on the natural numbers are primitive recursive. A concrete counterexample is the Goodstein function.

To explain this function, I need the fact that any number x can be expressed as a sum of powers of n, for any choice of $n > 1$. For example, choosing $n = 2$, we can express 266 as $2^8 + 2^3 + 2^1$. Here, the exponents are 8, 3, and 1. If we write these as powers of 2 as well, we get the "hereditary base-2 representation" of 266:

$$266 = 2^{2^{2+1}} + 2^{2+1} + 2^1.$$

Starting with any number n, we can now define a sequence of numbers, called the *Goodstein sequence for n*. The first item in the sequence is n. For the second item, we replace each 2 in the hereditary base-2 representation of n by 3, and subtract 1. So the second item in the Goodstein sequence for 266 is

$$3^{3^{3+1}} + 3^{3+1} + 3^1 - 1 = 7,625,597,484,987.$$

For the third item, we replace each 3 in the hereditary base-3 representation of the second item by 4, and subtract 1. And so on. While Goodstein sequences initially grow large very quickly, Reuben Goodstein proved that their growth eventually stalls and reverses, until it reaches 0. (This is *Goodstein's Theorem*.) The *Goodstein function* now maps any number n to the length of the Goodstein sequence for n before it reaches 0. It can be shown that this function isn't primitive recursive. But it is clearly computable: from each item in the sequence, one can mechanically compute the next item. To compute the Goodstein function for n, we therefore simply need to compute all items in the sequence, one by one, until we reach 0, keeping count of how many items we've computed.

For another example of a computable function that isn't primitive recursive, note that we can effectively enumerate the primitive recursive functions: we start with the base functions, then we list all functions that can be obtained from these by one application of composition or primitive recursion, followed by all functions that require two applications of these operations, and so on. Let $f_1, f_2, f_3, \ldots$ be this enumeration, but omitting any functions with more than one argument. We can now define an antidiagonal function d by setting

$$d(n) = f_n(n) + 1.$$

Since all primitive recursive functions are total, this function is well-defined. It is evidently computable. But it can't be primitive recursive, since it differs from each primitive recursive function f_n at input n.

How would we compute $d(n)$? We would first identify the n-th one-place primitive recursive function f_n. Then we would compute $f_n(n)$ until we get the output, to which we would add 1. Like the computation of the Goodstein function, this computation involves an unbounded loop: we simply have to wait until $f_n(n)$ returns an output; we can't tell in advance how long this will take.

If we want to capture all computable functions, we need to add an operation that allows for this kind of unbounded search. This operation will search through all numbers $0, 1, 2, \ldots$ until it finds a number x for which a given condition $P(x)$ is satisfied.

We've already met such an operation above, in the form of the unbounded minimization operation μ: $\mu x\, P(x)$ is the least number x for which $P(x)$ holds. Given a two-place function $f(x, y)$, we can use μ to define a 1-place function h that maps any number x to the least number y for which $f(x,y)$ equals a desired value k:

$$h(x) = \mu y\,(f(x,y) = k).$$

Without loss of generality, we can assume that the desired value is always 0: if we want to find the least y such that $f(x,y) = k$, we can equivalently look for the least y such that $g(x,y) = 0$ where $g(x,y)$ is defined as $f(x,y) \dot{-} k$.

So assume that f is a total function of $n + 1$ arguments. (We'll deal with non-total functions in a moment.) Then the n-place function h defined by

$$h(x_1,\dots,x_n) = \mu x f(x_1,\dots,x_n,x) = 0$$

is called the *minimization* of f. We write $h = \mathrm{Mn}[f]$.

If f is computable then so is $\mathrm{Mn}[f]$. We simply need to compute $f(x_1,\dots,x_n,i)$ for each $i = 0, 1, 2, \dots$ until we find an i for which $f(x_1,\dots,x_n,i) = 0$:

```
function Mn_f(x1, ..., xn):
    let i = 0
    while f(x1, ..., xn, i) != 0:
        i = i + 1
    return i
```

If there is no i for which $f(x_1,\dots,x_n,i) = 0$, this algorithm runs forever. Thus $\mathrm{Mn}[f]$ may fail to be total, even if f is total. For example, $\mathrm{Mn}[+]$, the minimization of the addition function, returns 0 for input 0, but is undefined for every other input: if $x > 0$, there is no y such that $x + y = 0$.

A function $f(x_1,\dots,x_n,y)$ is called *regular* if it is total and for all $x_1,\dots,x_n$ there is some y such that $f(x_1,\dots,x_n,y) = 0$. When minimization is applied to a regular function f, the result is always total. In that case, we say that $\mathrm{Mn}[f]$ is defined by *regular minimization* from f.

So far, I've assumed that Mn is applied to a total function. We can also apply minimization to non-total functions. But we need a further constraint to ensure that $\mathrm{Mn}[f]$ is computable. Suppose $f(x,y)$ is 0 for some x,y, and undefined for the same x and some $z < y$. Then we may not be able to effectively search for the least y with $f(x,y) = 0$ by checking $f(x,0), f(x,1), f(x,2), \dots$: if the computation of, say, $f(x,2)$ never halts, the search doesn't proceed beyond 2. We therefore stipulate that if f is an $n + 1$-place function, then $\mathrm{Mn}[f]$ is the function that takes n numbers $x_1,\dots,x_n$ as input and returns the least number y for which

(i) $f(x_1, \dots, x_n, y) = 0$, and

(ii) $f(x_1, \dots, x_n, z)$ is defined for all $z < y$.

If there is no such y, $\mathrm{Mn}[f](x_1, \dots, x_n)$ is undefined.

Exercise 7.12 Let $f(x, y) = x \cdot y$. What is $\mathrm{Mn}[f]$? Is it total? Is it regular?

Exercise 7.13 Assume that g is a total recursive two-place function, and f is a total recursive one-place function. Show that we can construct a recursive function h such that $h(x)$ is the least y for which $g(x, y) = f(y)$, assuming such a y always exists.

Exercise 7.14 Assume that R is primitive recursive two-place relation. Show that we can construct a recursive function h such that $h(x)$ is the least number y for which $R(x, y)$ holds. (If there is no such y, $h(x)$ is undefined.)

Exercise 7.15 Consider the function $h(x) = \mu y\,(2y = x)$. What does this function do? Can you define h with the Mn notation?

Exercise 7.16 Use minimization to define a one-place function $h(x)$ that is undefined for every input x.

If we add minimization to our toolkit for constructing functions, we get the class of partial recursive functions. If we add regular minimization, we get the class of (total) recursive functions.

Definition 7.3

A function is *partial recursive* if it can be defined from the base functions s, z, and π^n_i by finitely many applications of composition, primitive recursion, and minimization.

Definition 7.4

A function is *(total) recursive* (a.k.a. *μ-recursive*) if it can be defined from the base functions s, z, and π^n_i by finitely many applications of composition, primitive recursion, and regular minimization.

As before, we can extend the concept of recursiveness to sets and relations.

Definition 7.5

A set is recursive if its characteristic function is (total) recursive. A relation is recursive if its extension is (total) recursive.

Above, I mentioned two functions that are computable but not primitive recursive: the Goodstein function and the antidiagonal of the primitive recursive functions. Both these functions are recursive. There is no known example of a computable function that is not recursive, and there are good reasons to believe that no such function exists. As we're going to show next, any such function would also be uncomputable by a Turing machine.

7.4 Recursiveness and Turing-computability

We'll now show that the class of partial recursive functions coincides precisely with the class of Turing-computable functions. We take the two directions in turn, starting with the easier direction: every partial recursive function is Turing-computable.

The proof idea is simple. Since every partial recursive function is built up from the base functions by composition, primitive recursion, and minimization, all we need to show is that the base functions are Turing-computable, and that the Turing-computable functions are closed under composition, primitive recursion, and minimization.

Theorem 7.1

Every partial recursive function is Turing-computable.

Proof sketch. The proof is by induction on the construction of partial recursive functions. I assume the same coding convention as in section 6.2, so that a number n is represented by a block of $n + 1$ strokes.

Base functions. You designed a Turing machine for the successor function in exercise 6.4. A machine for the zero function erases the input, writes a stroke, and halts. A machine for the projection functions erases all but one of its input blocks. These machines are trivial to design.

Composition. I assume for readability that $n = 1$ and $m = 2$. Suppose we have Turing machines for computing g_1, g_2 and f. We can design a machine for computing $h = \text{Cn}[f, g_1, g_2]$ on any input x as follows. The machine first calls the g_1 and g_2 machines (as subroutines) on input x, and stores the results next to each other, separated by a blank. It then calls the f machine on this pattern of strokes and blanks. The output is $f(g_1(x), g_2(x))$.

Primitive recursion. Suppose we have Turing machines for computing f and g. Let $h = \text{Pr}[f, g]$. That is, $h(x, 0) = f(x)$ and $h(x, y{+}1) = g(x, y, h(x, y))$, assuming for readability that f is 1-place. The machine for computing h works as follows. Given input x, y, it first calls the f-machine on x and stores the result in a block that will eventually hold $h(x, y)$; call this the "result block". The input y is kept on the tape in a separate "y block". In yet another block, we initialize a counter to 0 (represented by a single stroke). The machine then enters a loop. If the counter has the same length as the y block, the machine erases everything except the result block and halts. Otherwise it calls the g machine on x, the current counter value, and the current result block, and stores the output in the result block. The machine then increments the counter (by adding one stroke) and enters the next iteration of the loop. The loop will run exactly y times before halting. At that point, the result block will contain $h(x, y)$.

Minimization. Suppose we have a Turing machine for computing f. Let $h = \text{Mn}[f]$. that is, $h(x) = \mu y\, [\, f(x, y) = 0\,]$, assuming for readability that h is 1-place. We can construct a machine for computing h as follows. First, the machine initialises a "y block" to 0, represented by a single stroke. It then goes into a loop. In each iteration, it runs the machine for f on x and the current y block. If the output is 0, the machine halts and erases everything except the y block. Otherwise, the machine adds a stroke to the y block and enters the next iteration of the loop. If there is some y such that $f(x, y) = 0$ and $f(x, z)$ is defined for all $z < y$, this machine will output the least such y. □

Now for the other direction: every Turing-computable function (on $\mathbb{N}$) is recursive. Let M be a Turing machine that computes some (possibly partial) function f on $\mathbb{N}$. For simplicity, let's assume that f is a 1-place function. Our task is to find a recursive definition of f. Here's an outline of how this can be done.

Remember that each stage of a Turing machine computation is captured by a *configuration*. A configuration records the current state of the machine, the position of the head, and the contents of the tape. The initial configuration of our machine M on some input x, for example, specifies that the machine is in state q_0 and that its head is scanning the leftmost stroke of a block of $x+1$ strokes on an otherwise blank tape. We can code any such configuration as a natural number. Let init be a function that takes a number x as input and outputs the code number of M's initial configuration for input x. With a suitable coding scheme, this function will be primitive recursive.

Let $\text{next}(c)$ be a function that takes the code number c of a configuration as input and outputs the code number of the next configuration, according to the rules of M. If there are no applicable rules (i.e., if M halts in configuration c), we let $\text{next}(c)$ equal c. The function next is also primitive recursive.

From init and next, we can define (by primitive recursion) another function conf that takes an input x and a step number y, and outputs the code number of M's configuration after y steps on input x:

$$\begin{aligned} \text{conf}(x, 0) &= \text{init}(x) \\ \text{conf}(x, s(y)) &= \text{next}(\text{conf}(x, y)). \end{aligned}$$

We need two more functions. Let runs map the code number of any halting configuration of M to 0 and any other number to 1. Let out take the code number of a halting configuration as input and extract the content of the tape as output. Both of these are primitive recursive. We can now define the function f computed by M:

$$f(x) = \text{out}(\text{conf}(x, \mu y[\text{runs}(\text{conf}(x, y)) = 0])).$$

This says that $f(x)$ is the output extracted from the configuration at the first step at which M halts on input x.

The following proof sketch fills in a few more details.

Theorem 7.2

Every Turing-computable function is partial recursive.

Proof sketch. Let M be a Turing machine computing a (partial) function f on $\mathbb{N}$.

Each configuration of M can be coded as a quadruple $\langle q,\ L,\ s,\ R\rangle$, where q is the current state, $s \in \{0, 1\}$ is the scanned symbol (0 for blank, 1 for a stroke), and R is a finite sequence of 0s and 1s giving the contents of the tape to the right of the head (0 for blank, 1 for stroke) up to the last non-blank symbol, and L is a finite sequence of 0s and 1s giving the contents of the tape to the left of the head, in reverse order, up to the last non-blank symbol. For example, if the tape is 0 1 0 1 1 1 1 1 0 0 and the head is at the shaded cell, s would be 1, R would be 1, and L would be 1101. We can read s, R, and L as numbers in binary notation. (In the example, $s = 1$, $R = 1$, and $L = 2^3 + 2^2 + 2^0 = 13$.) If we code the state q as a number (using i for q_i), the entire configuration becomes a quadruple of natural numbers. We can code this quadruple as a single natural number using Gödel's prime-exponent coding (section 5.5). To extract the components of a coded configuration, we can use the primitive recursive functions len and entry from section 7.2.

The initial configuration for input x has state q_0, scanned symbol 1 (since the input is a block of $x + 1$ strokes), an empty left sequence, and a right sequence consisting of x many 1s (the input block minus the scanned stroke). The corresponding quadruple of numbers is

$$\langle 0,\ 0,\ 1,\ 2^x - 1 \rangle.$$

The init function therefore takes x as input and outputs the code number of this quadruple:

$$\text{init}(x) = 2^0 \cdot 3^0 \cdot 5^1 \cdot 7^{2^x - 1}.$$

This function is evidently primitive recursive.

To define next, we need a predicate $\text{HasRule}(x, y)$ that tests whether the machine table of M has a rule for state x and symbol y. (So $\text{HasRule}(2, 1)$ is true iff the machine has a rule for what to do in state q_2 when scanning a stroke.) Since the machine table is finite, this is a finite boolean combination of equalities, hence primitive recursive.

We can similarly define functions $\text{nextState}(x, y)$, $\text{write}(x, y)$, and $\text{move}(x, y)$ that extract the relevant components of the rule for state x and symbol y in the machine table (and return some arbitrary default value if no such rule exists).

With these in place, we can define next by cases. Given a coded configuration c as input, from which we can extract the quadruple $\langle q, L, s, R \rangle$ using entry, the next function first checks if $\text{HasRule}(q, s)$. If no, it returns c. If yes, it computes $\text{move}(q, s)$, $\text{write}(q, s)$, and $\text{nextState}(q, s)$. If the move is to the right, the new L becomes the old L with the written symbol appended at the front (in binary), the new scanned symbol becomes the first symbol of R, and the new R becomes the rest of R. If the move is to the left, the new R becomes the old R with the written symbol appended at the front, the new scanned symbol becomes the first symbol of L, and the new L becomes the rest of L. These operations on binary numbers (appending/deleting/extracting the first symbol) are primitive recursive. So next is primitive recursive.

We define conf as described above:

$$\begin{aligned}\mathrm{conf}(x,0) &= \mathrm{init}(x)\\ \mathrm{conf}(x,s(y)) &= \mathrm{next}(\mathrm{conf}(x,y)).\end{aligned}$$

The function runs is easily defined from HasRule:

$$\mathrm{runs}(c) = \begin{cases} 1 & \text{if } \mathrm{HasRule}(\mathrm{entry}(c,0),\mathrm{entry}(c,2)),\\ 0 & \text{otherwise.}\end{cases}$$

It remains to define the out function that extracts the output number represented by the tape contents in a halting configuration c. This simply needs to add the number of 1s in the left sequence, the scanned symbol, and the right sequence, and subtract 1.

Finally, we can define the function f computed by M, as announced above:

$$f(x) = \mathrm{out}(\mathrm{conf}(x, \mu y[\mathrm{runs}(\mathrm{conf}(x,y)) = 0])).$$

Equivalently,

$$f = \mathrm{Cn}[\mathrm{Cn}[\mathrm{out},\mathrm{conf}], \pi^1_1, \mathrm{Mn}[\mathrm{Cn}[\mathrm{runs},\mathrm{conf}]]].$$

□

Notice that the entire construction only uses a single unbounded μ, at the very end. We've therefore discovered an interesting corollary:

Theorem 7.3

Every partial recursive function can be defined using a single instance of Mn.

Proof. Immediate from the proof of Theorem 7.2.

What about total recursive functions? We can show that the total recursive functions are precisely the Turing-computable total functions. It follows that a function is recursive iff it is partial recursive and total.

Theorem 7.4

A total function is recursive iff it is Turing-computable.

Proof sketch. The left-to-right direction is immediate from Theorem 7.1. For the right-to-left direction, we show that whenever minimization yields a total function, it can be replaced by regular minimization.

Let f be a partial recursive function, and $h = \text{Mn}[f]$. By Theorem 7.1, there is a Turing machine M that computes f. Assume h is total. This means that for any x there is a y such that M halts on input x, y with output 0, and M halts on input x, z with some nonzero output for all $z < y$. (I assume without loss of generality that f has one argument.) It follows that for any x there is a y and a bound t such that

(i) M halts within t steps on input x, y with output 0, and

(ii) for all $z < y$, M halts within t steps on input x, z with some nonzero output.

We can express (i) and (ii) in terms of conf, out, and out from the proof of Theorem 7.2. That is, we can define a primitive recursive 3-ary predicate H so that $H(x, y, t)$ holds iff x, y, and t satisfy conditions (i) and (ii). Define

$$g(x, w) = \begin{cases} 0 & \text{if } H(x, \text{entry}(w, 0), \text{entry}(w, 1)), \\ 1 & \text{otherwise.} \end{cases}$$

So $g(x, w)$ returns 0 iff w encodes a pair $\langle y, t \rangle$ such that (i) and (ii) hold of x, y, and t. The function g is regular. We can define h by regular minimization from g:

$$h = \text{Cn}[\text{entry}, \pi^1_1, \text{Mn}[g]].$$

□

Since the Turing-computable functions and the partial recursive functions coincide, it doesn't matter if we state the Church-Turing thesis (Section 5.2) as the claim that the computable functions are precisely the partial recursive functions or as the claim that the computable functions are precisely the Turing-computable functions. The two claims are equivalent.

I can now also explain – if it isn't clear already – why the Church-Turing Thesis is true. Take any partial recursive function: a function that can be defined from zero, successor, and projection by composition, primitive recursion, and minimization. The definition effectively *gives us* an algorithm for computing the function. That such an algorithm exists, or that one could in principle follow it mechanically, is not a speculative conjecture. This direction of the Church-Turing thesis is beyond doubt.

The other direction, that every computable function is recursive, requires a little more argument. Here, we may draw on the fact that any mechanical computation must be representable as a rule-based, step-by-step manipulation of symbols, where the manipulations at each step are determined by a finite set of predefined rules. This isn't fully precise, but every way of making it precise leads to the same conclusion: the computation can be simulated by a Turing machine, and so it computes a partial recursive function.

Exercise 7.17 Give a diagonal argument to show that the set of total computable functions is not computably enumerable. Using the Church-Turing thesis, show that (a) the set of total recursive functions is not computably enumerable, and therefore (b) the set of regular recursive functions is not decidable.

7.5 Feasible computation

I mentioned in section 7.1 that the base functions s, z, and π^n_i are computable "in one step", without subroutines or loops. We can also count the number of steps needed to compute more complex functions. For example, if h is defined by composition from f and g, so that $h(x) = f(g(x))$, then one can compute h by first computing $g(x)$, then feeding the result into f; the total number of steps is the sum of the number of steps needed to compute f and g, for the given input x. We may also count the steps in a Turing machine computation that executes a given algorithm. Again, the number of steps will generally depend on the input.

Either way, the "step count" gives us a way to measure the *computational complexity* of an algorithm. The field of computational complexity theory studies different types of complexity. For example, in the class of *linear-time* algorithms, the number of steps it takes to compute an output is (at most) proportional to the size of the input. In the broader class of *polynomial-time* algorithms, the number of steps is bound by a polynomial function of the input size. For example, if the input has size n, the number of steps might be bound by n^2, or by $10n^{10} + 3n^7$.

The class of polynomial-time algorithms turns out to be very natural. It doesn't depend on details of how we count steps or how we measure the size of the input; it is also closed under composition and "subroutine insertion", wherein an arbitrary part of an algorithm is replaced by another algorithm. In analogy to the Church-Turing theses, the polynomial-time algorithms have been suggested to formalize the informal concept of a *feasible* algorithm.

Consider, for example, the task of checking whether a given sentence S of propositional logic is true in a given model σ, which assigns a truth-value to every sentence letter. It's easy to show that this can be achieved by a polynomial-time algorithm, using the truth-table method. This algorithm is feasible. By contrast, consider the task of checking whether an $\mathfrak{L}_0$-sentence S is true relative to *some* assignment of truth-values – that is, whether it is satisfiable. How could we do this? The obvious "brute-force" algorithm is to try out all possible assignments of truth-values to the sentence letters in S. This algorithm is not polynomial, but *exponential* in the number of sentence letters. For 100 sentence letters, it requires $2^{100} \approx 10^{30}$ steps. This is clearly not a feasible algorithm.

Oddly, it is not known whether there is also a feasible, polynomial-time algorithm for deciding whether an $\mathfrak{L}_0$-sentence is satisfiable. Nobody has yet found such an algorithm, and it is generally believed that none exists. But we don't know for sure. This is an instance of the notorious *P vs NP problem*, which has yet to be resolved.

8 Arithmetical Definability

In this chapter, we'll show that all computable functions and relations on the natural numbers can be defined in the language $\mathfrak{L}_A$ of arithmetic. As foreshadowed at the end of Chapter 5, it will follow that there can be no true, axiomatizable, and complete theory of arithmetic. Further limitative consequences will be explored in the next two chapters.

