Computability and Incompleteness

Lecture notes

Jeremy Avigad

Version: January 9, 2007

Contents

Chapter 1

Preliminaries

1.1 Overview

Three themes are developed in this course. The first is *computability*, and its flip side, *uncomputability* or *unsolvability*.

The informal notion of a *computation* as a sequence of steps performed according to some kind of recipe goes back to antiquity. In Euclid, one finds algorithmic procedures for constructing various geometric objects using a compass and straightedge. Throughout the middle ages Chinese and Arabic mathematicians wrote treatises on arithmetic calculations and methods of solving equations and word problems. The word "algorithm" comes from the name "al-Khowârizmi," a mathematician who, around the year 825, wrote such a treatise. It was titled *Hiŝab al-jabr w'al-muqâ-balah*, "science of the reunion and the opposition." The phrase "al-jabr" was also used to describe the procedure of setting broken bones, and is the source of the word *algebra*.

I have just alluded to computations that were intended to be carried out by human beings. But as technology progressed there was also an interest in mechanization. Blaise Pascal built a calculating machine in 1642, and Gottfried Leibniz built a better one a little later in the century. In the early 19th century Charles Babbage designed two grand mechanical computers, the *Difference Engine* and the *Analytic Engine*, and Ada Lovelace wrote some of the earliest computer programs. Alas, the technology of the time was incapable of machining gears fine enough to meet Babbage's specifications.

What is lacking in all these developments is a precise definition of what it means for a function to be *computable*, or for a problem to be *solvable*. For most purposes, this absence did not cause any difficulties; in a sense, computability is similar to the Supreme Court Justice Stewart's character-

ization of pornography, "it may be hard to define precisely, but I know it when I see it." Why, then, is such a definition desirable?

In 1900 the great mathematician David Hilbert addressed the international congress of mathematicians in Paris, and presented a list of 23 problems that he hoped would be solved in the next century. The tenth problem called for a decision procedure for Diophantine equations (a certain type of equation involving integers) or a demonstration that no such procedure exists. Much later in the century this problem was solved in the negative. For this purpose, having a formal model of computability was essential: in order to show that *no* computational procedure can solve Diophantine equations, you have to have a characterization of *all* possible computational procedures. Showing that something *is* computable is easier: you just describe an algorithm, and assume it will be recognized as such. Showing that something is *not* computable needs more conceptual groundwork.

Surprisingly, formal models of computation did not arise until the 1930's, and then, all of a sudden, they shot up like weeds. Turing provided a notion of mechanical computability, Gödel and Herbrand characterized computability in terms of the recursive functions, Church presented the notion of lambda computability, Post offered another notion of mechanical computability, and so on. Today, we can add a number of models to the list, such as computability by register machines, or programmability in any number of programming languages, like ML, C++, or Java.

The astounding fact is that though the various descriptions of computability are quite different, exactly the same functions (say, from numbers to numbers) turn out to be computable in each model. This is one form of evidence that the various definitions capture the intuitive notion of computability. The assertion that this is the case has come to be known as the "Church-Turing thesis."

Incidentally, theoreticians are fond of pointing out that the *theory* of computation predates the invention of the modern computer by about a decade. In 1944, a joint venture between IBM and Harvard produced the "Automatic sequence controlled calculator," and the coming years saw the development of the ENIAC, MANIAC, UNIVAC, and more. Was the theory of computation ahead of its time, or late in coming? Your answer may depend on your perspective.

The second theme developed in this course is the notion of *incompleteness*, and, more generally, the notion of formal mathematical provability.

Mathematical logic has a long and colorful history, but the subject really came of age in the nineteenth century. The first half of the century brought the rigorization of the calculus, providing analysis with a firm mathematical

foundation. In 1879, in a landmark paper called *Begriffsschrift* (concept writing), Frege presented a formal system of logic that included quantifiers and relations, treated much as we treat them today. Frege's goal was a wholesale reduction of mathematics to logic, a topic we will come back to.

Towards the end of the century, mathematicians like Cantor and Dedekind used new and abstract methods to reason about infinitary mathematical objects. This has come to be called the Cantor-Dedekind revolution, and the innovations were controversial at the time. They led to a flurry of work in foundations, aimed at finding both precise descriptions of the new methods and philosophical justifications.

By the end of the century, it was clear that a naive use of Cantor's "set theoretic" methods could lead to paradoxes. (Cantor was well aware of this, and to deal with it developed a vague distinction between various ordinary infinite totalities, and the "absolute" infinite.) In 1902 Russell showed that Frege's formal system was inconsistent, i.e. it led to contradictions as well. These problems led to what is now called the "crisis of foundations," involving rival foundational and methodological stances, and heated debates between their proponents.

Hilbert had a longstanding interest in foundational issues. He was a leading exponent of the new Cantor-Dedekind methods in mathematics, but, at the same time, was sensitive to foundational worries. By the early 1920's he had developed a detailed program to address the foundational crisis. The idea was to represent abstract mathematical reasoning using formal systems of deduction; and then prove, using indubitable, "finitary" methods, that the formal systems are consistent.

Consistency was, however, not the only issue that was important to Hilbert. His writings from the turn of the century suggest that a system of axioms for a mathematical structure, like the natural numbers, is inadequate unless it allows one to derive all true statements about the structure. Combined with his later interest in formal systems of deduction, this suggests that one should try to guarantee that, say, the formal system one is using to reason about the natural numbers is not only consistent, but also *complete*, i.e. every statement is either provable or refutable.

It was exactly these two goals that Gödel shot down in 1931. His first incompleteness theorem shows that there is no complete, consistent, effectively axiomatized formal system for arithmetic. And his second incompleteness theorem shows that no reasonable formal system can prove its own consistency; so, the consistency of "abstract mathematics" cannot even be proved using all of abstract mathematics, much less a safe, finitary portion.

I mentioned above that there are three themes to this course. The first

is "computability" and the second is "incompleteness". There is only one word left in the title of the course: the third theme is the "and".

On the surface, the phrase "computability and incompleteness" is no more coherent than the phrase "French cooking and auto repair." Perhaps that is not entirely fair: the two topics we have discussed share a common emphasis on philosophical and conceptual clarification, of "computation" in the first case, and "proof" in the second. But we will see that the relationship is much deeper than that. Computability is needed to define the notion of an "effectively axiomatized" formal system; after proving his incompleteness theorems in their original form, Gödel needed the theory of computability to restate them as strongly as possible. Furthermore, the methods and tools used in exploring the two subjects overlap a good deal. For example, we will see that the unsolvability of the halting problem can be used to prove Gödel's first incompleteness theorem in an easy way. Finally, the formal analysis of computability helps clarify the foundational issues that gave rise to Hilbert's program, including the constructive view of mathematics.

Before going on, let me emphasize that there *are* prerequisites for this course. The first, and more important one, is some previous background in mathematics. I will assume that you are comfortable with mathematical notation and definitions; and, more crucially, I will assume that you are capable of reading and writing mathematical proofs.

The second prerequisite is some background in formal logic: I will assume that you are familiar with the language of first-order logic and its uses, and that you have worked with at least one deductive system in the past.

In the philosophy department, 80-211 Arguments and Inquiry is designed to meet both needs, but there are many other ways of acquiring the necessary background.

1.2 The set-theoretic view of mathematics

What I am about to describe is the modern understanding of mathematical objects, which is, oddly enough, usually called the "classical" viewpoint.

One starts with basic mathematical objects, like natural numbers, rational numbers, real numbers, points, lines, and triangles. For our purposes, it is best to think of these as fundamental. But nineteenth century mathematicians knew that, for example, the other number systems could be defined "in terms of" the natural numbers, prompting Kronecker's dictum that "God created the natural numbers, everything else is the work of Man." In fact, the modern understanding is that all mathematical objects, including the

natural numbers, can be defined in terms of the single notion of a "set." That is why what I am describing here is also often called the "set-theoretic foundation" of mathematics.

If A is a set and x is some other mathematical object (possibly another set), the relation "x is an element of A" is written $x \in A$. If A and B are sets, A is a subset of B, written $A \subseteq B$, if every element of A is an element of B. A and B are equal, i.e. the same set, if $A \subseteq B$ and $B \subseteq A$. Notice that $A = B$ is equivalent to saying that every element of A is an element of B and vice-versa; so two sets are equal if they have exactly the same elements.

If A and B are sets, $A \cup B$ denotes their union, i.e. the set of things that are in either one, and $A \cap B$ denotes their intersection, i.e. the set of things that are in both. If $\mathcal{A}$ is a collection of sets, $\bigcup \mathcal{A}$ and $\bigcap \mathcal{A}$ denote the union and intersection, respectively, of all the sets in $\mathcal{A}$; if $A_0, A_1, A_2, \ldots$ is a sequence of sets indexed by natural numbers, then $\bigcup_i A_i$ and $\bigcap_i A_i$ denote their union and intersection. There are other ways of building more sets. For example, if A is any set, $\mathcal{P}(A)$, "the power set of A," denotes the set of all subsets of A. The empty set, i.e. the set with no elements, is denoted $\emptyset$.

$\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ denote the sets of natural numbers, rationals, and real numbers respectively. Given a set A, one can describe a subset of A by a property; if P is such a property, the notation

$$\{x \in A \mid P(x)\}$$

is read "the set of all elements of A satisfying P" or "the set of $x \in A$ such that $P(x)$." For example, the set

$$\{x \in \mathbb{N} \mid \text{for some } y \in \mathbb{N},\ x = 2y\}$$

is just a fancy way of describing the set of even numbers. Here are some other examples:

1. $\{x \in \mathbb{N} \mid x \text{ is prime}\}$
2. $\{n \in \mathbb{N} \mid \text{for some nonzero natural numbers } x, y, z,\ x^n + y^n = z^n\}$
3. $\{x \in \mathcal{P}(\mathbb{N}) \mid x \text{ has three elements}\}$

One can also describe a set by listing its elements, as in $\{1, 2\}$. Note that by Fermat's last theorem this is the *same* set as the one described in the second example above, because they have the same elements; but a proof that the different descriptions denote the same set is a major accomplishments of contemporary mathematics. In philosophical terms, this highlights the

difference between a description's *intension*, which is the manner in which it is presented, and its *extension*, which is the object that the description denotes.

One needs to be careful in presenting the rules for forming sets. Russell's paradox amounts to the observation that allowing definitions of the form

$$\{x \mid P(x)\}$$

is inconsistent. For example, it allows us to define the set

$$S = \{x \mid x \notin x\},$$

the set of all sets that are not elements of themselves. The paradox arises from asking whether or not $S \in S$. By definition, if $S \in S$, then $S \notin S$, a contradiction. So $S \notin S$. But then, by definition, $S \in S$. And this is contradictory too.

This is the reason for restricting the set formation property above to elements of a previously formed set A. Note that Russell's paradox also tells us that it is inconsistent to have a "set of all sets." If A were such a thing, then $\{x \in A \mid P(x)\}$ would be no different from $\{x \mid P(x)\}$.

If A and B are sets, $A \times B$, "the cross product of A and B," is the set of all ordered pairs $\langle a, b\rangle$ consisting of an element $a \in A$ and an element $b \in B$. Iterating this gives us notions of ordered triple, quadruple, and so on; for example, one can take $\langle a, b, c\rangle$ to abbreviate $\langle a, \langle b, c\rangle\rangle$. I noted above that on the set-theoretic point of view, everything can be construed as a set. This is true for ordered pairs as well; I will ask you to show, for homework, that if one defines $\langle a, b\rangle$ to be $\{\{a\}, \{a, b\}\}$, the definiendum has the right properties; in particular, $\langle a, b\rangle = \langle c, d\rangle$ if and only if $a = c$ and $b = d$. (It is a further exercise to show that the definition of $A \times B$ can be put in the form $\{x \in C \mid P(x)\}$, where C is constructed using operations, like power-set, described above.) This definition of ordered pairs is due to Kuratowski.

A binary *relation* R on A and B is just a subset of $A \times B$. For example, the relation "divides" on $\{1, 2, 3, 4, 5, 6\} \times \{1, 2, 3, 4, 5, 6\}$ is formally defined to be the set of ordered pairs

$$\{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 1, 3\rangle, \langle 1, 4\rangle, \langle 1, 5\rangle, \langle 1, 6\rangle, \langle 2, 2\rangle, \langle 2, 4\rangle, \\ \langle 2, 6\rangle, \langle 3, 3\rangle, \langle 3, 6\rangle, \langle 4, 4\rangle, \langle 5, 5\rangle, \langle 6, 6\rangle\}.$$

It is convenient to write $R(a, b)$ instead of $\langle a, b\rangle \in R$. Sometimes I will resort to binary notation, aRb, instead of $R(a, b)$. Of course, these considerations can be extended to ternary relations, and so on.

What about functions? If A and B are sets, I will write $f : A \to B$ to denote that f is a function from A to B. One view is that a function is a kind of "black box"; you put an input into the left side of the box, and an output comes out of the right. Another way of thinking about functions is to associate them with "rules" or "procedures" that assign an output to any given input.

The modern conception is that a function from A to B is just a certain type of abstract relationship, or an "arbitrary correspondence" between A and B. More precisely, a function f from A to B is a binary relation R_f on A and B such that

- For every $a \in A$, there is a $b \in B$ such that $R_f(a, b)$
- For every $a \in A$, $b \in B$, and $b' \in B$, if $R_f(a, b)$ and $R_f(a, b')$ then $b = b'$

The first clause says that for every a there is *some* b such that $R_f(a, b)$, while the second clause says there is *at most* one such b. So, the two can be combined by saying that for every a there is *exactly* one b such that $R_f(a, b)$.

Of course, we write $f(a) = b$ instead of $R_f(a, b)$. (Similar considerations hold for binary functions, ternary functions, and so on.) The important thing to keep in mind is that in the official definition, a function is just a set of ordered pairs. The advantage to this definition is that it provides a lot of latitude in defining functions. Essentially, you can use any methods that you use to define sets. According to the recipe above, you can define any set of the form $\{x \in C \mid P(x)\}$, so the challenge is just to find a set C that is big enough and a clearly stated property $P(x)$. For example, consider the function $f : \mathbb{R} \to \mathbb{R}$ defined by

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is irrational} \\ 0 & \text{if } x \text{ is rational} \end{cases}$$

(Try to draw the graph of this!) For nineteenth century mathematicians, it was unclear whether or not the above should be counted as a legitimate "function". But, with our broad definition, it is clear that it should: it is just the set

$$\{\langle x, y\rangle \in \mathbb{R} \times \{0, 1\} \mid x \text{ is rational and } y = 0, \text{ or } x \text{ is irrational and } y = 1\}.$$

In modern terms, we can say that an outcome of foundational investigations of the 1930's is a precise definition of what it means for an "arbitrary"

function from the natural numbers to the natural numbers to be a *computable* function; and the awareness that some very basic, easily definable functions are *not* computable.

Before going on to the next section we need some more definitions. If $f : A \to B$, A is called the *domain* of f, and B is called the *codomain* or *range*. It is important to note that the range of a function is not uniquely determined. For example, if f is the function defined on the natural numbers by $f(x) = 2x$, then f can be viewed in many different ways:

- $f : \mathbb{N} \to \mathbb{N}$
- $f : \mathbb{N} \to \{\text{even numbers}\}$
- $f : \mathbb{N} \to \mathbb{R}$

So writing $f : A \to B$ is a way of specifying which range we have in mind.

Definition 1.2.1 *Suppose f is a function from A to B.*

1. *f is* injective *(or* one-one*) if whenever x and x' are in A and $x \neq x'$, then $f(x) \neq f(x')$*
2. *f is* surjective *(or* onto*) if for every y in B there is an x in A such that $f(x) = y$.*
3. *f is* bijective *(or a* one-to-one correspondence*) if it is injective and surjective.*

I will draw the corresponding picture on the board. If $f : A \to B$, the *image* of f is said to be the set of all $y \in B$ such that for some $x \in A$, $f(x) = y$. So f is surjective if its image is the entire domain.

(For those of you who are familiar with the notion of an inverse function, I will note that f is injective if and only if it has a left inverse, surjective if and only if it has a right inverse, and bijective if and only if it has an inverse.)

Definition 1.2.2 *Suppose f is a function from A to B, and g is a function from B to C. Then the* composition *of g and f, denoted $g \circ f$, is the function from A to C satisfying*

$$g \circ f(x) = g(f(x))$$

for every x in C.

Again, I will draw the corresponding picture on the board. You should think about what the equation above says in terms of the relations R_f and R_g. It is not hard to argue from the basic axioms of set theory that for every such f and g there is a function $g \circ f$ meeting the specification. (So the "definition" has a little "theorem" built in.)

Later in the course we will need to use the notion of a *partial* function.

Definition 1.2.3 *A* partial function *f from A to B is a binary relation R_f on A and B such that for every x in A there is* at most one *y in B such that $R_f(x, y)$.*

Put differently, a *partial function* from A to B is a really a function from some *subset* of A to B. For example, we can consider the following partial functions:

1. $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(x) = \begin{cases} x/2 & \text{if } x \text{ is even} \\ \text{undefined} & \text{otherwise} \end{cases}$$

2. $g : \mathbb{R} \to \mathbb{R}$ defined by

$$g(x) = \begin{cases} \sqrt{x} & \text{if } x \geq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

3. $h : \mathbb{N} \to \mathbb{N}$, where h is not defined for any input.

An ordinary function from A to B is sometimes called a *total* function, to emphasize that it is defined everywhere. But keep in mind that if I just say "function" then, by default, I mean a total function.

1.3 Cardinality

The abstract style of reasoning in mathematics is nicely illustrated by Cantor's theory of cardinality. Later, what has come to be known as Cantor's "diagonal method" will also play a central role in our analysis of computability.

