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 **??**, 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 **??**, 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 $\times$. 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 $\times$.

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 $\times$?)

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 $\times$. 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 $\quad \forall x \forall y\, (s(x) = s(y) \to x = y)$

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

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

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 $\quad \forall x\,(x \neq 0 \rightarrow \exists y\, x = s(y))$

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 **??**). Still, we can get close by adding the following axiom schema, called the *induction schema*:

Ind $\quad (A(0) \wedge \forall x\,(A(x) \rightarrow A(s(x)))) \rightarrow \forall x\, A(x)$

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

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

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 $A(x) \to A(s(x))$ 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 **??**. 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($\mathfrak{N}$). We'll prove this in ch. **??**. 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:

$$V_0 = \emptyset.$$

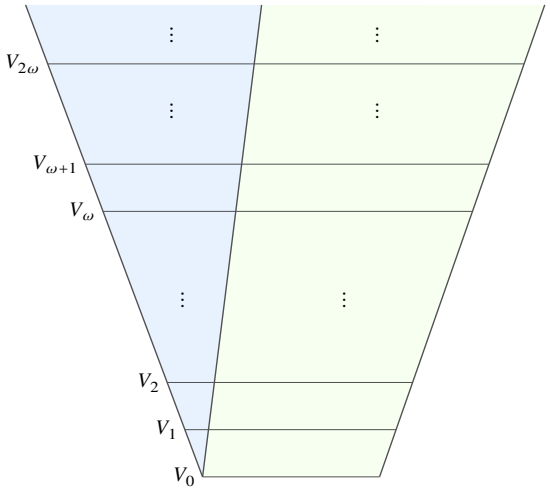
From any stage V_k, we recursively define the next stage V_{k+1} as the set of all sets whose elements are in V_k. This is just the power set of V_k:

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

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

But we don't stop there. After all the stages $V_0, V_1, V_2, \ldots$, 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 $V_0, V_1, V_2, \ldots$ 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, $V_{\omega+1} = \mathcal{P}(V_\omega)$. We then form $V_{\omega+2} = \mathcal{P}(V_{\omega+1})$, and so on. After all the stages $V_\omega, V_{\omega+1}, V_{\omega+2}, \ldots$, 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 $V_\omega, \ldots, V_{\omega\cdot 2}, \ldots, V_{\omega\cdot 3}, \ldots$, 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 $\quad \forall y \exists z \forall x(x \in z \leftrightarrow (x \in y \wedge A(x)))$

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 $\mathcal{P}(t)$. $\bigcup t$ is the union of all sets in t; $\mathcal{P}(t)$ 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 $\mathcal{P}(x)$ 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 $\exists x (\emptyset \in x \wedge \forall y (y \in x \to y \cup \{y\} \in x))$.

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 $\quad \forall x(x \neq \emptyset \to \exists y(y \in x \ \wedge \ x \cap y = \emptyset))$

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 **??**, 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 $\exists x(A(x) \wedge \forall y(A(y) \rightarrow y = x))$,

$\exists! x A(x)$ 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 $\quad \forall v((\forall x(x \in v \rightarrow \exists! y A(x, y)) \rightarrow \exists w \forall x(x \in v \leftrightarrow \exists y(y \in w \wedge A(x, y)))))$

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 $\quad \forall x[\forall z(z \in x \rightarrow z \neq \emptyset) \rightarrow \exists y \forall z(z \in x \rightarrow \exists! v(v \in z \wedge v \in y))]$

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 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 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 '$\forall x A$' becomes '$\forall x(x \in \omega \to A)$' 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 $\quad \forall x\, 0 \neq s(x)$

Under the above translation scheme, this turns into $\forall x(x \in \omega \to (\emptyset \neq x \cup \{x\}))$. 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:

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

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 **??**, 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 $\mathcal{P}(\omega)$ is greater than $\aleph_0$. How much greater? Cantor conjectured, but was unable to prove, that the cardinality of $\mathcal{P}(\omega)$ is $\aleph_1$. Since $\mathcal{P}(\omega)$ 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 **??** 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, $\mathrm{Th}(\mathfrak{N})$, 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 $\mathcal{P}(\omega)$, even more in $\mathcal{P}(\mathcal{P}(\omega))$, 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 $\mathcal{P}(\omega)$ is uncountable by proving that there is no injective function from $\mathcal{P}(\omega)$ 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 $x = \mathcal{P}(\omega)$. In the countable model $\mathfrak{M}$, there may, in fact, be a bijection between the objects denoted by ω and $\mathcal{P}(\omega)$. 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 a "Thesis" rather than a theorem because it does not admit a mathematical proof. (A rigorous proof would first require a mathematically precise definition of 'mechanically computable'.)

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

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

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

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

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

5.3 Uncomputable functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Decidable and semidecidable sets

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

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

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

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

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

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

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

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

Exercise 5.9 Explain why every finite set is decidable.

Exercise 5.10 Are there undecidable sets? Explain briefly.

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

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

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

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

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

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

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

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

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

Exercise 5.13 Explain why every decidable set is computably enumerable.

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

Proposition 5.1: (Kleene's Theorem)

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

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

Proposition 5.2

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

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

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

Proposition 5.2 has a converse:

Proposition 5.3

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

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

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

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

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

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

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

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

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

Proposition 5.4

Every computably axiomatizable first-order theory is computably enumerable.

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

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

Proposition 5.5

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

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

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

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

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

5.5 Coding

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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		

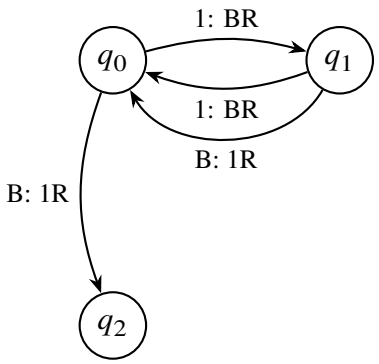
Each cell in the table holds the instruction for what to do in a given state (the row) when reading a given symbol (the column). I've added a third state, q_2, so that the directive to halt can be represented as the directive to go into a new state for which there are no instructions.

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 should 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.

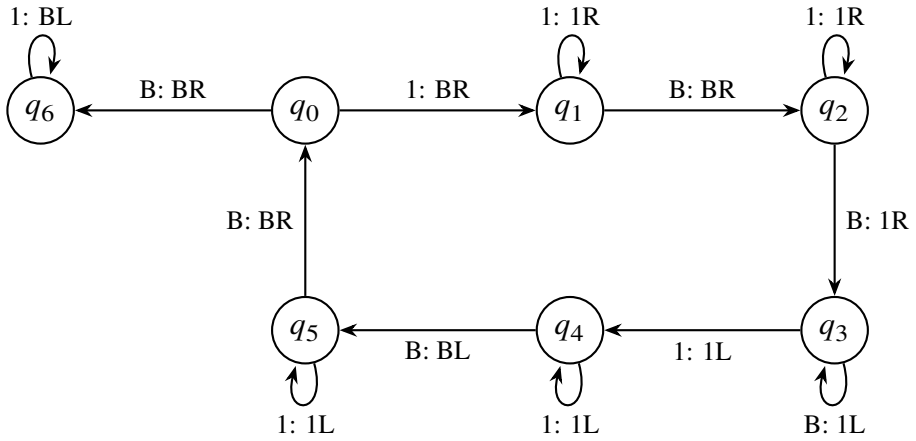
We say that a Turing machine *computes a function f from $\mathbb{N}$ to $\mathbb{N}$* if, whenever it starts on the left-most stroke of a sequence of $n + 1$ strokes on an otherwise blank tape, it eventually halts on a tape with a sequence of $f(n) + 1$ strokes on an otherwise blank tape. A function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

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.

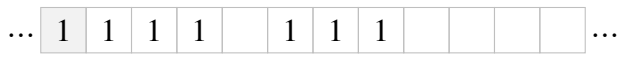
To implement more complex algorithms, it helps to think in terms of subroutines. Let's tackle multiplication. A Turing machine that computes multiplication would start at the left end of a block of $n + 1$ strokes, followed by a blank, followed by another block of $m + 1$ strokes, and eventually halt 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?)

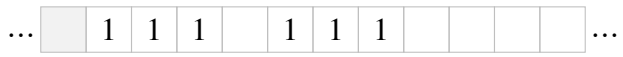
But how can we repeatedly add m strokes to the second block? After i iterations, the first block would have $n + 1 - i$ strokes and the second $i \times m + 1$. It's hard to extract from this the original value m. So here's a better idea: instead of directly adding m strokes to the second block, we insert *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.