8.1 Definability

In Section 4.1, I introduced the language $\mathfrak{L}_A$, with non-logical symbols for the number 0 ('0'), the successor function ('s'), addition ('+'), and multiplication ('×'). Other arithmetical concepts can be defined in terms of these primitives. Let's think about what this involves.

Back in Section 4.1, I suggested that we can define the less-than relation $<$ by stipulating that for any terms t_1 and t_2, '$t_1 < t_2$' is short for '$\exists z(t_1 + s(z) = t_2)$', where z is a variable not occurring in t_1 or t_2. This works because every $\mathfrak{L}_A$-term denotes a natural number (in the standard model $\mathfrak{A}$), and a number a is less than a number b iff there is a non-zero number c such that $a + c = b$.

An $\mathfrak{L}_A$-*numeral* is an $\mathfrak{L}_A$-term constructed from 0 and s alone. Every natural number is denoted by a unique $\mathfrak{L}_A$-numeral (in $\mathfrak{A}$): 0 is denoted by '0', 1 by '$s(0)$', 2 by '$s(s(0))$', and so on. To avoid clutter, I will use '$\overline{n}$' as a shorthand for the $\mathfrak{L}_A$-numeral of the number n. So $\overline{5}$ is '$s(s(s(s(s(0)))))$'. Since every $\mathfrak{L}_A$-term denotes (in $\mathfrak{A}$) a number n that is also denoted by an $\mathfrak{L}_A$-numeral $\overline{n}$, it suffices to focus on $\mathfrak{L}_A$-numerals when examining whether an $\mathfrak{L}_A$-expression defines a relation on the natural numbers.

Officially, we'll say that an n-ary relation is defined by an $\mathfrak{L}_A$-formula with n free variables. The less-than relation $<$, for example, is defined by the formula $\exists z(x + s(z) = y)$, with free variables x and y. The free variables are placeholders. $\exists z(x + s(z) = y)$ defines the less-than relation because, for any numbers a and b, $\exists z(\overline{a} + s(z) = \overline{b})$ is true (in $\mathfrak{A}$) iff $a < b$. In general:

Definition 8.1

An $\mathfrak{L}_A$-formula $A(x_1, \dots, x_n)$ *defines* an n-place relation R on $\mathbb{N}$ iff, for all numbers $a_1, \dots, a_n$, $A(\overline{a_1}, \dots, \overline{a_n})$ is true in $\mathfrak{A}$ iff $a_1, \dots, a_n$ stand in the relation R.

Exercise 8.1 Give two other $\mathfrak{L}_A$-formulas that define the less-than relation, and explain why they do so.

So far, I've talked about relations. We can also define new functions in $\mathfrak{L}_A$. For example, we might say that the expression $x \times x$ defines the squaring function that maps any number a to a^2. Often, however, a function won't be definable in this way by an $\mathfrak{L}_A$-term. Instead, we have to resort to what I called a "syncategorematic" definition in Section 4.2. For example, we'll see that there is an $\mathfrak{L}_A$-formula $F(x, y)$ such that $F(\overline{a}, \overline{b})$ is true (in $\mathfrak{A}$) iff b is the factorial $a!$ of a. (That is, $b = 1 \times 2 \times \dots \times a$.) With the help of this formula, any statement about factorials can be expressed in $\mathfrak{L}_A$. The (false) statement that every number is less than its factorial, for example, can be expressed as

$$\forall x \forall y (F(x, y) \to x < y).$$

It's important to clarify what kind of "definition" we are after. In the previous chapter, I talked about defining functions by primitive recursion. The primitive recursive definition of the factorial function would look as follows:

$$\begin{aligned} 0! &= 1 \\ s(y)! &= s(y) \times y!. \end{aligned}$$

But this doesn't define the factorial function *in* $\mathfrak{L}_A$. A definition in $\mathfrak{L}_A$ would be a single $\mathfrak{L}_A$-formula $F(x, y)$ that is true of x and y iff y is the factorial of x.

Definition 8.2

An $\mathfrak{L}_A$-formula $A(x_1, \dots, x_n, y)$ *defines* a (total) n-ary function f on $\mathbb{N}$ iff, for all numbers $a_1, \dots, a_n, b$, $A(\overline{a_1}, \dots, \overline{a_n}, \overline{b})$ is true in $\mathfrak{A}$ iff $f(a_1, \dots, a_n) = b$.

Definition **??** could be extended to partial functions, but we'll only be interested in total functions in this chapter.

We say that a relation or function is *definable* in $\mathfrak{L}_A$ if it is defined by some $\mathfrak{L}_A$-formula.

Exercise 8.2 Give an $\mathfrak{L}_A$-formula that defines the addition function.

Exercise 8.3 Give an $\mathfrak{L}_A$-formula that defines the switcheroo function δ that maps any positive number to 0 and 0 to 1.

We'll show that all recursive functions and relations – and therefore, by the Church-Turing thesis, all computable functions and relations – are definable in $\mathfrak{L}_A$. This is not obvious. The factorial function, for example, is recursive, but it is hard to find an $\mathfrak{L}_A$-formula $F(x, y)$ that defines it. (Try it!)

For a different type of example, consider the halting-with-bound relation that holds between the code number of a Turing machine M, a number n, and a number k iff M halts on input n within k steps. This relation is computable: we can simply run M on n for k steps, and return 'yes' if M has halted by then, and 'no' otherwise. Since all computable relations are definable in $\mathfrak{L}_A$, there must be an $\mathfrak{L}_A$-expression $H(x, y, z)$ so that $H(\overline{m}, \overline{n}, \overline{k})$ is true (in $\mathfrak{A}$) iff the Turing machine coded by m halts on input n within k steps. It's not at all obvious what such a formula would look like.

As I mentioned in Section 5.5, the definability of halting-with-bound leads to a version of Gödel's incompleteness theorem. If $H(x, y, z)$ defines halting-with-bound then $\exists z H(x, y, z)$ defines the halting relation: $\exists z H(\overline{m}, \overline{n}, z)$ is true (in $\mathfrak{A}$) iff the Turing machine coded by m halts on input n. We know from Theorem 6.1 that there is no algorithm that decides the halting relation. It follows that there can be no complete, axiomatizable, and true theory of arithmetic: otherwise we could decide the halting relation by checking, for any numbers m and n, whether $\exists z H(\overline{m}, \overline{n}, z)$ or its negation is entailed by the axioms.

This version of the incompleteness theorem is sometimes called "semantic", as it concerns theories that are *true*. Gödel himself gave prominence to a "syntactic" version of incompleteness that (in its contemporary form) only requires the relevant theories to be consistent. The syntactic theorem relies not on definability in $\mathfrak{L}_A$ but on a slightly stronger concept – representability – that I'm going to introduce next.

Exercise 8.4 Explain why every finite set is definable in $\mathfrak{L}_A$.

Exercise 8.5 Explain why the Busy Beaver function Σ is definable in $\mathfrak{L}_A$. (Hint: consider the relation that holds between four numbers m, n, t, k iff m codes a Turing machine with n states that halts, on blank input, after t steps leaving k strokes on the tape.)

8.2 Representability

'+', '×', 's' and '0' are non-logical symbols. They have an *intended interpretation*, but this interpretation isn't built into the language. An axiomatic $\mathfrak{L}_A$-theory doesn't automatically "know" what the non-logical symbols mean. '0' and '$s(0)$', for example, denote different numbers in the intended interpretation, but an axiomatic theory may not be able to prove '$0 \neq s(0)$'.

Exercise 8.6 Specify an $\mathfrak{L}_A$-theory that proves $0 = s(0)$.

The standard axiomatic theory of arithmetic, Peano Arithmetic (PA), can prove $0 \neq s(0)$. Indeed, whenever $a \neq b$, PA contains $\overline{a} \neq \overline{b}$. Trivially, if $a = b$ then PA contains $\overline{a} = \overline{b}$, for then $\overline{a}$ and $\overline{b}$ are the same term. So PA knows all particular facts about equality between numbers: that $\overline{17} = \overline{17}$, that $\overline{17} \neq \overline{29}$, and so on.

PA also knows all particular facts about the less-than relation: whenever $a < b$, PA contains $\overline{a} < \overline{b}$, and whenever $a \nless b$, PA contains $\neg(\overline{a} < \overline{b})$. Of course, '<' isn't really part of the language. What I mean is that whenever $a < b$, PA contains $\exists z(\overline{a} + s(z) = \overline{b})$, and whenever $a \nless b$, PA contains $\neg\exists z(\overline{a} + s(z) = \overline{b})$. In that sense, $\exists z(x + s(z) = y)$ *represents* the less-than relation in PA.

Definition 8.3

An $\mathfrak{L}_A$-formula $A(x_1, \ldots, x_n)$ *represents* an n-ary relation R on $\mathbb{N}$ in a theory T iff, for all numbers $a_1, \ldots, a_n$,

(i) if R holds of $a_1, \ldots, a_n$, then $\vdash_T A(\overline{a_1}, \ldots, \overline{a_n})$, and

(ii) if R does not hold of $a_1, \ldots, a_n$, then $\vdash_T \neg A(\overline{a_1}, \ldots, \overline{a_n})$.

Exercise 8.7 Explain why every relation is representable in the inconsistent theory.

Exercise 8.8 Explain why a formula defines a relation in $\mathfrak{L}_A$ iff it represents the relation in $\mathrm{Th}(\mathfrak{A})$. ($\mathrm{Th}(\mathfrak{A})$ is the set of all $\mathfrak{L}_A$-sentences that are true in $\mathfrak{A}$).

We can also talk about representing functions:

Definition 8.4

An $\mathfrak{L}_A$-formula $A(x_1, \dots, x_n, y)$ *represents* a (total) n-ary function f on $\mathbb{N}$ in T iff, for all numbers $a_1, \dots, a_n$,

(i) $\vdash_T A(\overline{a_1}, \dots, \overline{a_n}, \overline{f(a_1, \dots, a_n)})$, and

(ii) $\vdash_T \forall y (A(\overline{a_1}, \dots, \overline{a_n}, y) \to y = \overline{f(a_1, \dots, a_n)})$.

Here, $\overline{f(a_1, \dots, a_n)}$ is the $\mathfrak{L}_A$-term with $f(a_1, \dots, a_n)$ occurrences of 's'. Condition (ii) effectively requires T to know that $A(x_1, \dots, x_n, y)$ expresses a functional relationship (as discussed in Section 4.2).

An example may help. What is needed for a formula $F(x, y)$ to represent the factorial function in a theory T? Condition (i) requires that whenever b is the factorial of a then T proves $F(\overline{a}, \overline{b})$. This leaves open that T also proves $F(\overline{a}, \overline{c})$ for some $c \neq b$. Indeed, the formula $x = x \land y = y$ passes condition (i) in any theory T, but there's no good sense in which this formula represents the factorial function. By condition (ii), T must know that there is no y other than b for which $F(\overline{a}, y)$ holds.

If a function or relation is represented in a theory T by some formula, we say that the function or relation is *representable* in T.

Proposition 8.1

If an $\mathfrak{L}_A$-formula represents a function in a theory $T \subseteq \mathrm{Th}(\mathfrak{A})$, then the formula also defines that function.

Proof. Assume $A(x, y)$ represents a function f in a theory T, where $T \subseteq \mathrm{Th}(\mathfrak{A})$. I'll assume for readability that f is one-place. We have to show that for all numbers a, b, $A(\overline{a}, \overline{b})$ is true in $\mathfrak{A}$ iff $f(a) = b$. For the 'if' direction, assume that $f(a) = b$. By condition (i) in definition 8.4, $\vdash_T A(\overline{a}, \overline{b})$. Since $T \subseteq \mathrm{Th}(\mathfrak{A})$, $A(\overline{a}, \overline{b})$ is true in $\mathfrak{A}$. For the 'only if' direction, assume that $f(a) \neq b$. By condition (ii) in definition 8.4, $\vdash_T \forall y (A(\overline{a}, y) \to y = \overline{f(a)})$. So $\forall y (A(\overline{a}, y) \to y = \overline{f(a)})$ is true in $\mathfrak{A}$. Since $\overline{f(a)} \neq \overline{b}$ is true in $\mathfrak{A}$, it follows that $\neg A(\overline{a}, \overline{b})$ is true in $\mathfrak{A}$, and so $A(\overline{a}, \overline{b})$ is false in $\mathfrak{A}$. □

Exercise 8.9 '$x+y = z$' represents addition in PA. Let PA* be the theory obtained by swapping '+' and '×' everywhere in PA. Can you find a formula that represents addition in PA*?

In the following sections, we'll show that all recursive functions are representable in any arithmetical theory that knows some basic facts about arithmetic. It will follow by Proposition 8.1 that all recursive functions are definable in $\mathfrak{L}_A$.

We can focus on functions because a relation is recursive iff its characteristic function is recursive. As the next proposition shows, it follows that the representability of all recursive functions entails the representability of all recursive relations, at least in any theory that can prove $0 \neq 1$.

Proposition 8.2

A relation R is representable in a theory T iff its characteristic function χ_R is representable in T, provided that $\vdash_T 0 \neq \overline{1}$.

Proof. I'll assume for readability that R is a one-place relation.

For the left-to-right direction, assume that R is represented in T by some formula $A(x)$. Let $C(x,y)$ be the formula $(A(x) \land y = \overline{1}) \lor (\neg A(x) \land y = 0)$. I claim that $C(x,y)$ represents the characteristic function χ_R of R. Assume first that $\chi_R(a) = 1$. Then R holds of a, and T proves $A(\overline{a})$. Since $A(\overline{a})$ entails both $C(\overline{a},\overline{1})$ and $\forall y(C(\overline{a},y) \to y = \overline{1})$, T proves both of these as well. Similarly, if $\chi_R(a) = 0$, then T proves $\neg A(\overline{a})$, which entails $C(\overline{a},0)$ and $\forall y(C(\overline{a},y) \to y = 0)$. So whenever $\chi_R(a) = b$ then T proves $C(\overline{a},\overline{b})$ and $\forall y(C(\overline{a},y) \to y = \overline{b})$. So $C(x,y)$ satisfies the two conditions in definition 8.4.

For the other direction, assume $A(x,y)$ represents χ_R in T. This means that whenever R holds of a, then $\vdash_T A(\overline{a},\overline{1})$, by condition (i) in definition 8.4. Moreover, when R does not hold of a, then $\vdash_T A(\overline{a},0)$ by condition (i) and $\vdash_T \forall y A(\overline{a},y) \to y = 0$ by condition (ii). Assuming that $\vdash_T 0 \neq \overline{1}$, it follows that $A(x,1)$ represents R in T. □

8.3 Conditions for Representability I

We want to show that every recursive function is representable in any theory that knows some basic facts about arithmetic. We proceed by induction on the construction of recursive functions. By definition 7.4 in Section 7.3, every recursive function is constructed from the base functions zero, successor, and projection by composition, primitive recursion, and regular minimization. We'll first show that the base functions are representable in any theory. Then we'll show that representability in a theory T is preserved under composition, primitive recursion, and regular minimization, provided that T knows certain facts about arithmetic.

Let's start with the zero function z that maps every number a to 0. It's not hard to find a formula $A(x, y)$ so that $A(\overline{a}, \overline{b})$ is true (in $\mathfrak{A}$) iff $b = 0$. The formula $x = x \land y = 0$ does the job. So $x = x \land y = 0$ defines z in $\mathfrak{L}_A$. We need to confirm that it also represents z in any theory T.

Lemma 8.1

The zero function z is representable in every theory T.

Proof. I claim that z is represented in every theory T by the formula $x = x \land y = 0$. By definition 8.4, this means that, for all numbers a, T can prove

(i) $\overline{a} = \overline{a} \land 0 = 0$, and

(ii) $\forall y((\overline{a} = \overline{a} \land y = 0) \to y = 0)$.

Both of these are logical truths. □

Lemma 8.2

The successor function s is representable in every theory T.

Proof. I claim that $s(x) = y$ represents the successor function in every theory T: for all numbers a, T can prove

(i) $s(\overline{a}) = \overline{s(a)}$, and

(ii) $\forall y(s(\overline{a}) = y \to y = \overline{s(a)})$.

Since $\overline{s(a)}$ and $s(\overline{a})$ are the same term, both of these are logical truths. □

Lemma 8.3

Each projection function π^n_i is representable in every theory T.

Proof. Exercise.

Exercise 8.10 Prove Lemma 8.3.

Now for the closure operations. We start with composition.

Lemma 8.4

If an m-place function f is representable in a theory T and m n-place functions $g_1, \dots, g_m$ are representable in T, then the composition $h = Cn[f, g_1, \dots, g_m]$ is representable in T.

Proof. For readability, I only give the proof for $n = 1$ and $m = 2$.

Assume that f is represented in a theory T by a formula $F(x_1, x_2, y)$, and that g_1, g_2 are represented in T by $G_1(x, y_1)$ and $G_2(x, y_2)$, respectively. I claim that $h = Cn[f, g_1, g_2]$ is represented in T by the formula

$$\exists v_1 \exists v_2 (G_1(x, v_1) \wedge G_2(x, v_2) \wedge F(v_1, v_2, y)).$$

Condition (i) for representations requires that whenever $h(a) = b$ then

$$\vdash_T \exists v_1 \exists v_2 (G_1(\overline{a}, v_1) \wedge G_2(\overline{a}, v_2) \wedge F(v_1, v_2, \overline{b})).$$

So assume $h(a) = b$. Then there are c_1, c_2 such that $g_1(a) = c_1$, $g_2(a) = c_2$, and $f(c_1, c_2) = b$. Since g_1 and g_2 are represented by $G_1(x, y_1)$ and $G_2(x, y_2)$, respectively, and f is represented by $F(x_1, x_2, y)$, we have

$$\vdash_T G_1(\overline{a}, \overline{c_1})$$
$$\vdash_T G_2(\overline{a}, \overline{c_2})$$
$$\vdash_T F(\overline{c_1}, \overline{c_2}, \overline{b}).$$

The desired claim follows by the fact that T is closed under first-order consequence.

For condition (ii), we have to show that

$$\vdash_T \forall y (\exists v_1 \exists v_2 (G_1(\overline{a}, v_1) \wedge G_2(\overline{a}, v_2) \wedge F(v_1, v_2, y)) \rightarrow y = \overline{f(g_1(a), g_2(a))}).$$

This, too, follows from the representability conditions for F, G_1, and G_2, which yield

$$\vdash_T \forall y(G_1(\overline{a}, y) \to y = \overline{c_1})$$
$$\vdash_T \forall y(G_2(\overline{a}, y) \to y = \overline{c_2})$$
$$\vdash_T \forall y(F(\overline{c_1}, \overline{c_2}, y) \to y = \overline{f(c_1, c_2)}).$$ □

I leave the case of primitive recursion for last: it is by far the hardest. Let's turn to regular minimization. Suppose $h = \text{Mn}[f]$, where f is a regular function. Recall that $f(x, y)$ is regular if it is total and for all x there is some y such that $f(x, y) = 0$. The function h takes a number x and returns the least y for which $f(x, y) = 0$. Assuming that f is represented in T by some formula $F(x, y, z)$, we have: $h(x) = y$ iff $F(x, y, 0)$ and there is no $z < y$ such that $F(x, z, 0)$. We can directly translate this into $\mathfrak{L}_A$:

$$F(x, y, 0) \land \forall z(z < y \to \neg F(x, z, 0)).$$

This formula defines h in $\mathfrak{L}_A$. Some assumptions are needed for it to represent h in a theory T. Informally speaking, the theory must have some idea of what $<$ means. The following conditions are sufficient.

R1 For all a, $\vdash_T \forall x(\overline{a} < x \lor x = \overline{a} \lor x < \overline{a})$.
R2 $\vdash_T \neg\exists x(x < 0)$.
R3 For all $a > 0$, $\vdash_T \forall x(x < \overline{a} \to (x = 0 \lor \ldots \lor x = \overline{a - 1}))$.
R4$^-$ For all $a > 0, \vdash_T \overline{a} \neq 0$

Officially, '<' isn't part of the language. I assume here, and in what follows, that '$t_1 < t_2$' is short for '$\exists z(s(z) + t_1 = t_2)$', where z is a variable that doesn't occur in t_1 or t_2.

Lemma 8.5

If a regular function f is representable in a theory T, and T satisfies the conditions R1, R2, R3, and R4$^-$, then the minimization Mn$[f]$ of f is representable in T.

Proof. Assume that $F(x, y, z)$ represents f in T, and T satisfies R1, R2, R3, and R4$^-$. For readability, I'll assume that f is a two-place function. I claim that $h = \text{Mn}[f]$ is represented in T by $F(x, y, 0) \land \forall z(z < y \to \neg F(x, z, 0))$. We have to confirm that whenever $h(a) = b$ then T can prove

(i) $F(\overline{a}, \overline{b}, 0) \land \forall z(z < \overline{b} \to \neg F(\overline{a}, z, 0))$.

(ii) $\forall y(F(\overline{a}, y, 0) \land \forall z(z < y \to \neg F(\overline{a}, z, 0)) \to y = \overline{b})$.

From the fact that $h = \mathrm{Mn}[f]$ and $h(a) = b$, we know that $f(a, b) = 0$ and $f(a, c) \neq 0$ for all $c < b$. Since f is represented in T by F, T can prove

$$F(\overline{a}, \overline{b}, 0) \tag{1}$$

as well as (for $c < b$)

$$\forall y(F(\overline{a}, \overline{c}, y) \to y = \overline{f(a, c)}). \tag{2}$$

From (2) and R4$^-$, it follows that T can prove $\neg F(\overline{a}, \overline{c}, 0)$ for all $c < b$. By R3, T can prove $\forall z(z < \overline{b} \to (z = 0 \lor \ldots \lor z = \overline{b-1}))$ whenever $b > 0$. So if $b > 0$ then T can prove

$$\forall z(z < \overline{b} \to \neg F(\overline{a}, z, 0)). \tag{3}$$

For $b = 0$, (3) follows from R2. (i) is the conjunction of (1) and (3).