The following definition suggests a sense in which two sets can be said to have the same "size":

Definition 1.3.1 *Two sets A and B are* equipollent *(or* equinumerous*), written $A \approx B$, if there is a bijection from A to B.*

This definition agrees with the usual notion of the size of a finite set (namely, the number of elements), so it can be seen as a way of extending size comparisons to the infinite. The definition has a lot of pleasant properties. For example:

Proposition 1.3.2 *Equipollence is an equivalence relation: for every A, B, and C,*

- $A \approx A$
- *if* $A \approx B$, *then* $B \approx A$
- *if* $A \approx B$ *and* $B \approx C$ *then* $A \approx C$

Definition 1.3.3 *1. A set A is* finite *if it is equinumerous with the set $\{1, \ldots, n\}$, for some natural number n.*

2. A is countably infinite *if it is equinumerous with $\mathbb{N}$.*

3. A is countable *if it is finite or countably infinite.*

(An aside: one can define an ordering $A \preceq B$, which holds if and only if there is an injective map from A to B. Under the axiom of choice, this is a linear ordering. It is true but by no means obvious that if $A \preceq B$ and $B \preceq A$ then $A \approx B$; this is known as the Schröder-Bernstein theorem.)

Here are some examples.

1. The set of even numbers is countably infinite: $f(x) = 2x$ is a bijection from $\mathbb{N}$ to this set.
2. The set of prime numbers is countably infinite: let $f(x)$ be the xth prime number.
3. More generally, as illustrated by the previous example, if A is any subset of the natural numbers, then A is countable. In fact, any subset of a countable set is countable.
4. A set A is countable if and only if there is a surjective function from $\mathbb{N}$ to A. Proof: suppose A is countable. If A is countably infinite, then there is a bijective function from $\mathbb{N}$ to A. Otherwise, A is finite, and there is a bijective function f from $\{1, \ldots, n\}$ to A. Extend f to a surjective function f' from $\mathbb{N}$ to A by defining

$$f'(x) = \begin{cases} f(x) & \text{if } x \in \{1, \ldots, n\} \\ f(1) & \text{otherwise} \end{cases}$$

Conversely, suppose $f : \mathbb{N} \to A$ is a surjective function. If A is finite, we're done. Otherwise, let $g(0)$ be $f(0)$, and for each natural number i, let $g(i+1)$ be $f(k)$, where k is the smallest number such that $f(k)$ is not in the set $\{g(0), g(1), \ldots, g(i)\}$. Then g is a bijection from $\mathbb{N}$ to A.

5. If A and B are countable then so is $A \cup B$.

6. $\mathbb{N} \times \mathbb{N}$ is countable. To see this, draw a table of ordered pairs, and enumerate them by "dovetailing," that is, weaving back and forth. In fact, one can show that the function

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

is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$.

7. $\mathbb{Q}$ is countable. The function f from $\mathbb{N} \times \mathbb{N}$ to the nonnegative rational numbers

$$f(\langle x, y \rangle) = \begin{cases} x/y & \text{if } y \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

is surjective, showing that the set of nonnegative rational numbers is countable. Similarly, the set of negative rational numbers is countable, and hence so is their union.

Theorem 1.3.4 *The set of real numbers is not countable.*

Proof. Let us show that in fact the real interval $[0, 1]$ is not countable. Suppose $f : \mathbb{N} \to [0, 1]$ is any function; it suffices to construct a real number that is not in the range of f. Note that every real number $f(i)$ can be written as a decimal of the form

$$0.a_{i,0}a_{i,1}a_{i,2}\ldots$$

writing 1 as 0.99999. (If $f(i)$ is a terminating decimal, it can also be written as a decimal ending with 9's. For concretness, choose the latter representation.) Now define a new number $0.b_0b_1b_2\ldots$ by making each b_i different from $a_{i,i}$. Specifically, set b_i to be 3 if $a_{i,i}$ is any number other than 3, and 7 otherwise. Then the number $0.b_0b_1b_2\ldots$ is not in the range of $f(i)$, because it differs from $f(i)$ at the ith digit. □

Similar arguments can be used to show that the set of all functions $f : \mathbb{N} \to \mathbb{N}$, and even the set of all functions $f : \mathbb{N} \to \{0, 1\}$ are uncountable. In fact, both these sets have the same cardinality, namely, that of $\mathbb{R}$. Cantor's

continuum hypothesis is that there is no infinite set whose cardinality is strictly greater than that of $\mathbb{N}$, but strictly less than that of $\mathbb{R}$. We now know (thanks to Gödel and Paul Cohen) that whether or not CH is true is independent of the axioms of set theory.

The diagonal argument also shows that for any set A, $\mathcal{P}(A)$ has a cardinality greater than A. So given any set, you can always find one that is bigger.

By the way, pay close attention to the methods of proof, and the manner of presenting proofs, in these notes. For example, the conventional way of proving "if A then B" is to suppose that A is true and show that B follows from this assumption. You will often see proofs by contradiction: to prove that a statement A is true, we can show that the assumption that it is false leads to a contradiction. If you are not entirely comfortable reading and writing such proofs, please talk to me about ways to fill that gap.

Chapter 2

Models of computation

In this chapter we will consider a number of definitions of what it means for a function from $\mathbb{N}$ to $\mathbb{N}$ to be computable. Among the first, and most well known, is the notion of a Turing machine. The beauty of Turing's paper, "On computable numbers," is that he presents not only a formal definition, but also an argument that the definition captures the intuitive notion. (In the paper, Turing focuses on computable real numbers, i.e. real numbers whose decimal expansions are computable; but he notes that it is not hard to adapt his notions to computable functions on the natural numbers, and so on.)

From the definition, it should be clear that any function computable by a Turing machine is computable in the intuitive sense. Turing offers three types of argument that the converse is true, i.e. that any function that we would naturally regard as computable is computable by such a machine. They are (in Turing's words):

1. A direct appeal to intuition.

2. A proof of the equivalence of two definitions (in case the new definition has a greater intuitive appeal).

3. Giving examples of large classes of numbers which are computable.

We will discuss Turing's argument of type 1 in class. Most of this chapter is devoted to filling out 2 and 3. But once we have the definitions in place, we won't be able to resist pausing to discuss Turing's key result, the unsolvability of the halting problem. The issue of unsolvability will remain a central theme throughout this course.

This is a good place to inject an important note: our goal is to try to define the notion of computability "in principle," i.e. without taking into account practical limitations of time and space. Of course, with the broadest definition of computability in place, one can then go on to consider computation with bounded resources; this forms the heart of the subject known as "computational complexity." We may consider complexity issues briefly at the end of this course.

2.1 Turing machines

Turing machines are defined in Chapter 9 of Epstein and Carnielli's textbook. I will draw a picture, and discuss the various features of the definition:

- There is a finite symbol alphabet, including a "blank" symbol.
- There are finitely many states, including a designated "start" state.
- The machine has a two-way infinite tape with discrete cells. Note that "infinite" really means "as big as is needed for the computation"; any halting computation will only have used a finite piece of it.
- There is a finite list of instructions. Each is either of the form "if in state i with symbol j, write symbol k and go to state l" or "if in state i with symbol j, move the tape head right and go to state l" or "if in state i with symbol j, move the tape head left and go to state l."

To start a computation, you put the machine in the start state, with the tape head to the right of a finite string of symbols (on an otherwise blank tape). Then you keep following instructions, until you end up in a state/symbol pair for which no further instruction applies.

The textbook describes Turing machines with only two symbols, 0 and 1; but one can show that with only two symbols, it is possible to simulate machines with more. Similarly, some authors use Turing machines with "one-way" infinite tapes; with some work, one can show how to simulate two way tapes, or even multiple tapes or two-dimensional tapes, etc. Indeed, we will argue that with the Turing machines we have described, it is possible to simulate any mechanical procedure at all.

The book has a standard but clunky notation for describing Turing machine programs. We will use a more convenient type of diagram, which I will describe in class. Roughly, circles with numbers in them represent states. An arrow between states i and l labelled (j, k) stands for the instruction

"if in state i and scanning j, write k and go to state l." "Move right" and "move left" are indicated with arrows, $\rightarrow$ and $\leftarrow$ respectively. This is the notation used in the program *Turing's World*, which allows you to design Turing machines and then watch them run. If you have never played with Turing machines before, I recommend this program to you.

It is easy to design machines that never halt; for example, you can use one state and loop indefinitely. In class, I will go over an example from *Turing's world* called "The Lone Rearranger."

I have described the notion of a Turing machine informally. Now let me present a precise mathematical definition. For starters, if the machine has n states, I will assume that they are numbered $0, \ldots, n-1$, and that 0 is the start state; similarly, it is convenient to assume that the symbols are numbered $0, \ldots, m-1$, where 0 is the "blank" character. For such a Turing machine, it is also convenient to use m to stand for "move left" and $m+1$ for "move right."

Definition 2.1.1 *A* Turing machine *consists of a triple* $\langle n, m, \delta \rangle$ *where*

- n *is a natural number (intuitively, the number of states);*
- m *is a natural number (intuitively, the number of symbols);*
- δ *is a* partial function *from* $\{0, \ldots, n-1\} \times \{0, \ldots, m-1\}$ *to* $\{0, \ldots, m+1\} \times \{0, \ldots, n-1\}$ *(intuitively, the instructions).*

Notice that we are not specifying whether a Turing machine is made of metal or wood, or manufactured by Intel or Motorola; we also have nothing to say about the size, shape, or processor speed. In our account, a Turing machine is an *abstract specification* that can be *instantiated* in many different ways, using physical machines, programs like Turing's world, or even human agents at a blackboard. (In discussing the mind, philosophers sometimes make use of this distinction, and argue that a mind is an abstract object like a Turing machine, and should therefore not be identified with a particular physical instantiation.)

We will have to say what it means for such a machine to "compute" something. The first step is to say what a *configuration* is. Intuitively, a configuration specifies

- the current state,
- the contents of the tape,
- the position of the head.

Since we only care about the position of the tape head relative to the data, it is convenient to replace the last two pieces of information with these three: the symbol under the tape head, the string to the left of the tape head (in reverse order), and the string to the right of the tape head.

Definition 2.1.2 *If M is a Turing machine, a* configuration *of M consists of a 4-tuple $\langle i, j, r, s\rangle$ where*

- *i is a state, i.e. a natural number less than the number of states of M*
- *j is a symbol, i.e. a natural number less than the number of symbols of M*
- *r is a finite string of symbols, $\langle r_0, \ldots, r_k\rangle$*
- *s is a finite string of symbols, $\langle s_0, \ldots, s_l\rangle$*

Now, suppose $\langle i, j, r, s\rangle$ is a configuration of a machine M. I will call this a *halting configuration* if no instruction applies; i.e. the pair $\langle i, j\rangle$ is not in the domain of δ, where δ is machine M's set of instructions. Otherwise, the *configuration after c according to M* is obtained as follows:

- If $\delta(i, j) = \langle k, l\rangle$, where k is a symbol, the desired configuration is $\langle l, k, r, s\rangle$.
- If $\delta(i, j) = \langle m, l\rangle$, a "move left" instruction, the desired configuration is $\langle l, j', r', s'\rangle$, where j' is the first symbol in r, r' is the rest of r, and s' consists of j prepended to s; or, if r is empty, j' is 0, r' is empty, and s' consists of j prepended to s.
- If $\delta(i, j) = \langle m + 1, l\rangle$, a "move right" instruction, the desired configuration is $\langle l, j', r', s'\rangle$, where j' is the first symbol in s, s' is the rest of s, and r' consists of j prepended to r; or, if s is empty, j' is 0, s' is empty, and r' consists of j prepended to r.

Now suppose M is a Turing machine and s is a string of symbols for M (i.e. a sequence of numbers, each less than the number of symbols of M). Then the *start configuration for M with input s* is the configuration $\langle 0, i, \emptyset, s'\rangle$, where i is the first symbol in s, s' is the rest of s, and $\emptyset$ is the empty string. This corresponds to the configuration where the machine is in state 0 and s written on the input tape, with the head at the beginning of the string.

Definition 2.1.3 *Let M be a Turing machine and s a sequence of symbols of M. A* partial computation sequence *for M on input s is a sequence of configurations $c_0, c_1, \ldots, c_k$ such that:*

- *c_0 is the start configuration for M with input s*
- *For each $i < k$, c_{i+1} is the configuration after c_i, according to M.*

A halting computation sequence *for M on input s is a partial computation sequence where the last configuration is a halting configuration. M* halts *on input s if and only if there is a halting computation of M on input s.*

Whew! We are almost done. Suppose we want to compute functions from $\mathbb{N}$ to $\mathbb{N}$. We need to assume that the Turing machine has at least one non-blank symbol (otherwise, it's hard to read the output!), i.e. the number of symbols is at least 2. Like the book, we will use 1^x to denote the string consisting of symbol 1 repeated x times.

Definition 2.1.4 *Let f be a (unary) function from $\mathbb{N}$ to $\mathbb{N}$, and let M be a Turing machine. Then M* computes *f if the following holds: for every natural number x, on input 1^{x+1}, M halts with output $1^{f(x)}$ extending to the right of the tape head.*

More precisely, the final requirement is that if M halts in configuration $\langle i, j, r, s\rangle$ when started on input 1^{x+1}, then j followed by s is a string consisting of $f(x)$ 1's, followed by at least one blank, possibly with other stuff beyond that. Notice that we require an extra 1 on the input string, but not on the output string. These input / output conventions are somewhat arbitrary, but convenient. Note also that we haven't said anything about what the machine does if the input is not in the right format.

More generally, for functions which take more than one argument, we will adopt the convention that these arguments are written sequentially as input, separated by blanks.

Definition 2.1.5 *Let f be a k-ary function from $\mathbb{N}$ to $\mathbb{N}$, and let M be a Turing machine. Then M* computes *f if the following holds: for every sequence of natural numbers $x_0, \ldots, x_{k-1}$, on the input string*

$$1^{x_1+1}, 0, 1^{x_2+1}, 0, \ldots, 0, 1^{x_{k-1}+1}$$

M halts with output $1^{f(x_0,\ldots,x_{k-1})}$ extending to the right of the tape head.

Of course, we can now say that a function from $\mathbb{N}$ to $\mathbb{N}$ is *computable* if there is a Turing machine that computes it. We can also say what it means for a Turing machine to compute a *partial* function. If f is a partial k-ary function from $\mathbb{N}$ to $\mathbb{N}$ and M is a Turing machine, we will say that M computes f is M behaves as above, whenever f is defined for some input; and M does not halt on inputs where f is not defined.

(For some purposes, it is useful to modify this definition so that we can say that every Turing machine computes some partial function. One way to do this is to say that whenever M halts, the output is the longest string of consecutive 1's extending to the right of the input head, ignoring any "junk" that comes afterwards.)

It may seem that we have put entirely too much effort into the formal definition of a Turing machine computation, when you probably had a pretty clear sense of the notion to start with. But, on reflection, our formal development should seem like nothing more than a precise mathematical formulation of your intuitions. The advantage to having a rigorous definition is that now there is no ambiguity as to what we mean by "Turing computable," and we can prove things *about* Turing computability with mathematical precision.

2.2 Some Turing computable functions

Here are some basic examples of computable functions.

Theorem 2.2.1 *The following functions are Turing computable:*

1. $f(x) = x + 1$
2. $g(x, y) = x + y$
3. $h(x, y) = x \times y$

Proof. In fact, it will later be useful to know we have Turing machines computing these functions that never move to the left of the starting point, end up with the head on the same tape cell on which is started, and (contrary to the parenthetical remark above) don't leave any extra nonblank symbols to the right of the output. I will let you think about how to state these requirements formally; the point is that if the Turing machine is started with some stuff to the left of the tape head, it performs the computation leaving the stuff to the left alone.

To compute f, by our input and output conventions, the Turing machine can just halt right away!

To compute g, the following algorithm works:

1. Replace the first 1 by a blank. (This marks the beginning.)
2. Move past the end of the first block of 1's.
3. Print a 1.
4. Move to the end of the second block of 1's.
5. Delete 3 1's, moving backwards.
6. Move back to the first blank, and replace it with a 1.

I will design a Turing machine that does this, in class.

For the third, we need to take an input of $1^{x+1}, 0, 1^{y+1}$ and return an output of 1^{xy}. I will only describe the algorithm in general terms, and let you puzzle over the implementation in the book. The idea is to use the first block of 1's as a counter, to move the second block of 1's (minus 1) over x times; and then fill in the blanks. I will not worry about leaving the output in the starting position; I will leave it to you to make suitable modifications to this effect.

Here is the algorithm:

1. Delete the leftmost 1.
2. If there are no more 1's in the first block (i.e. $x = 0$), delete the second block, and halt.
3. Otherwise, delete the rightmost 1 in the second block. If there are no more 1's (i.e. $y = 0$), erase the first block, and halt.
4. Otherwise, now the string on the tape reads $1^x, 0, 1^y$. Delete a 1 from the left side of the first block.
5. Repeat the following
   (a) Shift the second block y places to the right.
   (b) Delete a 1 from the left side of the first block.

   until the first block is empty.
6. Now the tape head is on a blank (i.e. a 0); to the right of the blank are $(x-1)y$ blanks, followed by y 1's. Fill in the blanks to the right of the tape head with 1's.

This completes (a sketch of) the proof. □

These examples are far from convincing that Turing machines can do anything a Cray supercomputer can do, even setting issues of efficiency aside. Beyond the direct appeal to intuition, Turing suggested two ways of making the case stronger: first, showing that lots more functions can be computed by such machines; and, second, showing that one can simulate other models of computation. For example, many of you would be firmly convinced if we had a mechanical way of compiling C++ source down to Turing machine code!