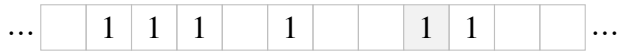
For example, consider the case of $n = 3$ and $m = 2$. The input tape is



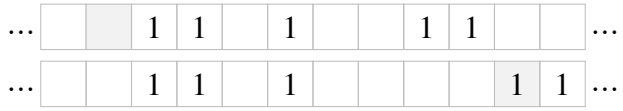
The desired output is a sequence of $3 \times 2 + 1 = 7$ strokes. We begin by erasing the first stroke in the left block (the counter block). We now have



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



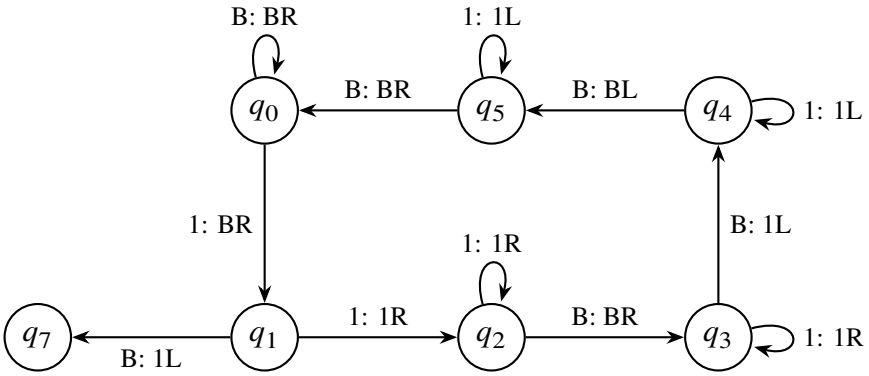
Then we start over, erasing another stroke in the counter block and shifting the two strokes on the right by two more squares:



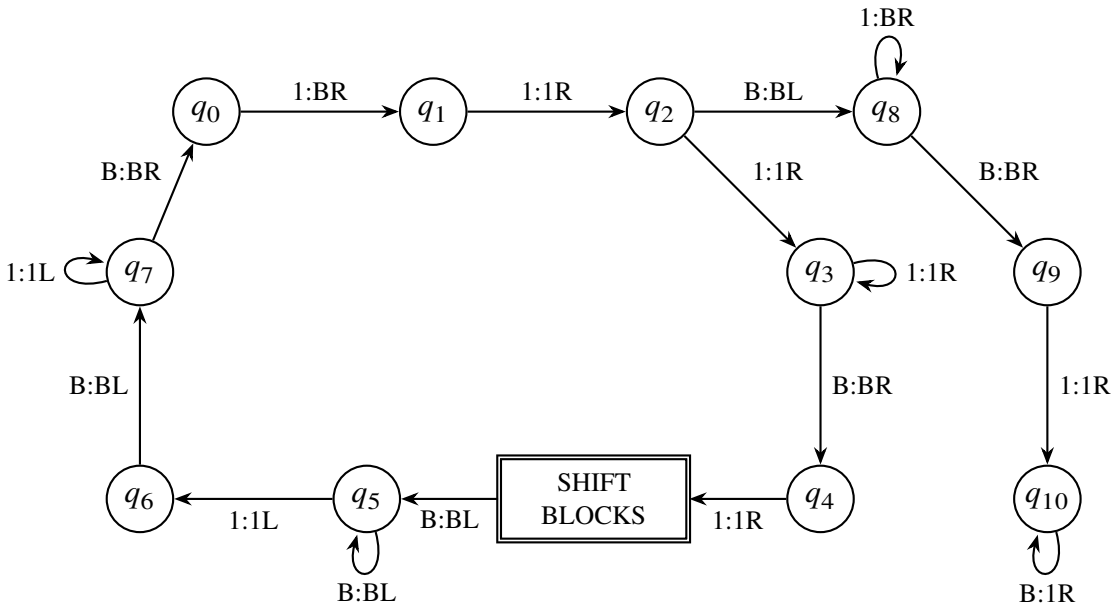
Now there are only two strokes left in the counter. We erase these two strokes and fill in all the blanks we've inserted in the right block:

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 1 B 1 1 B 1 B 1 1 B 1 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, as well as all the even-numbered squares in that area, 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, insert 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 $g(x) = f(x, x)$.

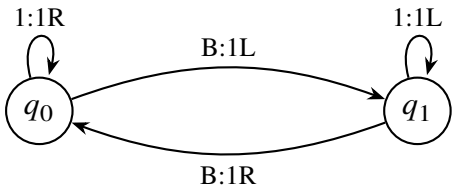
Exercise 6.11 Can the universal Turing machine simulate itself? What happens if you feed U its own machine code as input, together with some further input I for U?

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 use 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.

$$Q_0(0) \wedge @(0, o) \wedge 1(0, o) \wedge 1(0, r(o)) \wedge \forall y(y \neq o \wedge y \neq r(o) \to \neg 1(0, y)).$$

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:

$$\begin{aligned}&\forall x \forall y((Q_0(x) \wedge @(x, y) \wedge 1(x, y)) \to \\ &\quad (Q_1(s(x)) \wedge @(s(x), r(y)) \wedge \neg 1(s(x), y) \wedge \forall z(z \neq y \to (1(s(x), z) \leftrightarrow 1(x, z))))).\end{aligned}$$

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 $\forall x \forall y (s(x) = s(y) \to x = y)$
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 $\forall x \forall y (Q_i(x) \to \neg Q_j(x))$ for $i \neq j$
T6 $\forall x \exists y @(x, y)$
T7 $\forall x \forall y \forall z ((@(x, y) \wedge @(x, z)) \to y = z)$.

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 **??**, 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 on natural numbers. (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 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 $\mathrm{Cn}[f, g_1, \ldots, g_m]$. For example, $\mathrm{Cn}[s, z]$ is the function that takes a number as input, passes it to the zero function and passes the output to the successor function: $\mathrm{Cn}[s, z](x) = s(z(x))$. This is the constant function that always outputs 1.

Exercise 7.1 What is (a) $\mathrm{Cn}[s, s]$? (b) $\mathrm{Cn}[s, \mathrm{Cn}[s, z]]$?

Exercise 7.2 Define the 1-place function that always returns 4, using composition and the base functions.

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 $x + s(y)$.) The algorithm, which resembles the counting-on strategy, could be stated more explicitly as follows:

```
function add(x, y):
    let z = x
    for i from 1 to y:
        z = s(z)
    return z
```

In the most general case, a function h is defined by primitive recursion from two other functions, f and g: f specifies the starting point, $h(x_1, \ldots, x_n, 0)$; g specifies $h(x_1, \ldots, x_n, s(y))$ in terms of $h(x_1, \ldots, x_n, y)$. We allow g to also depend on $x_1, \ldots, x_n$, and y. So the general format for primitive recursion looks like this:

$$\begin{aligned} h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n, s(y)) &= g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)). \end{aligned}$$

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 also write $h = \mathrm{Pr}[f, g]$. For example, addition is $\mathrm{Pr}[\pi^1_1, \mathrm{Cn}[s, \pi^3_3]]$. That's because

$$\begin{aligned} x + 0 &= \pi^1_1(x) = x \\ x + s(y) &= \mathrm{Cn}[s, \pi^3_3](x, y, x + y) = s(x + y). \end{aligned}$$

For another example, consider the *truncated predecessor* function pred that returns the predecessor of a number if that number is positive, and otherwise 0. This can be defined by primitive recursion as follows:

$$\begin{aligned} \mathrm{pred}(0) &= 0 \\ \mathrm{pred}(s(y)) &= y \end{aligned}$$

Here, there are no extra arguments $x_1, \ldots, x_n$. Using pred, we can define a *truncated*

difference function:

$$x \dot{-} 0 = x$$
$$x \dot{-} s(y) = \text{pred}(x \dot{-} y)$$

$x \dot{-} y$ is $x - y$ if $x \geq y$, and 0 otherwise.

Another useful function is *switcheroo* function δ ("delta") that takes every positive integer to 0, and 0 to 1:

$$\delta(0) = 1$$
$$\delta(k + 1) = 0$$

Exercise 7.3 Write the definitions of pred, $\dot{-}$, and δ using the Pr notation.