For (ii), we show (by reasoning "inside T") that T can derive $y = \overline{b}$ from

$$F(\overline{a}, y, 0) \land \forall z(z < y \to \neg F(\overline{a}, z, 0)). \tag{4}$$

(1) and (4) imply $\neg(\overline{b} < y)$. From (3) and (4), we have $\neg(y < \overline{b})$. By R1, it follows that $y = \overline{b}$. □

Now for the hard part: primitive recursion.

8.4 Conditions for Representability II

Return to the factorial function that maps each number n to $n! = 1 \times 2 \times \ldots \times n$. The construction by primitive recursion goes as follows:

$$0! = 1$$
$$s(y)! = s(y) \times y!$$

We need to find a formula $F(x, y)$ that expresses this function in $\mathfrak{L}_A$, so that $F(x, y)$ is true of numbers a and b iff $a! = b$.

The trick is to see the recursive construction as defining a sequence: $\langle 0!, 1!, 2!, \ldots, n! \rangle$. Our formula $F(x, y)$ will say that "y is the last element of the sequence $\langle 0!, 1!, \ldots x! \rangle$".

Of course, $\mathfrak{L}_A$ doesn't have terms for sequences. But we know that sequences of numbers can be coded as single numbers. Suppose we can find a formula $\text{ENTRY}(x, i, y)$ that expresses "y is the i-th entry in the sequence coded by x". Using $\text{ENTRY}(x, i, y)$, we can then define a formula $\text{SEQ}(z, x)$ saying that

(i) z codes a sequence whose first entry is 1, and

(ii) for all $i < x$, the $s(i)$-th entry in the sequence coded by z is the product of the ith entry and $s(i)$.

So $\text{SEQ}(z, x)$ will say that z codes the sequence $\langle 0!, 1!, \dots, x! \rangle$. From this, we can define $F(x, y)$ as $\exists z(\text{SEQ}(z, x) \land \text{ENTRY}(z, s(x), y))$. This says that there is a number z that codes $\langle 0!, 1!, \dots, x! \rangle$ and that y is the $s(x)$-th entry in that sequence. (We need an $s(x)$ here because $F(x, 0)$ is the *first*, not the *zero*-th, entry in the sequence.)

The main task, then, is to find the formula $\text{ENTRY}(x, i, y)$ that holds of numbers x, i, y iff y is the i-th entry in the sequence coded by x.

As you may remember, in Section 7.2 I showed how to construct a primitive recursive function entry(x, y) that returns the y-th number in the sequence coded by x, using the prime exponents method to code sequences of numbers. Unfortunately, my construction of the entry function involved primitive recursion, so we can't assume that this function is definable in $\mathfrak{L}_A$. In fact, we won't code sequences of numbers in terms of prime exponents, as we don't even have an $\mathfrak{L}_A$-formula for exponentiation yet. We'll use a different coding method.

To explain that method, assume first that we want to code a sequence $\langle a_1, a_2, \dots, a_n \rangle$ of numbers all of which are below 9. We could simply concatenate their decimal representation: $\langle 1, 7, 0, 7 \rangle$ would be coded as 1707. To simplify accessing individual elements of the sequence, we might store the indices (the position numbers) of the elements in the code, so that $\langle 1, 7, 0, 7 \rangle$ gets coded as 11273047. The third element can now be identified as the digit to the right of the '3' (in decimal representation). As it stands, this doesn't quite work because the indices can also be among the coded elements, as is the case for the number 1 in the example: there are two digits to the right of a '1'. We can disambiguate the indices by prefixing them with yet another digit, 9, that doesn't occur among the coded numbers. The code of $\langle 1, 7, 0, 7 \rangle$ becomes 911927930947. The i-th element can now be retrieved as the unique digit to the right of '9i' (in decimal representation).

We'll adapt this scheme to code arbitrary sequences of numbers. We obviously can't assume that all of the numbers are below 9. We therefore code sequences not to base 10, but to some base p that is at least 2 greater than all numbers in the sequence and all index numbers ($p - 1$ is used to mark the index numbers). For convenience, we'll always use a prime number as the base p. The sequence $\langle 1, 12, 0 \rangle$, for example, would be coded in base 17 as

$$16 \overset{17}{\frown} 1 \overset{17}{\frown} 1 \overset{17}{\frown} 16 \overset{17}{\frown} 2 \overset{17}{\frown} 12 \overset{17}{\frown} 16 \overset{17}{\frown} 3 \overset{17}{\frown} 0,$$

where '$\overset{17}{\frown}$' is the operation of concatenation in base 17. (I'll define this formally below.) If q is the code number of a sequence in base p, the i-th element of the sequence can be retrieved as

> alpha(p, q, i) = the unique number x for which $(p-1) \overset{p}{\frown} i \overset{p}{\frown} x$ is part of the base-p numeral of q.

We'll see that this can be expressed in $\mathfrak{L}_A$.

The alpha function retrieves elements from the code q of a sequence, but it also needs the base p as a key to the code. We can get rid of the extra argument by coding $\langle p, q \rangle$ into a single number. We have to use a different coding method here. We'll use the pairing function that we met in Section 3.1 when we discussed Cantor's zig-zag method:

$$\mathrm{j}(x, y) = \frac{1}{2}(x+y)(x+y+1) + y$$

The j function is easily defined in $\mathfrak{L}_A$. We can also define two functions l and r that extract the elements of a pair encoded by j, so that $\mathrm{l}(\mathrm{j}(x, y)) = x$ and $\mathrm{r}(\mathrm{j}(x, y)) = y$. If we have coded a sequence of numbers $\langle a_1, \dots, a_n \rangle$ by p and q as described above, and packaged these into a single number $c = \mathrm{j}(p, q)$, we can now retrieve any element a_i of the doubly coded sequence as

$$\mathrm{beta}(c, i) = \mathrm{alpha}(\mathrm{l}(c), \mathrm{r}(c), i).$$

We'll show that this function beta is definable in $\mathfrak{L}_A$. The formula BETA(x, y, z) that defines it is the formula ENTRY(x, y, z) that we were looking for.

I'm not actually going to write down BETA(x, y, z) as an $\mathfrak{L}_A$-formula. Instead, I'll show that the function beta can be constructed by composition and minimization from certain functions and relations that are easily definable in $\mathfrak{L}_A$: projection, addition, multiplication, and the characteristic function $\chi_=$ of identity. $\chi_=$ is the two-place function that returns 1 if its two arguments are equal and 0 otherwise. It is defined in $\mathfrak{L}_A$ by $(x = y \land z = 1) \lor (x \neq y \land z = 0)$. Projection, addition, and multiplication are also easily definable. We've dealt with projection in Lemma 8.3. Addition is defined by $x + y = z$; multiplication by $x \times y = z$. To ensure that these formulas also *represent* the relevant functions in a theory T, we need the following assumptions:

R4 For all a, b, if $a \neq b$ then $\vdash_T \overline{a} \neq \overline{b}$.
R5 For all a, b, $\vdash_T \overline{a} + \overline{b} = \overline{a + b}$.
R6 For all a, b, $\vdash_T \overline{a} \times \overline{b} = \overline{a \times b}$.

Note that R4 subsumes R4$^-$.

Lemma 8.6

Projection, addition, multiplication, and $\chi_=$ are representable in every theory that satisfies R4–R6.

Proof. Projection is representable in every theory by Lemma 8.3.

The addition function is represented by $x + y = z$ in every theory T that satisfies R5: condition (i) in the definition of representation is given by R5; condition (ii) then holds by first-order logic.

The multiplication function is represented by $x \times y = z$ in every theory T that satisfies R6: condition (i) is given by R6; (ii) holds by first-order logic.

$\chi_=$ is represented by $(x = y \wedge z = 1) \vee (x \neq y \wedge z = 0)$ in every theory T that satisfies R4. For condition (i), we need to show that if $\chi_=(a, b) = c$ then $\vdash_T (\overline{a} = \overline{b} \wedge \overline{1} = \overline{c}) \vee (\overline{a} \neq \overline{b} \wedge \overline{0} = \overline{c})$. There are two ways in which $\chi_=(a, b) = c$ can hold: either $a = b$ and $c = 1$ or $a \neq b$ and $c = 0$. In the first case, $\overline{a} = \overline{b}$ and $\overline{1} = \overline{c}$ are logical truths. In the second case, $\overline{0} = \overline{c}$ is a logical truth and $\overline{a} \neq \overline{b}$ holds by R4. Condition (ii) holds by first-order logic. □

Now we need to show that beta can be constructed from projection, addition, multiplication, and $\chi_=$ by composition and minimization. To this end, let's first introduce a name for the class of functions that can be so constructed. I'll call a function *legit* if it can be constructed from projection, addition, multiplication, and $\chi_=$ by composition and minimization; I'll call a relation *legit* if its characteristic function is legit.

Lemma 8.7

All legit functions and relations are representable in any theory that satisfies R1–R6.

Proof. By Lemma 8.6, projection, addition, multiplication, and $\chi_=$ are representable in any theory that satisfies R4–R6. By Lemmas 8.4 and 8.5, composition and minimization preserve representability in any theory that satisfies R1–R4. □

The following lemmas show that the class of legit relations is closed under truth-functional combinations, bounded quantification, and regular minimization. We've met the first two of these operations in Section 7.2. For the third, recall that the minimization $\mu y\, R(x_1, \ldots, x_n, y)$ of an $n+1$-ary relation R is the n-ary function that maps $x_1, \ldots, x_n$ to the least y such that $R(x_1, \ldots, x_n, y)$. *Regular* minimization is minimization applied to a relation that is *regular* in the sense that for all $x_1, \ldots, x_n$ there is some y such that $R(x_1, \ldots, x_n, y)$.

Lemma 8.8

The legit relations are closed under truth-functional combinations.

Proof. Let R and S be legit relations (of arity 1, for readability), and χ_R and χ_S their characteristic functions. Then

$$\begin{aligned} \chi_{R \wedge S}(x) &= \chi_R(x) \times \chi_S(x) \\ \chi_{\neg R}(x) &= \chi_=(\chi_R(x), 0). \end{aligned}$$

All truth-functional combinations can be constructed from conjunction and negation. □

Lemma 8.9

The legit relations are closed under regular minimization.

Proof. Let $R(x_1, \ldots, x_n, y)$ be a regular legit relation. Then $\mu y\, R(x_1, \ldots, x_n, y)$ is $\mathrm{Mn}[\chi_{\neg R}]$. Since $\chi_{\neg R}$ is legit by Lemma 8.8, so is $\mathrm{Mn}[\chi_{\neg R}]$. □

Lemma 8.10

The legit relations are closed under bounded quantification: if $R(x_1, \ldots, x_n, z)$ is a legit $n+1$-ary relation, then so are the $n+1$-ary relations $\forall z \leq y\, R(x_1, \ldots, x_n, z)$ and $\exists z \leq y\, R(x_1, \ldots, x_n, z)$.

Proof. Let $R(x_1, \ldots, x_n, z)$ be a legit relation. For readability, I assume $n = 1$. Let $d(x, y) = \mu z[\neg R(x, z) \vee y = z]$. By Lemma 8.8, μz is here applied to a legit relation, and the final disjunct ensures that the relation is regular. So d is legit by Lemma 8.9. If $d(x,y) < y$ then $\neg R(x, d(x,y))$; if $d(x,y) = y$ then $\forall z \leq y\, R(x,z)$. So $\forall z \leq y\, R(x,z)$ holds iff $d(x,y) = y$. (I.e., $\chi_{\forall z \leq y\, R(x,z)}(x,y) = \chi_{=}(d(x,y),y)$.)

Since $\exists z \leq y\, R(x,z)$ is equivalent to $\neg \forall z \leq y\, \neg R(x,z)$, it is legit by Lemma 8.8. □

Exercise 8.11 We know from the proof of Lemma 8.6 that addition is representable in every theory that satisfies R5. Use Lemmas 8.7 and Lemma 8.10 to infer that $\leq$ is representable in any theory that satisfies R1–R5, by defining $\leq$ in terms of addition and bounded quantification.

Lemma 8.11: Beta Function Lemma

There is a function beta such that for any finite sequence $\langle a_1, \ldots, a_n \rangle$ of natural numbers, there is a number c such that $\text{beta}(c,i) = a_i$ for all $1 \leq i \leq n$. Moreover, beta is representable in any theory that satisfies R1–R6.

Proof. We use the coding method described above: given a sequence $\langle a_1, \ldots, a_n \rangle$, let p be the smallest prime that's at least 2 greater than all of $a_1, \ldots, a_n$ and n; let q be the base-p numeral built as $(p-1) \overset{p}{\frown} 1 \overset{p}{\frown} a_1 \overset{p}{\frown} \ldots \overset{p}{\frown} (p-1) \overset{p}{\frown} n \overset{p}{\frown} a_n$; let $c = \text{j}(p,q)$. I'll now show how we can construct a function $\text{beta}(c,i)$ that retrieves the i-th element from the sequence coded by c:

$$
\begin{aligned}
x < y &\Leftrightarrow \exists z \leq x (s(z) = y). \\
\text{Divides}(x,y) &\Leftrightarrow \exists z \leq y\, (y = x \times z). \\
\text{Prime}(x) &\Leftrightarrow x \neq 0 \wedge x \neq 1 \wedge \forall y \leq x\, (\text{Divides}(y,x) \rightarrow y = 1 \vee y = x). \\
\text{Pow}(x,p) &\Leftrightarrow x \neq 0 \wedge \text{Prime}(p) \wedge \forall y \leq x\, (\text{Divides}(y,x) \rightarrow y = 1 \vee \text{Divides}(p,y)) \\
&\qquad \text{(in words: } x \text{ is a power of the prime } p\text{).} \\
\eta(p,x) &= \mu y\, ((\text{Pow}(y,p) \wedge x < y \wedge 1 < y) \vee (\neg\text{Prime}(p) \wedge y = 0)) \\
&\qquad \text{(the smallest power of prime } p \text{ greater than } x\text{).} \\
x \overset{p}{\frown} y &= x \times \eta(p,y) + y \\
&\qquad \text{(the base-}p\text{ numeral of } y \text{ appended to that of } x\text{).} \\
\text{Part}(x,y,p) &\Leftrightarrow \exists v \leq y\, \exists w \leq y\, (v \overset{p}{\frown} x \overset{p}{\frown} w = y \vee v \overset{p}{\frown} x = y \vee x \overset{p}{\frown} v = y \vee x = y) \\
&\qquad \text{(the base-}p\text{ numeral of } x \text{ is part of the base-}p\text{ numeral of } y\text{).}
\end{aligned}
$$

$$\begin{aligned}
x \dot{-} y &= \mu z\,((y{<}x \to y{+}z{=}x) \;\land\; (\neg(y{<}x) \to z{=}0)) \\
&\qquad \text{(truncated subtraction, as in Section 7.1).} \\
\mathrm{alpha}(p,q,i) &= \mu x\,(\mathrm{Part}((p \dot{-} 1)\overset{p}{\frown} i \overset{p}{\frown} x, q, p) \lor x{=}q) \\
&\qquad \text{(the } x \text{ for which } (p-1)\overset{p}{\frown} i \overset{p}{\frown} x \text{ is part of the base-}p\text{ numeral of } q\text{).} \\
\mathrm{J}(x,y,q) &\Leftrightarrow 2 \times q = (x+y) \times (x+y+1) + 2 \times y. \\
\mathrm{r}(q) &= \mu y\, \exists x{\le}q(\mathrm{J}(x,y,q)). \\
\mathrm{l}(q) &= \mu x\, \exists y{\le}q(\mathrm{J}(x,y,q)). \\
\mathrm{beta}(c,i) &= \mathrm{alpha}(\mathrm{l}(c),\mathrm{r}(c),i).
\end{aligned}$$

To be clear: this chain of definitions doesn't directly define the relevant functions and relations in $\mathfrak{L}_A$. The expressions on the right-hand side aren't $\mathfrak{L}_A$-formulas; they are metalinguistic descriptions of certain functions and relations. The chain of definitions merely shows how beta can be constructed from addition, multiplication, projection, and $\chi_=$ by composition, truth-functional combination, bounded quantification, and regular minimization, By Lemmas 8.8, 8.10, and 8.9, it follows that beta is legit. By Lemma 8.7, beta is representable in any theory T that satisfies R1–R6. □

Now we can show that representability is closed under primitive recursion.

Lemma 8.12

If f and g are representable in a theory T that satisfies R1–R6, and $h = \mathrm{Pr}[f,g]$, then h is also representable in T.

Proof. Assume $h = Pr[f,g]$. For readability, I assume that f is one-place. Let $F(x,y)$, $G(x,y,z,w)$, and $\mathrm{BETA}(x,y,z)$ be $\mathfrak{L}_A$-formulas that represent f, g, and beta in T, respectively. Let $\mathrm{SEQ}(c,x,k)$ be the formula

$$\begin{aligned}
&\exists u(\mathrm{BETA}(c,1,u) \land F(x,u)) \land \\
&\qquad\qquad \forall i(i<k \to \exists t \exists u(\mathrm{BETA}(c,s(i),t) \land \mathrm{BETA}(c,s(s(i)),u) \land G(x,i,t,u))).
\end{aligned}$$

This represents (in T) the relation that holds of c,x,k iff c codes $\langle h(x,0),\dots,h(x,k)\rangle$. Let $H(x,k,y)$ be

$$\exists c(\mathrm{SEQ}(c,x,k) \land \mathrm{BETA}(c,s(k),y)).$$

This represents the relation that holds of x,k,y iff y is the last element of $\langle h(x,0),\dots,h(x,k)\rangle$.

It therefore represents h in T. □

Exercise 8.12 Show that all legit functions are recursive. If you are ambitious, you may also want to show the converse: that all recursive functions are legit. (Hint: if $h = \mathrm{Pr}[f, g]$ then h can be constructed from f, g, and beta, roughly as in the proof of Lemma 8.12.)

8.5 Summing up

The lemmas from the previous two sections combine to give us our main result:

Theorem 8.1

All recursive functions are representable in any theory that satisfies R1–R6.

Proof. Let T be any theory that satisfies R1–R6. By Lemmas 8.1, 8.2, and 8.3, the zero function, successor function, and projection functions are representable in T. By Lemmas 8.4, 8.5, and 8.12, any function constructed by composition, regular minimization, or primitive recursion from functions that are representable in T is itself representable in T. Since all recursive functions can be constructed from zero, successor, and projection by these operations, it follows that all recursive functions are representable in T. □

Theorem 8.2

All recursive relations are representable in any theory that satisfies R1–R6.

Proof. Immediate from Theorem 8.1, Proposition 8.2, and the fact that any theory T that satisfies R4 can prove $0 \neq \overline{1}$. □

The conditions R1–R6 can be seen as defining a minimal arithmetical theory in which all recursive functions and relations are representable. This theory is not especially elegant. More elegant is the theory Q (or "Robinson Arithmetic") that we met in Section 4.1. As a reminder, here are the axioms of Q:

Q1 $\forall x \forall y\, (s(x) = s(y) \rightarrow x = y)$

Q2 $\forall x\, 0 \neq s(x)$

Q3 $\forall x\, (x \neq 0 \rightarrow \exists y\, x = s(y))$

Q4 $\forall x(x + 0 = x)$

Q5 $\forall x \forall y(x + s(y) = s(x + y))$

Q6 $\forall x(x \times 0 = 0)$

Q7 $\forall x \forall y(x \times s(y) = (x \times y) + x)$

The following proof shows that Q, and therefore every extension of Q, satisfies R1–R6. An *extension* of Q is a theory that is at least as strong as Q, in the sense that it contains all sentences in Q.

Theorem 8.3

All recursive functions and relations are representable in every extension of Q.

Proof. By Theorems 8.1 and 8.2, it suffices to show that Q (and therefore any extension of Q) satisfies R1–R6.

R1. We show by induction on a that $\vdash_Q \forall x(\overline{a} < x \;\vee\; x = \overline{a} \;\vee\; x < \overline{a})$. *Base:* $a = 0$. (I now reason "inside Q".) By Q3, for all x either $x = 0$ or $\exists y\, x = s(y)$. In the second case, $\exists y(s(y) + 0 = x)$ by Q4, and so $0 < x$. So $\forall x(x = 0 \;\vee\; 0 < x)$. *Induction step:* Let x be any number. We show that $s(\overline{a}) < x \;\vee\; x = s(\overline{a}) \;\vee\; x < s(\overline{a})$. By Q3, either $x = 0$ or $\exists y\, x = s(y)$. If $x = 0$ then $x < s(\overline{a})$ because $s(\overline{a}) + 0 = s(\overline{a})$ by Q4 and hence $\exists z(s(z) + 0 = s(a))$. Assume $\exists y\, x = s(y)$. By induction hypothesis, $\overline{a} < y \;\vee\; y = \overline{a} \;\vee\; y < \overline{a}$. If $\overline{a} < y$ then $\exists z(s(z) + \overline{a} = y)$ and $s(z) + s(\overline{a}) = s(y)$ by Q5; so $s(\overline{a}) < x$. If $y = \overline{a}$ then $x = s(\overline{a})$. If $y < \overline{a}$ then $\exists z(s(z) + y = \overline{a})$ and $s(z) + s(y) = s(\overline{a})$ by Q5; so $x < s(\overline{a})$.

R2. We show that Q contains $\neg\exists x(x < 0)$. Fix any x, z. By Q3, either $x = 0$ or $\exists y\, x = s(y)$. If $x = 0$, $s(z) + x = s(z) + 0 = s(z)$ by Q4, hence $s(z) + x \neq 0$ by Q2. If, alternatively, $\exists y\, x = s(y)$, then $s(z) + x = s(z) + s(y) = s(s(z) + y)$ by Q5, hence $s(z) + x \neq 0$ by Q2. So Q2–Q5 entail $\forall x \neg\exists z(s(z) + x = 0)$, which is equivalent to $\neg\exists x(x < 0)$.