One way to proceed towards both these ends would be to build up a library of computable functions, as well as build up methods of executing subroutines, passing arguments, and so on. But designing Turing machines with diagrams and lists of 4-tuples can be tedious, so we will take another tack. I will describe another class of functions, the primitive recursive functions, and show that this class is very flexible and robust; and then we will show that every primitive recursive function is Turing computable.

2.3 Primitive recursion

Suppose I specify that a certain function l from $\mathbb{N}$ to $\mathbb{N}$ satisfies the following two clauses:

$$\begin{aligned} l(0) &= 1 \\ l(x+1) &= 2 \cdot l(x). \end{aligned}$$

It is pretty clear that there is only one function, l, that meets these two criteria. This is an instance of a *definition by primitive recursion*. We can define even more fundamental functions like addition and multiplication by

$$\begin{aligned} f(x,0) &= x \\ f(x,y+1) &= f(x,y)+1 \end{aligned}$$

and

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

Exponentiation can also be defined recursively, by

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

We can also compose functions to build more complex ones; for example,

$$\begin{aligned} k(x) &= x^x + (x+3)\cdot x \\ &= f(h(x,x), g(f(x,3),x)). \end{aligned}$$

Remember that the *arity* of a function is the number of arguments. For convenience, I will consider a constant, like 7, to be a 0-ary function. (Send it zero arguments, and it returns 7.) The set of *primitive recursive functions* is the set of functions from $\mathbb{N}$ to $\mathbb{N}$ that you get if you start with 0 and the successor function, $S(x) = x + 1$, and iterate the two operations above, primitive recursion and composition. The idea is that primitive recursive functions are defined in a very straightforward and explicit way, so that it is intuitively clear that each one can be computed using finite means.

We will need to be more precise in our formulation. If f is a k-ary function and $g_0, \ldots, g_{k-1}$ are l-ary functions on the natural numbers, the *composition* of f with $g_0, \ldots, g_{k-1}$ is the l-ary function h defined by

$$h(x_0, \ldots, x_{l-1}) = f(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{k-1}(x_0, \ldots, x_{l-1})).$$

And if $f(z_0, \ldots, z_{k-1})$ is a k-ary function and $g(x, y, z_0, \ldots, z_{k-1})$ is a $k+2$-ary function, then the function defined by *primitive recursion from f and g* is the $k+1$-ary function h, defined by the equations

$$\begin{aligned} h(0, z_0, \ldots, z_{k-1}) &= f(z_0, \ldots, z_{k-1}) \\ h(x+1, z_0, \ldots, z_{k-1}) &= g(x, h(x, z_0, \ldots, z_{k-1}), z_0, \ldots, z_{k-1}) \end{aligned}$$

In addition to the constant, 0, and the successor function, $S(x)$, we will include among primitive recursive functions the projection functions,

$$P_i^n(x_0, \ldots, x_{n-1}) = x_i,$$

for each natural number n and $i < n$. In the end, we have the following:

Definition 2.3.1 *The set of primitive recursive functions is the set of functions of various arities from the set of natural numbers to the set of natural numbers, defined inductively by the following clauses:*

- *The constant, 0, is primitive recursive.*
- *The successor function, S, is primitive recursive.*
- *Each projection function P_i^n is primitive recursive.*

- *If f is a k-ary primitive recursive function and $g_0, \ldots, g_{k-1}$ are l-ary primitive recursive functions, then the composition of f with $g_0, \ldots, g_{k-1}$ is primitive recursive.*
- *If f is a k-ary primitive recursive function and g is a $k+2$-ary primitive recursive function, then the function defined by primitive recursion from f and g is primitive recursive.*

Put more concisely, the set of primitive recursive functions is the smallest set containing the constant 0, the successor function, and projection functions, and closed under composition and primitive recursion.

Another way of describing the set of primitive recursive functions keeps track of the "stage" at which a function enters the set. Let S_0 denote the set of starting functions: zero, successor, and the projections. Once S_i has been defined, let S_{i+1} be the set of all functions you get by applying a single instance of composition or primitive recursion to functions in S_i. Then

$$S = \bigcup_{i \in \mathbb{N}} S_i$$

is the set of primitive recursive functions

Our definition of composition may seem too rigid, since $g_0, \ldots, g_{k-1}$ are all required to have the same arity, l. But adding the projection functions provides the desired flexibility. For example, suppose f and g are ternary functions and h is the binary function defined by

$$h(x, y) = f(x, g(x, x, y), y).$$

Then the definition of h can be rewritten with the projection functions, as

$$h(x, y) = f(P_0^2(x, y), g(P_0^2(x, y), P_0^2(x, y), P_1^2(x, y)), P_1^2(x, y)).$$

Then h is the composition of f with P_0^2, l, P_1^2, where

$$l(x, y) = g(P_0^2(x, y), P_0^2(x, y), P_1^2(x, y)),$$

i.e. l is the composition of g with P_0^2, P_0^2, P_1^2.

For another example, let us consider one of the informal examples given at the beginning of this section, namely, addition. This is described recursively by the following two equations:

$$\begin{aligned} x + 0 &= x \\ x + (y + 1) &= S(x + y). \end{aligned}$$

In other words, addition is the function g defined recursively by the equations

$$\begin{aligned} g(0,x) &= x \\ g(y+1,x) &= S(g(y,x)). \end{aligned}$$

But even this is not a strict primitive recursive definition; we need to put it in the form

$$\begin{aligned} g(0,x) &= k(x) \\ g(y+1,x) &= h(y,g(y,x),x) \end{aligned}$$

for some 1-ary primitive recursive function k and some 3-ary primitive recursive function h. We can take k to be P_0^1, and we can define h using composition,

$$h(y,w,x) = S(P_1^3(y,w,x)).$$

The function h, being the composition of basic primitive recursive functions, is primitive recursive; and hence so is g. (Note that, strictly speaking, we have defined the function $g(y,x)$ meeting the recursive specification of $x+y$; in other words, the variables are in a different order. Luckily, addition is commutative, so here the difference is not important; otherwise, we could define the function g' by

$$g'(x,y) = g(P_1^2(y,x)), P_0^2(y,x)) = g(y,x),$$

using composition.)

As you can see, using the strict definition of primitive recursion is a pain in the neck. I will make you do it once or twice for homework. After that, both you and I can use more lax presentations like the definition of addition above, confident that the details can, in principle, be filled in.

One advantage to having the precise description of the primitive recursive functions is that we can be systematic in describing them. For example, we can assign a "notation" to each such function, as follows. Use symbols 0, S, and P_i^n for zero, successor, and the projections. Now suppose f is defined by composition from a k-ary function h and l-ary functions $g_0, \ldots, g_{k-1}$, and we have assigned notations $H, G_0, \ldots, G_{k-1}$ to the latter functions. Then, using a new symbol $Comp_{k,l}$, we can denote the function f by $Comp_{k,l}[H, G_0, \ldots, G_{k-1}]$. For the functions defined by primitive recursion, we can use analogous notations of the form $Rec_k[G, H]$, where k denotes that arity of the function being defined. With this setup, we can denote the addition function by

$$Rec_2[P_0^1, Comp_{1,3}[S, P_1^3]].$$

Having these notations will prove useful later on.

2.4 Some primitive recursive functions

Here are some examples of primitive recursive functions:

- Constants: for each natural number n, n is a 0-ary primitive recursive function, since it is equal to $S(S(\ldots S(0)))$.
- The identity function: $id(x) = x$, i.e. P_0^1
- Addition, $x + y$
- Multiplication, $x \cdot y$
- Exponentiation, x^y (with 0^0 defined to be 1)
- Factorial, $x!$
- The predecessor function, $pred(x)$, defined by

$$pred(0) = 0, \quad pred(x+1) = x$$

- Truncated subtraction, $x \dot{-} y$, defined by

$$x \dot{-} 0 = x, \quad x \dot{-} (y+1) = pred(x \dot{-} y)$$

- Maximum, $max(x, y)$, defined by

$$\max(x, y) = x + (y \dot{-} x)$$

- Minimum, $min(x, y)$
- Distance between x and y, $|x - y|$

The set of primitive recursive functions is further closed under the following two operations:

- Finite sums: if $f(x, \vec{z})$ is primitive recursive, then so is the function

$$g(y, \vec{z}) \equiv \Sigma_{x=0}^{y} f(x, \vec{z}).$$

- Finite products: if $f(x, \vec{z})$ is primitive recursive, then so is the function

$$h(y, \vec{z}) \equiv \Pi_{x=0}^{y} f(x, \vec{z}).$$

For example, finite sums are defined recursively by the equations

$$g(0, \vec{z}) = f(0, \vec{z}), \quad g(y+1, \vec{z}) = g(y, \vec{z}) + f(y+1, \vec{z}).$$

We can also define boolean operations, where 1 stands for true, and 0 for false:

- Negation, $not(x) = 1 \dot{-} x$
- Conjunction, $and(x, y) = x \cdot y$

Other classical boolean operations like $or(x, y)$ and $implies(x, y)$ can be defined from these in the usual way.

A relation $R(\vec{x})$ is said to be primitive recursive if its characteristic function,

$$\chi_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive. In other words, when one speaks of a primitive recursive relation $R(\vec{x})$, one is referring to a relation of the form $\chi_R(\vec{x}) = 1$, where χ_R is a primitive recursive function which, on any input, returns either 1 or 0. For example, the relation

- $Zero(x)$, which holds if and only if $x = 0$,

corresponds to the function χ_{Zero}, defined using primitive recursion by

$$\chi_{Zero}(0) = 1, \quad \chi_{Zero}(x+1) = 0.$$

It should be clear that one can compose relations with other primitive recursive functions. So the following are also primitive recursive:

- The equality relation, $x = y$, defined by $Zero(|x - y|)$
- The less-than relation, $x \leq y$, defined by $Zero(x \dot{-} y)$

Furthermore, the set of primitive recursive relations is closed under boolean operations:

- Negation, $\neg P$
- Conjunction, $P \wedge Q$
- Disjunction, $P \vee Q$
- Implication $P \rightarrow Q$

One can also define relations using bounded quantification:

- Bounded universal quantification: if $R(x, \vec{z})$ is a primitive recursive relation, then so is the relation

  $$\forall x < y \; R(x, \vec{z})$$

  which holds if and only if $R(x, \vec{z})$ holds for every x less than y.

- Bounded existential quantification: if $R(x, \vec{z})$ is a primitive recursive relation, then so is

  $$\exists x < y \; R(x, \vec{z}).$$

By convention, we take expressions of the form $\forall x < 0 \; R(x, \vec{z})$ to be true (for the trivial reason that there *are* no x less than 0) and $\exists x < 0 \; R(x, \vec{z})$ to be false. A universal quantifier functions just like a finite product; it can also be defined directly by

$$g(0, \vec{z}) = 1, \quad g(y + 1, \vec{z}) = \chi_{and}(g(y, \vec{z}), \chi_R(y, \vec{z})).$$

Bounded existential quantification can similarly be defined using or. Alternatively, it can be defined from bounded universal quantification, using the equivalence, $\exists x < y \; \varphi \equiv \neg \forall x < y \; \neg \varphi$. Note that, for example, a bounded quantifier of the form $\exists x \leq y$ is equivalent to $\exists x < y + 1$.

Another useful primitive recursive function is:

- The conditional function, $cond(x, y, z)$, defined by

  $$cond(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{otherwise} \end{cases}$$

This is defined recursively by

$$cond(0, y, z) = y, \quad cond(x + 1, y, z) = z.$$

One can use this to justify:

- Definition by cases: if $g_0(\vec{x}), \ldots, g_m(\vec{x})$ are functions, and $R_1(\vec{x}), \ldots, R_{m-1}(\vec{x})$ are relations, then the function f defined by

  $$f(\vec{x}) = \begin{cases} g_0(\vec{x}) & \text{if } R_0(\vec{x}) \\ g_1(\vec{x}) & \text{if } R_1(\vec{x}) \text{ and not } R_0(\vec{x}) \\ \vdots & \\ g_{m-1}(\vec{x}) & \text{if } R_{m-1}(\vec{x}) \text{ and none of the previous hold} \\ g_m(\vec{x}) & \text{otherwise} \end{cases}$$

  is also primitive recursive.

When $m = 1$, this is just the the function defined by

$$f(\vec{x}) = cond(\chi_{\neg R_0}(\vec{x}), g_0(\vec{x}), g_1(\vec{x})).$$

For m greater than 1, one can just compose definitions of this form.

We will also make good use of bounded minimization:

- Bounded minimization: if $R(x, \vec{z})$ is primitive recursive, so is the function $f(y, \vec{z})$, written

  $$min\ x < y\ R(x, \vec{z}),$$

  which returns the least x less than y such that $R(x, \vec{z})$ holds, if there is one, and 0 otherwise.

The choice of "0 otherwise" is somewhat arbitrary. It is easier to recursively define a function that returns the least x less than y such that $R(x, \vec{z})$ holds, and y otherwise, and then define *min* from that. As with bounded quantification, $min\ x \leq y\ \ldots$ can be understood as $min\ x < y + 1\ \ldots$.

All this provides us with a good deal of machinery to show that natural functions and relations are primitive recursive. For example, the following are all primitive recursive:

- The relation "x divides y", written $x|y$, defined by

  $$x|y \equiv \exists z \leq y\ (x \cdot z = y).$$

- The relation $Prime(x)$, which asserts that x is prime, defined by

  $$Prime(x) \equiv (x \geq 2 \wedge \forall y \leq x\ (y|x \rightarrow y = 1 \vee y = x)).$$

- The function $nextPrime(x)$, which returns the first prime number larger than x, defined by

  $$nextPrime(x) = min\ y \leq x! + 1(y > x \wedge Prime(y))$$

  Here we are relying on Euclid's proof of the fact that there is always a prime number between x and $x! + 1$.

- The function $p(x)$, returning the xth prime, defined by $p(0) = 2, p(x + 1) = nextPrime(p(x))$. For convenience we will write this as p_x (starting with 0; i.e. $p_0 = 2$).

We have seen that the set of primitive recursive functions is remarkably robust. But we will be able to do even more once we have developed an adequate means of handling *sequences*. I will identify finite sequences of natural numbers with natural numbers, in the following way: the sequence $\langle a_0, a_1, a_2, \ldots, a_k \rangle$ corresponds to the number

$$p_0^{a_0} \cdot p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k+1}.$$

I have added one to the last exponent, to guarantee that, for example, the sequences $\langle 2, 7, 3 \rangle$ and $\langle 2, 7, 3, 0, 0 \rangle$ have distinct numeric codes. I will take both 0 and 1 to code the empty sequence; for concreteness, let $\emptyset$ denote 0. (This coding scheme is slightly different from the one used in the book.)

Let us define the following functions:

- $length(s)$, which returns the length of the sequence s:

$$length(s) = \begin{cases} 0 & \text{if } s = 0 \text{ or } s = 1 \\ min\ i < s\ (p_i | s \wedge \forall j < s & \\ (j > i \rightarrow p_j \not| s)) + 1 & \text{otherwise} \end{cases}$$

  Note that we need to bound the search on i; clearly s provides an acceptable bound.

- $append(s, a)$, which returns the result of appending a to the sequence s:

$$append(s, a) = \begin{cases} 2^{a+1} & \text{if } s = 0 \text{ or } s = 1 \\ \frac{s \cdot p_{length(s)}^{a+1}}{p_{length(s)-1}} & \text{otherwise} \end{cases}$$

  I will leave it to you to check that integer division can also be defined using minimization.

- $element(s, i)$, which returns the ith element of s (where the initial element is called the 0th), or 0 if i is greater than or equal to the length of s:

$$element(s, i) = \begin{cases} 0 & \text{if } i \geq length(s) \\ min\ j < s\ (p_i^{j+1} \not| s) - 1 & \text{if } i + 1 = length(s) \\ min\ j < s\ (p_i^{j+1} \not| s) & \text{otherwise} \end{cases}$$

I will now resort to more common notation for sequences. In particular, I will use $(s)_i$ instead of $element(s, i)$, and $\langle s_0, \ldots, s_k \rangle$ to abbreviate $append(append(\ldots append(\emptyset, s_0) \ldots), s_k)$. Note that if s has length k, the elements of s are $(s)_0, \ldots, (s)_{k-1}$.

It will be useful for us to be able to bound the numeric code of a sequence, in terms of its length and its largest element. Suppose s is a sequence of length k, each element of which is less than equal to some number x. Then s has at most k prime factors, each at most p_{k-1}, and each raised to at most $x+1$ in the prime factorization of s. In other words, if we define

$$\mathit{sequenceBound}(x,k) = p_{k-1}^{k\cdot(x+1)},$$

then the numeric code of the sequence, s, described above, is at most $\mathit{sequenceBound}(x,k)$.

Having such a bound on sequences gives us a way of defining new functions, using bounded search. For example, suppose we want to define the function $\mathit{concat}(s,t)$, which concatenates two sequences. One first option is to define a "helper" function $\mathit{hconcat}(s,t,n)$ which concatenates the first n symbols of t to s. This function can be defined by primitive recursion, as follows:

- $\mathit{hconcat}(s,t,0) = s$
- $\mathit{hconcat}(s,t,n+1) = \mathit{append}(\mathit{hconcat}(s,t,n),(t)_n)$

Then we can define *concat* by

$$\mathit{concat}(s,t) = \mathit{hconcat}(s,t,\mathit{length}(t)).$$

But using bounded search, we can be lazy. All we need to do is write down a primitive recursive *specification* of the object (number) we are looking for, and a bound on how far to look. The following works:

$$\begin{aligned}\mathit{concat}(s,t) = \mathit{min}\ v < \mathit{sequenceBound}(s+t, \mathit{length}(s)+\mathit{length}(t))\\ (\mathit{length}(v) = \mathit{length}(s)+\mathit{length}(t) \wedge\\ \forall i < \mathit{length}(s)\ ((v)_i = (s)_i) \wedge \forall j < \mathit{length}(t)\ ((v)_{\mathit{length}(s)+j} = (t)_j))\end{aligned}$$

I will write $s\hat{}t$ instead of $\mathit{concat}(s,t)$.