Exercise 7.4 Define the multiplication function using primitive recursion. Can you express your definition using the Pr notation?

The class of primitive recursive functions is now defined as follows:

Definition 7.1

A function is *primitive recursive* if it can be defined from the base functions s, z, and π_i^n 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 $\text{Pr}[f, g](x_1, \ldots, x_n, y)$, 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 any member of the set to 1 and any 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 can be defined by primitive recursion, as follows:

$$\begin{aligned} \chi(0) &= 0 \\ \chi(s(y)) &= \delta(\chi(y)) \end{aligned}$$

The property of being odd is also primitive recursive. A property is a 1-place relation. So 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.

An example of a 2-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:

$$\mathrm{Leq}(x, y) \;=\; \delta(x \dot{-} y).$$

Another example is the identity relation on $\mathbb{N}$, with the following characteristic function:

$$\mathrm{Eq}(x, y) \;=\; \delta\big((x \dot{-} y) + (y \dot{-} x)\big).$$

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 7.5 Show that the set of even numbers is primitive recursive.

Exercise 7.6 Show that the less-than relation on $\mathbb{N}$ is primitive recursive. You may, if you want, use the functions Leq and Eq.

Exercise 7.7 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 7.8 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. Immediately from exercises 7.7 and 7.8. □

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 $R(x, y)$; the generalization to more arguments is straightforward. 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 $R(x, 0), R(x, 1), \ldots, R(x, k)$ 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:

$$\text{Prime}(x) \text{ iff } 1 < x \wedge \forall y \leq x\, (\text{Divides}(y, x) \rightarrow (y = 1 \vee y = x)).$$

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; '$\wedge$', '$\rightarrow$', '$\vee$', 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:

$$\begin{aligned} \max(x, y) &= x \cdot \text{Leq}(y, x) + y \cdot \delta(\text{Leq}(y, x)) \\ &= x \cdot \delta(x \dot{-} y) + y \cdot \delta(\delta(x \dot{-} y)). \end{aligned}$$

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 7.9 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 7.10 Let $h(x) = \mu y \leq x\,(y + y = x)$. What is $h(0)$? What is $h(3)$?

Proposition 7.5

If $R(x_1, \ldots, x_n, y)$ is a primitive recursive relation, then the function defined by

$$f(x) = \mu y \leq x\, R(x_1, \ldots, x_n, y)$$

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 $f(x, 0), f(x, 1), f(x, 2), \ldots$. At each step $k = 0, 1, 2, \ldots$, $f(x, k)$ 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 powerful 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:

$$\begin{aligned} \mathrm{pri}(0) &= 2, \\ \mathrm{pri}(s(y)) &= \mu x \leq \mathrm{pri}(y)! + 1\ (\mathrm{Prime}(x) \wedge x > \mathrm{pri}(y)). \end{aligned}$$

Here I use Euclid's observation that if p is prime then the next prime is no greater than $p! + 1$.

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, $\mathrm{entry}(2^3 \cdot 3^2 \cdot 5^1 \cdot 7^4, 1) = 2$.

Finally, we can define a primitive recursive function $\mathrm{len}(x)$ that returns the length of the sequence coded by x:

$$\mathrm{len}(x) = \mu y \leq x\ \forall z \leq x\ (z \geq y \to \mathrm{entry}(x, z) = 0).$$

Thus $\mathrm{len}(2^3 \cdot 3^2 \cdot 5^1 \cdot 7^4) = 4$.

Exercise 7.11 What are $\mathrm{entry}(1, 0)$ and $\mathrm{len}(1)$?

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 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 (with, say, a single argument): we start with the base functions, then we list all (one-place) 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. 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 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 al-

lows for this kind of unbounded search. The 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 briefly met such an operation above, in the form of the unbounded minimization operator 'μ': $\mu x\, P(x)$ is the least number x for which $P(x)$ holds. We'll introduce a version of this operation that applies to functions rather than relations.

Given a two-place function $f(x, y)$, we can 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 later.) Then the n-place function h defined by