R3. We show by induction on a that whenever $a > 0$ then $\vdash_Q \forall x(x < \overline{a} \rightarrow (x = 0 \;\vee\; \ldots \;\vee\; x = \overline{a - 1}))$. *Base:* $a = 1$. We show that $\vdash_Q \forall x(x < \overline{1} \rightarrow x = 0)$. Assume $x < \overline{1}$; i.e. $\exists z(s(z) + x = s(0))$. Suppose for reductio that $x \neq 0$. By Q3, $\exists y\, x = s(y)$; so

$s(z) + s(y) = s(0)$; so $s(s(z) + y) = s(0)$ by Q5, and $s(z) + y = 0$ by Q1; if $y = 0$ then $s(z) + 0 = s(z) = 0$ contradicting Q2; if $y = s(w)$ then $s(z) + s(w) = s(s(z) + w) = 0$, again contradicting Q2. *Induction step:* Assume $x < s(\overline{a})$, i.e. $\exists z(s(z) + x = s(\overline{a}))$. By Q3, either $x = 0$ or $\exists y\, x = s(y)$. In the second case, $s(z) + s(y) = s(\overline{a})$, so $s(s(z) + y) = s(\overline{a})$ by Q5, and $s(z) + y = \overline{a}$ by Q1; so $y < \overline{a}$. By induction hypothesis, $y = 0 \lor \dots \lor y = \overline{a-1}$, so $x = s(y)$ is one of $1, \dots, \overline{a}$. Combining both cases, we have $x = 0 \lor \dots \lor x = \overline{a}$.

R4. We show that if $a \neq b$, then $\vdash_Q \overline{a} \neq \overline{b}$. Assume $a < b$. We show by induction on a that $\vdash_Q \overline{a} \neq \overline{b}$. *Base:* $a = 0$. Then $\overline{b} = \overline{s(b-1)})$ and hence $0 \neq \overline{b}$ by Q2. *Induction step:* $a = s(c)$. Then $b = s(d)$ for some d with $c < d$. By induction hypothesis, $\vdash_Q \overline{c} \neq \overline{d}$. So $\vdash_Q s(\overline{c}) \neq s(\overline{d})$ by Q1. The case for $b < a$ is analogous.

R5. We show by induction on b that for all a, b, $\vdash_Q \overline{a} + \overline{b} = \overline{a+b}$. *Base*: $b = 0$. Then $\overline{a} + \overline{b} = \overline{a+b}$ by Q4. *Induction step:* $b = s(c)$. By induction hypothesis, $\overline{a} + \overline{c} = \overline{a+c}$. By Q5, $\overline{a} + s(\overline{c}) = s(\overline{a} + \overline{c}) = s(\overline{a+c}) = \overline{a + s(c)} = \overline{a+b}$.

R6. We show by induction on b that for all a, b, $\vdash_Q \overline{a} \times \overline{b} = \overline{a \times b}$. *Base*: $b = 0$. Then $\overline{a} \times \overline{b} = \overline{a \times b}$ by Q6. *Induction step:* $b = s(c)$. Then $\overline{a} \times s(\overline{c}) = \overline{a} \times \overline{c} + \overline{a}$ by Q7, $= \overline{a \times c} + \overline{a}$ by induction hypothesis, $= \overline{a \times c + a}$ by R5, $= \overline{a \times b}$. □

By Proposition 4.1, Peano Arithmetic (PA) is an extension of Q. So all recursive functions and relations are representable in PA.

We can also prove a result that goes in the other direction:

Theorem 8.4

Every relation that is representable in an axiomatizable and consistent $\mathfrak{L}_A$-theory is recursive.

Proof. Assume $A(x_1, \dots, x_n)$ represents R in an axiomatizable and consistent theory T. That is, if R holds of some numbers $a_1, \dots, a_n$, then T can prove $A(\overline{a_1}, \dots, \overline{a_n})$, and if R does not hold of $a_1, \dots, a_n$, then T can prove $\neg A(\overline{a_1}, \dots, \overline{a_n})$. Since T is consistent, it never proves both $A(\overline{a_1}, \dots, \overline{a_n})$ and $\neg A(\overline{a_1}, \dots, \overline{a_n})$. By Proposition 5.4, every axiomatizable first-order theory is computably enumerable: we can define an algorithm that lists all sentences provable in T. This gives us an algorithm for deciding R: to check whether R holds of some numbers $a_1, \dots, a_n$, we wait until either $A(\overline{a_1}, \dots, \overline{a_n})$ or $\neg A(\overline{a_1}, \dots, \overline{a_n})$ appears on the list of sentences provable in T. By the Church-Turing thesis, it follows that R is recursive. □

Finally, let's return to definability. As I announced in Section 8.2, our result about representability shows that all recursive functions and relations can be defined in $\mathfrak{L}_A$:

Theorem 8.5

All recursive functions and relations are definable in $\mathfrak{L}_A$.

Proof. By Theorem 8.3, all recursive functions and relations are representable in Q. All axioms of Q are true in the standard model of arithmetic $\mathfrak{A}$. So $\mathrm{Q} \subseteq \mathrm{Th}(\mathfrak{A})$. By Proposition 8.1, every formula that represents a function or relation in Q therefore also defines that function or relation in $\mathfrak{L}_A$ □

Exercise 8.13 Are all recursive functions and relations representable in $\mathrm{Th}(\mathfrak{A})$?

Exercise 8.14 Using the Church-Turing theses, explain why all computably enumerable relations are definable in $\mathfrak{L}_A$. (Hint: use Proposition 5.3.)

Exercise 8.15 Are all computably enumerable relations representable in Q?

Exercise 8.16 Say that a relation R is *weakly represented* by an $\mathfrak{L}_A$-formula $A(x_1, \dots, x_n)$ in a theory T iff for all numbers $a_1, \dots, a_n$,

$$R(a_1, \dots, a_n) \text{ iff } \vdash_T A(\overline{a_1}, \dots, \overline{a_n}).$$

Explain the following facts:

(a) A relation can be weakly representable in a theory without being representable in that theory.

(b) If R is weakly representable in an axiomatizable and consistent theory then R is computably enumerable. (Compare Theorem 8.4.)

(c) All recursive relations are weakly representable in any $\mathfrak{L}_A$-theory that is consistent with Q. (Hint: We know that every recursive relation R is represented in Q by some formula A. Let $\hat{Q}$ be the conjunction of the seven axioms of Q, and consider the formula $\hat{Q} \rightarrow A$.)

9 Incompleteness

In this chapter, we're going to prove several versions of Gödel's First Incompleteness Theorem. We're also going to prove Tarski's Theorem on the undefinability of truth, and Church's Theorem on the undecidability of first-order logic.

9.1 Preview

In Chapter 4, we studied axiomatic theories. The aim of axiomatizing an area of mathematics (or other discipline) is to put it on a firm foundation: instead of relying on a hodgepodge of intuition and imperfectly understood techniques, all results in the area should be derivable by pure logic from a set of precise and explicitly stated assumptions: the axioms.

What should we expect of an axiomatic theory? Obviously, all the axioms should be true on their intended interpretation. Ideally, they should suffice to derive *all* truths in the relevant area. These are semantic properties of theories, related to their intended interpretation. But they entail syntactic properties. If all axioms of a theory are true, the theory must be *consistent*: it won't contain a sentence A and its negation $\neg A$; equivalently, it won't contain $\bot$. If a theory contains all truths on its intended interpretation, it will be *complete* in the sense that for every sentence A in its language, it contains either A or $\neg A$.

Note that this is not the sense of 'complete' in which the first-order calculus is complete. When we talk about completeness *of a proof system*, we mean that the system can prove every valid sentence. When we talk about completeness *of a theory*, we mean that the theory decides every sentence: it contains A or $\neg A$, for every sentence A. (This notion of completeness is sometimes called *negation-completeness*.)

Confusingly, 'sound' also has two meanings. A proof system is sound if it can only prove valid sentences. A theory is called *sound* if all sentences in the theory are true on their intended interpretation.

Consistency and completeness are defined syntactically, without reference to the meaning of the axioms. This makes them easier to study formally than semantic properties like soundness, which requires pinning down the intended interpretation independently of the proposed axioms, so that one can compare what the axioms say with the structure they are meant to describe.

Consistency may seem trivial: surely nobody would propose an inconsistent set of axioms? But remember that this is exactly what happened to Frege. It also happened to others, especially when trying to develop powerful systems to unify diverse areas of mathematics. Many mathematicians were therefore wary of ZFC when it was first proposed. Couldn't it also turn out to be inconsistent?

David Hilbert saw how such fears could be put to rest. To check whether an axiomatic theory is consistent, we only need to check whether there is deduction of $\bot$ from its axioms. Even if the theory itself, like ZFC, talks about highly infinitary matters, any deduction from its axioms is finite. We should therefore be able to establish the consistency of ZFC in a much weaker, finitary branch of mathematics that doesn't study sets, but proofs and deductions. In the same way, Hilbert hoped that we could prove the completeness of axiomatic theories: we should be able to verify (as seemed plausible in the 1920s) that the axioms of, say, Peano Arithmetic or ZFC decide every sentence in their language.

This project for establishing the consistency and completeness of axiomatic theories is known as *Hilbert's program*. It was shattered by Gödel's incompleteness theorems. Gödel showed that sufficiently strong axiomatic theories can never be complete, unless they are inconsistent. He also showed that there is no hope of establishing the consistency of sufficiently strong theories from safe, finitary grounds. In the present chapter, we'll focus on the first of these results. We'll turn to the second in Chapter 10.

Gödel realized that we don't need a separate branch of mathematics to study proofs. Since sentences and deductions are finite strings of symbols, they can be coded as numbers. We can therefore use arithmetical theories to reason indirectly about proofs. We can, for example, construct an $\mathfrak{L}_A$-formula $\text{PROV}_{\text{PA}}(x)$ so that $\text{PROV}_{\text{PA}}(\overline{n})$ is true (in the standard model of arithmetic $\mathfrak{A}$) iff there is a deduction of the sentence coded by n from the axioms of Peano Arithmetic.

Gödel then showed how to construct a sentence G, coded by some number n, such that $G \leftrightarrow \neg\text{PROV}_{\text{PA}}(\overline{n})$ is true in $\mathfrak{A}$. Informally, G says of itself that it is not provable (in PA): it is true iff it is unprovable. It swiftly follows that PA can't decide G, assuming that PA is sound. To see this, note first that G is true: if it were is false, it would be provable (because G is true iff it is unprovable), and so PA would prove a falsehood, contradicting the assumption that PA is sound. So G is true. And so G can't be proved in PA (because G is true iff it is unprovable). Its negation $\neg G$ can't be proved either, as otherwise PA would prove a falsehood.

This argument assumes that PA is sound. For that reason, it is known as the “semantic” version of the First Incompleteness Theorem. Gödel’s main result is a “syntactic” version of the theorem that doesn’t require soundness. In its standard formulation, it shows that every consistent axiomatic theory that is at least as strong as Q is incomplete.

The restriction to *axiomatic* theories is crucial. Consider the theory $\mathrm{Th}(\mathfrak{A})$ consisting of all $\mathfrak{L}_A$-sentences that are true in the standard model of arithmetic $\mathfrak{A}$. This theory is complete: every $\mathfrak{L}_A$-sentence is either true or false in $\mathfrak{A}$; if true, it is in $\mathrm{Th}(\mathfrak{A})$; if false, its negation is in $\mathrm{Th}(\mathfrak{A})$. By definition, $\mathrm{Th}(\mathfrak{A})$ is also sound, and therefore consistent. But it is not an axiomatic theory: I haven’t specified $\mathrm{Th}(\mathfrak{A})$ by giving a set of axioms, and the Incompleteness Theorem implies that I couldn’t have done so.

Officially, theories are just sets of sentences closed under first-order consequence. The same set of sentences can always be specified in many ways. Instead of speaking of *axiomatic* theories, we should therefore speak of *axiomatizable* theories. Recall that a theory is axiomatizable if there is a decidable set of axioms from which all and only the sentences in the theory can be deduced: a set of axioms for which there is a mechanical algorithm to check whether a given sentence is in it or not.

In Section 5.4, I introduced these concepts informally, using the pre-theoretical notions of computability and algorithms. We can now give more formal definitions.

We say that a set of numbers is *recursively decidable* if the property of being in the set is a recursive relation, as defined in Chapter 7; a set of sentences is recursively decidable if the set of their code numbers is recursively decidable. A theory is *recursively axiomatizable* if there is a recursively decidable set of axioms from which all and only the sentences in the theory can be deduced. By Theorem 7.4, this is equivalent to saying that there is a Turing machine that determines whether a given $\mathfrak{L}_A$-sentence is among the axioms or not.

The syntactic version of Gödel’s First Incompleteness Theorem can now be stated as follows: *every consistent and recursively axiomatizable theory of arithmetic that is at least as strong as Q is incomplete.*

Exercise 9.1 Let T_1 be the set of all $\mathfrak{L}_A$-sentences. Is T_1 (a) a theory? (b) recursively axiomatizable? (c) complete? (d) consistent? (Explain.)

Exercise 9.2 Let T_2 be the set of $\mathfrak{L}_A$-sentences that are valid in first-order logic. Is T_2 (a) a theory? (b) recursively axiomatizable? (c) complete? (d) consistent?

Exercise 9.3 Can you find an $\mathfrak{L}_A$-theory that is recursively axiomatizable, complete, and consistent? (Hint: you only need one simple axiom.)

9.2 Arithmetization of syntax

As I mentioned above, Gödel's proof draws on the insight that we can use arithmetical theories like PA to reason about their own syntax. After the work we've done in the previous chapter, this should not be surprising. We've shown in Theorem 8.5 that every computable property or relation is definable in $\mathfrak{L}_A$. Syntactic properties like *coding an $\mathfrak{L}_A$-sentence* or *coding a deduction from the axioms of PA* are clearly computable; so they are definable in $\mathfrak{L}_A$: there is an $\mathfrak{L}_A$-formula $\textsc{prf}_{\text{PA}}(x, y)$ such that $\textsc{prf}_{\text{PA}}(\overline{n}, \overline{m})$ is true (in $\mathfrak{A}$) iff n codes a proof of the sentence coded by m from the axioms of PA. This is all we need to run Gödel's argument. To fix ideas, I'll nonetheless fill in some more details.

We want to talk about sentences and deductions in the language $\mathfrak{L}_A$, whose non-logical symbols are 0, s, +, and ×. To this end, we code $\mathfrak{L}_A$-strings as numbers, so that we can indirectly refer an $\mathfrak{L}_A$-string by the $\mathfrak{L}_A$-numeral of its code. We'll use Gödel's own coding scheme, which I introduced in Section 5.5.

We first assign a *symbol code* to each primitive symbol of $\mathfrak{L}_A$, like so:

Symbol:	0	s	+	×	=	¬	→	∀	(	)	,	x_1	c_1	x_2	c_2	…
Code:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…

Then we use the prime exponent method to code sequences of symbol codes, and thereby $\mathfrak{L}_A$-strings. The string '0 = 0', for example, determines the sequence of symbol codes $\langle 1, 5, 1 \rangle$, which is coded as $2^1 \cdot 3^5 \cdot 5^1 = 2430$. The exponents of the primes are the symbol codes. In general, if p_i is the ith prime number then the code number of an $\mathfrak{L}_A$-string A composed of symbols $s_1 s_2 \ldots s_n$ with symbol codes $c_1, c_2, \ldots, c_n$ is

$$\#[A] = p_1^{c_1} \cdot p_2^{c_2} \cdot \ldots \cdot p_n^{c_n}.$$

From now on, we'll call $\#[A]$ the *Gödel number* of A. Note that individual symbols of $\mathfrak{L}_A$ have both a Gödel number and a symbol code: '→' has symbol code 7 and Gödel number $2^7 = 128$. We won't talk about symbol codes any more.

Since deductions are finite sequences of $\mathfrak{L}_A$-sentences, we can use the prime exponent method again to code them: the Gödel number of a deduction $A_1, A_2, \dots, A_n$ is

$$\#[A_1, A_2, \dots, A_n] = p_1^{\#[A_1]} \cdot p_2^{\#[A_2]} \cdot \dots \cdot p_n^{\#[A_n]}.$$

The Gödel number function # converts any $\mathfrak{L}_A$-string A into a number $\#[A]$. This number $\#[A]$ is denoted in $\mathfrak{L}_A$ by some numeral $\overline{\#[A]}$. So we can indirectly refer to any $\mathfrak{L}_A$-string A by the numeral $\overline{\#[A]}$ of its Gödel number.

We'll abbreviate $\overline{\#[A]}$ as $\ulcorner A \urcorner$. For example, since $\#[0 = 0] = 2430$, $\ulcorner 0 = 0 \urcorner$ is $\overline{2430}$, which is $s(s(\dots s(0) \dots))$ with 2430 occurrences of s. In practice, you should treat the corner quotes as a special kind of quote marks: we use $\ulcorner 0 = 0 \urcorner$ to denote the string '$0 = 0$' via its Gödel number.

Exercise 9.4 What are (a) $\#[0]$? (b) $\overline{\#[0]}$? (c) $\ulcorner 0 \urcorner$? (d) $\#[\ulcorner 0 \urcorner]$?

Now consider a simple syntactic property: being a variable. In our coding scheme, variables have Gödel numbers $2^{12}, 2^{14}, 2^{16}, \dots$. That is, a number n codes a variable iff $n = 2^{12+2y}$, for some y. This is a purely arithmetical property that can be expressed in $\mathfrak{L}_A$: there is an $\mathfrak{L}_A$-formula $\text{VAR}(x)$ such that $\text{VAR}(\overline{n})$ is true (in $\mathfrak{A}$) iff n codes a variable. In this sense, $\text{VAR}(\overline{n})$ "says that" n codes a variable. But what it actually, explicitly says is simply that there is a number y such that $n = 2^{12+2y}$.

In the terminology of the previous chapter, the formula $\text{VAR}(x)$ *defines* the property of coding a variable. By Theorem 8.5, every recursive relation and function is definable in L_A. We can use this result to show that a wide range of syntactic notions are definable in L_A. Since our coding scheme can be implemented mechanically, it maps every computable relation or function on $\mathfrak{L}_A$-strings to a computable relation or function on $\mathbb{N}$. By the Church-Turing Thesis, that relation or function is recursive. By Theorem 8.5, it is definable in L_A.

For example, there is a mechanical procedure for checking whether a given string is a well-formed sentence of $\mathfrak{L}_A$. So there is also a mechanical procedure for checking whether a given number is the Gödel number of an $\mathfrak{L}_A$-sentence. By the Church-Turing Thesis, it follows that the property (call it Sent) of coding an $\mathfrak{L}_A$-sentence is recursive. By Theorem 8.5, it follows that there is an $\mathfrak{L}_A$-formula $\text{SENT}(x)$ such that $\text{SENT}(\overline{n})$ is true (in $\mathfrak{A}$) iff n is the Gödel number of an $\mathfrak{L}_A$-sentence.

Similarly, if a theory T is axiomatized by a decidable set of axioms then there is a mechanical procedure for checking whether a given sequence of $\mathfrak{L}_A$-sentences is a deduction of a given target sentence from these axioms: we only need to check whether the last sentence in the sequence is the target sentence, and whether each sentence in the sequence is either an axiom of T, an instance of the logical axioms A1–A6, or follows from previous sentences by MP or Gen. All these checks can be performed mechanically. Let Prf_T be the relation that holds between numbers n and m iff n codes a deduction (informally, a "proof") of the sentence coded by m from a set of axioms that generates T. If T is computably axiomatizable, Prf_T is computable. By the Church-Turing Thesis, it is recursive. By Theorem 8.5, it follows that it is definable in L_A: there is an $\mathfrak{L}_A$-formula $\text{PRF}_T(x, y)$ such that $\text{PRF}_T(\overline{n}, \overline{m})$ is true (in $\mathfrak{A}$) iff n codes a proof of the sentence coded by m from the axioms of T.

For a final example, let's look at a function on $\mathfrak{L}_A$-strings. Consider the *concatenation* function that takes two $\mathfrak{L}_A$-strings and returns the string consisting of the first followed by the second. This is clearly computable. By the Church-Turing Thesis, the corresponding function on Gödel numbers is recursive: there is a recursive function $*$ that maps the Gödel numbers of any two $\mathfrak{L}_A$-strings to the Gödel number of the concatenation of these strings. By Theorem 8.5, it follows that there is an $\mathfrak{L}_A$-formula $\text{CONCAT}(x, y, z)$ that defines $*$ in L_A, so that $\text{CONCAT}(\overline{n}, \overline{m}, \overline{k})$ is true (in $\mathfrak{A}$) iff k codes the concatenation of the strings coded by n and m. (I'll write this as $k = n * m$, rather than $k = *(n, m)$).

Exercise 9.5 What is $\#[0 =] * \#[0]$?

I've appealed to the Church-Turing Thesis to argue that Sent, Prf_T and $*$ are recursive, but we could have shown this directly: we could show that Sent, Prf_T and $*$ can be constructed from zero, successor, and projection by composition, primitive recursion, and regular minimization. In fact, we don't need minimization: Sent, Prf_T and $*$ are primitive recursive. I won't go through the details for each case. But let me illustrate what's involved with the concatenation function $*$ (which will play an important role in the next section).

Recall that $*$ maps two Gödel numbers $\#[A]$ and $\#[B]$ to the Gödel number $\#[AB]$ of the concatenation of A and B. If B is a single symbol, it is easy to define this operation arithmetically:

$$\#[A] * \#[s] = \#[A] \cdot \text{pri}(\text{len}(\#[A]))^{\#[s]},$$

where $\text{pri}(i)$ is the ith prime number and $\text{len}(n)$ is the length of the string coded by n. In Section 7.2, I showed that pri and len are primitive recursive. So the function

$$\text{append}(x, y) = x \cdot \text{pri}(\text{len}(x))^y$$

is also primitive recursive.