Using pairing and sequencing, we can go on to justify more exotic (and useful) forms of primitive recursion. For example, it is often useful to define two more functions simultaneously, such as in the following definition:

$$\begin{aligned} f_0(0,\vec{z}) &= k_0(\vec{z}) \\ f_1(0,\vec{z}) &= k_1(\vec{z}) \\ f_0(x+1,\vec{z}) &= h_0(x, f_0(x,\vec{z}), f_1(x,\vec{z}), \vec{z}) \\ f_1(x+1,\vec{z}) &= h_1(x, f_0(x,\vec{z}), f_1(x,\vec{z}), \vec{z}) \end{aligned}$$

This is an instance of *simultaneous recursion*. Another useful way of defining functions is to give the value of $f(x+1, \vec{z})$ in terms of *all* the values $f(0, \vec{z}), \ldots, f(x, \vec{z})$, as in the following definition:

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

The following schema captures this idea more succinctly:

$$f(x, \vec{z}) = h(x, \langle f(0, \vec{z}), \ldots, f(x-1, \vec{z}) \rangle)$$

with the understanding that the second argument to h is just the empty sequence when x is 0. In either formulation, the idea is that in computing the "successor step," the function f can make use of the entire sequence of values computed so far. This is known as a *course-of-values* recursion. For a particular example, it can be used to justify the following type of definition:

$$f(x, \vec{z}) = \begin{cases} h(x, f(k(x, \vec{z}), \vec{z}), \vec{z}) & \text{if } k(x, \vec{z}) < x \\ g(x, \vec{z}) & \text{otherwise} \end{cases}$$

In other words, the value of f at x can be computed in terms of the value of f at *any* previous value, given by k.

You should think about how to obtain these functions using ordinary primitive recursion. One final version of primitive recursion is more flexible in that one is allowed to change the *parameters* (side values) along the way:

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

This, too, can be simulated with ordinary primitive recursion. (Doing so is tricky. For a hint, try unwinding the computation by hand.)

Finally, notice that we can always extend our "universe" by defining additional objects in terms of the natural numbers, and defining primitive recursive functions that operate on them. For example, we can take an integer to be given by a pair $\langle m, n \rangle$ of natural numbers, which, intuitively, represents the integer $m - n$. In other words, we say

$$\mathit{Integer}(x) \equiv \mathit{length}(x) = 2$$

and then we define the following:

- $\mathit{iequal}(x, y)$

- $iplus(x, y)$
- $iminus(x, y)$
- $itimes(x, y)$

Similarly, we can define a rational number to be a pair $\langle x, y \rangle$ of integers with $y \neq 0$, representing the value x/y. And we can define *qequal*, *qplus*, *qminus*, *qtimes*, *qdivides*, and so on.

2.5 The recursive functions

We have seen that lots of functions are primitive recursive. Can we possibly have captured all the computable functions?

A moment's consideration shows that the answer is no. It should be intuitively clear that we can make a list of all the unary primitive recursive functions, $f_0, f_1, f_2, \ldots$ such that we can effectively compute the value of f_x on input y; in other words, the function $g(x, y)$, defined by

$$g(x, y) = f_x(y)$$

is computable. But then so is the function

$$\begin{aligned} h(x) &= g(x, x) + 1 \\ &= f_x(x) + 1. \end{aligned}$$

For each primitive recursive function f_i, the value of h and f_i differ at i. So h is computable, but not primitive recursive; and one can say the same about g. This is a an "effective" version of Cantor's diagonalization argument.

(One can provide more explicit examples of computable functions that are not primitive recursive. For example, let the notation $g^n(x)$ denote $g(g(\ldots g(x)))$, with n g's in all; and define a sequence $g_0, g_1, \ldots$ of functions by

$$\begin{aligned} g_0(x) &= x + 1 \\ g_{n+1}(x) &= g_n^x(x) \end{aligned}$$

You can confirm that each function g_n is primitive recursive. Each successive function grows much faster than the one before; $g_1(x)$ is equal to $2x$, $g_2(x)$ is equal to $2^x \cdot x$, and $g_3(x)$ grows roughly like an exponential stack of x 2's. Ackermann's function is essentially the function $G(x) = g_x(x)$, and one can show that this grows faster than any primitive recursive function.)

Let me come back to this issue of enumerating the primitive recursive functions. Remember that we have assigned symbolic notations to each primitive recursive function; so it suffices to enumerate notations. We can assign a natural number $\#(F)$ to each notation F, recursively, as follows:

$$\begin{aligned}
\#(0) &= \langle 0 \rangle \\
\#(S) &= \langle 1 \rangle \\
\#(P_i^n) &= \langle 2, n, i \rangle \\
\#(Comp_{k,l}[H, G_0, \ldots, G_{k-1}]) &= \langle 3, k, l, \#(H), \#(G_0), \ldots, \#(G_{k-1}) \rangle \\
\#(Rec_l[G, H]) &= \langle 4, l, \#(G), \#(H) \rangle
\end{aligned}$$

Here I am using the fact that every sequence of numbers can be viewed as a natural number, using the codes from the last section. The upshot is that every code is assigned a natural number. Of course, some sequences (and hence some numbers) do not correspond to notations; but we can let f_i be the unary primitive recursive function with notation coded as i, if i codes such a notation; and the constant 0 function otherwise. The net result is that we have an explicit way of enumerating the unary primitive recursive functions.

(In fact, some functions, like the constant zero function, will appear more than once on the list. This is not just an artifact of our coding, but also a result of the fact that the constant zero function has more than one notation. We will later see that one can not computably avoid these repetitions; for example, there is no computable function that decides whether or not a given notation represents the constant zero function.)

We can now take the function $g(x, y)$ to be given by $f_x(y)$, where f_x refers to the enumeration I have just described. How do we know that $g(x, y)$ is computable? Intuitively, this is clear: to compute $g(x, y)$, first "unpack" x, and see if it a notation for a unary function; if it is, compute the value of that function on input y. Many of you will be convinced that (with some work!) one can write a program in C++ that does this; and now we can appeal to the Church-Turing thesis, which says that anything that, intuitively, is computable can be computed by a Turing machine.

Of course, a more direct way to show that $g(x, y)$ is computable is to describe a Turing machine that computes it, explicitly. This would, in particular, avoid the Church-Turing thesis and appeals to intuition. But, as noted above, working with Turing machines directly is unpleasant. Soon we will have built up enough machinery to show that $g(x, y)$ is computable, appealing to a model of computation that can be *simulated* on a Turing machine: namely, the recursive functions.

To motivate the definition of the recursive functions, note that our proof that there are computable functions that are not primitive recursive actually establishes much more. The argument was very simple: all we used was the fact was that it is possible to enumerate functions $f_0, f_1, \ldots$ such that, as a function of x and y, $f_x(y)$ is computable. So the argument applies to *any class of functions that can be enumerated in such a way.* This puts us in a bind: we would like to describe the computable functions explicitly; but any explicit description of a collection of computable functions cannot be exhaustive!

The way out is to allow *partial* functions to come into play. We will see that it *is* possible to enumerate the partial Turing computable functions; in fact, we already pretty much know that this is the case, since it is possible to enumerate Turing machines in a systematic way. We will come back to our diagonal argument later, and explore why it does not go through when partial functions are included.

The question is now this: what do we need to add to the primitive recursive functions to obtain all the partial recursive functions? We need to do two things:

1. Modify our definition of the primitive recursive functions to allow for partial functions as well.

2. *Add* something to the definition, so that some new partial functions are included.

The first is easy. As before, we will start with zero, successor, and projections, and close under composition and primitive recursion. The only difference is that we have to modify the definitions of composition and primitive recursion to allow for the possibility that some of the terms in the definition are not defined. If f and g are partial functions, I will write $f(x) \downarrow$ to mean that f is defined at x, i.e. x is in the domain of f; and $f(x) \uparrow$ to mean the opposite, i.e. that f is not defined at x. I will use $f(x) \simeq g(x)$ to mean that either $f(x)$ and $g(x)$ are both undefined, or they are both defined and equal. We will these notations for more complicated terms, as well. We will adopt the convention that if h and $g_0, \ldots, g_k$ are all partial functions, then

$$h(g_0(\vec{x}), \ldots, g_k(\vec{x}))$$

is defined if and only if each g_i is defined at $\vec{x}$, and h is defined at $g_0(\vec{x}), \ldots, g_k(\vec{x})$. With this understanding, the definitions of composition and primitive recursion for partial functions is just as above, except that we have to replace "$=$" by "$\simeq$".

What we will add to the definition of the primitive recursive functions to obtain partial functions is the *unbounded search operator*. If $f(x, \vec{z})$ is any partial function on the natural numbers, define $\mu x\ f(x, \vec{z})$ to be

> the least x such that $f(0, \vec{z}), f(1, \vec{z}), \ldots, f(x, \vec{z})$ are all defined, and $f(x, \vec{z}) = 0$, if such an x exists

with the understanding that $\mu x\ f(x, \vec{z})$ is undefined otherwise. This defines $\mu x\ f(x, \vec{z})$ uniquely.

Note that our definition makes no reference to Turing machines, or algorithms, or any specific computational model. But like composition and primitive recursion, there is an operational, computational intuition behind unbounded search. Remember that when it comes to the Turing computability of a partial function, arguments where the function is undefined correspond to inputs for which the computation does not halt. The procedure for computing $\mu x\ f(x, \vec{z})$ will amount to this: compute $f(0, \vec{z}), f(1, \vec{z}), f(2, \vec{z})$ until a value of 0 is returned. If any of the intermediate computations do not halt, however, neither does the computation of $\mu x\ f(x, \vec{z})$.

If $R(x, \vec{z})$ is any relation, $\mu x\ R(x, \vec{z})$ is defined to be $\mu x\ (1 \dot{-} \chi_R(x, \vec{z}))$. In other words, $\mu x\ R(x, \vec{z})$ returns the least value of x such that $R(x, \vec{z})$ holds. So, if $f(x, \vec{z})$ is a total function, $\mu x\ f(x, \vec{z})$ is the same as $\mu x\ (f(x, \vec{z}) = 0)$. But note that our original definition is more general, since it allows for the possibility that $f(x, \vec{z})$ is not everywhere defined (whereas, in contrast, the characteristic function of a relation is always total).

Definition 2.5.1 *The set of* partial recursive functions *is the smallest set of* partial *functions from the natural numbers to the natural numbers (of various arities) containing zero, successor, and projections, and closed under composition, primitive recursion, and unbounded search.*

Of course, some of the partial recursive functions will happen to be total, i.e. defined at every input.

Definition 2.5.2 *The set of* recursive functions *is the set of partial recursive functions that are total.*

A recursive function is sometimes called "total recursive" to emphasize that it is defined everywhere, and I may adopt this terminology on occasion. But remember that when I say "recursive" without further qualification, I mean "total recursive," by default.

There is another way to obtain a set of total functions. Say a total function $f(x, \vec{z})$ is *regular* if for every sequence of natural numbers $\vec{z}$, there

is an x such that $f(x, \vec{z}) = 0$. In other words, the regular functions are exactly those functions to which one can apply unbounded search, and end up with a total function. One can, conservatively, restrict unbounded search to regular functions:

Definition 2.5.3 *The set of* general recursive functions *is the smallest set of functions from the natural numbers to the natural numbers (of various arities) containing zero, successor, and projections, and closed under composition, primitive recursion, and unbounded search applied to* regular *functions.*

Clearly every general recursive function is total. The difference between Definition 2.5.3 and the Definition 2.5.2 is that in the latter one is allowed to use partial recursive functions along the way; the only requirement is that the function you end up with at the end is total. So the word "general," a historic relic, is a misnomer; on the surface, the Definition 2.5.3 is *less* general than Definition 2.5.2. But, fortunately, we will soon see that the difference is illusory; though the definitions are different, the set of general recursive functions and the set of recursive functions are one and the same.

2.6 Recursive is equivalent to Turing computable

The aim of this section is to establish the following:

Theorem 2.6.1 *Every partial recursive function is a partial Turing computable function, and vice-versa.*

Since the recursive functions are just the partial recursive functions that happen to be total, and similarly for the Turing computable functions, we have:

Corollary 2.6.2 *Every recursive function is Turing computable, and vice-versa.*

There are two directions to proving Theorem 2.6.1. First, we will show that every partial recursive function is Turing computable. Then we will show that every Turing computable function is partial recursive.

For the first direction, recall the definition of the set of partial recursive functions as being the *smallest* set containing zero, successor, and projections, and closed under composition, primitive recursion, and unbounded search. To show that every partial recursive function is Turing computable,

it therefore suffices to show that the initial functions are Turing computable, and that the (partial) Turing computable functions are closed under these same operations. Indeed, we will show something slightly stronger: each initial function is computed by a Turing machine that never moves to the left of the start position, ends its computation on the same square on which it started, and leaves the tape after the output blank; and the set of functions computable in this way is closed under the relevant operations. I will follow the argument in the textbook.

Computing the constant zero is easy: just halt with a blank tape. Computing the successor function is also easy: again, just halt. Computing a projection function P_i^n is not much harder: just erase all the inputs other than the ith, copy the ith input to the beginning of the tape, and delete a single 1.

Closure under composition is slightly more interesting. Suppose f is the function defined by composition from $h, g_0, \ldots, g_k$, i.e.

$$f(x_0, \ldots, x_l) \simeq h(g_0(x_0, \ldots, x_l), \ldots, g_k(x_0, \ldots, x_l)).$$

Inductively, we have Turing machines $M_h, M_{g_0}, \ldots, M_{g_k}$ computing $h, g_0, \ldots, g_k$, and we need to design a machine that computes f.

Our Turing machine begins with input $1^{x_1+1}, 0, 1^{x_2+1}, 0, \ldots, 0, 1^{x_l+1}$. Call this block I. The idea is to run each of the machines $M_{g_0}, \ldots, M_{g_k}$ in turn on a copy of this input; and then run M_h on the sequence of outputs (remember that we have to add a 1 to each output). This is where we need to know that the activity of each Turing machine will not mess up information on the tape that lies to the left of the start position. Roughly put, the algorithm is as follows:

- Copy I: $I, 0, I$
- Run machine M_{g_0}: $I, 0, 1^{g_0(\vec{x})}$
- Add a 1: $I, 0, 1^{g_0(x_1)+1}$
- Copy I: $I, 0, 1^{g_0(x_1)+1}, 0, I$
- Run machine M_{g_1}: $I, 0, 1^{g_0(\vec{x})+1}, 0, 1^{g_1(\vec{x})}$
- Add a 1: $I, 0, 1^{g_0(\vec{x})+1}, 0, 1^{g_1(\vec{x})+1}$
- $\ldots$
- Run machine M_{g_k}: $I, 0, 1^{g_0(\vec{x})+1}, 0, 1^{g_1(\vec{x})+1}, 0, \ldots, 0, 1^{g_k(\vec{x})}$

- Add a 1: $I, 0, 1^{g_0(\vec{x})+1}, 0, 1^{g_1(\vec{x})+1}, 0, \ldots, 0, 1^{g_k(\vec{x})+1}$
- Run machine M_h: $I, 0, 1^{h(g_0(\vec{x}),\ldots,g_k(\vec{x}))}$
- Delete the initial block I, and move the output to the left.

Setting aside primitive recursion for the moment, let us consider unbounded search. Suppose we already have a machine M_f that computes $f(x, \vec{z})$, and we wish to find a machine to compute $\mu x\ f(x, \vec{z})$. Let I denote the block corresponding to the input $\vec{z}$. The algorithm, of course, is to iteratively compute $f(0, \vec{z}), f(1, \vec{z}), \ldots$ until the output is zero. We can do this as follows:

- Add a blank and a 1 after the input: $I, 0, 1$
- Loop, as follows:
  - Assuming the tape is of the from $I, 0, 1^{x+1}$, write down a blank, copy 1^{x+1}, write another blank, and copy I: $I, 0, 1^{x+1}, 0, 1^{x+1}, 0, I$
  - Run M_f on the block beginning with the second copy of 1^{x+1}: $I, 0, 1^{x+1}, 0, 1^{f(x,\vec{z})}$
  - If $f(x, \vec{z})$ is 0, leave an output of 1^x and halt
  - Otherwise, delete the block $1^{f(x,\vec{z})}$, and add a 1 to the block 1^{x+1}: $I, 0, 1^{x+2}$
  - Go back to the start of the loop.

Handling primitive recursion involves bookkeeping that is more ornery, but otherwise a straightforward reflection of the expected algorithm. I will leave the details to you.

Later in this course, we will have another way of finishing off the theorem. We will show that one can alternatively characterize the partial recursive functions as the smallest set of partial functions containing zero, successor, addition, multiplication, and the characteristic function for equality, and closed under composition and unbounded search. In other words, we can eliminate primitive recursion in favor of a few additional initial functions; and since the initial functions are easy to compute, we have the desired result. Either way, modulo a few details, we have established the forward direction of Theorem 2.6.1.

Going the other way, we need to show that any partial function computed by a Turing machine is partial recursive. The idea is to show that the notion of being a halting computation is recursive, and, in fact, primitive recursive;

and that the function that returns the output of a halting computation is also primitive recursive. Then, assuming f is computed by Turing machine M, we can describe f as a partial recursive function as follows: on input x, use unbounded search to look for a halting computation sequence for machine M on input x; and, if there is one, return the output of the computation sequence.

In fact, we did most of the work when we gave a precise definition of Turing computability in Section 2.1; we only have to show that all the definitions can be expressed in terms of primitive recursive functions and relations. It turns out to be convenient to use a sequence of 4-tuples to represent a Turing machine's list of instructions (instead of a partial function); otherwise, the definitions below are just the primitive recursive analogues of the ones given in Section 2.1.