$$h(x_1, \ldots, x_n) = \mu x f(x_1, \ldots, x_n, x) = 0$$

is 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 $f(x_1, \ldots, x_n, i)$ for each $i = 0, 1, 2, \ldots$ until we find an i for which $f(x_1, \ldots, x_n, i) = 0$:

```
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 $f(x_1, \ldots, x_n, i) = 0$, 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 $f(x_1, \ldots, x_n, y)$ is called *regular* if it is total and for all $x_1, \ldots, x_n$ there is some y such that $f(x_1, \ldots, x_n, y) = 0$. 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 partial 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 $f(x,0), f(x,1), f(x,2), \ldots$: if the computation of $f(x,z)$ never halts, the search never proceeds beyond z. We therefore stipulate that if f is an arbitrary $n+1$-place function, then $\mathrm{Mn}[f]$ is the function that takes n numbers $x_1, \ldots, x_n$ as input and returns the least number y for which

(i) $f(x_1, \ldots, x_n, y) = 0$, and
(ii) $f(x_1, \ldots, x_n, z)$ is defined for all $z < y$.

If there is no such y, $\mathrm{Mn}[f](x_1, \ldots, x_n)$ is undefined.

Exercise 7.12 Let $f(x,y) = x \cdot y$. What is $\mathrm{Mn}[f]$? Is it total? Is it regular? What is $\mathrm{Mn}[\mathrm{Mn}[f]]$?

Exercise 7.13 Consider the function $h(x) = \mu y\,(2y = x)$. What does this function do? Is it total? Is it regular? Can you define h with the Mn notation?

Exercise 7.14 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_i^n 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_i^n 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.

Exercise 7.15 Is the class of recursive relations closed under truth-functional operations?

Exercise 7.16 Can you give an example of a set that is recursive but not primitive 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 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. I write $\bar{x}$ for $x_1, \ldots, x_n$.

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. Suppose we have Turing machines for computing $g_1, \ldots, g_m$ and f. We can design a machine for computing $h = \mathrm{Cn}[f, g_1, \ldots, g_m]$ on any input $\bar{x}$ as follows. The machine first calls each g_i machine (as a subroutine) on input $\bar{x}$, and stores the results next to each other, separated by blanks. It then calls the f machine on this pattern of strokes and blanks. The output is $f(g_1(\bar{x}), \ldots, g_m(\bar{x}))$.

Primitive recursion. Suppose we have Turing machines for computing f and g. Let $h = \mathrm{Pr}[f, g]$. That is, $h(\bar{x}, 0) = f(\bar{x})$ and $h(\bar{x}, y{+}1) = g(\bar{x}, y, h(\bar{x}, y))$. The machine for computing h works as follows. Given input $\bar{x}, y$, it first calls the f-machine on $\bar{x}$ and stores the result in a block that will eventually hold $h(\bar{x}, y)$; 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 $\bar{x}$, 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 $h(\bar{x}, y)$.

Minimization. Suppose we have a Turing machine for computing f. Let $h = \mathrm{Mn}[f]$. that is, $h(\bar{x}) = \mu y\,[f(\bar{x}, y) = 0]$. 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 $\bar{x}$ 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 $f(\bar{x}, y) = 0$ and $f(\bar{x}, z)$ 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 $\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}$ 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:

$$\begin{aligned}\text{conf}(x, 0) &= \text{init}(x)\\ \text{conf}(x, s(y)) &= \text{next}(\text{conf}(x, y)).\end{aligned}$$

The function runs is easily defined from HasRule:

$$\text{runs}(c) = \begin{cases}1 & \text{if HasRule}(\text{entry}(c, 0), \text{entry}(c, 2)),\\ 0 & \text{otherwise.}\end{cases}$$

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:

$$f(x) = \text{out}(\text{conf}(x, \mu y[\text{runs}(\text{conf}(x, y)) = 0])).$$

Equivalently,

$$f = \text{Cn}[\text{Cn}[\text{out}, \text{conf}], \pi^1_1, \text{Mn}[\text{Cn}[\text{runs}, \text{conf}]]].$$

□

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: (Kleene's Normal Form Theorem)

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 recursive functions coincide, it doesn't matter if we state the Church-Turing thesis (Section 5.2) as the claim that every computable function is recursive or as the claim that every computable function is Turing-computable. The two claims are equivalent.

Exercise 7.17 Is the Busy Beaver function (section 6.4) recursive?

Exercise 7.18 Using the Church-Turing thesis, explain why the set of regular recursive functions is not decidable. Is it computably enumerable?

7.5 Feasible computation

I mentioned in section 7.1 that the base functions s, z, and π_i^n 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 Representability

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. We'll also show that these functions and relations are "representable" in moderately strong theories of arithmetic like Q and PA. As foreshadowed at the end of Chapter 5, it will follow that there can be no true, computably 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 $\mathfrak{L}_A$ of arithmetic, with non-logical symbols for 0 ('0'), the successor function ('s'), addition ('+'), and multiplication ('×'). I mentioned that other arithmetical concepts can be defined in terms of these primitives. For example, 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 x is a variable not occurring in t_1 or t_2. This works because the relation $<$ holds between natural numbers a and b iff there is a non-zero number c such that $a + c = b$. The formula $\exists z(x + s(z) = y)$, with free variables x and y, effectively expresses this relation: whenever we replace x and y by terms t_1, t_2 for numbers, the resulting sentence is true (in the standard model of arithmetic $\mathfrak{A}$) iff the number denoted by t_1 is less than the number denoted by t_2.

Every natural number has a canonical term in $\mathfrak{L}_A$, which we call its $\mathfrak{L}_A$-*numeral*. The $\mathfrak{L}_A$-numeral of 0 is '0', the $\mathfrak{L}_A$-numeral of 1 is '$s(0)$', and so on. It will be useful to have an abbreviation. I'll use '$\overline{n}$' as a shorthand for the $\mathfrak{L}_A$-numeral of the number n. So '$\overline{2} + \overline{3} = \overline{5}$' abbreviates '$s(s(0)) + s(s(s(0))) = s(s(s(s(s(0)))))$'. Since every $\mathfrak{L}_A$-term denotes (in $\mathfrak{A}$) a number that is also denoted by an $\mathfrak{L}_A$-numeral $\overline{n}$, we can focus our attention on $\mathfrak{L}_A$-numerals when discussing definability. We'll say that the formula $\exists z(x + s(z) = y)$ *defines* the less-than relation $<$ because for all natural numbers a and b, $a < b$ iff $\exists z(\overline{a} + s(z) = \overline{b})$ is true (in the standard model $\mathfrak{A}$). In general:

Definition 8.1

An $\mathfrak{L}_A$-formula $A(x_1, \ldots, x_n)$ *defines* an n-place relation R on $\mathbb{N}$ iff, for all numbers $a_1, \ldots, a_n$, $A(\overline{a_1}, \ldots, \overline{a_n})$ is true in $\mathfrak{A}$ iff $a_1, \ldots, a_n$ stand in the relation R.

Exercise 8.1 Give two other $\mathfrak{L}_A$-formulas that defines the less-than relation, and explain why they do so.

[Possible exercise: even]

We can also define further functions in $\mathfrak{L}_A$, besides the successor, addition, and multiplication functions. It's important to clarify what kind of definition we are after. In the previous chapter, we spoke of "definitions by primitive recursion". The factorial function, for example, can be recursively defined by the following clauses:

$$0! = 1$$
$$s(x)! = s(x) \times x!.$$

But these clauses don't give us a way to express statements about factorials in $\mathfrak{L}_A$. Ideally, we'd like to find an $\mathfrak{L}_A$-term $f(x)$ so that $f(\overline{a})$ denotes $a!$, for all natural numbers a. There is no such term, however. Still, we can find what in Section **??** I called a "syncategorematic" definition. We can find a formula $F(x, y)$ such that $F(\overline{a}, \overline{b})$ is true (in $\mathfrak{A}$) iff $b = a!$. With the help of this formula, any statements about factorials can be translated into $\mathfrak{L}_A$. For example, we can translate $\forall x(x < x!)$ into

$$\forall x \forall y(F(x, y) \to x < y).$$

Definition 8.2

An $\mathfrak{L}_A$-formula $A(x_1, \ldots, x_n, y)$ *defines* a (total) n-ary function f on $\mathbb{N}$ iff, for all numbers $a_1, \ldots, a_n, b$, $A(\overline{a_1}, \ldots, \overline{a_n}, \overline{b})$ is true in $\mathfrak{A}$ iff $f(a_1, \ldots, a_n) = b$.

(We won't be interested in non-total functions in this chapter.)

We say that a relation or function is *definable* in $\mathfrak{L}_A$ if it is defined by some formula in $\mathfrak{L}_A$.

Exercise 8.2 Give an $\mathfrak{L}_A$-expression 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 are definable in $\mathfrak{L}_A$. Since a relation is recursive iff its characteristic function is recursive, it will follow that all recursive relations are expressible in $\mathfrak{L}_A$.