Next, we need the function entry that takes two numbers n and i and returns the exponent of the ith prime in the prime factorization of n. I showed in Section 7.2 that this function, too, is primitive recursive. Using append and entry, we define (by primitive recursion) a function conc that takes three numbers n, m, and i and returns the code of the string consisting of the string coded by n followed by the first i symbols of the string coded by m:

$$\begin{aligned} \mathrm{conc}(x, y, 0) &= x \\ \mathrm{conc}(x, y, s(i)) &= \mathrm{append}(\mathrm{conc}(x, y, i), \mathrm{entry}(y, s(i))). \end{aligned}$$

From this, we can define $x * y$ as $\mathrm{conc}(x, y, \mathrm{len}(y))$.

9.3 The First Incompleteness Theorem

I'll now explain how Gödel managed to construct a sentence that is true iff it is unprovable. The construction is so perplexing that it may help to first give a version for English. I'll show how to construct an English sentence that is true iff it is unprovable. (Let's pretend we've specified what it means for an English sentence to be "provable". You'll see that nothing hangs on this.)

In English, we can use quote marks to denote expressions of English. For example,

> 'is English'

is a noun that denotes an English predicate. We can combine nouns like this with predicates to form sentences:

(1) 'is English' is English.
(2) 'is made of stone' is made of stone.
(3) 'is made of stone' is English.

In (1) and (2), a predicate is applied to itself, using quote marks. Let's call a sentence that results by applying a predicate to itself in this manner the *diagonalization* of that predicate. So (1) is the diagonalization of 'is English'.

Now consider the predicate 'has a diagonalization that is not provable'. If we diagonalize *this* predicate, we get

(4) 'has a diagonalization that is not provable' has a diagonalization that is not provable.

This is a sentence. What does it say? Well, it says that the predicate it quotes has an unprovable diagonalization. Every predicate has a unique diagonalization. So (4) says that the diagonalization of the quoted predicate ('has a diagonalization that is not provable') is not provable. But (4) *is* the diagonalization of that predicate. So (4) says of itself that it is not provable.

This trick obviously generalizes. We can replace 'is not provable' by any predicate A. The argument shows that for any English predicate A, there is a sentence G that says of itself that it is A. To construct G, we first construct another predicate F: 'has a diagonalization that is A'. G is then the diagonalization of F. It is true iff it is A.

We'll now run this argument for $\mathfrak{L}_A$. We use open formulas $A(x)$ as predicates, and Gödel numerals $\ulcorner A(x) \urcorner$ (instead of quote marks) to refer to these predicates. For example, if $\text{VAR}(x)$ is the formula that defines the property of coding a variable, then $\text{VAR}(\ulcorner \text{VAR}(x) \urcorner)$ is a sentence saying (falsely) that the code of $\text{VAR}(x)$ codes a variable – equivalently: that $\text{VAR}(x)$ is a variable. We might call $\text{VAR}(\ulcorner \text{VAR}(x) \urcorner)$ the diagonalization of $\text{VAR}(x)$. However, it proves convenient to use a slightly more roundabout definition.

For any $\mathfrak{L}_A$-formula A, we define the *diagonalization* of A as the formula

$$\exists x (x = \ulcorner A \urcorner \land A).$$

If x is free in A, which is the only case we care about, this is logically equivalent to $A(\ulcorner A(x) \urcorner)$.

With this definition, constructing the diagonalization of a formula is a trivial mechanical task. Let diag be the function that takes the Gödel number of a formula as input and returns the Gödel number of the formula's diagonalization. This function is recursive. In fact, it is primitive recursive, and easily definable with the concatenation function $*$:

$$\text{diag}(y) = \#[\exists x (x = \overline{y} \land] * y * \#[)].$$

By Theorem 8.5, all recursive functions are definable in $\mathfrak{L}_A$. So there is a formula $\text{DIAG}(x, y)$ such that $\text{DIAG}(\overline{n}, \overline{m})$ is true (in $\mathfrak{A}$) iff m codes the diagonalization of the formula coded by n. We use this formula to construct, for any formula $A(x)$ a sentence that "says of itself" that it has the property expressed by $A(x)$.

Lemma 9.1: The Semantic Diagonal Lemma

For every $\mathfrak{L}_A$-formula $A(x)$ there is a sentence G such that $\mathfrak{A} \Vdash G$ iff $\mathfrak{A} \Vdash A(\ulcorner G \urcorner)$.

Proof. Let $F(x)$ be the formula $\exists y(\text{DIAG}(x,y) \land A(y))$. Let G be the diagonalization of $F(x)$. So G is $\exists x(x = \ulcorner F(x)\urcorner \land F(x))$. This is logically equivalent to $F(\ulcorner F(x)\urcorner)$, which expands to $\exists y(\text{DIAG}(\ulcorner F(x)\urcorner, y) \land A(y))$. Since DIAG defines diag, G is true in $\mathfrak{A}$ iff there is a number n that codes the diagonalization of $F(x)$ and for which $A(\overline{n})$ is true (in $\mathfrak{A}$). The diagonalization of $F(x)$ is G. So G is true in $\mathfrak{A}$ iff $A(\overline{n})$ is true (in $\mathfrak{A}$) of the number n that codes G. In short G is true in $\mathfrak{A}$ iff $A(\ulcorner G\urcorner)$ is true in $\mathfrak{A}$. (If this proof baffles you, have another look at the English version above!) □

Now we're ready to prove the semantic version of Gödel's First Incompleteness Theorem. Let T be some recursively axiomatizable theory in $\mathfrak{L}_A$, so that there is a recursively decidable set of axioms Γ from which all and only the members of T can be deduced. I'll say that a sentence is *provable in T* if it is deducible from some such set Γ. As above, let Prf_T be the relation that holds between numbers n and m iff n codes a deduction of the sentence coded by m from Γ. As explained in the previous section, Prf_T is recursive; so there is an $\mathfrak{L}_A$-formula $\text{PRF}_T(x,y)$ such that $\mathfrak{A} \Vdash \text{PRF}_T(\overline{n},\overline{m})$ iff n codes a deduction from Γ of the sentence coded by m. Let $\text{PROV}_T(x)$ abbreviate $\exists y\, \text{PRF}_T(y,x)$. By construction, $\text{PROV}_T(\ulcorner A\urcorner)$ is true (in $\mathfrak{A}$) iff A is provable in T. So $\neg\text{PROV}_T(\ulcorner A\urcorner)$ is true iff A is unprovable in T. By diagonalising $\neg\text{PROV}_T(x)$, we get a sentence G that is true (in $\mathfrak{A}$) iff it is unprovable (in T).

Theorem 9.1: Gödel's First Incompleteness Theorem, semantic version

Every sound and recursively axiomatizable $\mathfrak{L}_A$-theory is incomplete.

Proof. Let T be a recursively axiomatizable $\mathfrak{L}_A$-theory. As I've just explained, there is then an $\mathfrak{L}_A$-formula $\text{PROV}_T(x)$ such that $\text{PROV}_T(\ulcorner A\urcorner)$ is true in $\mathfrak{A}$ iff A is provable in T. By the Semantic Diagonal Lemma (using $\neg\text{PROV}_T(x)$ for $A(x)$), there is a sentence G such that $\mathfrak{A} \Vdash G$ iff $\mathfrak{A} \Vdash \neg\text{PROV}_T(\ulcorner G\urcorner)$.

Suppose G is provable in T. Then $\mathfrak{A} \Vdash \text{PROV}_T(\ulcorner G\urcorner)$, and so $\mathfrak{A} \nVdash G$, contradicting our assumption that T is sound. So G is not provable in T. So $\mathfrak{A} \Vdash \neg\text{PROV}_T(\ulcorner G\urcorner)$, and so $\mathfrak{A} \Vdash G$. It follows that $\neg G$ isn't provable in T either, as otherwise T would prove a falsehood. □

This is a beautiful argument, although the conclusion isn't news to us: we've already derived it from the unsolvability of the Halting Problem in Section 5.5 (which, of course, wasn't known when Gödel published his result).

Exercise 9.6 Theorem 9.1 shows that there is a true sentence G that is not provable in a sound and recursively axiomatizable theory such as PA. Suppose we add G as a new axiom to PA. Is the resulting theory complete? Is it sound?

Exercise 9.7 Explain why there are infinitely many $\mathfrak{L}_A$-sentences that PA can't decide (assuming that PA is sound).

As I mentioned in Section 9.1, Gödel also proved a syntactic version of the Incompleteness Theorem that doesn't require the relevant theory to be sound (true in $\mathfrak{A}$), but merely imposes some syntactic conditions on it.

The idea is to run through the proof of Theorem 9.1 inside the theory T. Instead of relying on the equivalence of G and $\neg\text{PROV}_T(\ulcorner G \urcorner)$ in $\mathfrak{A}$, we'll use the fact that T can *prove* their equivalence: $\vdash_T G \leftrightarrow \neg\text{PROV}_T(\ulcorner G \urcorner)$. This requires a different version of the Diagonal Lemma, turning on the *representability* of diag in T, rather than on its definability. Recall that a (one-place) function f is representable in a theory T iff there is a formula $A(x,y)$ such that for all n,

(i) $\vdash_T A(\overline{n},\overline{f(n)})$, and

(ii) $\vdash_T \forall y(A(\overline{n},y) \to y = \overline{f(n)})$.

Equivalently: $\vdash_T \forall y(A(\overline{n},y) \leftrightarrow y = \overline{f(n)})$.

Lemma 9.2: The Syntactic Diagonal Lemma

If T is an $\mathfrak{L}_A$-theory in which diag is representable, then for every $\mathfrak{L}_A$-formula $A(x)$ there is a sentence G such that $\vdash_T G \leftrightarrow A(\ulcorner G \urcorner)$.

Proof. Let T be an $\mathfrak{L}_A$-theory in which diag is representable. Let $\text{DIAG}(x,y)$ be the formula that represents diag in T, and let $F(x)$ be the formula $\exists y(\text{DIAG}(x,y) \land A(y))$. Since DIAG represents diag in T, T can prove

$$\forall y\,(\text{DIAG}(\ulcorner F(x) \urcorner, y) \;\leftrightarrow\; y = \overline{\text{diag}(\ulcorner F(x) \urcorner)}). \tag{1}$$

Let G be the diagonalization of $F(x)$. So the following is logically true:

$$G \;\leftrightarrow\; \exists y(\text{DIAG}(\ulcorner F(x) \urcorner, y) \land A(y)). \tag{2}$$

From (1) and (2), first-order logic yields

$$G \leftrightarrow \exists y\,(y = \overline{\mathrm{diag}(\ulcorner F(x)\urcorner)} \land A(y)).$$

Since $\overline{\mathrm{diag}(\ulcorner F(x)\urcorner)}$ is $\ulcorner G\urcorner$, this simplifies to $G \leftrightarrow \exists y\,(y = \ulcorner G\urcorner \land A(y))$ and further to $G \leftrightarrow A(\ulcorner G\urcorner)$. □

Now assume that T is a recursively axiomatizable theory in which both diag and Prf_T are representable. As before, define $\mathrm{PROV}_T(x)$ as $\exists y\,\mathrm{PRF}_T(y, x)$. The Syntactic Diagonal Lemma gives us a sentence G (called the *Gödel sentence* for T) such that

$$\vdash_T G \leftrightarrow \neg\mathrm{PROV}_T(\ulcorner G\urcorner). \tag{D}$$

Let's go through the reasoning in the proof of Theorem 9.1 to show that T can't decide G.

One of the two directions goes through smoothly: we can show that G isn't provable in T, unless T is inconsistent. For suppose T can prove T. This means that there is a deduction of G from a suitable set of axioms for T. Since $\mathrm{PRF}_T(x, y)$ represents Prf_T in T, it follows that there is a number n (the code of the deduction) such that $\vdash_T \mathrm{PRF}_T(\overline{n}, \ulcorner G\urcorner)$. Since T is closed under first-order consequence, it follows that $\vdash_T \mathrm{PROV}_T(\ulcorner G\urcorner)$. By (D), we have $\vdash_T \neg G$, So T proves both G and $\neg G$.

The other direction is trickier. Suppose T can prove $\neg G$. By (D), T can then prove $\mathrm{PROV}_T(\ulcorner G\urcorner)$, which is short for $\exists y\,\mathrm{PRF}_T(y, \ulcorner G\urcorner)$. If T is consistent, there is no deduction of G from T's axioms. So $\mathrm{Prf}_T(n, \ulcorner G\urcorner)$ is false for every number n. Since $\mathrm{PRF}_T(x, y)$ represents Prf_T in T, it follows that $\vdash_T \neg\mathrm{PRF}_T(\overline{n}, \ulcorner G\urcorner)$ for every number n.

We now have the following situation: T proves $\exists y\,\mathrm{PRF}_T(y, \ulcorner G\urcorner)$, but also $\neg\mathrm{PRF}_T(\overline{n}, \ulcorner G\urcorner)$ for every number n. The theory says that *there is* a number of a certain kind, but also denies that any particular number 0,1,2,…is of that kind. This isn't inconsistency, but it is almost as bad. Gödel called it "ω-inconsistency": a theory is *ω-inconsistent* if there is a formula $A(x)$ such that

(i) $\vdash_T \exists x\,A(x)$, but

(i) for every number n, $\vdash_T \neg A(\overline{n})$.

A theory is *ω-consistent* if it is not ω-inconsistent.

Clearly, no sound theory can be ω-inconsistent. So ω-consistency is another purely syntactic condition (besides consistency) that is entailed by soundness.

We've established the main result of Gödel's 1931 paper:

Theorem 9.2: Gödel's First Incompleteness Theorem

Every recursively axiomatizable and ω-consistent theory in which all recursive functions are representable is incomplete.

I won't go through the details of the proof again, as we're going to prove a strictly stronger result in the next section, showing that mere consistency (as opposed to ω-consistency) is enough. We will derive this from another important result, Tarski's Theorem. But I want to mention that there is also a way to establish it directly, following Gödel's line of reasoning. The trick, due to J. Barkley Rosser, is to make a slight change to the sentence G. Instead of using a sentence that says of itself that it is unprovable, Rosser uses a sentence saying that for every proof of it, there is a shorter proof of its negation. More formally, Rosser's version of the argument uses the diagonalization R of the following formula in place of G:

$$\forall y(\mathrm{PRF}_T(y,x) \to \exists z(z<y \land \forall v(\mathrm{CONCAT}(\ulcorner\neg\urcorner,x,v) \to \mathrm{PRF}_T(z,v)))).$$

One can show that if Prf_T and diag are representable in T, T is consistent, and T knows a few facts about arithmetic, then it can prove neither R nor $\neg R$.

Exercise 9.8 Let G be the Gödel sentence for PA. We know that G is not provable in PA. How about $\mathrm{PROV}_{\mathrm{PA}}(\ulcorner G\urcorner)$? How about $\neg\mathrm{PROV}_{\mathrm{PA}}(\ulcorner G\urcorner)$?

Exercise 9.9 Explain why $\mathrm{PROV}_{\mathrm{PA}}(x)$ doesn't represent provability in PA. (Hint: use the previous exercise.)

Exercise 9.10 Show that every ω-consistent theory is consistent.

Exercise 9.11 Let T be an ω-inconsistent, but consistent theory. By the completeness of first-order logic, T has a model. Can you describe what such a model might look like?

9.4 Tarski's Theorem

Recall that a formula $A(x)$ represents a property P in a theory T iff for every $\mathfrak{L}_A$-sentence B,

(i) if $P(B)$, then $\vdash_T A(\ulcorner B \urcorner)$, and

(ii) if $\neg P(B)$, then $\vdash_T \neg A(\ulcorner B \urcorner)$.

In exercise 9.9, you showed that $\text{PROV}_{\text{PA}}(x)$ does not represent provability in PA. Officially, PA is just the set of all sentences that are provable in PA. You therefore showed that $\text{PROV}_{\text{PA}}(x)$ does not represent membership in PA.

This result can be strengthened. The following theorem, due to Alfred Tarski (1933), shows that no $\mathfrak{L}_A$-formula represents membership in PA. Indeed, no formula represents membership in any consistent theory in which diag is representable.

Theorem 9.3: Tarski's Theorem

If T is consistent and diag is representable in T, then membership in T is not representable in T.

Proof. Suppose $T(x)$ represents membership in T. By the Diagonal Lemma, there is a sentence G such that

$$\vdash_T G \leftrightarrow \neg T(\ulcorner G \urcorner) \tag{1}$$

Since $T(x)$ represents membership in T, we have

$$\text{if } \vdash_T G \text{, then } \vdash_T T(\ulcorner G \urcorner) \tag{2}$$

$$\text{if } \nvdash_T G \text{, then } \vdash_T \neg T(\ulcorner G \urcorner) \tag{3}$$

Either $\vdash_T G$ or $\nvdash_T G$. Suppose $\vdash_T G$. Then $\vdash_T \neg T(\ulcorner G \urcorner)$ by (1), and $\vdash_T T(\ulcorner G \urcorner)$ by (2); so T is inconsistent. Alternatively, suppose $\nvdash_T G$. Then $\vdash_T T(\ulcorner G \urcorner)$ by (1), and $\vdash_T \neg T(\ulcorner G \urcorner)$ by (3); again, T is inconsistent. □

Note that Tarski's Theorem isn't restricted to axiomatizable theories. It even holds for $\text{Th}(\mathfrak{A})$. Since representability in $\text{Th}(\mathfrak{A})$ implies definability in $\mathfrak{L}_A$, it follows that no $\mathfrak{L}_A$-formula defines membership in $\text{Th}(\mathfrak{A})$:

Theorem 9.4

Arithmetical truth is not definable in $\mathfrak{L}_A$: there is no $\mathfrak{L}_A$-formula $T(x)$ such that $\mathfrak{A} \Vdash T(\ulcorner A \urcorner)$ iff $\mathfrak{A} \Vdash A$.

Proof. Th($\mathfrak{A}$) is consistent extension of Q. By Theorem 8.3, it follows that diag is representable in Th($\mathfrak{A}$). By Theorem 9.3, it follows that membership in Th($\mathfrak{A}$) is not representable in Th($\mathfrak{A}$): there is no $\mathfrak{L}_A$-formula $T(x)$ such that

(i) if $\mathfrak{A} \Vdash A$ then $\mathfrak{A} \Vdash T(\ulcorner A \urcorner)$, and
(ii) if $\mathfrak{A} \nVdash A$ then $\mathfrak{A} \Vdash \neg T(\ulcorner A \urcorner)$.

So there is no $\mathfrak{L}_A$-formula that defines truth in $\mathfrak{A}$. □

Exercise 9.12 Use the Semantic Diagonal Lemma to prove Theorem 9.4, without invoking Theorem 9.3.

Tarski's Theorem shows that while $\mathfrak{L}_A$ can formalize its own syntax (we can define $\mathfrak{L}_A$-properties like being a variable or being a sentence), it can't express the most basic concept of its own semantics. This isn't just true for $\mathfrak{L}_A$. Loosely speaking, no sufficiently powerful language that can express its own syntax can express its own semantics.

We can bring this out a little more clearly by considering the concept of a truth predicate. As Tarski pointed out, the central feature of the predicate 'is true' in English is that when it is applied to a sentence, the result is equivalent to that sentence:

(1) 'Snow is white' is true iff snow is white.
(2) '2+2=4' is true iff 2+2=4.

Sentences like (1) and (2) are called *Tarski biconditionals*. A theory that can reason about truth should be able to prove all Tarski biconditionals for its language. Thus a formula $W(x)$ is called a *truth predicate for a theory T* iff $\vdash_T W(\ulcorner A \urcorner) \leftrightarrow A$ for every sentence A in T's language. An argument similar to the one used in Theorem 9.3 shows that no sufficiently powerful theory can have a truth predicate, unless it is inconsistent. This result is also called "Tarski's Theorem".

Theorem 9.5: Also Tarski's Theorem

If diag is representable in a consistent theory T then T has no truth predicate.

Proof. Suppose $W(x)$ is a truth predicate for T. By the Syntactic Diagonal Lemma, there is a sentence L such that $\vdash_T L \leftrightarrow \neg W(\ulcorner L \urcorner)$. Since $W(x)$ is a truth predicate for T, $\vdash_T W(\ulcorner L \urcorner) \leftrightarrow L$. So $\vdash_T L \leftrightarrow \neg L$. So T is inconsistent. □

While Gödel's sentence G says of itself that it is unprovable, the sentence L that figures in this proof says of itself that it is not true. It is a formal analogue of the Liar sentence 'This sentence is false'. The existence of such a sentence leads to paradox: if L is true then L is false, and if L is false then L is true. Theorem 9.5 concludes that L can't exist. By the Diagonal Lemma, it would exist if there were a truth predicate for T. So there can be no truth predicate for T. By contrast, it is not an option to deny the existence of G. By the Diagonal Lemma, G can be constructed from $\text{PROV}_T(x)$. The existence of $\text{PROV}_T(x)$ is guaranteed by the fact that (for suitable choices of T) Prf_T is recursive.

Exercise 9.13 Show that if T is a sound theory then there is no truth predicate for T.

We'll now use Tarski's Theorem to derive both the undecidability of first-order logic and strengthened versions of Gödel's First Incompleteness Theorem. We begin with two small lemmas.

Lemma 9.3

Every consistent theory in which all recursive functions are representable is recursively undecidable.

Proof. Let T be a consistent theory in which all recursive functions are representable. By Theorem 9.3, membership in T is not representable in T. So membership in T is not recursive: the set of Gödel numbers of sentences in T is not recursively decidable. □

Lemma 9.4

Let $\hat{Q}$ be the conjunction of the seven axioms of Q. The set of $\mathfrak{L}_A$-sentences of the form $\hat{Q} \to A$ is recursively decidable.