In the list below, names of functions begin with lower case letters, while the names of relations begin with upper case letters. I will not provide every last detail; the ones I leave out are for you to fill in.

1. Functions and relations related to instructions:

   $Instruction(k, n, m) \equiv length(k) = 4 \wedge (k)_0 < n \wedge (k)_1 < m \wedge (k)_2 < m + 2 \wedge (k)_3 < n$

   The relation above holds if and only if k codes a suitable instruction for a machine with n states and m symbols.

   $iState(k) = (k)_0$

   $iSymbol(k) = (k)_1$

   $iOperation(k) = (k)_2$

   $iNextState(k) = (k)_3$

   These four functions return the corresponding components of an instruction.

   $InstructionSeq(s, n, m) \equiv \forall i < length(s)\ Instruction((s)_i, n, m) \wedge$

   $\forall i < length(s)\ \forall j < length(s)\ (iState((s)_i) = iState((s)_j) \wedge$

   $iSymbol((s)_i) = iSymbol((s)_j) \rightarrow i = j)$

   This says that s is a suitable sequence of instructions for a Turing machine with n symbols and m states. The main requirement is that the list of 4-tuples corresponds to a function: for any symbol and state, there is at most one instruction that applies.

2. Functions and relations related to machines:

   $Machine(M) \equiv length(M) = 3 \wedge InstructionSeq((M)_2, (M)_1, (M)_0)$

   This says M represents a Turing machine.

   $mNumStates(M) = (M)_0$

   $mNumSymbols(M) = (M)_1$

   $mInstructions(M) = (M)_2$

   These are the corresponding components.

   $mLeft(M) = mNumSymbols(M)$

   $mRight(M) = mNumSymbols(M) + 1$

   These pick out the representatives for the "move left" and "move right" operations.

3. Functions and relations related to configurations:

   $Configuration(M, c) \equiv length(c) = 4 \wedge (c)_0 < mNumStates(M) \wedge$
   $(c)_1 < mNumSymbols(M) \wedge$
   $\forall i < length((c)_2)\ (((c)_2)_i < mNumSymbols(M)) \wedge$
   $\forall i < length((c)_3)\ (((c)_3)_i < mNumSymbols(M))$

   This holds if and only if c codes a configuration of Turing machine M: $(c)_0$ is the current state, $(c)_1$ is the symbol under the tape head, and $(c)_2$ and $(c)_3$ are the strings of symbols to the left and right of the tape head, respectively.

   $cState(c) = (c)_0$

   $cSymbol(c) = (c)_1$

   $cLeftString(c) = (c)_2$

   $cRightString(c) = (c)_3$

   These pick out the relevant components.

4. Operations on configurations:

   $addSymbol(a, s) = \langle a \rangle \hat{} s$

   $dropSymbol(s) = \ldots$

   This should return the result of dropping the first symbol of s, if s is nonempty, and s otherwise.

$$\mathit{firstSymbol}(s) = \begin{cases} (s)_0 & \text{if } \mathit{length}(s) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\mathit{changeState}(c, q) = \langle q, \mathit{cSymbol}(c), \mathit{cLeftString}(c), \mathit{cRightString}(c)\rangle$

This should change the state of configuration c to q.

$\mathit{moveLeft}(c) = \langle \mathit{cState}(c), \mathit{firstSymbol}(\mathit{cLeftString}(c)),$

$\mathit{dropSymbol}(\mathit{cLeftString}(c)), \mathit{addSymbol}(\mathit{cSymbol}(c), \mathit{cRightString}(c))\rangle$

$\mathit{moveRight}(c) = \ldots$

These return the result of moving left (resp. right) in configuration c.

5. Functions and relations to determine the next configuration:

   $\mathit{HaltingConfig}(M, c) \equiv \neg\exists i < \mathit{length}(\mathit{mInstructions}(M))$

   $(\mathit{iState}((\mathit{mInstructions}(M))_i) = \mathit{cState}(c) \wedge$

   $\mathit{iSymbol}((\mathit{mInstructions}(M))_i) = \mathit{cSymbol}(c))$

   This holds if c is a halting computation for Turing machine M.

   $\mathit{nextInstNum}(M, c) = \mathit{min}\ i < \mathit{length}(\mathit{mInstructions}(M))$

   $\mathit{iState}((\mathit{mInstructions}(M))_i) = \mathit{cState}(c) \wedge$

   $\mathit{iSymbol}((\mathit{mInstructions}(M))_i) = \mathit{cSymbol}(c))$

   $\mathit{nextInst}(M, c) = (\mathit{mInstructions}(M))_{\mathit{nextInstNum}(M,c)}$

   Assuming c is not a halting configuration, these return the index of the next instruction for M in configuration c, and the instruction itself.

   $\mathit{nextConfig}(M, c) = \ldots$

   Assuming c is not a halting configuration, this returns the next configuration for M after c.

6. Functions and relations to handle computation sequences:

   $\mathit{startConfig}(x) = \ldots$

   This should return the starting configuration corresponding to input x.

   $\mathit{CompSeq}(M, x, s) \equiv (s)_0 = \mathit{startConfig}(x) \wedge$

   $\forall i < \mathit{length}(s){-}1\ (\neg \mathit{HaltingConfig}(M, (s)_i) \wedge (s)_{i+1} = \mathit{nextConfig}(M, (s)_i)) \wedge$

   $\mathit{HaltingConfig}(M, (s)_{\mathit{length}(s)-1}))$

   This holds if s is a halting computation sequence for M on input x.

   $\mathit{cOutput}(c) = \ldots$

This should return the output represented by configuration c, according to our output conventions.

$output(s) = cOutput((s)_{length(s)-1})$

This returns the output of the (last configuration in the) computation sequence.

We can now finish off the proof of the theorem. Suppose $f(x)$ is a partial function computed by a Turing machine, coded by M. Then for every x, we have

$$f(x) = output(\mu s\ CompSeq(M, x, s)).$$

This shows that f is partial recursive. □

Let me remind you that we have intentionally set issues of efficiency aside. In practice, it would be ludicrous to compute the function f above by checking each natural number, $0, 1, 2, \ldots$ to see if it codes a halting computation sequence of Turing machine M on the given input.

The proof that every Turing computable function is partial recursive goes a long way towards explaining why we believe that *every* computable partial function is partial recursive. Intuitively, something is computable if one can describe a method of computing it, breaking the algorithm down into small, easily verified steps. As long as the notions of a "state" and a one-step transition are primitive recursive, our method of proof shows that the result will be partial recursive.

Indeed, this intuition is found in proposals to view a function as computable if it is represented in a "formal system." We will return to this idea later, but, for the moment, let me note that the definition is somewhat circular: a formal system of mathematics is one where the basic axioms and rules are computable, so this definition of computability *presupposes* a notion of computability to start with. Replacing the presupposed notion of computability with primitive recursion breaks the circularity, but then it is not entirely clear that we haven't lost anything with the restriction. It is really Turing's "appeals to intuition" in the 1936 paper that rounds out the argument that we have captured all the computable functions.

2.7 Theorems on computability

We have seen that there is a primitive recursive relation $CompSeq(M, x, s)$, which decides whether or not s is a halting computation of machine M

on input x, and a primitive recursive function $output(s)$ which returns the output of this computation.

Recall that if $f(\vec{x}, y)$ is a total or partial function, then $\mu y\ f(\vec{x}, y)$ is the function of $\vec{x}$ that returns the least y such that $f(\vec{x}, y) = 0$, assuming that all of $f(\vec{x}, 0), \ldots, f(\vec{x}, y-1)$ are defined; if there is no such y, $\mu y\ f(\vec{x}, y)$ is undefined. If $R(\vec{x}, y)$ is a relation, $\mu y\ R(\vec{x}, y)$ is defined to be the least y such that $R(\vec{x}, y)$ is true; in other words, the least y such that *one minus* the characteristic function of R is equal to zero at $\vec{x}, y$.

We have seen that if $f(x)$ is a partial function on the natural numbers computed by Turing machine M, then for each x, we have

$$f(x) \simeq output(\mu s\ CompSeq(M, x, s)).$$

If f is a total function, $\simeq$ is equivalent to $=$, and one minus the characteristic function $CompSeq(M, x, s)$ is regular; so we have shown that f is "general recursive" as well. In short, we have the following:

Theorem 2.7.1 *The following are all the same:*

1. *the set of partial Turing computable functions*
2. *the set of partial recursive functions*

Theorem 2.7.2 *The following are all the same:*

1. *the set of Turing computable functions*
2. *the set of recursive functions*
3. *the set of general recursive functions*

Our analysis gives us much more information.

Theorem 2.7.3 (Kleene's Normal Form Theorem) *There are a primitive recursive relation $T(M, x, s)$ and a primitive recursive function $U(s)$, with the following property: if f is any partial Turing computable function, then for some M,*

$$f(x) \simeq U(\mu s\ T(M, x, s))$$

for every x.

Proof. T and U are simply more conventional notations for any relation and function pair that behaves like our *CompSeq* and *output*. □

It is probably best to remember the proof of the normal form theorem in slogan form: $\mu s\ T(M, x, s)$ searches for a halting computation sequence of M on input x, and U returns the output of the computation sequence.

Theorem 2.7.4 *The previous theorem is true if we replace "partial Turing computable" by "partial recursive."*

Proof. Every partial recursive function is partial Turing computable. □

Note, incidentally, that we now have an enumeration of the partial recursive functions: we've shown how to translate any description of a partial recursive function into a Turing machine, and we've numbered Turing machines. Of course, this is a little bit roundabout. One can come up with a more direct enumeration, as we did when we enumerated the primitive recursive functions. This is done in Chapter 16 of Epstein and Carnielli.

A lot of what one does in computability theory doesn't depend on the particular model one chooses. The following tries to abstract away some of the important features of computability, that are not tied to the particular model. From now on, when I say "computable" you can interpret this as either "Turing computable" or "recursive"; likewise for "partial computable." If you believe Church's thesis, this use of the term "computable" corresponds exactly to the set of functions that we would intuitively label as such.

Theorem 2.7.5 *There is a universal partial computable function $Un(k, x)$. In other words, there is a function $Un(k, x)$ such that:*

1. *$Un(k, x)$ is partial computable.*
2. *If $f(x)$ is any partial computable function, then there is a natural number k such that $f(x) \simeq Un(k, x)$ for every x.*

Proof. Let $Un(k, x) \simeq U(\mu s\ T(k, x, s))$ in Kleene's normal form theorem. □

This is just a precise way of saying that we have an effective enumeration of the partial computable functions; the idea is that if we write f_k for the function defined by $f_k(x) = Un(k, x)$, then the sequence $f_0, f_1, f_2, \ldots$ includes all the partial computable functions, with the property that $f_k(x)$ can be computed "uniformly" in k and x. For simplicity, I am using a binary

function that is universal for unary functions, but by coding sequences of numbers you can easily generalize this to more arguments. For example, note that if $f(x, y, z)$ is a 3-ary partial recursive function, then the function $g(x) \simeq f((x)_0, (x)_1, (x)_2)$ is a unary recursive function.

Theorem 2.7.6 *There is no universal computable function $Un'(k, x)$.*

Proof. This theorem says that there is no *total* computable function that is universal for the total computable functions. The proof is a simple diagonalization: if $Un'(k, x)$ is total and computable, then

$$f(x) = Un'(x, x) + 1$$

is also total and computable, and for every k, $f(k)$ is not equal to $Un'(k, k)$. □

This proof is just the diagonalization argument that we have already used in the context of the primitive recursive functions. Theorem 2.7.5 above shows that we can get around the diagonalization argument, but only at the expense of allowing partial functions. It is worth trying to understand what goes wrong with the diagonalization argument, when we try to apply it in the partial case. In particular, the function $h(x) = Un(x, x) + 1$ *is* partial recursive. Suppose h is the kth function in the enumeration; what can we say about $h(k)$?

The following theorem hones in on the difference between the last two theorems.

Theorem 2.7.7 *Let*

$$f(k, x) = \begin{cases} 1 & \textit{if } Un(k, x) \textit{ is defined} \\ 0 & \textit{otherwise.} \end{cases}$$

Then f is not computable.

Since, in our construction, $Un(k, x)$ is defined if and only if the Turing machine coded by k halts on input x, the theorem asserts that the question as to whether a given Turing machine halts on a given input is computationally undecidable. I will provide two proofs below. The first continues the thread of our previous discussion, while the second is more direct.

Proof. If f were computable, we would have a universal computable function, as follows. Suppose f is computable, and define

$$Un'(k, x) = \begin{cases} Un(k, x) & \text{if } f(k, x) = 1 \\ 0 & \text{otherwise,} \end{cases}$$

To show that this is recursive, define g using primitive recursion, by

$$\begin{aligned} g(0, k, x) &\simeq 0 \\ g(y+1, k, x) &\simeq Un(k, x); \end{aligned}$$

then

$$Un'(k, x) \simeq g(f(k, x), k, x).$$

But now $Un'(k, x)$ is a total function. And since $Un'(k, x)$ agrees with $Un(k, x)$ wherever the latter is defined, Un' is universal for those partial computable functions that happen to be total. But this contradicts Theorem 2.7.6. □

Second proof of Theorem 2.7.7. Suppose $f(k, x)$ were computable. Define the function g by

$$g(x) = \begin{cases} 0 & \text{if } f(x, x) = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function g is partial computable; for example, one can define it as $\mu y f(x, x)$. So, for some k, $g(x) \simeq Un(k, x)$ for every x. Is g defined at k? If it is, then, by the definition of g, $f(k, k) = 0$. By the definition of f, this means that $Un(k, k)$ is undefined; but by our assumption that $g(k) \simeq Un(k, x)$ for every x, this means that $g(k)$ is undefined, a contradiction. On the other hand, if $g(k)$ is undefined, then $f(k, k) \neq 0$, and so $f(k, k) = 1$. But this means that $Un(k, k)$ is defined, i.e. that $g(k)$ is defined. □

We can describe this argument in terms of Turing machines. Suppose there were a Turing machine F that took as input a description of a Turing machine K and an input x, and decided whether or not K halts on input x. Then we could build another Turing machine G which takes a single input x, calls F to decide if machine x halts on input x, and does the opposite. In other words, if F reports that x halts on input x, G goes into an infinite loop, and if F reports that x doesn't halt on input x, then F just halts. Does F halt on input F? The argument above shows that it does if and only if it doesn't — a contradiction. So our supposition that there is a such Turing machine, F, is false.

I have found it instructive to compare and contrast the arguments in this section with Russell's paradox:

- Russell's paradox: let $S = \{x \mid x \notin x\}$. Then $x \in S$ if and only if $x \notin S$, a contradiction.

  Conclusion: S is not a set. Assuming the existence of a "set of all sets" is inconsistent with the other axioms of set theory.

- A modification of Russell's paradox: let F be the "function" from the set of all functions to $\{0, 1\}$, defined by

$$F(f) = \begin{cases} 1 & \text{if } f \text{ is in the domain of } f, \text{ and } f(f) = 0 \\ 0 & \text{otherwise} \end{cases}$$

  A similar argument shows that $F(F) = 0$ if and only if $F(F) = 1$, a contradiction.

  Conclusion: F is not a function. The "set of all functions" is too big to be the domain of a function.

- The diagonalization argument above: let $f_0, f_1, \ldots$ be the enumeration of the partial computable functions, and let $G : \mathbb{N} \to \{0, 1\}$ be defined by

$$G(x) = \begin{cases} 1 & \text{if } f_x(x) \downarrow = 0 \\ 0 & \text{otherwise} \end{cases}$$

  If G is computable, then it is the function f_k for some k. But then $G(k) = 1$ if and only if $G(k) = 0$, a contradiction.

  Conclusion: G is not computable. Note that according to axioms of set theory, G is still a function; there is no paradox here, just a clarification.

I have found that talk of partial functions, computable functions, partial computable functions, and so on can be confusing. The set of all partial functions from $\mathbb{N}$ to $\mathbb{N}$ is a big collection of objects. Some of them are total, some of them are computable, some are both total and computable, and some are neither. Keep in mind that when we say "function," by default, we mean a total function. Thus we have:

- computable functions
- partial computable functions that are not total
- functions that are not computable
- partial functions that are neither total nor computable

To sort this out, it might help to draw a big square representing all the partial functions from $\mathbb{N}$ to $\mathbb{N}$, and then mark off two overlapping regions, corresponding to the total functions and the computable partial functions, respectively. It is a good exercise to see if you can describe an object in each of the resulting regions in the diagram.

2.8 The lambda calculus

By now, we have discussed a number of early models of computation:

- Turing computability (Turing)
- The recursive functions (Kleene)
- The general recursive functions (Gödel, Herbrand)
- Representability in a formal system (Gödel, Church)

I have already noted that there are many more — including abacus computability, computability by register machines, computability by a C++ or Java program, and more. In fact, we will come across a few additional ones towards the end of the course. In this section we will consider one more model: computability by lambda terms.

The lambda calculus was originally designed by Alonzo Church in the early 1930's as a basis for constructive logic, and *not* as a model of the computable functions. But soon after the Turing computable functions, the recursive functions, and the general recursive functions were shown to be equivalent, lambda computability was added to the list. The fact that this initially came as a small surprise makes the characterization all the more interesting.

In Chapter 3, the textbook discusses some simple uses of λ notation. Instead of saying "let f be the function defined by $f(x) = x + 3$," one can say, "let f be the function $\lambda x\ (x + 3)$." In other words, $\lambda x\ (x + 3)$ is just a *name* for the function that adds three to its argument. In this expression, x is just a dummy variable, or a placeholder: the same function can just as well be denoted $\lambda y\ (y + 3)$. The notation works even with other parameters around. For example, suppose $g(x, y)$ is a function of two variables, and k is a natural number. Then $\lambda x\ g(x, k)$ is the function which maps any value of x to $g(x, k)$.