It's not at all obvious that all recursive functions are definable in $\mathfrak{L}_A$. Try to define the factorial function in terms of $+$, $\times$, s, and 0! We know that a function is recursive iff there is some algorithm for computing the output for any given input. It's surprising that any such algorithm can be built out of algorithms for addition and multiplication.

Consider, for example, the relation that holds between the code number of a Turing machine M, an "input" 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 input n for k steps, and return 'yes' if M has halted by then, and 'no' otherwise. Since all recursive relations are definable in $\mathfrak{L}_A$, there is an $\mathfrak{L}_A$-expression $A(x, y, z)$ such that $A(\overline{m}, \overline{n}, \overline{k})$ is true (in $\mathfrak{A}$) iff the Turing machine coded by m halts on input n within k steps. From this, we quickly get a version of Gödel's incompleteness theorem: $\exists z A(x, y, z)$ is true (in $\mathfrak{A}$) iff the Turing machine coded by x halts on input y. This is an $\mathfrak{L}_A$-expression whose instances can't be decided by any algorithm. So there can be no complete axiomatic theory that decides all its instances.

Gödel's own proof of incompleteness involved a different relation. Remember that proofs from computable axioms are "verifiable". There is an algorithm for checking that something is a proof. We've already seen that algorithms can generally be represented as operating on numbers, with a suitable coding and decoding of the inputs and outputs. We can code each $\mathfrak{L}_A$-sentence and each sequence of $\mathfrak{L}_A$-sentences by a number, called the gn. Since there is an algorithm for checking whether a sequence is a proof of some target sentence, it follows by Church's Thesis that the relation Prf(x,y) is recursive, and so represented by a formula $Prf(x, y)$. We can then construct a formula $Prov(y)$ as $\exists x Prf(x, y)$ that expresses provability. Gödel showed that we can then construct another $\mathfrak{L}_A$-formula G that is true iff $\neg Prov(x)$ is true of G. That is, we'll find an arithmetical sentence G that is true iff it is not provable.

Now suppose G is provable, in some computably axiomatized theory T. Then G is false, as it is true iff it is not provable. So if an arithmetical theory T doesn't prove false sentences, then it can't prove G. In this case, G is true. So T can't prove its negation

either, as the negation is false. So any sound (computably axiomatized) $\mathfrak{L}_A$-theory is incomplete.

These are "semantic" versions of Gödel's incompleteness theorem. They rely on the concept of truth in the standard model of arithmetic, Gödel's original proof was "syntactic". It relied not on definability in $\mathfrak{L}_A$ but on a related concept of representability.

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 H 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

In the previous section, I appealed to the standard model of arithmetic. '+', '×', '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. Just because '$\overline{1} + \overline{2} = \overline{3}$' is true, on the intended interpretation, doesn't mean that an axiomatic theory can prove '$\overline{1} + \overline{2} = \overline{3}$'. The theory axiomatized by the empty set of axioms, for example, can't prove this.

Exercise: exhibit a theory that can prove '$\overline{1} + \overline{2} \neq \overline{3}$'.

The standard axiomatic theory of arithmetic, Peano Arithmetic (PA), can prove '$\overline{1} + \overline{2} = \overline{3}$', Indeed, for any numbers a, b, c if $a + b = c$ then PA contains '$\overline{a} + \overline{b} = \overline{c}$', In that sense, PA "knows" all particular facts about addition: it knows that $1 + 2 = 3$, that $127 + 349 = 476$, and so on.

Similarly, PA knows all particular facts about the less-than relation: whenever $a < b$, PA contains $\overline{a} < \overline{b}$, whenever $a \not< b$, PA contains $\neg(\overline{a} < \overline{b})$. Of course, '<' isn't really part of the language. What I really mean is that whenever $a < b$ then PA contains $\exists z(\overline{a} + s(z) = \overline{b})$, and whenever $a \not< b$ then PA contains $\neg\exists z(\overline{a} + s(z) = \overline{b})$. We say that $\exists z(x + s(z) = y)$ *represents* the less-than relation in PA. In general:

Definition 8.3

An $\mathfrak{L}_A$-formula $A(x_1, \dots, x_n)$ *represents* an n-ary relation R on $\mathbb{N}$ in a theory T iff, for all numbers $a_1, \dots, a_n$,

(i) if R holds of $a_1, \dots, a_n$, then $\vdash_T A(\overline{a_1}, \dots, \overline{a_n})$, and
(ii) if R does not hold of $a_1, \dots, a_n$, then $\vdash_T \neg A(\overline{a_1}, \dots, \overline{a_n})$.

If a relation is represented in a theory T by some formula, we say that it is *representable* in T.

We can also apply this to functions. We'll say that $x + y = z$ represents the addition function in PA. In general:

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) $\vdash_T \forall y(A(\overline{a_1}, \dots, \overline{a_n}, y) \to y = \overline{f(a_1, \dots, a_n)})$.

A more direct analog of condition (ii) for relations would be to require that

(ii′) if $b \neq f(a_1, \dots, a_n)$, then $\vdash_T \neg A(\overline{a_1}, \dots, \overline{a_n}, \overline{b})$.

Our condition (ii) is strictly stronger than this. It effectively requires T to know that $A(x_1, \dots, x_n, y)$ expresses a functional relationship (as discussed in Section **??**).

To illustrate what these clauses do, return to the example of the factorial. What do we need for a formula $F(x, y)$ to represent the factorial function in a theory T? Condition (i) requires that whenever $b = a!$, then $\vdash_T F(\overline{a}, \overline{b})$. This leaves open that T also proves $F(\overline{a}, \overline{c})$ for some $c \neq b$. Indeed, the formula $x = x \wedge y = y$ passes condition (i) in any theory T, But there's no good sense in which this formula represents the factorial function. Condition (ii′) require that if $b \neq a!$ then $\vdash_T \neg F(\overline{a}, \overline{b})$. That is, there must be no other number of which T believes that it is the factorial of a. Our stronger condition (ii) demands that T must know that there is no other such number.

Proposition 8.1

If an $\mathfrak{L}_A$-formula represents a function in a theory $T \subseteq \mathfrak{A}$, then the formula also defines that function.

Proof. Assume that $A(x_1, \ldots, x_n, y)$ represents a function f in a theory T where $T \subseteq \mathfrak{A}$. We have to show that for all numbers $a_1, \ldots, a_n, b$, $A(\overline{a_1}, \ldots, \overline{a_n}, \overline{b})$ is true in $\mathfrak{A}$ iff $f(a_1, \ldots, a_n) = b$. For the 'if' direction, assume that $f(a_1, \ldots, a_n) = b$. By condition (i) in definition 8.4, $\vdash_T A(\overline{a_1}, \ldots, \overline{a_n}, \overline{b})$. Since $T \subseteq \mathfrak{A}$, $A(\overline{a_1}, \ldots, \overline{a_n}, \overline{b})$ is true in $\mathfrak{A}$. For the 'only if' direction, assume that $A(\overline{a_1}, \ldots, \overline{a_n}, \overline{b})$ is true in $\mathfrak{A}$. By condition (ii) for representability, $\vdash_T \forall y(A(\overline{a_1}, \ldots, \overline{a_n}, y) \rightarrow y = \overline{f(a_1, \ldots, a_n)})$. Since $T \subseteq \mathfrak{A}$, $A(\overline{a_1}, \ldots, \overline{a_n}, \overline{b})$ is also true in $\mathfrak{A}$. Therefore, $\overline{b} = \overline{f(a_1, \ldots, a_n)}$ is true in $\mathfrak{A}$, and so $b = f(a_1, \ldots, a_n)$. □

Exercise 8.6 Suppose we swap '+' and '×' everywhere in the Peano axioms. Does $x + y$ still represent addition in the resulting theory? Can you give an expression that does?

In the next two sections, we'll show that all recursive functions are representable in any arithmetical theory that knows some basic facts about arithmetic. Since the formula that *represents* a function in such a theory will also *define* the function in $\mathfrak{L}_A$, it will follow that all recursive functions are definable in $\mathfrak{L}_A$.

As in the case of expressibility, we can focus on functions because the representability of recursive functions entails the representability of recursive relations, in any theory that can prove that $0 \neq 1$.

Proposition 8.2

A relation R is representable in a theory T whenever its characteristic function χ_R is representable in T, provided that $\vdash_T 0 \neq \overline{1}$.

Proof. Assume $A(x_1, \ldots, x_n, y)$ represents the characteristic function χ_R of R in T. This means that whenever R holds of $a_1, \ldots, a_n$, then $\vdash_T A(\overline{a_1}, \ldots, \overline{a_n}, \overline{1})$, by condition (i) for representability of functions. Moreover, whenever R does not hold of $a_1, \ldots, a_n$, then $\vdash_T A(\overline{a_1}, \ldots, \overline{a_n}, 0)$ by condition (i) and $\vdash_T \forall y A(\overline{a_1}, \ldots, \overline{a_n}, y) \rightarrow y = 0$ by condition (ii). Assuming that $\vdash_T 0 \neq \overline{1}$, it follows that $A(x_1, \ldots, x_n, 1)$ represents R in