Proof. Let P be the property of coding sentences of the form $\hat{Q} \to A$. $P(n)$ can be defined as $\exists y \leq n\, (\text{Sent}(y) \land (n = \ulcorner \hat{Q} \to \urcorner * y))$. So $P(n)$ is (primitive) recursive. □

Theorem 9.6: Church's Theorem

The set of valid first-order sentence is recursively undecidable.

Proof. By Theorem 8.3, all recursive functions are representable in Q. Since Q is consistent, it follows by Lemma 9.3 that Q is recursively undecidable. As in the previous proof, let $\hat{Q}$ be the conjunction of Q's axioms. If the set of valid first-order sentences were recursively decidable, the set of valid $\mathfrak{L}_A$-sentences of the form $\hat{Q} \to A$ would also be recursively decidable, by Lemma 9.4 and the fact that the intersection of two recursively decidable sets is recursively decidable. By the soundness and completeness of first-order logic, $\hat{Q} \to A$ is valid iff $\vdash_Q A$. So Q would be recursively decidable, contradicting what we just established. □

Church's Theorem shows that Hilbert's Entscheidungsproblem has no solution: there is no mechanical procedure that decides whether an arbitrary first-order sentence is valid.

Exercise 9.14 In Section 6.4, I explained how Theorem 9.6 can be derived from the unsolvability of the Halting Problem. Explain in outline how we could derive the unsolvability of the Halting Problem from Theorem 9.6. (Hint: Given a first-order sentence A, we can mechanically go through all first-order proofs until we find a proof of A, in which case we halt and output 'yes'.)

Exercise 9.15 Explain why there can be no recursive bound on the length of a proof for a sentence in the first-order calculus: for every recursive function f, there is a sentence with length n that is provable, but whose proof requires more than $f(n)$ lines.

Now for the strengthened version of the syntactic Incompleteness Theorem. In Section 5.4 (Proposition 5.5), I showed that every axiomatizable and complete first-order theory is decidable. Together with Lemma 9.3, this immediately gives us the what we seek: every axiomatizable and consistent theory in which all recursive functions are representable is incomplete.

However, Proposition 5.5 used the informal concept of computable axiomatizability; so this argument relies on the Church-Turing Thesis. Let's prove a parallel result for recursive axiomatizability.

Lemma 9.5

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

Proof. Let T be a recursively axiomatizable and complete first-order theory. As in the previous section, let Prf_T be the relation that holds between numbers n and m iff n codes a deduction of m from some recursively decidable set of axioms for T. We know that Prf_T is recursive. We can now define the property W of coding a member of T as follows:

$$W(x) \text{ iff } \mathrm{Prf}_T(\mu p[\mathrm{Prf}_T(p,x) \vee \mathrm{Prf}_T(p,\#[\neg]*x)],x)$$

To see how this works, let A be any sentence, and x its Gödel number. $\mu p[\mathrm{Prf}_T(p,x) \vee \mathrm{Prf}_T(p,\#[\neg]*x)]$ finds the (Gödel number of the) first proof of either A or $\neg A$ in T. Since at least one of A and $\neg A$ must be in T by completeness, this search is guaranteed to terminate. The outer Prf_T then checks whether the proof that has been found is a proof of A. □

Theorem 9.7: Also Gödel's First Incompleteness Theorem

Every consistent and recursively axiomatizable theory in which all recursive functions are representable is incomplete.

Proof. Let T be a consistent and recursively axiomatizable theory in which all recursive functions are representable. By Lemma 9.3, T is undecidable. By Lemma 9.5, it follows that T is incomplete. □

Exercise 9.16 Let T be a consistent theory in which diag is representable. By Theorem 9.3, there is no formula $W(x)$ such that

(i) if $\vdash_T A$, then $\vdash_T W(\ulcorner A \urcorner)$, and
(ii) if $\nvdash_T A$, then $\vdash_T \neg W(\ulcorner A \urcorner)$.

But there could be formula $W(x)$ such that

(i*) $\vdash_T A$ iff $\vdash_T W(\ulcorner A \urcorner)$.

In the terminology of exercise 8.16, this formula *weakly represents* membership in T. Show that if such a formula exists then T is incomplete.

(Hint: use the Diagonal Lemma to infer that there is a sentence G such that $\vdash_T G \leftrightarrow \neg W(\ulcorner G \urcorner)$; show that neither G nor $\neg G$ is in T.)

Since all recursive functions are representable in every extension of Q (Theorem 8.3), Theorem 9.7 is often stated as saying that every consistent and recursively axiomatizable extension of Q is incomplete. We can prove an even stronger result by strengthening Lemma 9.3.

Lemma 9.6

Every $\mathfrak{L}_A$-theory consistent with Q is recursively undecidable.

Proof. Let T be an $\mathfrak{L}_A$-theory consistent with Q, and suppose for reductio that T is recursively decidable: the set W of Gödel numbers of sentences in T is recursive. Since diag is recursive, so is the property P that holds of a number x iff $\text{diag}(x)$ is not in W. By exercise 8.16.(c), all recursive relations are weakly representable in any $\mathfrak{L}_A$-theory consistent with Q. So P is weakly representable in T: there is a formula $A(x)$ such that $\vdash_T A(\overline{n})$ iff $P(n)$. So:

$$\begin{aligned}
\vdash_T A(\ulcorner A(x) \urcorner) &\text{ iff } P(\#[A(x)]) \\
&\text{ iff } \text{diag}(\#[A(x)]) \notin W \\
&\text{ iff } \#[\exists x(x = \ulcorner A(x) \urcorner \wedge A(x))] \notin W \\
&\text{ iff } \nvdash_T \exists x(x = \ulcorner A(x) \urcorner \wedge A(x)) \\
&\text{ iff } \nvdash_T A(\ulcorner A(x) \urcorner).
\end{aligned}$$

Contradiction. □

Theorem 9.8

Every recursively axiomatizable $\mathfrak{L}_A$-theory that is consistent with Q is incomplete.

Proof. Let T be a recursively axiomatizable $\mathfrak{L}_A$-theory that is consistent with Q. By Lemma 9.6, T is recursively undecidable. By Lemma 9.5, it follows that T is incomplete. □

9.5 The arithmetical hierarchy

Let's take stock. Since all recursive functions are representable in PA, Gödel's Theorem shows that PA is incomplete (unless it is inconsistent). The incompleteness can't be fixed by simply adding more axioms: as long as the resulting theory is consistent and axiomatizable, it will remain incomplete.

The result carries over to more powerful theories like ZFC, in virtue of the fact that PA is interpretable in these theories (see Section 4.3). More generally, Gödel's Theorem applies whenever a theory's language is rich enough to express central aspects of its own syntax. This isn't always the case. For example, consider a fragment of $\mathfrak{L}_A$ whose only non-logical symbols are 0, s, and $+$, without ×. In the previous chapter, we needed multiplication to define the recursive functions and relations. Without multiplication, Prf_T and $*$ are no longer definable. As a consequence, the Incompleteness Theorems don't apply. Indeed, if you restrict the axioms of PA to this weaker language, and remove the two axioms for multiplication, you get a complete theory. (This theory is called *Presburger Arithmetic*.)

Return to PA. We know that there are (infinitely many) true sentences that PA can't prove. But what do they look like? This is important to assess the practical significance of Gödel's result. If PA can't prove 2+2=4, that's a serious problem. If the only arithmetical truths that PA can't prove take a trillion years to state, incompleteness may be harmless in practice.

Gödel's original proof (unlike the proof via Tarski's Theorem) gives us an example of an unprovable sentence: the "Gödel sentence" G. As I'll explain below, this sentence states (in a very roundabout way) that a certain complicated equation between polynomials has no solution in the natural numbers. If it weren't for Gödel's Theorem, no one would ever have considered this equation. Until the 1960s, the only sentences known to be unprovable in PA were of this kind. Since then, more natural examples have been found. The simplest is probably Goodstein's Theorem. (See Section 7.3.) Goodstein's Theorem states an interesting fact about the natural numbers, but its proof involves transfinite ordinals: it is provable in ZFC, but not in PA. For ZFC itself, we already know of a "natural" statement that it can't decide: the Continuum Hypothesis. There are many other examples.

To get a sense of which $\mathfrak{L}_A$-sentences are provable and which might be unprovable in PA, it is useful to classify the $\mathfrak{L}_A$-sentences by their construction from atomic formulas. Since the only predicate letter in $\mathfrak{L}_A$ is the identity predicate '=', all atomic formulas of $\mathfrak{L}_A$ are identity statements: they have the form $t_1 = t_2$. From these, complex formulas are constructed using truth-functional connectives and quantifiers. We'll divide them into stages.

At the first stage, we have all identity statements $t_1 = t_2$, all inequalities of the form $t_1 < t_2$, and all formulas that can be constructed from these by truth-functional connectives and bounded quantification, where a bounded quantification of a formula A is a formula of the form $\forall x(x < t \to A)$ or $\exists x(x < t \land A)$, with x not occurring in t. (Officially, of course, $t_1 < t_2$ is short for $\exists z(t_1 + s(z) = t_2)$.) The formulas in this class are called Δ_0-*formulas*. Intuitively, a Δ_0-formula is any $\mathfrak{L}_A$-formula that doesn't involve unbounded quantification.

At the next stage, we consider all sentences that can be formed from Δ_0-formulas by prefixing unbounded universal quantifiers or unbounded existential quantifiers. A Δ_0-formula with a string of universal quantifiers in front is called a Π_1-*formula*; a Δ_0-formula with a string of existential quantifiers in front is called a Σ_1-*formula*. For example, $\forall x \forall y(x + y = y + x)$ is a Π_1-formula, while $\exists x \exists y(x + y = y + x)$ is a Σ_1-formula.

Prefixing universal quantifiers to a Σ_1-formula yields a Π_2-formula; prefixing existential quantifiers to a Π_1-formula yields a Σ_2-formula. Thus $\forall x \exists y(x+y = y+x)$ is Π_2, and $\exists x \forall y(x + y = y + x)$ is Σ_2. And so on.

This somewhat complicated classification is motivated by the computational properties of the relations defined by the relevant formulas. The relations expressed by Δ_0-formulas are all primitive recursive. Since Δ_0-formulas don't involve unbounded quantification, one can check whether they hold of some numbers by simple checks, without unbounded loops. By contrast, to check whether a Σ_1-formula $\exists x A(x)$ holds of some number, one may need to search through all numbers until one finds a witness for $A(x)$. Many Σ_1-formulas therefore express relations that are not primitive recursive. Some of them are merely recursive. In fact, every recursive relation is definable by a Σ_1-formula. But not every relation defined by a Σ_1-formula is recursive.

Some are just recursively enumerable. A relation R is *recursively enumerable* if there is a recursive relation S such that $R(x_1, \ldots, x_n)$ holds iff $\exists y\, S(x_1, \ldots, x_n, y)$. By the Church-Turing Thesis, the recursively enumerable relations are precisely the computably enumerable relations. (See Propositions 5.2 and 5.3 in Section 5.4.)

Theorem 9.9

A relation is recursively enumerable iff is definable in $\mathfrak{L}_A$ by a Σ_1-formula.

Proof sketch. I assume for readability that R is one-place.

From right to left, assume that R is defined by a Σ_1-formula $\exists y A(x, y)$, where A is Δ_0.

We can then mechanically enumerate all n for which $R(n)$ holds by going through all pairs of numbers (n, m) and check whether $A(n, m)$ holds.

For the other direction, we need show that every recursive relation is defined by a Σ_1-formula. Since prefixing existential quantifiers to a Σ_1-formula yields another Σ_1-formula, the result extends to every recursively enumerable relation.

The proof that every recursive function is defined by a Σ_1-formula proceeds by induction on the construction of recursive functions. In chapter 8, I showed that the base functions (zero, successor, projection) are definable in $\mathfrak{L}_A$, and that definability-in-$\mathfrak{L}_A$ is closed under composition, primitive recursion, and minimization. By going through each part of this proof, we can check that the defining formulas are all Σ_1. This is obvious for the base functions, which I explicitly defined using Δ_0-formulas. (For example, I showed that zero is defined by $x = 0$.) Closure under composition is also straightforward. I showed that $\mathrm{Cn}[f, g_1]$ is defined by $\exists v_1(F(y, v_1) \wedge G_1(v_1, x_1, \ldots, x_n))$. Since any initial existential quantifiers in F and G_1 can simply be pulled to the front, so the whole formula is Σ_1 if F and G_1 are.

Regular minimization requires a more work. I showed that $\mathrm{Mn}[f]$ is defined by $F(x, y, 0) \wedge \forall z(z < y \rightarrow \neg F(x, z, 0))$. We need to show that any initial existential quantifiers in F can be pulled to the front. This is possible because $\forall z(z < y \rightarrow \neg\exists w F(x, z, 0))$ is equivalent to $\exists c \forall z(z < y \rightarrow \neg F(x, z, \text{BETA}(c, z))$: the beta term $\text{BETA}(c, z)$ retrieves the witness for $F(x, z, 0)$ from the code c. By going through the construction of BETA, one can show that it is definable by a Δ_0-formula.

Finally, for primitive recursion, I showed that $\mathrm{Pr}[f, g_1]$ is defined by $\exists c(\text{SEQ}(c, x, k) \wedge \text{BETA}(c, s(k), y))$, where $\text{SEQ}(c, x, k)$ is defined in terms of F and G_1 and BETA. I've already mentioned that BETA is definable by a Δ_0-formula. Using the beta function trick that we've just used for minimization, one can show that $\text{SEQ}(c, x, k)$ is definable by a Σ_1-formula, by pulling existential quantifiers to the front. □

We can use Theorem 9.9 to get an idea of what the unprovable Gödel sentence G for PA might look like. Recall that G is equivalent in PA to $\neg\mathrm{Prov}_{\mathrm{PA}}(\ulcorner G \urcorner)$. Since $\mathrm{Prov}_{\mathrm{PA}}$ is defined by existential quantification from the recursive relation $\mathrm{Prf}_{\mathrm{PA}}$, it is recursively enumerable. By Theorem 9.9, it is definable by a Σ_1-formula. So the Gödel sentence G is equivalent in PA to the negation of a Σ_1-sentence. This makes it equivalent to a Π_1-sentence. Gödel's result therefore shows that there are undecidable Π_1-sentences.

Theorem 9.9 can be strengthened:

Theorem 9.10: The MRDP Theorem

A relation is recursively enumerable iff it is definable in $\mathfrak{L}_A$ by a formula of the form $\exists x_1 \ldots \exists x_n\, t_1 = t_2$.

Since every $\mathfrak{L}_A$-term expresses a polynomial, the MRDP Theorem shows that every recursively enumerable relation is expressed by a formula stating that a certain equation between polynomials has a solution in the natural numbers. That's why I said that the Gödel sentence is equivalent to the statement that some equation between polynomials has no solution in the natural numbers. The proof of the MRDP theorem is too difficult to be even sketched here.

Let's return once more to Tarski and Gödel. By Theorem 9.4, arithmetical truth is not definable in $\mathfrak{L}_A$. With our new understanding of the arithmetical hierarchy, we can now strengthen the semantic Incompleteness Theorem. As stated in Theorem 9.1, the semantic Theorem says that every sound and recursively axiomatizable $\mathfrak{L}_A$-theory is incomplete. It is easy to see that a theory is recursively axiomatizable iff the set of Gödel numbers of its members is recursively enumerable. Theorem 9.1 therefore applies to all theories whose members are defined by a Σ_1-formula. We can extend the result to non-axiomatizable theories that are only definable by Π_2-formulas or Σ_{12}-formulas.

Let's say that a theory T is *definable in* $\mathfrak{L}_A$ if there is an $\mathfrak{L}_A$-formula $W(x)$ such that for all sentences B, $\mathfrak{A} \Vdash W(\ulcorner B \urcorner)$ iff $B \in T$.

Theorem 9.11

Every sound $\mathfrak{L}_A$-theory that is definable in $\mathfrak{L}_A$ is incomplete.

Proof. If T is sound and complete then $T = \text{Th}(\mathfrak{A})$. By Theorem 9.4, $\text{Th}(\mathfrak{A})$ is not definable in $\mathfrak{L}_A$. So if T is sound and definable in $\mathfrak{L}_A$ then it is incomplete. □

Exercise 9.17 Explain why a theory is recursively axiomatizable iff the set of Gödel numbers of its members is recursively enumerable. (Hint: if T is recursively axiomatizable then Prf_T is recursive.)

10 The Unprovability of Consistency

Gödel's Second Incompleteness Theorem states that no sufficiently strong mathematical theory can prove its own consistency. In this chapter, we explore how this can be shown and what it implies for the foundations of mathematics and beyond.

10.1 The Second Incompleteness Theorem

Let's review some background. As in the previous chapter, we're going to focus on axiomatizable theories in the language of arithmetic $\mathfrak{L}_A$. A *theory* is simply a set of sentences that is closed under logical consequence. A *(recursively) axiomatizable* theory is one for which there is a (recursively) decidable set of axioms from which all and only the members of the theory are deducible.

By coding $\mathfrak{L}_A$-strings as numbers, we can use the language of arithmetic to reason about its own syntax. We use '$\#[A]$' to denote the code ("Gödel number") of an $\mathfrak{L}_A$-string A, and '$\ulcorner A \urcorner$' to denote the $\mathfrak{L}_A$-numeral for $\#[A]$, relative to some fixed coding scheme. (The details of the scheme don't matter, as long as it is effective.)

Given a recursively axiomatizable theory T, let Prf_T be the relation that holds between numbers n and m iff n codes a deduction of the sentence coded by m from some recursively decidable set of axioms for T. This relation is recursive. In Chapter 8 we proved that all recursive functions and relations are representable in any moderately strong theory of arithmetic – in particular, in any extension of Q. That is, if T is at least as strong as Q then there is an $\mathfrak{L}_A$-formula $\text{PRF}_T(x,y)$ such that

(i) if $\text{Prf}_T(n,m)$ holds, then $\vdash_T \text{PRF}_T(\ulcorner n \urcorner, \ulcorner m \urcorner)$;

(ii) if $\text{Prf}_T(n,m)$ doesn't hold, then $\vdash_T \neg\text{PRF}_T(\ulcorner n \urcorner, \ulcorner m \urcorner)$.

If T is sound, the formula $\text{PRF}_T(x,y)$ also defines Prf_T in $\mathfrak{L}_A$, so that

$$\mathfrak{A} \Vdash \text{PRF}_T(\ulcorner n \urcorner, \ulcorner m \urcorner) \text{ iff } \text{Prf}_T(n,m).$$

From $\text{PRF}_T(x,y)$, we can define another formula $\text{PROV}_T(x)$ as $\exists y\, \text{PRF}_T(y,x)$. Informally, $\text{PROV}_T(x)$ says that the sentence with Gödel number x is provable in T.

Now remember that a theory T is *consistent* if it doesn't prove a contradiction. This can be spelled out in multiple ways:

(1) There is no sentence A such that $\vdash_T A$ and $\vdash_T \neg A$;

(2) $\nvdash_T \bot$;

(3) There is some sentence A such that $\nvdash_T A$.

In classical logic, all three conditions are equivalent. Let's focus on (2), since it is the shortest. If we have an $\mathfrak{L}_A$-formula $\text{PROV}_T(x)$ that expresses provability in T, we can find another $\mathfrak{L}_A$ formula that expresses the consistency of T: $\neg\text{PROV}_T(\ulcorner\bot\urcorner)$.

(Officially, '$\bot$' is not part of $\mathfrak{L}_A$: in Chapter 1, I suggested that it abbreviates '$\neg(p \to p)$', but we also don't have sentence letters in $\mathfrak{L}_A$. Let's say that $\bot$ is the sentence '$\neg(0 = 0)$'. Since $0 = 0$ is provable in first-order logic, $\bot$ is refutable in first-order logic, which is all that matters.)

Now let T be some recursively axiomatizable extension of Q. Since the consistency of T can be expressed in T's language, we might wonder whether T can prove it: can an arithmetical theory prove its own consistency? At the end of the 1931 paper in which he proved the First Incompleteness Theorem, Gödel gave an answer: he claimed that no sufficiently strong, recursively axiomatizable, and consistent theory can prove its own consistency. This is Gödel's Second Incompleteness Theorem.

Gödel also outlined a proof of this claim. It goes as follows.

Remember that in his proof of the First Incompleteness Theorem, Gödel used the (Syntactic) Diagonal Lemma to construct a "Gödel sentence" G such that

$$\vdash_T G \leftrightarrow \neg\text{PROV}_T(\ulcorner G\urcorner). \tag{1}$$

He then showed that *if T is consistent then T can't prove G.* He also showed that if T is ω-consistent then it can't prove $\neg G$. But let's focus on the first result. Its proof required no advanced mathematics. (No transfinite ordinals or the Axiom of Choice or anything like that.) It can be carried out in any moderately strong theory that can reason about recursivity, representability, and the syntax of T. Suppose it can be carried out in T itself. Then the following sentence is provable in T:

$$\neg\text{PROV}_T(\ulcorner\bot\urcorner) \to \neg\text{PROV}_T(\ulcorner G\urcorner). \tag{2}$$

This says in $\mathfrak{L}_A$ that if T is consistent then T can't prove G.

Now suppose that T can prove its own consistency: it can prove $\neg\text{PROV}_T(\ulcorner\bot\urcorner)$. By Modus Ponens and (2), T can infer $\neg\text{PROV}_T(\ulcorner G\urcorner)$. By (1), it can then infer G. But we've assumed that T is a recursively axiomatizable extension of Q. And we know from Gödel's First Incompleteness Theorem that no recursively axiomatizable and *consistent* extension of Q can prove G. So T must be inconsistent.

In sum, if T is a consistent and recursively axiomatizable extension of Q, and (2) is provable in T, then T can't prove its own consistency.

It turns out that (2) isn't provable in Q. But it is provable in the standard axiomatization of arithmetic, Peano Arithmetic (PA). The Second Incompleteness Theorem therefore implies that PA can't prove its own consistency (assuming it is consistent).

Before we investigate why (2) is provable in PA, let's think about the significance of the result. Why should we care whether PA can prove its own consistency?