This way of defining a function from a symbolic expression is known as *lambda abstraction*. The flip side of lambda abstraction is *application*:

assuming one has a function f (say, defined on the natural numbers), one can *apply* it to any value, like 2. In conventional notation, of course, we write $f(2)$.

What happens when you combine lambda abstraction with application? Then the resulting expression can be simplified, by "plugging" the applicand in for the abstracted variable. For example,

$$(\lambda x\ (x + 3))(2)$$

can be simplified to $2 + 3$.

Up to this point, we have done nothing but introduce new notations for conventional notions. The lambda calculus, however, represents a more radical departure from the set-theoretic viewpoint. In this framework:

- Everything denotes a function.
- Functions can be defined using lambda abstraction.
- Anything can be applied to anything else.

For example, if F is a term in the lambda calculus, $F(F)$ is always assumed to be meaningful. This liberal framework is known as the *untyped* lambda calculus, where "untyped" means "no restriction on what can be applied to what." We will not discuss the *typed* lambda calculus, which is an important variation on the untyped version; but here I will note that although in many ways the typed lambda calculus is similar to the untyped one, it is much easier to reconcile with a classical set-theoretic framework, and has some very different properties.

Research on the lambda calculus has proved to be central in theoretical computer science, and in the design of programming languages. LISP, designed by John McCarthy in the 1950's, is an early example of a language that was influenced by these ideas. So, for the moment, let us put the set-theoretic way of thinking about functions aside, and consider this calculus.

One starts with a sequence of variables $x, y, z, \ldots$ and some constant symbols $a, b, c, \ldots$. The set of terms is defined inductively, as follows:

- Each variable is a term.
- Each constant is a term.
- If M and N are terms, so is (MN).

- If M is a term and x is a variable, then $(\lambda x\ M)$ is a term.

The system without any constants at all is called the *pure* lambda calculus.

Following the handout, we will follow a few notational conventions:

- When parentheses are left out, application takes place from left to right. For example, if M, N, P, and Q are terms, then $MNPQ$ abbreviates $(((MN)P)Q)$.

- Again, when parentheses are left out, lambda abstraction is to be given the widest scope possible. From example, $\lambda x\ MNP$ is read $\lambda x\ (MNP)$.

- A period can be used to abstract multiple variables. For example, $\lambda xyz.\ M$ is short for $\lambda x\ \lambda y\ \lambda z\ M$.

For example,

$$\lambda xy.\ xxyx\lambda z\ xz$$

abbreviates

$$\lambda x\ \lambda y\ ((((xx)y)x)\lambda z\ (xz)).$$

Memorize these conventions. They will drive you crazy at first, but you will get used to them, and after a while they will drive you less crazy than having to deal with a morass of parentheses.

Two terms that differ only in the names of the bound variables are called α-equivalent; for example, $\lambda x\ x$ and $\lambda y\ y$. It will be convenient to think of these as being the "same" term; in other words, when I say that M and N are the same, I also mean "up to renamings of the bound variables." Variables that are in the scope of a λ are called "bound", while others are called "free." There are no free variables in the previous example; but in

$$(\lambda z\ yz)x$$

y and x are free, and z is bound. More precise definitions of "free," "bound," and "α equivalent" can be found in the handout.

What can one do with lambda terms? Simplify them. If M and N are any lambda terms and x is any variable, the handout uses $[N/x]M$ to denote the result of substituting N for x in M, after renaming any bound variables of M that would interfere with the free variables of N after the substitution. For example,

$$[yyz/x](\lambda w\ xxw) = \lambda w\ (yyz)(yyz)w.$$

This notation is not the only one that is standardly used; I, myself, prefer to use the notation $M[N/x]$, and others use $M[x/N]$. Beware!

Intuitively, $(\lambda x\ M)N$ and $[N/x]M$ have the same meaning; the act of replacing the first term by the second is called β-contraction. More generally, if it is possible convert a term P to P' by β-contracting some subterm, one says P *β-reduces to P' in one step*. If P can be converted to P' with any number of one-step reductions (possibly none), then P *β-reduces* to P'. A term that can not be β-reduced any further is called β-irreducible, or β-normal. I will say "reduces" instead of "β-reduces," etc., when the context is clear.

Let us consider some examples.

1. We have

$$\begin{aligned}(\lambda x.\ xxy)\lambda z\ z &\triangleright_1 (\lambda z\ z)(\lambda z\ z)y \\ &\triangleright_1 (\lambda z\ z)y \\ &\triangleright_1 y\end{aligned}$$

2. "Simplifying" a term can make it more complex:

$$\begin{aligned}(\lambda x.\ xxy)(\lambda x.\ xxy) &\triangleright_1 (\lambda x.\ xxy)(\lambda x.\ xxy)y \\ &\triangleright_1 (\lambda x.\ xxy)(\lambda x.\ xxy)yy \\ &\triangleright_1 \ldots\end{aligned}$$

3. It can also leave a term unchanged:

$$(\lambda x.\ xx)(\lambda x.\ xx) \triangleright_1 (\lambda x.\ xx)(\lambda x.\ xx)$$

4. Also, some terms can be reduced in more than one way; for example,

$$(\lambda x\ (\lambda y\ yx)z)v \triangleright_1 (\lambda y\ yv)z$$

   by contracting the outermost application; and

$$(\lambda x\ (\lambda y\ yx)z)v \triangleright_1 (\lambda x\ zx)v$$

   by contracting the innermost one. Note, in this case, however, that both terms further reduce to the same term, zv.

The final outcome in the last example is not a coincidence, but rather illustrates a deep and important property of the lambda calculus, known as the "Church-Rosser property."

Theorem 2.8.1 *Let M, N_1, and N_2 be terms, such that $M \triangleright N_1$ and $M \triangleright N_2$. Then there is a term P such that $N_1 \triangleright P$ and $N_2 \triangleright P$.*

The proof of Theorem 2.8.1 goes well beyond the scope of this class, but if you are interested you can look it up in Hindley and Seldin, *Introduction to Combinators and λ Calculus*.

Corollary 2.8.2 *Suppose M can be reduced to normal form. Then this normal form is unique.*

Proof. If $M \triangleright N_1$ and $M \triangleright N_2$, by the previous theorem there is a term P such that N_1 and N_2 both reduce to P. If N_1 and N_2 are both in normal form, this can only happen if $N_1 = P = N_2$. □

Finally, I will say that two terms M and N are *β-equivalent*, or just *equivalent*, if they reduce to a common term; in other words, if there is some P such that $M \triangleright P$ and $N \triangleright P$. This is written $M \equiv N$. Using Theorem 2.8.1, you can check that $\equiv$ is an equivalence relation, with the additional property that for every M and N, if $M \triangleright N$ or $N \triangleright M$, then $M \equiv N$. (In fact, one can show that $\equiv$ is the *smallest* equivalence relation having this property.)

What is the lambda calculus doing in a chapter on models of computation? The point is that it *does* provide us with a model of the computable functions, although, at first, it is not even clear how to make sense of this statement. To talk about computability on the natural numbers, we need to find a suitable representation for such numbers. Here is one that works surprisingly well.

Definition 2.8.3 *For each natural number n, define the* numeral $\overline{n}$ *to be the lambda term $\lambda xy\,(x(x(x(\ldots x(y)))))$, where there are n x's in all.*

The terms $\overline{n}$ are "iterators": on input f, $\overline{n}$ returns the function mapping y to $f^n(y)$. Note that each numeral is normal. We can now say what it means for a lambda term to "compute" a function on the natural numbers.

Definition 2.8.4 *Let $f(x_0, \ldots, x_{n-1})$ be an n-ary partial function from $\mathbb{N}$ to $\mathbb{N}$. Say a lambda term X* represents *f if for every sequence of natural numbers $m_0, \ldots, m_{n-1}$,*

$$X\overline{m}_0\overline{m}_1 \ldots \overline{m}_{n-1} \triangleright \overline{f(m_0, m_1, \ldots, m_{n-1})}$$

if $f(m_0, \ldots, m_{n-1})$ is defined, and $X\overline{m}_0\overline{m}_1 \ldots \overline{m}_{n-1}$ has no normal form otherwise.

Theorem 2.8.5 *A function f is a partial computable function if and only if it is represented by a lambda term.*

This theorem is somewhat striking. As a model of computation, the lambda calculus is a rather simple calculus; the only operations are lambda abstraction and application! From these meager resources, however, it is possible to implement any computational procedure.

The "if" part of the theorem is easier to see: suppose a function, f, is represented by a lambda term X. Let us describe an informal procedure to compute f. On input $m_0, \ldots, m_{n-1}$, write down the term $X\overline{m}_0 \ldots \overline{m}_{n-1}$. Build a tree, first writing down all the one-step reductions of the original term; below that, write all the one-step reductions of those (i.e. the two-step reductions of the original term); and keep going. If you ever reach a numeral, return that as the answer; otherwise, the function is undefined.

An appeal to Church's thesis tells us that this function is computable. A better way to prove the theorem would be to give a recursive description of this search procedure, similar to our recursive description of Turing machine computations. For example, one could define a sequence primitive recursive functions and relations, "*IsASubterm*," "*Substitute*," "*ReducesToInOneStep*," "*ReductionSequence*," "*Numeral*," etc. The partial recursive procedure for computing $f(m_0, \ldots, m_{n-1})$ is then to search for a sequence of one-step reductions starting with $X\overline{m}_0 \ldots \overline{m}_{n-1}$ and ending with a numeral, and return the number corresponding to that numeral. The details are long and tedious but otherwise routine, and I will leave you to work them out to your own satisfaction.

In the other direction we need to show that every partial function f is represented by a lambda term, $\overline{f}$. By Kleene's normal form theorem, it suffices to show that every primitive recursive function is represented by a lambda term, and then that the functions so represented are closed under suitable compositions and unbounded search. To show that every primitive recursive function is represented by a lambda term, it suffices to show that the initial functions are represented, and that the partial functions that are represented by lambda terms are closed under composition, primitive recursion, and unbounded search.

I will resort to more conventional notation to make the rest of the proof more readable. For example, I will write $M(x, y, z)$ instead of $Mxyz$. While this is suggestive, you should remember that terms in the untyped lambda calculus do not have associated arities; so, for the same term M, it makes just as much sense to write $M(x, y)$ and $M(x, y, z, w)$. But using this notation indicates that we are treating M as a function of three variables, and helps

make the intentions behind the definitions clearer. In a similar way, I will resort to the old-fashioned way of saying "define M by $M(x, y, z) = \ldots$" instead of "define M by $M = \lambda x\ \lambda y\ \lambda z\ \ldots$."

Let us run through the list. Zero, $\overline{0}$, is just $\lambda xy.\ y$. The successor function, $\overline{S}$, is defined by $\overline{S}(u) = \lambda xy.\ x(uxy)$. You should think about why this works; for each numeral $\overline{n}$, thought of as an iterator, and each function f, $S(\overline{n}, f)$ is a function that, on input y, applies f n times starting with y, and then applies it once more.

There is nothing to say about projections: $\overline{P}_i^n(x_0, \ldots, x_{n-1}) = x_i$. In other words, by our conventions, $\overline{P}_i^n$ is the lambda term $\lambda x_0, \ldots, x_{n-1}.\ x_i$.

Closure under composition is similarly easy. Suppose f is defined by composition from $h, g_0, \ldots, g_{k-1}$. Assuming $h, g_0, \ldots, g_{k-1}$ are represented by $\overline{h}, \overline{g}_0, \ldots, \overline{g}_{k-1}$, respectively, we need to find a term $\overline{f}$ representing f. But we can simply define $\overline{f}$ by

$$\overline{f}(x_0, \ldots, x_{l-1}) = \overline{h}(\overline{g}_0(x_0, \ldots, x_{l-1}), \ldots, \overline{g}_{k-1}(x_0, \ldots, x_{l-1})).$$

In other words, the language of the lambda calculus is well suited to represent composition as well.

When it comes to primitive recursion, we finally need to do some work. We will have to proceed in stages. As before, on the assumption that we already have terms $\overline{g}$ and $\overline{h}$ representing functions g and h, respectively, we want a term $\overline{f}$ representing the function f defined by

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

So, in general, given lambda terms G' and H', it suffices to find a term F such that

$$\begin{aligned} F(\overline{0}, \vec{z}) &\equiv G'(\vec{z}) \\ F(\overline{n+1}, \vec{z}) &\equiv H'(n, F(\overline{n}, \vec{z}), \vec{z}) \end{aligned}$$

for every natural number n; the fact that G' and H' represent g and h means that whenever we plug in numerals $\overline{\vec{m}}$ for $\vec{z}$, $F(\overline{n+1}, \overline{\vec{m}})$ will normalize to the right answer.

But for this, it suffices to find a term F satisfying

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n, where

$$G = \lambda \vec{z}\, G'(\vec{z})$$

and

$$H(u, v) = \lambda \vec{z}\, H'(u, v(u, \vec{z}), \vec{z}).$$

In other words, with lambda trickery, we can avoid having to worry about the extra parameters $\vec{z}$ — they just get absorbed in the lambda notation.

Before we define the term F, we need a mechanism for handling ordered pairs. This is provided by the next lemma.

Lemma 2.8.6 *There is a lambda term D such that for each pair of lambda terms M and N, $D(M, N)(\overline{0}) \triangleright M$ and $D(M, N)(\overline{1}) \triangleright N$.*

Proof. First, define the lambda term K by

$$K(y) = \lambda x\ y.$$

In other words, K is the term $\lambda yx.\ y$. Looking at it differently, for every M, $K(M)$ is a constant function that returns M on any input.

Now define $D(x, y, z)$ by $D(x, y, z) = z(K(y))x$. Then we have

$$D(M, N, \overline{0}) \triangleright \overline{0}(K(N))M \triangleright M$$

and

$$D(M, N, \overline{1}) \triangleright \overline{1}(K(N))M \triangleright K(N)M \triangleright N,$$

as required. □

The idea is that $D(M, N)$ represents the pair $\langle M, N\rangle$, and if P is assumed to represent such a pair, $P(\overline{0})$ and $P(\overline{1})$ represent the left and right projections, $(P)_0$ and $(P)_1$. For clarity, I will use the latter notations.

Now, let us remember where we stand. We need to show that given any terms, G and H, we can find a term F such that

$$\begin{aligned} F(\overline{0}) &\equiv G \\ F(\overline{n+1}) &\equiv H(\overline{n}, F(\overline{n})) \end{aligned}$$

for every natural number n. The idea is roughly to compute sequences of *pairs*

$$\langle \overline{0}, F(\overline{0})\rangle, \langle \overline{1}, F(\overline{1})\rangle, \ldots,$$

using numerals as iterators. Notice that the first pair is just $\langle\overline{0}, G\rangle$. Given a pair $\langle\overline{n}, F(\overline{n})\rangle$, the next pair, $\langle\overline{n+1}, F(\overline{n+1})\rangle$ is supposed to be equivalent to $\langle\overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle$. We will design a lambda term T that makes this one-step transition.

The last paragraph was simply heuristic; the details are as follows. Define $T(u)$ by

$$T(u) = \langle S((u)_0), H((u)_0, (u)_1)\rangle.$$

Now it is easy to verify that for any number n,

$$T(\langle\overline{n}, M\rangle) \triangleright \langle\overline{n+1}, H(\overline{n}, M)\rangle.$$

As suggested above, given G and H, define $F(u)$ by

$$F(u) = (u(T, \langle\overline{0}, G\rangle))_1.$$

In other words, on input $\overline{n}$, F iterates T n times on $\langle\overline{0}, G\rangle$, and then returns the second component. To start with, we have

- $\overline{0}(T, \langle\overline{0}, G\rangle) \equiv \langle\overline{0}, G\rangle$
- $F(\overline{0}) \equiv G$

By induction on n, we can show that for each natural number one has the following:

- $\overline{n+1}(T, \langle\overline{0}, G\rangle) \equiv \langle\overline{n+1}, F(\overline{n+1})\rangle$
- $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$

For the second clause, we have

$$\begin{aligned} F(\overline{n+1}) &\triangleright (\overline{n+1}(T, \langle\overline{0}, G\rangle))_1 \\ &\equiv (T(\overline{n}(T, \langle\overline{0}, G\rangle)))_1 \\ &\equiv (T(\langle\overline{n}, F(\overline{n})\rangle))_1 \\ &\equiv (\langle\overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle)_1 \\ &\equiv H(\overline{n}, F(\overline{n})). \end{aligned}$$

Here we have used the induction hypothesis on the second-to-last line. For the first clause, we have

$$\begin{aligned} \overline{n+1}(T, \langle\overline{0}, G\rangle) &\equiv T(\overline{n}(T, \langle\overline{0}, G\rangle)) \\ &\equiv T(\langle\overline{n}, F(\overline{n})\rangle) \\ &\equiv \langle\overline{n+1}, H(\overline{n}, F(\overline{n}))\rangle \\ &\equiv \langle\overline{n+1}, F(\overline{n+1})\rangle. \end{aligned}$$

Here we have used the second clause in the last line. So we have shown $F(\overline{0}) \equiv G$ and, for every n, $F(\overline{n+1}) \equiv H(\overline{n}, F(\overline{n}))$, which is exactly what we needed.

The only thing left to do is to show that the partial functions represented by lambda terms are closed under the μ operation, i.e. unbounded search. But it will be much easier to do this later on, after we have discussed the fixed-point theorem. So, take this as an IOU. Modulo this claim (and some details that have been left for you to work out), we have proved Theorem 2.8.5.