T. □

Exercise 8.7 Show that the converse is also true: if a relation is representable in a theory, then its characteristic function is representable in that theory.

8.3 Conditions for Representability I

Here's the plan. We'll show that every recursive function can be defined by some $\mathfrak{L}_A$-formula. We'll then ask what an $\mathfrak{L}_A$-theory needs to know in order for that formula to represent the function. Of course, we can't go through the recursive functions one by one: there are infinitely many. Instead, we'll proceed by induction on the construction of recursive functions (as per definition 7.4 in Section 7.3). We'll start with the base functions z (zero), s (successor), and π^n_i (projection). These are easily definable in $\mathfrak{L}_A$, and we'll see that the formulas that define them also represent them, in any theory whatsoever. We'll then show that if a function is constructed from other functions by composition, primitive recursion, or regular minimization, and these other functions are defined in $\mathfrak{L}_A$ by certain formulas, then we can construct an $\mathfrak{L}_A$-formula out of these formulas that defines the new function. We'll see that this new formula represents the new function in any theory that satisfies six conditions.

let's start with the zero function that maps every number a to 0. We need to find a formula $A(x, y)$ so that $A(\overline{a}, \overline{b})$ is true (in $\mathfrak{N}$) iff $b = 0$. Such a formula is not hard to find.

Lemma 8.1

The zero function z is represented in every theory T by the formula $x = x \wedge y = 0$,

Proof. By definition 8.4, $x = x \wedge y = 0$ represents the zero function in a theory T iff, all numbers a,

(i) $\vdash_T \overline{a} = \overline{a} \wedge 0 = 0$, and
(ii) $\vdash_T \forall y((\overline{a} = \overline{a} \wedge y = 0) \rightarrow y = 0)$.

Both of these sentences are logical truths. So $x = x \wedge y = 0$ represents the zero function in every theory T.

Obviously, $x = x \wedge y = 0$ also *defines* the zero function: for any a, b, $\overline{a} = \overline{a} \wedge \overline{b} = 0$ is true (in $\mathfrak{N}$) iff $b = 0$. As I mentioned above, we don't need to check for definability

separately: as long as the theories T in which a formula represents a function include some true theory $T \subseteq \mathfrak{A}$, the formula also defines the function (by Proposition 8.1).

Lemma 8.2
The successor function s is represented in every theory T by the formula $s(x) = y$.

Proof. $s(x) = y$ represents the successor function in a theory T iff, for all numbers a,

(i) $\vdash_T s(\overline{a}) = \overline{s(a)}$, and
(ii) $\vdash_T \forall y(s(\overline{a}) = y \rightarrow y = \overline{s(a)})$.

Since $\overline{s(a)}$ is $s(\overline{a})$, both of these sentences are logical truths. So $s(x) = y$ represents the successor function in any theory T, including $Th(\mathfrak{A})$.

Lemma 8.3
Each projection function π_i^n is represented in every theory T by $x_1 = x_1 \wedge \ldots \wedge x_n = x_n \wedge y = x_i$.

Exercise.

Exercise 8.8 Prove Lemma 8.3.

Now for the closure operations. We start with composition.

Assume that h is the composition of two (one-place) functions f and g, so that $h(x) = f(g(x))$. Assume that f and g are defined by formulas $F(x, y)$ and $G(x, y)$, respectively. We want to find a formula that defines h. Note that $h(x) = y$ iff there is a v such that $g(x) = v$ and $f(v) = y$. So the following formula is a natural candidate:

$$\exists v(G(x, v) \wedge F(v, y)).$$

Lemma 8.4
If an m-place function f is represented in a theory T by a formula $F(x_1, \ldots, x_m, y)$, and m n-place functions $g_1, \ldots, g_m$ are represented in T by formulas and $G_1(x_1, \ldots, x_n, y_1), \ldots, G_m(x_1, \ldots, x_n, y_m)$, respectively, then the composition $h =$

$Cn[f, g_1, \ldots, g_m]$ is represented in T by the formula

$$\exists v_1 \ldots \exists v_m \, (G_1(x_1, \ldots, x_n, v_1) \wedge \ldots \wedge G_m(x_1, \ldots, x_n, v_m) \wedge F(v_1, \ldots, v_m, y)).$$

Proof. I'll give the proof for the case where $n = 1$ and $m = 2$. The proof for arbitrary n and m is analogous.

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

$$\begin{gathered} \vdash_T G_1(\overline{a}, \overline{c_1}) \\ \vdash_T G_2(\overline{a}, \overline{c_2}) \\ \vdash_T F(\overline{c_1}, \overline{c_2}, \overline{b}). \end{gathered}$$

The desired claim follows by the fact that T is closed under first-order consequence.

For condition (ii), we have to show that

$$\vdash_T \forall y (\exists v_1 \exists v_2 (G_1(\overline{a}, v_1) \wedge G_2(\overline{a}, v_2) \wedge F(v_1, v_2, y)) \rightarrow y = \overline{f(a)}).$$

This, too, follows from the representability conditions for F, G_1, and G_2, which yield

$$\begin{gathered} \vdash_T \forall y (G_1(\overline{x}, y) \rightarrow y = \overline{c_1}) \\ \vdash_T \forall y (G_2(\overline{x}, y) \rightarrow y = \overline{c_2}) \\ \vdash_T \forall y (F(c_1, c_2, y) \rightarrow y = \overline{f(a)}). \end{gathered}$$

□

I leave the case of primitive recursion for last: it is by far the hardest. Let's turn to regular minimization. Suppose $h = Mn[f]$, where f is a regular function. This means that for any input number x, h returns the smallest number y for which $f(x, y)$ is 0. Assuming that f is represented in T by some formula $F(x, y)$, 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 $\mathfrak{L}_A$:

$$F(x, y, 0) \wedge \forall z(z < y \rightarrow \neg F(x, z, 0)).$$

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, $\vdash_T \forall x(\overline{a} < x \vee x = \overline{a} \vee x < \overline{a})$.
R2 $\vdash_T \neg\exists x(x < 0)$.
R3 For all $a > 0$, $\vdash_T \forall x(x < \overline{a} \rightarrow (x = 0 \vee \ldots \vee x = \overline{a - 1}))$.
R4$^-$ for all $a > 0, \vdash_T \overline{a} \neq 0$

Officially, of course, '$<$' 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 an n-place regular function f is represented in a theory T by a formula $F(x_1, \ldots, x_n, y)$, and T satisfies the conditions (R1), (R2), (R3), and (R4$^-$), then the minimization $Mn[f]$ of f is represented in T by the formula

$$F(x_1, \ldots, x_n, y, 0) \wedge \forall z(z < y \rightarrow \neg F(x_1, \ldots, x_n, z, 0)).$$

Proof. For readability, I assume that $n = 1$. We have to show that the formula

$$F(x, y, 0) \wedge \forall z(z < y \rightarrow \neg F(x, z, 0))$$

satisfies the two conditions for representing the function $h = Mn[f]$ in T. That is, we have to show that whenever $h(a) = b$, then T can prove:

(i) $F(\overline{a}, \overline{b}, 0) \wedge \forall z(z < \overline{b} \rightarrow \neg F(\overline{a}, z, 0))$.
(ii) $\forall y(F(\overline{a}, y, 0) \wedge \forall z(z < y \rightarrow \neg F(\overline{a}, z, 0)) \rightarrow y = \overline{b})$.

From the fact that $h = Mn[f]$ and $f(a) = b$, we know that $f(a, b) = 0$ and that $f(a, c) \neq 0$ 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

$$\forall y(F(\overline{a},\overline{c},y) \to y = \overline{f(a,c)}), \tag{2}$$

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 $\forall z(z < \overline{b} \to (z = 0 \vee \ldots \vee z = \overline{b-1}))$ whenever $b > 0$. In that case, it follows that 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 that T can derive $y = \overline{b}$ from

$$F(\overline{a},y,0) \wedge \forall z(z < y \to \neg F(\overline{a},z,0)). \tag{4}$$

By (1), T knows that $F(\overline{a},\overline{b},0)$. Together with (4), this implies $\neg(\overline{b} < y)$. From (3) and (4), T knows that $\neg(y < \overline{b})$. By R3, T can infer $y = \overline{b}$. □

Now for the hard part: primitive recursion.

8.4 Conditions for Representability II

Consider the factorial function that maps each number n to $n! = 1 \cdot 2 \cdot 3 \cdots n$. The definition by primitive recursion is simple:

$$0! = 1$$
$$(n+1)! = (n+1) \cdot n!$$

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 $b = a!$.

The trick is to see the recursive definition of $n!$ as defining a sequence: $0!, 1!, 2!, \ldots, n!$. The sequence has $n+1$ elements, whose last element is $n!$. Our formula $F(x,y)$ will say that *y is the $x+1$st element of the sequence* $0!, 1!, \ldots n!$.

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 $\mathrm{Entry}(x,i,y)$ that expresses "y is the i-th entry in the sequence coded by x". We can then define a formula $\mathrm{Fact}_n(x)$ saying that

(i) the first entry in the sequence coded by x is 1, and

(ii) for all $i < n$, the $(i+1)$th entry in the sequence coded by x is the product of the ith entry and $(i+1)$.

In other words, $Fact_n(x)$ will say that x codes the sequence $0!, 1!, \dots, n!$. We can then define $F(x, y)$ as

$$\exists z(\text{fact}_x(z) \wedge \text{Entry}(z, s(x), y)).$$