Note that any *inconsistent* theory can trivially prove its own consistency (as long as this is expressible in its language): an inconsistent theory can prove everything. Even if a consistent theory could prove its own consistency, this would therefore provide no reason to think that the theory really is consistent.

The significance of the Second Incompleteness Theorem comes from what it implies about the powers of theories to prove the consistency of *other* theories. Take a theory like ZFC in which most of mathematics can be formalized. Hilbert had hoped that one could prove the consistency of such theories in a much weaker, "finitary" theory that studies deductions as finite syntactic objects. Gödel's Second Incompleteness Theorem implies that this can't be done. ZFC is certainly strong enough to prove (2). By Gödel's Theorem, it follows that ZFC can't prove its own consistency. But then *no weaker theory* can prove the consistency of ZFC either: if something isn't even provable in ZFC, it can't be provable in, say, PA or Q – for anything that's provable in PA or Q is also provable (under a suitable translation, see Section 4.3) in ZFC.

Exercise 10.1 The Second Incompleteness Theorem allows that the consistency of ZFC could be proved in a theory stronger than ZFC. Explain why this would hardly reassure skeptics who doubt the consistency of ZFC.

To complete the proof of the Second Incompleteness Theorem, we'd need to show that sufficiently powerful theories like PA prove (2). Fortunately, this doesn't require formalizing the entire proof of the First Incompleteness Theorem. It is enough to show that the formula $\text{PROV}_T(x)$ satisfies some basic conditions. These conditions are known as the *Hilbert-Bernays-Löb provability conditions*, because they were first formulated explicitly in a 1939 textbook by Hilbert and Bernays and later streamlined by Martin Löb. They are as follows.

P1 If $\vdash_T A$, then $\vdash_T \text{PROV}_T(\ulcorner A \urcorner)$.

P2 $\vdash_T \text{PROV}_T(\ulcorner A \to B \urcorner) \to (\text{PROV}_T(\ulcorner A \urcorner) \to \text{PROV}_T(\ulcorner B \urcorner))$.

P3 $\vdash_T \text{PROV}_T(\ulcorner A \urcorner) \to \text{PROV}_T(\ulcorner \text{PROV}_T(\ulcorner A \urcorner) \urcorner)$.

We'll see below how (2) can be derived from P1–P3. First, let's examine what it takes to show that, say, $\text{PROV}_{\text{PA}}(x)$ satisfies the three conditions.

P1 is easy. Assume that A is provable in PA. Then there is number n that codes a deduction of A from the axioms of PA. Since $\text{PRF}_{\text{PA}}(x, y)$ represents Prf_{PA} in PA, we have $\vdash_{\text{PA}}$ $\text{PRF}_{\text{PA}}(\overline{n}, \ulcorner A \urcorner)$. So we also have $\vdash_{\text{PA}} \exists y\, \text{PRF}_{\text{PA}}(y, \ulcorner A \urcorner)$ and therefore $\vdash_{\text{PA}} \text{PROV}_{\text{PA}}(\ulcorner A \urcorner)$. So $\text{PROV}_{\text{PA}}(x)$ satisfies P1.

The argument for P2 isn't much harder. Suppose we have a proof of $A \to B$ and a proof of A. From these, we can construct a proof of B by concatenating the two proofs and adding B as a final line (applying Modus Ponens). All this can be formalized in PA, showing that if there are numbers m and n that code proofs of A and $A \to B$, then there is a number k that codes a proof of B. Which is what P2 says.

The argument for P3 is more involved. It essentially requires formalizing within PA the proof that PROV_{PA} satisfies P1. This takes about a dozen pages of tedious tinkering. I'll spare us the details. (You can find them, for example, in George Boolos, *The Logic of Provability*, 1993.) Let's just accept that $\text{PROV}_{\text{PA}}(x)$ satisfies P1–P3.

Exercise 10.2 Using the soundness of PA, show that $\text{PROV}_{\text{PA}}(x)$ also satisfies the following condition:

CNec If $\vdash_T \text{PROV}_T(\ulcorner A \urcorner)$ then $\vdash_T A$.

Exercise 10.3 Explain why it doesn't follow from the previous exercise that $\text{PROV}_{\text{PA}}(x)$ satisfies this condition:

Ref $\vdash_T \text{PROV}_T(\ulcorner A \urcorner) \to A$.

We can now finish the proof of the Second Incompleteness Theorem, by deriving (2) from P1–P3. To remove clutter, I'll abbreviate $\text{PROV}_T(\ulcorner A \urcorner)$ as $\Box A$. This is a little misleading because it suggests that the sentence A occurs in $\Box A$, while in fact only its Gödel numeral does. But it will make the proof more readable.

With this convention, P1–P3 can be written as follows:

P1 If $\vdash_T A$, then $\vdash_T \Box A$.

P2 $\vdash_T \Box(A \to B) \to (\Box A \to \Box B)$.

P3 $\vdash_T \Box A \to \Box\Box A$.

Theorem 10.1: Gödel's Second Incompleteness Theorem

If T is a consistent and recursively axiomatizable theory in which diag is representable, and $\mathrm{PROV}_T(x)$ is a formula that satisfies P1–P3, then T cannot prove $\neg\mathrm{PROV}_T(\ulcorner\bot\urcorner)$.

Proof. Applying the (Syntactic) Diagonal Lemma to the formula $\neg\mathrm{PROV}_T(x)$, we get a sentence G such that $\vdash_T G \leftrightarrow \neg\mathrm{PROV}_T(\ulcorner G\urcorner)$. Using the fact that $\mathrm{PROV}_T(x)$ satisfies P1–P3, we now reason as follows.

1.	$\vdash_T G \leftrightarrow \neg\Box G$	(Diagonal Lemma)
2.	$\vdash_T G \to \neg\Box G$	(from 1)
3.	$\vdash_T G \to (\Box G \to \bot)$	(from 2)
4.	$\vdash_T \Box(G \to (\Box G \to \bot))$	(from 3 by P1)
5.	$\vdash_T \Box G \to \Box(\Box G \to \bot)$	(from 4 by P2)
6.	$\vdash_T \Box G \to (\Box\Box G \to \Box\bot)$	(from 5 by P2)
7.	$\vdash_T \Box G \to \Box\Box G$	(P3)
8.	$\vdash_T \Box G \to \Box\bot$	(from 6 and 7)
9.	$\vdash_T \neg\Box\bot \to \neg\Box G$	(from 8)

Line 9 is (2) in the box notation. From here, the argument continues as explained above:

10.	$\vdash_T \neg\Box\bot$	(Assumption)
11.	$\vdash_T \neg\Box G$	(from 9 and 10)
12.	$\vdash_T G$	(from 1 and 11)
13.	$\vdash_T \Box G$	(from 12 by P1)

This shows that if T proves its own consistency (line 10) then it is inconsistent: it proves both $\mathrm{PROV}_T(\ulcorner G\urcorner)$ (line 13) and $\neg\mathrm{PROV}_T(\ulcorner G\urcorner)$ (line 11). By contraposition: if T is consistent, it can't prove its own consistency. □

Why is this an *incompleteness* theorem? Because it shows that a certain sentence, $\neg\text{PROV}_T(\ulcorner\bot\urcorner)$, is unprovable in any sufficiently strong and consistent theory T, even though it is true. In a way, this form of incompleteness is more striking than the one established by Gödel's First Theorem. Here is how Gödel put it in a 1951 lecture:

> [The second theorem] makes the incompletability of mathematics particularly evident. For, it makes it impossible that someone should set up a certain well-defined system of axioms and rules and consistently make the following assertion about it: All of these axioms and rules I perceive (with mathematical certitude) to be correct, and moreover I believe that they contain all of mathematics. If someone makes such a statement he contradicts himself. For if he perceives the axioms under consideration to be correct, he also perceives (with the same certainty) that they are consistent. Hence he has a mathematical insight not derivable from his axioms.

Take the standard axiomatization of set theory, ZFC. Suppose we "perceive" that ZFC is sound, and therefore consistent. If this "perception" is correct, it goes beyond what ZFC can prove, although the consistency part can be expressed in the language of ZFC: we perceive the truth of $\neg\text{PROV}_{\text{ZFC}}(\ulcorner\bot\urcorner)$. We might propose a strengthened version of ZFC, with $\neg\text{PROV}_{\text{ZFC}}(\ulcorner\bot\urcorner)$ as a new axiom. Call this theory ZFC$^+$. The Second Incompleteness Theorem still applies: ZFC$^+$ can't prove its own consistency (if it is consistent). But if ZFC is sound, then so is ZFC$^+$. Our "perception" of the soundness of ZFC therefore allows us to strengthen ZFC$^+$ by adding another consistency statement, $\neg\text{PROV}_{\text{ZFC}^+}(\ulcorner\bot\urcorner)$. Call this theory ZFC^{++}. The Second Incompleteness Theorem still applies …. In this way, our perception of the soundness of ZFC allows us to construct an infinite sequence of ever stronger theories, each of which must be sound. We might even propose a theory ZFC$^\omega$ that combines all these theories: the union of ZFC, ZFC$^+$, ZFC^{++}, etc. This theory is still recursively axiomatizable, so the Second Incompleteness Theorem still applies: it can't prove its own consistency (if it is consistent). So we can keep adding consistency statements, creating ZFC$^{\omega+1}$, ZFC$^{\omega+2}$, …, ZFC$^{\omega+\omega}$, and so on, up through the ordinals until we reach a point (the "Church-Kleene ordinal" ω_1^{CK}) where the union of the previous theories can no longer be captured in the language of ZFC.

Exercise 10.4 Formulas that satisfy P1–P3 are often called *provability predicates*. But this is misleading. Show that the formula $\text{SENT}(x)$ that represents (in PA) the property of coding an $\mathfrak{L}_A$-sentence satisfies P1–P3.

Exercise 10.5 The Second Incompleteness Theorem applies to any predicate satisfying P1–P3: the predicate $\text{PROV}_T(x)$ that figures in the Theorem doesn't have to be defined as $\exists y\, \text{PRF}_T(y,x)$ from a predicate $\text{PRF}_T(x,y)$ that represents the proof relation of T. What do we learn if we apply the Second Incompleteness Theorem to $\text{SENT}(x)$?

Exercise 10.6 Let PA$^+$ be obtained from PA by adding $\text{PROV}_{\text{PA}}(\ulcorner \bot \urcorner)$ as a new axiom. Is this theory consistent? Is it ω-consistent? (Hint: remember that $\text{PROV}_{\text{PA}}(\ulcorner \bot \urcorner)$ abbreviates $\exists y\, \text{PRF}_{\text{PA}}(y, \ulcorner \bot \urcorner)$.)

Exercise 10.7 Using exercise 10.2, explain why PA can't prove the negation of $\neg\text{PROV}_{\text{PA}}(\ulcorner \bot \urcorner)$, unless it is inconsistent.

10.2 Löb's Theorem

For a suitable theory T, the Diagonal Lemma allows us to construct a Gödel sentence G that says of itself that it is unprovable in T. We can also construct a sentence H that says of itself that it is provable. More formally, if we apply the (Syntactic) Diagonal Lemma to the formula $\text{PROV}_T(x)$, we get a sentence H such that

$$\vdash_T H \leftrightarrow \text{PROV}_T(\ulcorner H \urcorner). \tag{D}$$

This is called the *Henkin sentence* for T. It's easy to show that no consistent theory can prove its Gödel sentence. For the Henkin sentence, the situation is less clear. Is H provable in T? This question was raised by Leon Henkin in 1952, and answered by Martin Löb in 1955. Löb showed that if T is consistent, recursively axiomatizable, and sufficiently strong, and it proves $\text{PROV}_T(\ulcorner A \urcorner) \to A$ for some sentence A, then it also proves that sentence A. This is known as *Löb's Theorem*. Since (D) gives us $\vdash_T \text{PROV}_T(\ulcorner H \urcorner) \to H$, the Theorem entails $\vdash_T H$. The Henkin sentence H is indeed provable in T.

The proof of Löb's Theorem resembles the following proof that Santa Claus exists.

Let S be the sentence 'if S is true then Santa Claus exists'. This being a conditional, we can try to prove it by deriving the consequent from the antecedent. So assume the antecedent: S is true. So if S is true then Santa Claus exists (for this is what S says). Still assuming that S is true, we can infer (by Modus Ponens) that Santa Claus exists.

Now we have derived the consequent of S from its antecedent. So we've proved S: we've proved that if S is true then Santa Claus exists. And since we've proved S, we can infer by Modus Ponens that Santa Claus exists.

This line of reasoning is known as *Curry's Paradox*. It clearly works for any consequent A in place of 'Santa Claus exists'. In the following proof of Löb's Theorem, we replace 'is true' (which is not expressible in $\mathfrak{L}_A$) by 'is provable'. The assumption $\vdash_T \textsc{prov}_T(\ulcorner A \urcorner) \to A$ of Löb's Theorem gives us one direction of the Tarski biconditional for A. In the presence of P1–P3, that's enough to run a Curry-style argument and show that $\vdash_T A$.

Theorem 10.2: (Löb's Theorem)

If T is a consistent, recursively axiomatizable theory in which diag is representable, and $\textsc{prov}_T(x)$ satisfies P1–P3, then the following holds for any sentence A in the language of T:

$$\text{If } \vdash_T \textsc{prov}_T(\ulcorner A \urcorner) \to A \text{ then } \vdash_T A.$$

Assume that $\vdash_T \textsc{prov}_T(\ulcorner A \urcorner) \to A$. Applying the Syntactic Diagonal Lemma to the formula $\textsc{prov}_T(x) \to A$, we get a sentence S such that $\vdash_T S \leftrightarrow (\textsc{prov}_T(\ulcorner S \urcorner) \to A)$. Using the box notation, we now reason as follows.

1. $\vdash_T S \leftrightarrow (\Box S \to A)$ (Diagonal Lemma)
2. $\vdash_T S \to (\Box S \to A)$ (from 1)
3. $\vdash_T \Box(S \to (\Box S \to A))$ (from 2 by P1)
4. $\vdash_T \Box S \to \Box(\Box S \to A)$ (from 3 by P2)
5. $\vdash_T \Box S \to (\Box\Box S \to \Box A)$ (from 4 by P2)
6. $\vdash_T \Box S \to \Box\Box S$ (P3)
7. $\vdash_T \Box S \to \Box A$ (from 5 and 6)
8. $\vdash_T \Box A \to A$ (assumption)
9. $\vdash_T \Box S \to A$ (from 7 and 8)
10. $\vdash_T S$ (from 1 and 9)
11. $\vdash_T \Box S$ (from 10 by P1)
12. $\vdash_T A$ (from 9 and 11) □

In line 1 of this proof, the Diagonal Lemma is used to construct the sentence S that "says that" (is provably equivalent to) 'if S is provable then A'. As in Curry's Paradox, T can derive that if S is provable then A (line 9). From this, T infers that S is true (line 10), and thereby that S provable (line 11), from which it infers A by Modus Ponens (line 12).

Exercise 10.8 Prove the converse of Löb's Theorem: if $\vdash_T A$, then $\vdash_T \text{PROV}_T(\ulcorner A \urcorner) \to A$.

From Löb's Theorem, it is a short step to the Second Incompleteness Theorem. Assume T can prove $\neg\text{PROV}_T(\ulcorner \bot \urcorner)$. By propositional logic, this entails $\text{PROV}_T(\ulcorner \bot \urcorner) \to \bot$. By Löb's Theorem, it follows that T can prove $\bot$. So if T can prove its own consistency, then T is inconsistent.

Löb's Theorem also entails a version of Tarski's Theorem on the undefinability of truth. Recall that a predicate $W(x)$ is a *truth predicate* for a theory T if T can prove all the Tarski biconditionals

$$W(\ulcorner A \urcorner) \leftrightarrow A.$$

Suppose that $W(x)$ is a truth predicate for T. Then $W(x)$ is also a provability predicate for T. Assume that T is a recursively axiomatizable theory in which diag is representable. Since $\vdash_T W(\ulcorner A \urcorner) \to A$ for all A, it follows by Löb's Theorem that $\vdash_T A$ for all A. So T is inconsistent.

Exercise 10.9 Explain why a truth predicate for T is also a provability predicate for T.

Löb's Theorem highlights the difference between the formal concept of provability and the concept of truth. One might have expected that a sufficiently powerful theory would "know that" whatever it can prove is the case. That is, one might have expected that PROV_T should satisfy the Reflection principle from exercise 10.3:

Ref $\quad \vdash_T \text{PROV}_T(\ulcorner A \urcorner) \to A.$

(A truth predicate would satisfy both Ref and its converse.) Löb's Theorem shows that if T is sufficiently strong and consistent then PROV_T satisfies only those instances of Ref for which A is already provable in T, in which case $\text{PROV}_T(\ulcorner A \urcorner) \to A$ follows by propositional logic.

Exercise 10.10 What's the difference between the hypothesis $\vdash_{\text{PA}} A$ and the arithmetical hypothesis $\text{PROV}_{\text{PA}}(A)$? Can one be true without the other?

Exercise 10.11 Return to the theory PA$^+$ axiomatized by adding $\text{PROV}_{\text{PA}}(\ulcorner\bot\urcorner)$ to the axioms of PA. (a) Explain why $\vdash_{\text{PA}^+} \text{PROV}_{\text{PA}^+}(\ulcorner\bot\urcorner)$. (b) Show that there is a sentence A for which $\vdash_{\text{PA}^+} \neg(\text{PROV}_{\text{PA}^+}(\ulcorner A\urcorner) \to A)$.

10.3 The logic of provability

If you are familiar with modal logic, the provability conditions P1–P3 will look familiar, especially when written with the box operator $\Box$. If you aren't familiar with modal logic, let me quickly introduce some background.

Propositional modal logic is an extension of classical propositional logic. The language $\mathfrak{L}_M$ of propositional modal logic is obtained from $\mathfrak{L}_0$ by adding the unary sentence operator '$\Box$'. Thus whenever A is an $\mathfrak{L}_M$-sentence, then so is $\Box A$. The intended meaning of the box varies from application to application; often it is used to express some kind of necessity.

Hilbert-style proof systems for propositional modal logic extend the propositional calculus by axioms and inference rules that govern the behaviour of the box. A well-known proof system adds one axiom schema and one rule of inference:

K $\quad \Box(A \to B) \to (\Box A \to \Box B)$
Nec $\quad$ From A one may infer $\Box A$

This system is known as K. (So 'K' names both a proof system and an axiom schema. The letter stands for Saul Kripke.) Stronger systems can be obtained by adding further axioms. For example, the system K4 adds an axiom schema known (for obscure historical reasons) as '4':

4 $\quad \Box A \to \Box\Box A$;

the system S4 adds both 4 and Ref (more commonly known as 'T'):

Ref $\quad \Box A \to A$.

Any system that can be obtained by adding axiom schemas to K is called a *normal modal logic*.

Above I used the “box notation” to simplify sentences of $\mathfrak{L}_A$. Sentences in the “box notation” look just like sentences of $\mathfrak{L}_M$. To make this more explicit, let t be a function that maps each sentence letter p of $\mathfrak{L}_M$ to an $\mathfrak{L}_A$-sentence $t(p)$. We can extend any such t to a mapping from $\mathfrak{L}_M$-sentences to $\mathfrak{L}_A$-formulas, as follows:

$$\begin{aligned}
\widetilde{p}^{\,t} &= t(p)\\
\widetilde{\neg A}^{\,t} &= \neg\widetilde{A}^{\,t}\\
\widetilde{A \to B}^{\,t} &= \widetilde{A}^{\,t} \to \widetilde{B}^{\,t}\\
\widetilde{\Box A}^{\,t} &= \text{PROV}_T(\ulcorner \widetilde{A}^{\,t} \urcorner)
\end{aligned}$$

Here I use ‘$\widetilde{A}^{\,t}$’ to denote the output of the mapping for input A. For example, assume that $t(p)$ is ‘$0 = 0$’. Then $\widetilde{p}^{\,t}$ is ‘$0 = 0'$, $\widetilde{\neg p}^{\,t}$ is ‘$\neg(0 = 0)$’, and $\widetilde{\Box p}^{\,t}$ is ‘$\text{PROV}_T(\ulcorner 0 = 0 \urcorner)$’.

Now assume that $\text{PROV}_T(x)$ satisfies P1–P3. In “box notation”, P1, P2, and P3 are Nec, K, and 4, respectively. We might therefore expect that whenever an $\mathfrak{L}_M$-sentence A is provable in the modal system K4, then $\widetilde{A}^{\,t}$ is provable in T. We’ll confirm this below.

The full modal logic of provability isn’t K4, however. It has another axiom schema, known as GL (for Gödel-Löb):

GL $\quad \Box(\Box A \to A) \to \Box A.$

The system of modal logic obtained by adding GL to K4 is also called GL. (It turns out that the 4 schema is redundant: it can be derived from GL.)

GL is the modal translation of a formalized version of Löb’s Theorem. Löb’s Theorem states that

If $\vdash_T \text{PROV}_T(\ulcorner A \urcorner) \to A$ then $\vdash_T A$.

The formalized version of Löb’s Theorem lifts this into T:

$$\vdash_T \text{PROV}_T(\ulcorner \text{PROV}_T(\ulcorner A \urcorner) \to A \urcorner) \to \text{PROV}_T(\ulcorner A \urcorner).$$

It is easily derivable from Löb’s Theorem itself:

Lemma 10.1

If T is a consistent, recursively axiomatizable theory in which diag is representable, and $\text{PROV}_T(x)$ satisfies P1–P3, then the following holds for any sentence A in the language of T:

$$\vdash_T \text{PROV}_T(\ulcorner \text{PROV}_T(\ulcorner A \urcorner) \to A \urcorner) \to \text{PROV}_T(\ulcorner A \urcorner).$$

Proof. Using the box notation, we need to show that T proves $\Box(\Box A \to A) \to \Box A$. Call this sentence S. We show $\vdash_T S$ by showing $\vdash_T \Box S \to S$, from which $\vdash_T S$ follows by Löb's Theorem.