Chapter 3

Computability Theory

3.1 Generalities

The branch of logic known as *Computability Theory* deals with issues having to do with the computability, or relative computability, of functions and sets. From the last chapter, we know that we can take the word "computable" to mean "Turing computable" or, equivalently, "recursive." It is a evidence of Kleene's influence that the subject used to be known as *Recursion Theory*, and today, both names are commonly used.

Most introductions to Computability Theory begin by trying to abstract away the general features of computability as much as possible, so that one can explore the subject without having to refer to a specific model of computation. For example, we have seen that there is a universal partial computable function, $Un(n, x)$. This allows us to enumerate the partial computable functions; from now on, we will adopt the notation φ_n to denote the nth unary partial computable function, defined by $\varphi_n(x) \simeq Un(n, x)$. (Kleene used $\{n\}$ for this purpose, but this notation has not been used as much recently.) Slightly more generally, we can uniformly enumerate the partial computable functions of arbitrary arities, and I will use φ_n^k to denote the nth k-ary partial recursive function. The key fact is that there is a universal function for this set. In other words:

Theorem 3.1.1 *There is a partial computable function $f(x, y)$ such that for each n and k and sequence of numbers $a_0, \ldots, a_{k-1}$ we have*

$$f(n, \langle a_0, \ldots, a_{k-1} \rangle) \simeq \varphi_n^k(a_0, \ldots, a_{k-1}).$$

In fact, we can take $f(n, x)$ to be $Un(n, x)$, and define $\varphi_n^k(a_0, \ldots, a_{k-1}) \simeq Un(n, \langle a_0, \ldots, a_{k-1} \rangle)$. Alternatively, you can think of f as the partial com-

putable function that, on input n and $\langle a_0, \ldots, a_{k-1} \rangle$, returns the output of Turing machine n on input $a_0, \ldots, a_{k-1}$.

Remember also Kleene's normal form theorem:

Theorem 3.1.2 *There is a primitive recursive relation $T(n, x, s)$ and a primitive recursive function U such that for each recursive function f there is a number n, such that*

$$f(x) \simeq U(\mu s T(n, x, s)).$$

In fact, T and U can be used to define the enumeration $\varphi_0, \varphi_1, \varphi_2, \ldots$. From now on, we will assume that we have fixed a suitable choice of T and U, and take the equation

$$\varphi_n(x) \simeq U(\mu s T(n, x, s))$$

to be the *definition* of φ_n.

The next theorem is known as the "s-m-n theorem," for a reason that will be clear in a moment. The hard part is understanding just what the theorem says; once you understand the statement, it will seem fairly obvious.

Theorem 3.1.3 *For each pair of natural numbers n and m, there is a primitive recursive function s_n^m such that for every sequence $x, a_0, \ldots, a_{m-1}, y_0, \ldots, y_{n-1}$, we have*

$$\varphi^n_{s_n^m(x, a_0, \ldots, a_{m-1})}(y_0, \ldots, y_{n-1}) \simeq \varphi_x^{m+n}(a_0, \ldots, a_{m-1}, y_0, \ldots, y_{n-1}).$$

It is helpful to think of s_n^m as acting on *programs*. That is, s_n^m takes a program, x, for an $(m+n)$-ary function, as well as fixed inputs $a_0, \ldots, a_{m-1}$; and it returns a program, $s_n^m(x, a_0, \ldots, a_{m-1})$, for the n-ary function of the remaining arguments. It you think of x as the description of a Turing machine, then $s_n^m(x, a_0, \ldots, a_{m-1})$ is the Turing machine that, on input $y_0, \ldots, y_{n-1}$, prepends $a_0, \ldots, a_{m-1}$ to the input string, and runs x. Each s_n^m is then just a primitive recursive function that finds a code for the appropriate Turing machine.

Here is another useful fact:

Theorem 3.1.4 *Every partial computable function has infinitely many indices.*

Again, this is intuitively clear. Given any Turing machine, M, one can design another Turing machine M' that twiddles its thumbs for a while, and then acts like M.

Throughout this chapter, we will reason about what types of things are computable. To show that a function *is* computable, there are two ways one can proceed:

1. Rigorously: describe a Turing machine or partial recursive function explicitly, and show that it computes the function you have in mind;

2. Informally: describe an algorithm that computes it, and appeal to Church's thesis.

There is no fine line between the two; a very detailed description of an algorithm should provide enough information so that it is relatively clear how one could, in principle, design the right Turing machine or sequence of partial recursive definitions. Fully rigorous definitions (like sequences of 4-tuples or Turing machine diagrams) are unlikely to be informative, and we will try to find a happy medium between these two approaches; in short, we will try to find intuitive yet rigorous proofs that the precise definitions could be obtained.

3.2 Computably enumerable sets

Remember that a function is *computable* if and only if there is a Turing machine that computes it. We can extend the notion of computability to sets:

Definition 3.2.1 *Let S be a set of natural numbers. Then S is* computable *if and only if its characteristic function is; i.e. the function*

$$\chi_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

is computable. Similarly, a relation $R(x_0, \ldots, x_{k-1})$ is computable if and only if its characteristic function is.

Notice that we now have a number of notions of computability: for partial functions, for functions, and for sets. Do not get them confused! The Turing machine computing a partial function returns the output of the function, for input values at which the function is defined; the Turing machine computing a set returns either 1 or 0, after deciding whether or not the input value is in the set or not. Computable sets are sometimes also called *decidable*.

Here is yet another notion:

Definition 3.2.2 *A set is* computably enumerable *if it is empty or the range of a computable function.*

The textbook uses the term *recursively enumerable* instead. This is the original terminology, and today both are commonly used, as well as the abbreviations "c.e." and "r.e." You should think about what the definition means, and why the terminology is appropriate. The idea is that if S is the range of the computable function f, then

$$S = \{f(0), f(1), f(2), \ldots\},$$

and so f can be seen as "enumerating" the elements of S. Note that according to the definition, f need not be an increasing function, i.e. the enumeration need not be in increasing order. In fact, f need not even be injective, so that the constant function $f(x) = 0$ enumerates the set $\{0\}$.

Any computable set is computably enumerable. To see this, suppose S is computable. If S is empty, then by definition it is computably enumerable. Otherwise, let a be any element of S. Define f by

$$f(x) = \begin{cases} x & \text{if } \chi_S(x) = 1 \\ a & \text{otherwise.} \end{cases}$$

Then f is a computable function, and S is the range of f.

The following gives a number of important equivalent statements of what it means to be computably enumerable.

Theorem 3.2.3 *Let S be a set of natural numbers. Then the following are equivalent:*

1. *S is computably enumerable.*
2. *S is the range of a* partial *computable function.*
3. *S is empty or the range of a primitive recursive function.*
4. *S is the* domain *of a partial computable function.*

The first three clauses say that we can equivalently take any nonempty computably enumerable set to be enumerated by either a computable function, a partial computable function, or a primitive recursive function. The fourth clause tells us that if S is computably enumerable, then for some index e,

$$S = \{x \mid \varphi_e(x) \downarrow\}.$$

If we take e to code a Turing machine, then S is the set of inputs on which the Turing machine halts. For that reason, computably enumerable sets are

sometimes called *semi-decidable*: if a number is in the set, you eventually get a "yes," but if it isn't, you never get a "no"!

Proof. Since every primitive recursive function is computable and every computable function is partial computable, 3 implies 1 and 1 implies 2. (Note that if S is empty, S is the range of the partial computable function that is nowhere defined.) If we show that 2 implies 3, we will have shown the first three clauses equivalent.

So, suppose S is the range of the partial computable function φ_e. If S is empty, we are done. Otherwise, let a be any element of S. By Kleene's normal form theorem, we can write

$$\varphi_e(x) = U(\mu s T(e, x, s)).$$

In particular, $\varphi_e(x) \downarrow = y$ if and only if there is an s such that $T(e, x, s)$ and $U(s) = y$. Define $f(z)$ by

$$f(z) = \begin{cases} U((z)_1) & \text{if } T(e, (z)_0, (z)_1) \\ a & \text{otherwise.} \end{cases}$$

Then f is primitive recursive, because T and U are. Expressed in terms of Turing machines, if z codes a pair $\langle (z)_0, (z)_1 \rangle$ such that $(z)_1$ is a halting computation of machine e on input $(z)_0$, then f returns the output of the computation; otherwise, it returns a.

We need to show that S is the range of f, i.e. for any natural number y, y is in S if and only if it is in the range of f. In the forwards direction, suppose y in S. Then y is in the range of φ_e, so for some x and s, $T(e, x, s)$ and $U(s) = y$; but then $y = f(\langle x, s \rangle)$. Conversely, suppose y is in the range of f. Then either $y = a$, or for some z, $T(e, (z)_0, (z)_1)$ and $U((z)_1) = y$. Since, in the latter case, $\varphi_e(x) \downarrow = y$, either way, y is in S.

(Note that I am using the notation $\varphi_e(x) \downarrow = y$ to mean "$\varphi_e(x)$ is defined and equal to y." I could just as well have written $\varphi_e(x) = y$, but the extra arrow is sometimes helpful in reminding us that we are dealing with a partial function.)

To finish up the proof of Theorem 3.2.3, it suffices to show that 1 and 4 are equivalent. First, let us show that 1 implies 4. Suppose S is the range of a computable function f, i.e.

$$S = \{y \mid \text{for some } x,\ f(x) = y\}.$$

Let

$$g(y) = \mu x (f(x) = y).$$

Then g is a partial computable function, and $g(y)$ is defined if and only if for some x, $f(x) = y$; in other words, the domain of g is the range of f. Expressed in terms of Turing machines: given a Turing machine M_f that enumerates the elements of S, let M_g be the Turing machine the semi-decides S, by searching through the range of f to see if a given element is in the set.

Finally, to show 4 implies 1, suppose that S is the domain of the partial computable function φ_e, i.e.

$$S = \{x \mid \varphi_e(x) \downarrow\}.$$

If S is empty, we are done; otherwise, let a be any element of S. Define f by

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

Then, as above, a number x is in the range of f if and only if $\varphi_e(x) \downarrow$, i.e. if and only if x is in S. Expressed in terms of Turing machines: given a machine M_e that semi-decides S, enumerate the elements of S by running through all possible Turing machine computations, and returning the inputs that correspond to halting computations. □

The fourth clause of Theorem 3.2.3 provides us with a convenient way of enumerating the computably enumerable sets: for each e, let W_e denote the domain of φ_e. Then if A is any computably enumerable set, $A = W_e$, for some e.

The following provides yet another characterization of the computably enumerable sets.

Theorem 3.2.4 *A set S is computably enumerable if and only if there is a computable relation $R(x, y)$ such that*

$$S = \{x \mid \exists y\ R(x, y)\}.$$

Proof. In the forward direction, suppose S is computably enumerable. Then for some e, $S = W_e$. Then for this value of e we can write S as

$$S = \{x \mid \exists y\ T(e, x, y)\}.$$

In the reverse direction, suppose $S = \{x \mid \exists y\ R(x, y)\}$. Define f by

$$f(x) \simeq \mu y R(x, y).$$

Then f is partial computable, and S is the domain of f. □

We saw above that every computable set is computably enumerable. Is the converse true? The following shows that, in general, it is not.

Theorem 3.2.5 *Let K_0 be the set $\{\langle e, x\rangle \mid \varphi_e(x) \downarrow\}$. Then K_0 is computably enumerable but not computable.*

Proof. To see that K_0 is computably enumerable, note that it is the domain of the function f defined by

$$f(z) = \mu y(\mathit{length}(z) = 2 \wedge T((z)_0, (z)_1, y)).$$

For, if $\varphi_e(x)$ is defined, $f(\langle e, x\rangle)$ finds a halting computation sequence; if $\varphi_e(x)$ is undefined, so is $f(\langle e, x\rangle)$; and if z doesn't even code a pair, then $f(z)$ is also undefined.

The fact that K_0 is not computable is just Theorem 2.7.7 from Chapter 2. □

The following theorem gives some closure properties on the set of computably enumerable sets.

Theorem 3.2.6 *Suppose A and B are computably enumerable. Then so are $A \cap B$ and $A \cup B$.*

Proof. Theorem 3.2.3 allows us to use various characterizations of the computably enumerable sets. By way of illustration, I will use provide a few different proofs; I will also word the proofs both in terms of partial computable functions, and Turing machine computations.

For the first proof, suppose A is enumerated by a computable function f, and B is enumerated by a computable function g. Let

$$h(x) = \mu y(f(y) = x \vee g(y) = x)$$

and

$$j(x) = \mu y(f((y)_0) = x \wedge g((y)_1) = x).$$

Then $A \cup B$ is the domain of h, and $A \cap B$ is the domain of j. Here is what is going on, in computational terms: given procedures that enumerate A and B, we can semi-decide if an element x is in $A \cup B$ by looking for x in either enumeration; and we can semi-decide if an element x is in $A \cap B$ for looking for x in both enumerations at the same time.

For the second proof, suppose again that A is enumerated by f and B is enumerated by g. Let

$$k(x) = \begin{cases} f(x/2) & \text{if } x \text{ is even} \\ g((x-1)/2) & \text{if } x \text{ is odd.} \end{cases}$$

Then k enumerates $A \cup B$; the idea is that k just alternates between the enumerations offered by f and g. Enumerating $A \cap B$ is tricker. If $A \cap B$ is empty, it is trivially computably enumerable. Otherwise, let c be any element of $A \cap B$, and define l by

$$l(x) = \begin{cases} f((x)_0) & \text{if } f((x)_0) = g((x)_1) \\ c & \text{otherwise.} \end{cases}$$

In computational terms, l runs through pairs of elements in the enumerations of f and g, and outputs every match it finds; otherwise, it just stalls by outputting c.

For the last proof, suppose A is the *domain* of the partial function $m(x)$ and B is the domain of the partial function $n(x)$. Then $A \cap B$ is the domain of the partial function $m(x) + n(x)$. In computational terms, if A is the set of values for which m halts and B is the set of values for which n halts, $A \cap B$ is the set of values for which both procedures halt. Expressing $A \cup B$ as a set of halting values is more difficult, because one has to simulate m and n in parallel. Let d be an index for m and let e be an index for n; in other words, $m = \varphi_d$ and $n = \varphi_e$. Then $A \cup B$ is the domain of the function

$$p(x) = \mu y(T(d, x, y) \vee T(e, x, y)).$$

In computational terms, on input x, p searches for either a halting computation for m or a halting computation for n, and halts if it finds either one. □

Suppose A is computably enumerable. Is the complement of A, $\overline{A}$, necessarily computably enumerable as well? The following theorem and corollary show that the answer is "no."

Theorem 3.2.7 *Let A be any set of natural numbers. Then A is computable if and only if both A and $\overline{A}$ are computably enumerable.*

Proof. The forwards direction is easy: if A is computable, then $\overline{A}$ is computable as well ($\chi_A = 1 \dot{-} \chi_{\overline{A}}$), and so both are computably enumerable.

In the other direction, suppose A and $\overline{A}$ are both computably enumerable. Let A be the domain of φ_d, and let $\overline{A}$ be the domain of φ_e. Define h by

$$h(x) = \mu s(T(e, x, s) \vee T(f, x, s)).$$

In other words, on input x, h searches for either a halting computation of φ_d or a halting computation of φ_e. Now, if x is in A, it will succeed in the first case, and if x is in $\overline{A}$, it will succeed in the second case. So, h is a total computable function. But now we have that for every x, $x \in A$ if and only if $T(e, x, h(x))$, i.e. if φ_e is the one that is defined. Since $T(e, x, h(x))$ is a computable relation, A is computable.

It is easier to understand what is going on in informal computational terms: to decide A, on input x search for halting computations of φ_e and φ_f. One of them is bound to halt; if it is φ_e, then x is in A, and otherwise, x is in $\overline{A}$. □

Corollary 3.2.8 *$\overline{K}_0$ is not computably enumerable.*

Proof. We know that K_0 is computably enumerable, but not computable. If $\overline{K}_0$ were computably enumerable, then K_0 would be computable by Theorem 3.2.7. □

3.3 Reducibility and Rice's theorem

We now know that there is at least one set, K_0, that is computably enumerable but not computable. It should be clear that there are others. The method of reducibility provides a very powerful method of showing that other sets have these properties, without constantly having to return to first principles.

Generally speaking, a "reduction" of a set A to a set B is a method of transforming answers to whether or not elements are in B into answers as to whether or not elements are in A. We will focus on a notion called "many-one reducibility," but there are many other notions of reducibility available, with varying properties. Notions of reducibility are also central to the study of computational complexity, where efficiency issues have to be considered as well. For example, a set is said to be "NP-complete" if it is in NP and every NP problem can be reduced to it, using a notion of reduction

that is similar to the one described below, only with the added requirement that the reduction can be computed in polynomial time.

We have already used this notion implicitly. Define the set K by

$$K = \{x \mid \varphi_x(x) \downarrow\},$$

i.e. $K = \{x \mid x \in W_x\}$. Our proof that the halting problem in unsolvable, Theorem 2.7.7, shows most directly that K is not computable. Recall that K_0 is the set

$$K_0 = \{\langle e, x\rangle \mid \varphi_e(x) \downarrow\}.$$

i.e. $K_0 = \{x \mid x \in W_e\}$. It is easy to extend any proof of the uncomputability of K to the uncomputability of K_0: if K_0 were computable, we could decide whether or not an element x is in K simply by asking whether or not the pair $\langle x, x\rangle$ is in K_0. The function f which maps x to $\langle x, x\rangle$ is an example of a *reduction* of K to K_0.

Definition 3.3.1 *Let A and B be sets. Then A is said to be* many-one reducible *to B, written $A \leq_m B$, if there is a computable function f such that for every natural number x,*

$$x \in A \quad \textit{if and only if} \quad f(x) \in B.$$

If A is many-one reducible to B and vice-versa, then A and B are said to be many-one equivalent*, written $A \equiv_m B$.*

If the function f in the definition above happens to be injective, A is said to be *one-one reducible* to B. Most of the reductions described below meet this stronger requirement, but we will not use this fact. As an aside, let me remark that it is true, but by no means obvious, that one-one reducibility really is a stronger requirement than many-one reducibility; in other words, there are infinite sets A and B such that A is many-one reducible to B but not one-one reducible to B.