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.

In Section 7.2, we showed that there is a primitive recursive function $\text{entry}(x, y)$ that returns the exponent of the y-th prime in the prime factorization of x. Now we're looking at an $\mathfrak{L}_A$-formula $\text{Entry}(x, i, y)$ though, and since the construction of entry used primitive recursion, our previous work doesn't help us here. In fact, we won't code sequences of numbers in terms of prime exponents this time.

To explain the coding method we're going to use instead, assume first that we want to code a sequence $a_1, a_2, \dots, a_n$ of numbers all of which are below 9. We could then simply concatenate their decimal representation: $1, 7, 0, 7$ would be coded as 1707. To simplify accessing individual elements of the sequence, we might store the indices of the elements in the code, so that $1, 7, 0, 7$ 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 $1, 7, 0, 7$ becomes 911927930947. The i-th element can be retrieved as the unique digit to the right of '9i' (in decimal representation).

We'll now 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 it 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 $1, 12, 0$, for example, would therefore 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. 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 $\mathfrak{L}_A$.

Our present coding scheme effectively codes sequences of numbers as *pairs* of numbers $\langle p, q \rangle$: q is the actual code; but to retrieve the elements, one also needs to know the base p. We can combine p and q in a single number by using the pairing function that we met way back in Section **??**:

$$\mathrm{J}(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$$

Let L and R be functions that extract the elements of a pair encoded by J, so that $L(J(x, y)) = x$ and $R(J(x, y)) = y$. If we have coded a sequence of numbers $a_1, \ldots, a_n$ by p and q as described above, and packaged these into a single number $c = \mathrm{J}(p, q)$, we can retrieve any element a_i of the doubly coded sequence as

$$\mathrm{beta}(c, i) = \mathrm{alpha}(\mathrm{L}(c), \mathrm{R}(c), i).$$

This is also expressible in $\mathfrak{L}_A$. That is, there is a formula $\mathrm{Beta}(x, i, y)$ that holds of x, i, y iff y is the i-th element of the sequence (doubly) coded by x. This is formula $\mathrm{Entry}(x, i, y)$ that we were looking for.

I'm not going to write down the $\mathfrak{L}_A$-formula $\mathrm{Beta}(x, i, z)$. Instead, I'll show that beta can be constructed by composition and regular minimization from functions and relations that are obviously definable in $\mathfrak{L}_A$ – namely, from addition, multiplication, and identity. These are defined, respectively, by $x + y = z$, $x \times y = z$, and $x = y$. To ensure that they are also *represented* by these formulas 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, $\vdash_T \overline{a} \cdot \overline{b} = \overline{a \cdot b}$.

Note that R4 subsumes R4$^-$.

Lemma 8.6
The identity relation is represented by $x = y$ in every theory T that satisfies R4.

Proof. We need to show that for all a, b:

(i) if $a = b$ then $\vdash_T \overline{a} = \overline{b}$.
(ii) if $a \neq b$ then $\vdash_T \neg(\overline{a} = \overline{b})$.

(i) holds in any theory because $\overline{a}$ and $\overline{b}$ are the same term. (ii) holds by R4. □

Lemma 8.7

The addition function is represented by $x + y = z$ in every theory T that satisfies R5.

Proof. Condition (i) in the definition of representation is given by R5. Condition (ii) is a theorem of the predicate calculus. □

Lemma 8.8

The multiplication function is represented by $x \cdot y = z$ in every theory T that satisfies R6.

Proof. Condition (i) in the definition of representation is given by R6. Condition (ii) is a theorem of the predicate calculus. □

To show that beta can be constructed by composition and regular minimization from these functions, I'll use the following two lemmas.

Lemma 8.9

If some relations (with the same arity) are representable in a theory T then so are all truth-functional combinations of these relations.

Proof. Suppose R_1 and R_2 are n-ary relations and represented in T by A_1 and A_2, respectively. I show that $A_1 \wedge A_2$ represents the conjunction of R_1 and R_2, assuming $n = 1$. We have to show that

(i) if $R_1(a)$ and $R_2(a)$, then $\vdash_T A_1(\overline{a}) \wedge A_2(\overline{a})$;
(ii) if it is not the case that $R_1(a)$ and $R_2(a)$, then $\vdash_T \neg(A_1(\overline{a}) \wedge A_2(\overline{a}))$.

For (i), assume $R_1(a)$ and $R_2(a)$. By the assumption that A_1 and A_2 represent R_1 and R_2, T can prove $A_1(\overline{a})$ and $A_2(\overline{a})$. Then T can also prove their conjunction.

For (ii), assume it is not the case that $R_1(a)$ and $R_2(a)$. By the assumption that A_1 and A_2 represent R_1 and R_2, T can prove $\neg A_1(\overline{a})$ or $\neg A_2(\overline{a})$. Either way, T can prove their disjunction.

The other cases are similarly trivial. □

Lemma 8.10

If an $n+1$-ary relation $R(x_1, \ldots, x_n, y)$ is representable in a theory T, and T satisfies R2, R3, and R5, then for all k, the n-ary relations $\forall y < k\, R(x_1, \ldots, x_n, y)$ and $\exists y < k\, R(x_1, \ldots, x_n, y)$ are representable in T.

Proof. Assume that $R(x_1, \ldots, x_n, y)$ is represented in T by $A(x_1, \ldots, x_n, y)$. For simplicity, I'll assume that $n = 1$. I claim that $\forall y < k\, R(x, y)$ is represented in T by $\forall y(y < \overline{k} \to A(x, y))$, and $\exists y < k\, R(x, y)$ by $\exists y(y < \overline{k} \wedge A(x, y))$.

For the universal quantifier, we have to show that $\forall y(y < \overline{k} \to A(\overline{a}, y))$ satisfies the two representation conditions for relations:

(i) if $\forall y < kR(a, y)$ then $\vdash_T \forall y(y < \overline{k} \to A(\overline{a}, y))$,
(ii) if $\neg\forall y < kR(a, y)$ then $\vdash_T \neg\forall y(y < \overline{k} \to A(\overline{a}, y))$.

For (i), assume $\forall y < kR(a, y)$. We first consider the case where $k > 0$. Since A represents R in T, T can prove $A(\overline{a}, \overline{b})$ for all $b < k$. So T can prove $\forall y(y = \overline{b} \to A(\overline{a}, y))$ for all $b < k$. By R3, T can prove $\forall y(y < \overline{k} \to (y = 0 \vee \ldots \vee y = \overline{k-1}))$. So T can prove $\forall y(y < \overline{k} \to A(\overline{a}, y))$. For $k = 0$, T can prove $\neg\exists y(y < 0)$ by R2, which entails $\forall y(y < 0 \to A(\overline{a}, y))$.

For (ii), assume $\neg\forall y < kR(a, y)$. Then there is some $b < k$ such that $\neg R(a, b)$. Since A represents R, T can prove $\neg A(\overline{a}, \overline{b})$. Pick any c with $k = b + (c + 1)$. By R5, T can prove $\overline{b} + \overline{c + 1} = \overline{k}$. Since T can also prove $\overline{c + 1} = s(\overline{c})$ (because the two sides are the same term), T can prove $\overline{b} + s(\overline{c}) = \overline{k}$ and thus also $\overline{b} < \overline{k}$, which is short for $\exists z(\overline{b} + s(z) = \overline{k})$. Hence T can prove $\exists y(y < \overline{k} \wedge \neg A(\overline{a}, y))$, which is equivalent to $\neg\forall y(y < \overline{k} \to A(\overline{a}, y)$.

The proof for $\exists y < kR(x_1, \ldots, x_n, y)$ is analogous.

The proof of the following lemma shows the construction of beta.

Lemma 8.11: Beta function Lemma

There is a function beta such that for any finite sequence $a_1, \ldots, a_n$ 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 T that satisfies R1–R6.

Proof.

I've explained above how beta works. I'll now show how it can be constructed from addition, multiplication, and identity by composition and regular minimization. (If you pay close attention, you'll see that the construction also involves the projection functions, which we get for free by Proposition **??**.)

As explained above, beta is defined by composition from alpha, L and R.

$$\text{beta}(c, i) = \text{alpha}(\text{L}(c), \text{R}(c), i).$$

L and R can be constructed by minimization and bounded quantification from J and identity:

$$\begin{aligned} \text{L}(q) &= \mu x \, \exists y \leq q (\text{J}(x, y) = q) \\ \text{R}(q) &= \mu y \, \exists x \leq q (\text{J}(x, y) = q) \end{aligned}$$

J can be constructed by composition and regular minimization from addition, multiplication, and identity (and the constants 1 and 2, which can constructed from s and 0):

$$\text{J}(x, y) = \mu z \, (2 \cdot z = (x + y) \cdot (x + y + 1) + 2 \cdot y).$$

It remains to define alpha. Here is a more explicit version of the construction given above:

$$\text{alpha}(p, q, i) = \mu x \, ((p \dot{-} 1) \overset{p}{\frown} i \overset{p}{\frown} x \text{ is part}_p \text{ of } q).$$

'x is part$_p$ of y' means 'the base-p numeral of x is part of the base-p numeral of y', and '$\dot{-}$' is truncated subtraction, which we can construct as follows:

$$\begin{aligned} x \dot{-} y &= \mu z \, ((y < x \rightarrow y + z = x) \wedge (\neg (y < x) \rightarrow z = 0)) \\ x < y &\Leftrightarrow \exists z \leq y (x + s(z) = y). \end{aligned}$$

The part$_p$ relation can be constructed in terms of base-p concatenation:

$$\begin{aligned} x \text{ is part}_p \text{ of } y \Leftrightarrow \exists v \leq y \, \exists w \leq y \, (v \overset{p}{\frown} x \overset{p}{\frown} w = y \vee v \overset{p}{\frown} x = y \\ \vee x \overset{p}{\frown} v = y \vee x = y) \end{aligned}$$

To construct base-p concatenation, we use the function $\eta(p,x)$ that returns the smallest power of p greater than x. That is, $\eta(p,x)$ is the next-greater number after x whose base-p numeral is longer than the base-p numeral of x.

$$\begin{aligned} x \overset{p}{\frown} y &= x \cdot \eta(p,y) + y \\ \eta(p,x) &= \mu y\,((y \text{ is power of prime } p \\ &\qquad \wedge\ y > x\ \wedge\ y > 1)\ \vee\ (\neg(p \text{ is prime})\ \wedge\ y = 0)) \end{aligned}$$

Finally, we construct 'x is power of p' and 'x is prime':

$$\begin{aligned} x \text{ is power of prime } p &\Leftrightarrow x \neq 0\ \wedge\ p \text{ is prime} \\ &\qquad \wedge\ \forall y \leq x\,(y \text{ divides } x \rightarrow y = 1\ \vee\ p \text{ divides } y) \\ x \text{ is prime} &\Leftrightarrow x \neq 0 \text{ and } x \neq 1 \text{ and } \forall y < x\,(y \text{ divides } x \rightarrow y = 1\ \vee\ y = x) \\ x \text{ divides } y &\Leftrightarrow \exists z \leq y\,(y = x \cdot z) \end{aligned}$$

To be clear, this chain of definitions doesn't directly show how to express 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 shows how the beta function can be constructed from simpler functions and relations by composition, regular minimization, truth-functional combination, and bounded quantification. A few simple functions and relations remain undefined: addition, multiplication, identity, projection, and the constants 0, 1, 2. [I guess we should reduce 1, 2 to 0 and addition?] By Lemmas 8.6, 8.7, 8.8, these are representable in any theory T that satisfies R1–R6. By Lemmas 8.4, **??**, 8.9, and 8.10, composition, regular minimization, truth-functional combination, and bounded quantification preserve representability in any theory T that satisfies R1–R6. It follows that beta is representable in any such theory. □

With this, we can finally show that representability is closed under primitive recursion. Assume that $h = Pr[f,g]$, With the help of the beta function lemma, we could construct a formula $S(c,x,k)$ for "c codes the sequence $h(x,0), h(x,1), \ldots, h(x,k)$". We could then define $H(x,y,z)$ as $\exists c(S(c,x,y) \wedge \text{Beta}(c,y,z))$, and show that H represents h in T. But we can take a shortcut:

Lemma 8.12

If h is defined from an f and g by primitive recursion, and f and g are representable in a theory T that satisfies R1–R6, then h is also representable in T.

Proof. Assume that $h = Pr[f, g]$, meaning that

$$\begin{aligned} h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n, n+1) &= g(x_1, \ldots, x_n, n, h(x_1, \ldots, x_n, n)). \end{aligned}$$

For readability, I assume that $n = 1$.

Let Seq be the relation that holds between numbers c, x, k iff c codes the sequence $h(x, 0), \ldots, h(x, k)$. This can be defined from f, g, and beta by boolean combination and bounded quantification:

$$\mathrm{Seq}(c, x, k) = \mathrm{beta}(c, 1) = f(x) \text{ and } \forall i \leq k(\mathrm{beta}(c, i+1) = g(x, i, \mathrm{beta}(c, i))).$$

By the assumption that f and g are representable in T and lemmas 8.11, 8.10 and 8.9, Seq is representable in T. By lemma **??**, so is the function

$$d(x, k) = \mu c \, \mathrm{Seq}(c, x, k)$$

that returns the code of $h(x, 0), \ldots, h(x, k)$. From this, we can define h by composition, projection, and successorhood:

$$h(x, k) = \mathrm{beta}(d(x, k), s(k)).$$

By lemmas 8.2, 8.4 and 8.3, h is representable in T. □

8.5 Putting the pieces together

Theorem 8.1

All total recursive functions are representable in any theory that satisfies R1–R6.

By induction on the definition of total recursive functions, using lemmas **??**, 8.1, 8.7, 8.8, **??**, **??**, and 8.12.

Corollary 8.1

All recursive relations are representable in any theory that satisfies R1–R6.

By the theorem, the lemma on characteristic functions, and the fact that any theory T that satisfies R6 can prove $0 \neq 1$.

Let's reflect on what we've achieved. We've shown that all recursive functions are representable in any theory that knows a few facts about the natural numbers, summarized in R1–R6.

R1 $\vdash_T \neg\exists x(x < 0)$

R2 For all $a > 0$, $\vdash_T \forall x(x < \overline{a} \rightarrow (x = 0 \vee \ldots \vee x = \overline{a-1}))$.

R3 For all a, $\vdash_T \forall x(\overline{a} < x \vee x = \overline{a} \vee x < \overline{a})$.

R4 For all a, b, $\vdash_T \overline{a} + \overline{b} = \overline{a+b}$.

R5 For all a, b, $\vdash_T \overline{a} \cdot \overline{b} = \overline{a \cdot b}$.

R6 For all a, b, if $a \neq b$ then $\vdash_T \overline{a} \neq \overline{b}$.

Each of R1-R6 specifies one or more sentences that must be contained in T. As such, R1-R6 can be seen as an axiomatization of a theory. This minimal theory is not especially elegant, especially if we were to unpack the '<' relation in R1-R3. An elegant alternative that satisfies R1-R6 is the theory Q, also known as Robinson arithmetic.

Theorem 8.2

All recursive functions and relations are representable in Q.

Proof. We need to show that Q satisfies R1–R6.

R1. 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 that $\forall x \neg\exists z(s(z) + x = 0)$.

R2. 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, $y = 0 \vee \ldots \vee y = \overline{a - 1}$, so $x = s(y)$ is one of $1, \ldots, \overline{a}$. Combining both cases, we have $x = 0 \vee \ldots \vee x = \overline{a}$.

R3. We show by induction on a that $\vdash_Q \forall x(\overline{a} < x \vee x = \overline{a} \vee x < \overline{a})$. *Base:* $a = 0$. 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 $0 < x$ by definition. 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 $\exists z(s(z) + 0 = s(z))$. 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 $x < s(\overline{a})$.

R4. 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, $\overline{a} + s(\overline{c}) = s(\overline{a} + \overline{c}) = s(\overline{a + c}) = \overline{a + s(c)} = \overline{a + b}$.

R5. We show by induction on b that for all a, b, $\vdash_Q \overline{a} \cdot \overline{b} = \overline{a \cdot b}$. *Base*: $b = 0$. Then $\overline{a} \cdot \overline{b} = \overline{a \cdot b}$ by Q6. *Induction step:* $b = s(c)$. Then $\overline{a} \cdot s(\overline{c}) = \overline{a} \cdot \overline{c} + \overline{a}$ by Q7, $= \overline{a \cdot c} + \overline{a}$ by induction hypothesis, $= \overline{a \cdot c + a}$ by R4, $= \overline{a \cdot b}$.

R6. We need to 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. □

Theorem 8.3

All recursive functions and relations are representable in PA.

Proof. Immediate from Theorem 8.2 and the fact (Proposition 4.1) that PA extends Q. □

We also have a converse:

Theorem 8.4

If a relation is representable in an axiomatizable and consistent $\mathfrak{L}_A$-theory, then it is recursive.

Proof. Assume $A(x_1, \ldots, x_n)$ represents R in an axiomatizable and consistent $\mathfrak{L}_A$-theory T. This means that if R holds of some numbers $a_1, \ldots, a_n$, then T can prove $A(\overline{a_1}, \ldots, \overline{a_n})$, and if R does not hold of $a_1, \ldots, a_n$, then T can prove $\neg A(\overline{a_1}, \ldots, \overline{a_n})$. Now one can computably enumerate all (codes of) theorems of T. To check whether R holds of some numbers $a_1, \ldots, a_n$, we can wait until either $A(\overline{a_1}, \ldots, \overline{a_n})$ or $\neg A(\overline{a_1}, \ldots, \overline{a_n})$ appears on this list. By a lazy appeal to the Church-Turing thesis, it follows that R is recursive. □

At the beginning of section xx, I asked about definability. I announced that all recursive functions and relations are definable in the language of arithmetic $\mathfrak{L}_A$. Can you see why this holds?

Theorem 8.5

All recursive functions and relations are definable in $\mathfrak{L}_A$.

Proof. All axioms of Q are true in the standard model of arithmetic. It follows that the formulas that represent recursive functions in Q also define those functions. □

We know that r.e. relations result from recursive relations by existential quantification. Now remember that an n-ary relation R is r.e. iff there is an $n-1$-ary recursive relation Q such that $R(a_1, \ldots, a_n)$ iff $\exists y Q(a_1, \ldots, a_n, y)$. It follows that $\mathfrak{L}_A$ can express all r.e. relations. But we also know that not all r.e. relations are recursive. It follows that $\mathfrak{L}_A$ can express more than just the recursive relations.

It follows that the complete truth of $\mathfrak{L}_A$ is not computably axiomatizable! This is an abstract version of Gödel's Incompleteness Theorem.