1. $\vdash_T \Box(\Box(\Box A \to A) \to \Box A) \to (\Box\Box(\Box A \to A) \to \Box\Box A)$ (P2)
2. $\vdash_T \Box(\Box A \to A) \to \Box\Box(\Box A \to A)$ (P3)
3. $\vdash_T \Box(\Box A \to A) \to (\Box\Box A \to \Box A)$ (P2)
4. $\vdash_T \Box(\Box(\Box A \to A) \to \Box A) \to (\Box(\Box A \to A) \to \Box A)$ (1–3)
5. $\vdash_T \Box(\Box A \to A) \to \Box A$ (4, Löb's Thm.) □

The following theorem confirms that if $\text{PROV}_T(x)$ satisfies P1–P3, and an $\mathfrak{L}_M$-sentence A is provable in the modal logic GL, then $\widetilde{A}^t$ is provable in T.

Theorem 10.3: The Arithmetical Soundness Theorem

If $\text{PROV}_T(x)$ satisfies P1–P3, then for any $\mathfrak{L}_M$-sentence A, if $\vdash_{\text{GL}} A$ then $\vdash_T \widetilde{A}^t$.

Proof. Assume $\vdash_{\text{GL}} A$. This means that there is a sequence $A_1, A_2, \dots, A_n$ of $\mathfrak{L}_M$-sentences such that A_n is A and each A_k in the sequence is either an axiom of GL, an instance of the classical propositional axioms A1–A3, or follows from previous sentences by MP or Nec. We show by induction on k that $\vdash_T \widetilde{A_k}^t$.

If A_k is an instance of A1-A3 then $\widetilde{A_k}^t$ is also an instance of A1-A3, and so $\vdash_T \widetilde{A_k}^t$.

If A_k is an instance of K then $\vdash_T \widetilde{A_k}^t$ by P2,

If A_k is an instance of 4 then $\vdash_T \widetilde{A_k}^t$ by P3.

If A_k is an instance of GL, then $\vdash_T \widetilde{A_k}^t$ by Lemma 10.1.

If A_k follows by MP from A_i and A_j with $i,j < k$, then $\widetilde{A_k}^t$ follows by MP from $\widetilde{A_i}^t$

and $\widetilde{A_j}^{t}$. By induction hypothesis, $\vdash_T \widetilde{A_i}^{t}$ and $\vdash_T \widetilde{A_j}^{t}$. So $\vdash_T \widetilde{A_k}^{t}$.

Assume A_k follows by Nec from A_i with $i < k$. Then $\widetilde{A_k}^{t}$ is $\text{PROV}_T(\ulcorner \widetilde{A_i}^{t} \urcorner)$. By induction hypothesis, $\vdash_T \widetilde{A_i}^{t}$. So $\vdash_T \widetilde{A_k}^{t}$ by P1. □

The converse of Theorem 10.3 also holds: whenever $\vdash_T \widetilde{A}^{t}$, then $\vdash_{\text{GL}} A$. This *arithmetical completeness theorem* was proved by Robert Solovay in 1976. The proof is too hard to get into here.

Theorem 10.3 shows that we can use propositional modal logic to establish results about provability in theories like PA. Solovay's theorem shows that *all* general facts about provability in such theories can be established in this way.

For a simple illustration of how this works, here is a GL-proof of a formalized version of the Second Incompleteness Theorem, showing that if T meets the conditions for the Theorem, then it can prove that if it can prove its own consistency, then it is inconsistent.

1. $\vdash_{GL} \neg\Box\bot \to (\Box\bot \to \bot)$ (propositional logic)
2. $\vdash_{GL} \Box(\neg\Box\bot \to (\Box\bot \to \bot))$ (from 1 by Nec)
3. $\vdash_{GL} \Box\neg\Box\bot \to \Box(\Box\bot \to \bot)$ (from 2 by K)
4. $\vdash_{GL} \Box(\Box\bot \to \bot) \to \Box\bot$ (GL)
5. $\vdash_{GL} \Box\neg\Box\bot \to \Box\bot$ (from 3 and 4)

By Theorem 10.3, it follows that $\vdash_T \text{PROV}_T(\ulcorner \neg\text{PROV}_T(\ulcorner\bot\urcorner)\urcorner) \to \text{PROV}_T(\ulcorner\bot\urcorner)$.

For another illustration, one can show that (for any sentence A)

$$\vdash_{GL} \Box(A \leftrightarrow \Box\neg A) \to \Box(A \leftrightarrow \Box\bot).$$

Among other things, this tells us what theories like PA will make of a sentence N that "says of itself" that its negation is provable. We get such a sentence by applying the Diagonal Lemma to the formula $\text{PROV}_T(\ulcorner\neg x\urcorner)$:

$$\vdash_T N \leftrightarrow \text{PROV}_T(\ulcorner\neg N\urcorner).$$

By P1, this entails

$$\vdash_T \text{PROV}_T(\ulcorner N \leftrightarrow \text{PROV}_T(\ulcorner\neg N\urcorner)\urcorner).$$

The above result from GL gives us

$$\vdash_T \text{PROV}_T(\ulcorner N \leftrightarrow \text{PROV}_T(\ulcorner\neg N\urcorner)\urcorner) \to \text{PROV}_T(\ulcorner N \leftrightarrow \text{PROV}_T(\ulcorner\bot\urcorner)\urcorner).$$

So by Modus Ponens,

$$\vdash_T \text{PROV}_T(\ulcorner N \leftrightarrow \text{PROV}_T(\ulcorner \bot \urcorner)\urcorner).$$

If T is sound, this entails

$$\vdash_T N \leftrightarrow \text{PROV}_T(\ulcorner \bot \urcorner).$$

So the sentence N that says that its negation is provable in PA is equivalent in PA to the statement that PA is inconsistent. Since PA can neither prove nor disprove its own consistency (assuming it is consistent), it can neither prove nor disprove N.

Exercise 10.12 Show that $\vdash_{GL} \Box\neg\Box A \to \Box A$.

Exercise 10.13 Let GL+Ref be the system obtained by adding the axiom schema Ref to GL. Show that $\Box\bot$ is provable in GL+Ref. Is GL+Ref consistent?

Exercise 10.14 Show that $\Box\bot$ is provable in the system GL+5 obtained by adding the axiom schema 5 to GL:

5 $\neg\Box A \to \Box\neg\Box A$.

Exercise 10.15 Show that $\vdash_{GL} (G \leftrightarrow \neg\Box G) \to (G \leftrightarrow \neg\Box\bot)$. (What does this tell us about the Gödel sentence G?)

Exercise 10.16 How should we restrict the Nec rule if we wanted to allow for deductions from premises in our calculus for normal modal logics?

10.4 Chaitin's Incompleteness Theorem

In 1974, Gregory Chaitin proved a version of the First Incompleteness Theorem that connects it to issues of computability. We'll see that it also opens a new route to the Second Incompleteness Theorem.

Recall that a Turing machine takes as input a pattern of strokes and blanks on its tape. Its output, if it halts, is also a pattern of strokes and blanks. If we ignore the blank cells to the left of the first stroke and to the right of the last stroke, all these patterns are finite. Every finite pattern can be produced by a Turing machine. In fact, for every finite pattern of strokes and blanks, there are infinitely many Turing machines that produce it. We might be interested in the most efficient way to generate a given pattern: what is the shortest Turing machine program that produces it (starting on blank tape)?

The *program* of a Turing machine is a string that lists all its instructions – all quintuples like $\langle q_0, B, 1, R, q_1\rangle$ that specify what machine does if it scans a certain symbol in a certain state. We've seen in Section 6.3 that these instructions can themselves be coded as patterns of strokes and blanks and fed to a universal Turing machine, which will execute the program.

Let's define the *complexity* of a pattern as the length of the shortest program that produces it. (This is somewhat imprecise, but the imprecision will do us no harm. The more precise concept is called *Kolmogorov complexity*.)

Some patterns can be produced by very short programs, others require very long programs. As a rule, short and regular patterns can be produced by short programs, while long and disorderly patterns tend to require long programs: they have high complexity.

It's easy to see that there is no limit to how complex a pattern can be. For any number n, only finitely many patterns can be produced by programs of length up to n; all other patterns require longer programs: they have greater complexity.

You'd think that for any number n, we can easily find examples of patterns whose complexity is demonstrably greater than n. Chaitin's Incompleteness Theorem shows that this is not so. In essence, it says that for any recursively axiomatized and consistent theory that knows some basic facts about Turing machines, there is a particular number L such that the theory can't prove of any pattern that its complexity is greater than L.

To state this more precisely, consider the relation H that holds between a Turing machine program M, a pattern of strokes and blanks S, and a number t if M halts with output S after t steps. This relation is decidable. (With a concrete coding of programs and patterns, it can easily be shown to be recursive.) So it is representable in any sufficiently strong arithmetical theory – e.g., in any extension of Q – by some formula $\mathrm{H}(x, y, z)$. Let $\mathrm{LEN}(x, y)$ represent the (obviously computable) function that maps each program to its length. We can now express "the complexity of pattern S exceeds n" in $\mathfrak{L}_A$ as the claim that there is no program x of length $\leq n$ that produces S after some number of steps t: $\forall x \forall y \forall t\, ((\mathrm{LEN}(x, y) \land y \leq \overline{n}) \to \lnot \mathrm{H}(x, \ulcorner S \urcorner, t))$. Abbreviate this as $\mathrm{COMPEXC}(S, n)$. Chaitin's Theorem says that beyond some limit L, no such statement is provable.

Theorem 10.4: Chaitin's Incompleteness Theorem

If T is a consistent, recursively axiomatizable extension of Q, there is a number L such that for any pattern S, T cannot prove that the complexity of S exceeds L: $\nvdash_T \text{COMPEXC}(S, L)$.

Proof sketch. Since T is recursively axiomatizable, we can design a Turing machine M_T that takes a number k as input and searches through all proofs in T until it finds a proof of a sentence of the form $\text{COMPEXC}(S, \overline{k})$; it then outputs S and halts.

We know from section 6.2 that there's a (fairly simple) machine that doubles the number of strokes on the tape. For any number $n > 0$, we can therefore design another machine M_n that (when started on a blank tape)

1. writes a single stroke, then
2. doubles the number of strokes on the tape n times, then
3. moves to the left end of all the strokes, then
4. calls M_T.

This machine writes 2^n strokes on the tape and then calls M_T. If there is a proof in T that the complexity of some pattern S is greater than 2^n, M_n will find some such proof; it will then output S and halt. If there is no such proof, M_n runs forever.

The program for M_n has length $d \cdot n + c$, where d is the length of the doubling machine's program and c is a constant for the length of M_T's program plus whatever is needed for writing the initial stroke and for moving to the left end of a block of strokes.

Since $d \cdot n + c$ grows more slowly than 2^n, there must be a number k such that

$$d \cdot k + c < 2^k.$$

Now suppose for reductio that there is a pattern S for which T can prove that its complexity exceeds $L = 2^k$. Then M_k will output some such S and halt. Since the program for M_k has length $d \cdot k + c$ and outputs S, the complexity of S is at most $d \cdot k + c$, which is less than L.

Since H represents the halting-with-bound relation, and M_k produces S after some number of steps t, T proves $\text{H}(\ulcorner M_k \urcorner, \ulcorner S \urcorner, \overline{t})$. T also knows the length of M_k's program: it can prove $\text{LEN}(\ulcorner M_k \urcorner) = \overline{d \cdot k + c}$. So T proves that there is a program of length $d \cdot k + c$ that produces S within t steps. By assumption, however, T also proves

COMPEXC$(S, \overline{2^k})$, which states that there is no program of length $\leq 2^k$ that produces S after any number of steps. So T is inconsistent. □

A nice aspect of this proof is that it works for any reasonable definition of complexity. (That's why we didn't need to be very precise about this.) It also works for programs in ordinary programming languages like Python or JavaScript, rather than Turing machine programs, and for data structures that aren't just patterns of strokes and blanks. Among other things, it shows that there is a number L such that we can't prove (in, say, ZFC) of any specific string of bits that it requires a Python program longer than L to be produced.

We can estimate the value of L. One would certainly need a lot of instructions to program the machine M_T (in the proof of Theorem 10.4) that takes a number k as input and then searches through all proofs in T until it finds a proof of some statement of the form 'the complexity of S is greater than k'. But one wouldn't need an astronomical number of instructions. 10,000 should be enough. If the doubling machine needs 10 instructions, M_n needs around $10n+10,000$ instructions, each of which has a fixed length. If the average length of these instructions is, say, 10 (never mind how this is measured), $d \cdot k + c$ will be around $100,000 + 100k$. The smallest k with $100,000 + 100k < 2^k$ is $k = 17$. This puts L at $2^{17} = 131,072$. For less cumbersome kinds of programs, L is even smaller. For Python and bit strings, it is well under 10,000.

That's an astonishingly small number. Imagine a 3D movie of the entire observable universe for its first billion years, at atomic resolution. Could you write a Python program that generates this movie, in under 10 KB, without using external resources? Surely not! Chaitin's Incompleteness Theorem implies that while this may be true, it can't be proved.

Exercise 10.17 Show that the complexity function that assigns to any finite pattern of strokes and blanks its complexity is not computable.
(Hint: Assume some machine M computes the complexity function. For any number n, one can then design a machine M_n that goes through all finite patterns until it finds one whose complexity exceeds n, then outputs that string and halts. The program for M_n is not much longer than that of M. Now derive a contradiction.)

Like Gödel's First Incompleteness Theorem, Chaitin's Incompleteness Theorem shows that certain arithmetical truths are unprovable in PA or ZFC In 2010, Shira Kritchman and Ran Raz pointed out that there is also a route from Chaitin's Theorem to the Second Incompleteness Theorem.

We begin with an apparent paradox. There is a finite number of programs of length $\leq L$. Let N be that number plus 1. Each number n between 1 and N can be coded as a pattern of $n + 1$ strokes. Since there are more such numbers than programs of length $\leq L$, at least one of them must require a program longer than L to be printed. How many, exactly, require a program longer than L?

Suppose there is exactly one number x between 1 and N that requires a program longer than L to be printed. In that case, we can find that number. We can run all Turing machines whose program has length $\leq L$. By assumption, at some point, all the $N - 1$ numbers that can be printed by such machines have been printed. We know that there is at least one number between 1 and N that requires a longer program. So we know that the remaining number must be the one that requires a longer program. We have a proof that the complexity of that remaining number is greater than L. But by Chaitin's Incompleteness Theorem, we can't prove this!

This argument seems to show that there can't be a single number between 1 and N that requires a program longer than L to be printed. Well, suppose there are exactly two. In that case, we can find them. We run through all Turing machines with programs of length $\leq L$ until all the $N - 2$ numbers that can be printed by such machines have been printed. We've just shown that there must be at least two numbers between 1 and N that require a program longer than L to be printed. So we can identify these two numbers as the ones that haven't been printed yet. We have therefore proved that the complexity of these two numbers is greater than L. By Chaitin's Incompleteness Theorem, this is impossible.

So there can't be exactly two numbers between 1 and N that require a program longer than L to be printed. Well, suppose there are exactly three…

Continuing this line of thought up to N, we can seemingly show that every number between 1 and N is too complex to be printed with a program of length $\leq L$. But this is evidently false. We can easily show that, say, the number 1 can be printed with a program much shorter than L.

Kritchman and Raz show where the paradoxical argument fails if it is spelled out formally. This requires a theory in which one can formalize the proof of Chaitin's Theorem. Q is too weak for this, but PA is strong enough. More specifically, PA can prove that there is a number L such that PA can't prove of any number x that the complexity of x is greater than L – unless PA is inconsistent, in which case, of course, it proves everything. PA can also prove that there is at least one number between 1 and N whose complexity is greater than L. And it can prove that if there is exactly one such number then PA can find it, by showing that all the other numbers $y < N$ have complexity at most L, and inferring that the remaining number has complexity greater than L. So PA can prove that if there is exactly one number between 1 and N then PA can prove that the complexity of that number is greater than L.

In the argument above, I said that this contradicts Chaitin's Incompleteness Theorem. But Chaitin's Theorem for PA doesn't quite say that PA can't prove that the complexity of x is greater than L. It says that PA can't prove this *if PA is consistent*.

If PA could prove its own consistency, it could use Chaitin's Theorem to conclude that there can't be exactly one number between 1 and N with complexity greater than L. It could similarly show that there can't be exactly two such numbers, and so on. It would reach the conclusion that every number between 1 and N has complexity greater than L, which it can also refute, as it can prove that 1 has complexity less than L. Thus, if PA could prove its own consistency, it would be inconsistent.

10.5 Philosophy of Mind

Gödel's Incompleteness Theorems, and their corollaries, have profound implications for the foundations of mathematics. They derailed Hilbert's program. They draw a wedge between truth and provability, showing that there is no way to capture all mathematical truths in an axiomatic system. Some have argued that Gödel's theorems also have profound implications for our understanding of the human mind: they show that our mathematical capacity goes beyond what any mechanical system (any computer) can achieve.

In its basic form, this argument goes as follows.

Let S be the set of mathematical sentences that I accept as true. S includes the axioms of PA. Let S^+ be the set of sentences entailed by S. If my mind is equivalent to a Turing machine, S is computably enumerable, and S^+ is a computably axiomatizable extension of PA. I can then go through the proof of Gödel's First Incompleteness Theorem to construct a sentence G that is true, but not in S^+. This is impossible: since I realize that G is true, G is part of my mathematical knowledge: it must be in S^+! Hence my mind is not equivalent to a Turing machine.

An initial problem with this argument is that the inference to the truth of G requires the assumption that S^+ is consistent. Gödel's proof shows that if S^+ is axiomatizable *and consistent*, then G is not in S^+. To reach the conclusion that G is true, the argument therefore assumes that the set S^+ includes the statement that S^+ is consistent. Obviously, S^+ is consistent iff S is consistent. So the argument assumes that the set S of my mathematical beliefs includes the statement that this very set is consistent. It's not enough to believe that "my mathematical beliefs are consistent"; I would also have to be aware of the fact that my mathematical beliefs comprise precisely the set S. Thus one way to escape the argument is to hold that we cannot be fully aware of our mathematical beliefs.

There are other problems with the argument. The concepts of mathematical knowledge and belief are surprisingly difficult to model consistently, especially if we want to allow for beliefs about one's own beliefs.

Suppose we extend the language of arithmetic by a predicate B for belief. On its intended interpretation, $B(\ulcorner A \urcorner)$ is meant to express that I accept the sentence A. (We could use special quote marks to refer to sentences, rather than Gödel numbers. This would make no difference.) We could also add a predicate K for knowledge.

We might now want to lay down some axioms for B and K. For example, since knowledge is factive (what is known is true), a minimal theory of mathematical knowledge and belief should include the schema Ref_K:

$$\text{Ref}_K \quad \vdash_T K(\ulcorner A \urcorner) \to A.$$

Suppose such a theory also includes the axioms of Q. In fact, let's simply consider the theory T that adds Ref_K to Q. By the Diagonal Lemma, there is a sentence G such that

$$\vdash_T G \leftrightarrow \neg K(\ulcorner G \urcorner). \tag{D}$$

By Ref_K, $\vdash_T K(\ulcorner G \urcorner) \to G$. Combining this with (D), we get $\vdash_T K(\ulcorner G \urcorner) \to \neg K(\ulcorner G \urcorner)$, which entails $\vdash_T \neg K(\ulcorner G \urcorner)$. By (D) again, we get $\vdash_T G$.

This shows that the sentence G that figures in (D) is derivable from the axioms of Q and an instance of Ref_K. One would think that I could know the axioms of Q and the relevant instance of Ref_K. Going through the above reasoning, I could thereby come to know G. But this is incompatible with T, for we've just seen that $\vdash_T \neg K(\ulcorner G \urcorner)$!

This puzzle is known as the *Knower Paradox*. If we hold fixed classical logic and the factivity of knowledge (as expressed by Ref_K), the only way to avoid contradiction is to deny that I can come to know G by competently going through its proof. Notice that the above argument against the equivalence between my mind and a Turing machine involved a very similar assumption: that I could come to know the Gödel sentence G of S^+ by going through the proof of Gödel's Theorem.

The situation for belief is arguably even worse. If we add B to $\mathfrak{L}_A$, we can construct a formula $\textsc{bc}(x, y)$ saying that there is a code c of a sequence of sentences $A_1, \ldots, A_n$ such that $B(\textsc{entry}(c, i))$ for each $1 \leq i \leq n$, and $\exists y \textsc{prf}_{\text{PA}}(y, \ulcorner A_1 \wedge \ldots \wedge A_n \to A \urcorner)$. That is, $\textsc{bc}(\ulcorner A \urcorner)$ is true iff A is provable in PA from premises that I believe.

Now assume that the set of sentences that I believe is decidable and consistent with PA. (This is surely possible.) Let B^* be the set of sentences that are provable in PA from premises that I believe. B^* is a consistent, axiomatizable extension of PA. Without further assumptions about B, one can show that $\textsc{bc}(x)$ satisfies P1–P3. So Löb's Theorem applies: whenever B^* contains $\textsc{bc}(\ulcorner A \urcorner) \to A$, it also contains $\textsc{bc}(\ulcorner A \urcorner)$.

This is highly counterintuitive. For example, one might think that I could believe that $1+1=3$ does not follow in PA from my mathematical beliefs. (It certainly seems to me as if I believe this!) But if $\neg\textsc{bc}(\ulcorner 1+1=3 \urcorner)$ is in B^* then so is its tautological consequence $\textsc{bc}(\ulcorner 1+1=3 \urcorner) \to 1+1=3$. By Löb's Theorem, it follows that '$1+1=3$' is in B^*. That is, I can only believe that '$1+1=3$' does not follow from my mathematical beliefs if it actually follows from my mathematical beliefs!