The intuition behind writing $A \leq_m B$ is that A is "no harder than" B. The following two propositions support this intuition.

Proposition 3.3.2 *If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.*

Proof. Composing a reduction of A to B with a reduction of B to C yields a reduction of A to C. (You should check the details!) □

Proposition 3.3.3 *Let A and B be any sets, and suppose A is many-one reducible to B.*

- *If B is computably enumerable, so is A.*
- *If B is computable, so is A.*

Proof. Let f be a many-one reduction from A to B. For the first claim, just check that if B is the domain of partial function g, then A is the domain of $g \circ f$:

$$\begin{aligned} x \in A &\leftrightarrow f(x) \in B \\ &\leftrightarrow g(f(x)) \downarrow . \end{aligned}$$

For the second claim, remember that if B is computable then B and $\overline{B}$ are computably enumerable. It is not hard to check that f is also a many-one reduction of $\overline{A}$ to $\overline{B}$, so, by the first part of this proof, A and $\overline{A}$ are computably enumerable. So A is computable as well. (Alternatively, you can check that $\chi_A = \chi_B \circ f$; so if χ_B is computable, then so is χ_A.) □

As another aside, let me mention that a more general notion of reducibility called *Turing reducibility* is useful in other contexts, especially for proving undecidability results. Note that by Corollary 3.2.8, the complement of K_0 is not reducible to K_0, since it is not computably enumerable. But, intuitively, if you knew the answers to questions about K_0, you would know the answer to questions about its complement as well. A set A is said to be Turing reducible to B if one can determine answers to questions in A using a computable procedure that asks questions about B. This is more liberal than many-one reducibility, in which (1) you are only allowed to ask one question about B, and (2) a "yes" answer has to translate to a "yes" answer to the question about A, and similarly for "no." It is still the case that if A is Turing reducible to B and B is computable then A is computable as well (though, as we have seen, the analogous statement does not hold for computable enumerability).

You should think about the various notions of reducibility we have discussed in this section, and understand the distinctions between them. We will, however, only deal with many-one reducibility in this chapter. Incidentally, both types of reducibility discussed in the last paragraph have analogues in computational complexity, with the added requirement that the Turing machines run in polynomial time: the complexity version of many-one reducibility is known as *Karp reducibility*, while the complexity version of Turing reducibility is known as *Cook reducibility*.

Let us consider an application of Proposition 3.3.3.

Proposition 3.3.4 *Let*

$$K_1 = \{e \mid \varphi_e(0) \downarrow\}.$$

Then K_1 is computably enumerable but not computable.

Proof. Since $K_1 = \{e \mid \exists s\ T(e, 0, s)\}$, K_1 is computably enumerable by Theorem 3.2.4.

To show that K_1 is not computable, let us show that K_0 is reducible to it. This is a little bit tricky, since using K_1 we can only ask questions about computations that start with a particular input, 0. Suppose you have a smart friend that can answer questions of this type (friends like this are known as "oracles"). The suppose someone comes up to you and asks you whether or not $\langle e, x\rangle$ is in K_0, that is, whether or not machine e halts on input x. One thing you can do is build another machine, e_x, that, for *any* input, ignores that input and runs e on input x. Then clearly the question as to whether machine e halts on input x is equivalent to the question as to whether machine e_x halts on input 0 (or any other input). So, then you ask your friend whether this new machine, e_x, halts on input 0; your friend's answer to the modified question provides the answer to the original one. This provides the desired reduction of K_0 to K_1.

More formally, using the universal partial computable function, let f be the 3-ary function defined by

$$f(x, y, z) \simeq \varphi_x(y).$$

Note that f ignores its third input entirely. Pick an index e such that $f = \varphi^3_e$; so we have

$$\varphi^3_e(x, y, z) \simeq \varphi_x(y).$$

By the s-m-n theorem, there is a function $s(e, x, y)$ such that, for every z,

$$\begin{aligned} \varphi_{s(e,x,y)}(z) &\simeq \varphi^3_e(x, y, z) \\ &\simeq \varphi_x(y). \end{aligned}$$

In terms of the informal argument above, $s(e, x, y)$ is an index for the machine that, for any input z, ignores that input and computes $\varphi_x(y)$. In particular, we have

$$\varphi_{s(e,x,y)}(0) \downarrow \quad \text{if and only if} \quad \varphi_x(y) \downarrow .$$

In other words, $\langle x, y\rangle \in K_0$ if and only if $s(e, x, y) \in K_1$. So the function g defined by

$$g(w) = s(e, (w)_0, (w)_1)$$

is a reduction of K_0 to K_1. □

Definition 3.3.5 *A set A is said to be a* complete computably enumerable set *(under many-one reducibility) if*

- *A is computably enumerable, and*
- *for any other computably enumerable set B, $B \leq_m A$.*

In other words, complete computably enumerable sets are the "hardest" computably enumerable sets possible; they allow one to answer questions about *any* computably enumerable set.

Theorem 3.3.6 *K, K_0, and K_1 are all complete computably enumerable sets.*

Proof. To see that K_0 is complete, let B be any computably enumerable set. Then for some index e,

$$B = W_e = \{x \mid \varphi_e(x) \downarrow\}.$$

Let f be the function $f(x) = \langle e, x\rangle$. Then for every natural number x, $x \in B$ if and only if $f(x) \in K_0$. In other words, f reduces B to K_0.

To see that K_1 is complete, note that in the last theorem we reduced K_0 to it. So, by Proposition 3.3.2, any computably enumerable set can be reduced to K_1 as well. K can be reduced to K_0 in much the same way. □

So, it turns out that all the examples of computably enumerable sets that we have considered so far are either computable, or complete. This should seem strange! Are there any examples of computably enumerable sets that are neither computable nor complete? The answer is yes, but it wasn't until the middle of the 1950's that this was established by Friedberg and Muchnik, independently.

Let us consider one more example of using the s-m-n theorem to show that something is noncomputable. Let *Tot* be the set of indices of total computable functions, i.e.

$$Tot = \{x \mid \text{for every } y,\ \varphi_x(y) \downarrow\}.$$

Proposition 3.3.7 *Tot is not computable.*

Proof. It turns out that *Tot* is not even computably enumerable — its complexity lies further up on the "arithmetic hierarchy." But we will not worry about this strengthening here.

To see that *Tot* is not computable, it suffices to show that K is reducible to it. Let $h(x, y)$ be defined by

$$h(x,y) \simeq \begin{cases} 0 & \text{if } x \in K \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that $h(x, y)$ does not depend on y at all. By now, it should not be hard to see that h is partial computable: on input x, y, the program computing h first simulates machine x on input x; if this computation machine halts, $h(x, y)$ outputs 0 and halts. So $h(x, y)$ is just $Z(\mu s\ T(x, x, s))$, where Z is the constant zero function.

Using the s-m-n theorem, there is a primitive recursive function $k(x)$ such that for every x and y,

$$\varphi_{k(x)}(y) = \begin{cases} 0 & \text{if } x \in K \\ \text{undefined} & \text{otherwise} \end{cases}$$

So $\varphi_{k(x)}$ is total if $x \in K$, and undefined otherwise. Thus, k is a reduction of K to *Tot*. □

If you think about it, you will see that the specifics of *Tot* do not play into the proof above. We designed $h(x, y)$ to act like the constant function $j(y) = 0$ exactly when x is in K; but we could just as well have made it act like any other partial computable function under those circumstances. This observation lets us state a more general theorem, which says, roughly, that no nontrivial property of computable functions is decidable.

Keep in mind that $\varphi_0, \varphi_1, \varphi_2, \ldots$ is our standard enumeration of the partial computable functions.

Theorem 3.3.8 *(Rice's theorem) Let C be any set of partial computable functions, and let $A = \{n \mid \varphi_n \in C\}$. If A is computable, then either C is $\emptyset$ or C is the set of all the partial computable functions.*

An *index set* is a set A with the property that if n and m are indices which "compute" the same function, then either both n and m are in A, or neither is. It is not hard to see that the set A in the theorem has this

property. Conversely, if A is an index set and C is the set of functions computed by these indices, then $A = \{n \mid \varphi_n \in C\}$.

With this terminology, Rice's theorem is equivalent to saying that no nontrivial index set is decidable. To understand what the theorem says, it is helpful to emphasize the distinction between *programs* (say, in your favorite programming language) and the functions they compute. There are certainly questions about programs (indices), which are syntactic objects, that are computable: does this program have more than 150 symbols? does it have more than 22 lines? Does it have a "while" statement? Does the string "hello world" every appear in the argument to a "print" statement? Rice's theorem says that no nontrivial question about the program's *behavior* is computable. This includes questions like these: does the program halt on input 0? Does it ever halt? Does it ever output an even number?

Proof of Rice's theorem. Suppose C is neither $\emptyset$ nor the set of all the partial computable functions, and let A be the set of indices of functions in C. I will show that if A were computable, we could solve the halting problem; so A is not computable.

Without loss of generality, we can assume that the function f which is nowhere defined is not in C (otherwise, switch C and its complement in the argument below). Let g be any function in C. The idea is that if we could decide A, we could tell the difference between indices computing f, and indices computing g; and then we could use that capability to solve the halting problem.

Here's how. Using the universal computation predicate, we can define a function

$$h(x,y) \simeq \begin{cases} \text{undefined} & \text{if } \varphi_x(x) \uparrow \\ g(y) & \text{otherwise} \end{cases}$$

To compute h, first we try to compute $\varphi_x(x)$; if that computation halts, we go on to compute $g(y)$; and if *that* computation halts, we return the output. More formally, we can write

$$h(x,y) \simeq P_0^2(g(y), Un(x,x)).$$

This is a composition of partial computable functions, and the right side is defined and equal to $g(y)$ just when $Un(x,x)$ and $g(y)$ are both defined.

Notice that for a fixed x, if $\varphi_x(x)$ is undefined, then $h(x,y)$ is undefined for every y; and if $\varphi_x(x)$ is defined, then $h(x,y) \simeq g(y)$. So, for any fixed value of x, either $h(x,y)$ acts just like f or it acts just like g, and deciding whether or not $\varphi_x(x)$ is defined amounts to deciding which of these two

cases holds. But this amounts to deciding whether or not $\lambda y\ h(x,y)$ is in C or not, and if A were computable, we could do just that.

More formally, since h is partial computable, it is equal to the function φ_k for some index k. By the s-m-n theorem there is a primitive recursive function s such that for each x, $\varphi_{s(k,x)} = \lambda y\ h(x,y)$. Now we have that for each x, if $\varphi_x(x)\downarrow$, then $\varphi_{s(k,x)}$ is the same function as g, and so $s(k,x)$ is in A. On the other hand, if $\varphi_x(x)\uparrow$, then $\varphi_{s(k,x)}$ is the same function as f, and so $s(k,x)$ is not in A. In other words we have that for every x, $x \in K$ if and only if $s(k,x) \in A$. If A were computable, K would be also, which is a contradiction. So A is not computable. □

Rice's theorem is very powerful. The following immediate corollary shows some sample applications.

Corollary 3.3.9 *The following sets are undecidable.*

- $\{x \mid$ *17 is in the range of* $\varphi_x\}$
- $\{x \mid \varphi_x$ *is constant*$\}$
- $\{x \mid \varphi_x$ *is total*$\}$
- $\{x \mid$ *whenever* $y < y'$, $\varphi(y)\downarrow$, *and* $\varphi(y')\downarrow$, *then* $\varphi(y) < \varphi(y')\}$

Proof. These are all nontrivial index sets. □

3.4 The fixed-point theorem

Let's consider the halting problem again. As temporary notation, let us write $\ulcorner\varphi_x(y)\urcorner$ for $\langle x,y\rangle$; think of this as representing a "name" for the value $\varphi_x(y)$. With this notation, we can reword one of our proofs that the halting problem is undecidable.

Question: is there a computable function h, with the following property? For every x and y,

$$h(\ulcorner\varphi_x(y)\urcorner) = \begin{cases} 1 & \text{if } \varphi_x(y)\downarrow \\ 0 & \text{otherwise} \end{cases}$$

Answer: No; otherwise, the partial function

$$g(x) \simeq \begin{cases} 0 & \text{if } h(\ulcorner\varphi_x(x)\urcorner) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

would be computable, with some index, e. But then we have

$$\varphi_e(e) \simeq \begin{cases} 0 & \text{if } h(\ulcorner \varphi_e(e) \urcorner) = 0 \\ \text{undefined} & \text{otherwise,} \end{cases}$$

in which case $\varphi_e(e)$ is defined if and only if it isn't, a contradiction.

Now, take a look at the equation with φ_e. There is an instance of self-reference there, in a sense: we have arranged for the value of $\varphi_e(e)$ to depend on $\ulcorner \varphi_e(e) \urcorner$, in a certain way. The fixed-point theorem says that we *can* do this, in general – not just for the sake of proving contradictions.

Lemma 3.4.1 gives two equivalent ways of stating the fixed-point theorem. Logically speaking, the fact that the statements are equivalent follows from the fact that they are both true; but what I really mean is that each one follows straightforwardly from the other, so that they can be taken as alternative statements of the same theorem.

Lemma 3.4.1 *The following statements are equivalent:*

1. *For every partial computable function $g(x, y)$, there is an index e such that for every y,*
$$\varphi_e(y) \simeq g(e, y).$$
2. *For every computable function $f(x)$, there is an index e such that for every y,*
$$\varphi_e(y) \simeq \varphi_{f(e)}(y).$$

Proof. $1 \Rightarrow 2$: Given f, define g by $g(x, y) \simeq Un(f(x), y)$. Use 1 to get an index e such that for every y,

$$\begin{aligned} \varphi_e(y) &= Un(f(e), y) \\ &= \varphi_{f(e)}(y). \end{aligned}$$

$2 \Rightarrow 1$: Given g, use the s-m-n theorem to get f such that for every x and y, $\varphi_{f(x)}(y) \simeq g(x, y)$. Use 2 to get an index e such that

$$\begin{aligned} \varphi_e(y) &= \varphi_{f(e)}(y) \\ &= g(e, y). \end{aligned}$$

This concludes the proof. □

Before showing that statement 1 is true (and hence 2 as well), consider how bizarre it is. Think of e as being a computer program; statement 1 says that given any partial computable $g(x, y)$, you can find a computer program e that computes $\lambda y\ g(e, y)$. In other words, you can find a computer program that computes a function that references the program itself.

Theorem 3.4.2 *The two statements in Lemma 3.4.1 are true.*

Proof. It suffices to prove statement 1. The ingredients are already implicit in the discussion of the halting problem above. Let $diag(x)$ be a computable function which for each x returns an index for the function $\lambda y\ \varphi_x(x, y)$, i.e.

$$\varphi_{diag(x)} \simeq \lambda y\ \varphi_x(x, y).$$

Think of *diag* as a function that transforms a program for a 2-ary function into a program for a 1-ary function, obtained by fixing the original program as its first argument. The function *diag* can be defined formally as follows: first define s by

$$s(x, y) \simeq Un^2(x, x, y),$$

where *Un* is a 3-ary function that is universal for computable 2-ary functions. Then, by the s-m-n theorem, we can find a primitive recursive function *diag* satisfying

$$\varphi_{diag(x)}(y) \simeq s(x, y).$$

Now, define the function l by

$$l(x, y) \simeq g(diag(x), y).$$

and let $\ulcorner l \urcorner$ be an index for l. Finally, let $e = diag(\ulcorner l \urcorner)$. Then for every y, we have

$$\begin{aligned} \varphi_e(y) &\simeq \varphi_{diag(\ulcorner l \urcorner)}(y) \\ &\simeq \varphi_{\ulcorner l \urcorner}(\ulcorner l \urcorner, y) \\ &\simeq l(\ulcorner l \urcorner, y) \\ &\simeq g(diag(\ulcorner l \urcorner), y) \\ &\simeq g(e, y), \end{aligned}$$

as required. □

What's going on? The following heuristic might help you understand the proof.

Suppose you are given the task of writing a computer program that prints itself out. Suppose further, however, that you are working with a programming language with a rich and bizarre library of string functions. In particular, suppose your programming language has a function *diag* which works as follows: given an input string s, *diag* locates each instance of the symbol 'x' occuring in s, and replaces it by a quoted version of the original string. For example, given the string

```
hello x world
```

as input, the function returns

```
hello 'hello x world' world
```

as output. In that case, it is easy to write the desired program; you can check that

```
print(diag('print(diag(x))'))
```

does the trick. For more common programming languages like C++ and Java, the same idea (with a more involved implementation) still works.

We are only a couple of steps away from the proof of the fixed-point theorem. Suppose a variant of the print function $print(x, y)$ accepts a string x and another numeric argument y, and prints the string x repeatedly, y times. Then the "program"

```
getinput(y);print(diag('getinput(y);print(diag(x),y)'),y)
```

prints itself out y times, on input y. Replacing the *getinput–print–diag* skeleton by an arbitrary funtion $g(x, y)$ yields

```
g(diag('g(diag(x),y)'),y)
```

which is a program that, on input y, runs g on the program itself and y. Thinking of "quoting" with "using an index for," we have the proof above.

For now, it is o.k. if you want to think of the proof as formal trickery, or black magic. But you should be able to reconstruct the details of the argument given above. When we prove the incompleteness theorems (and the related "fixed-point theorem") we will discuss other ways of understanding why it works.

Let me also show that the same idea can be used to get a "fixed point" combinator. Suppose you have a lambda term g, and you want another term k with the property that k is β-equivalent to gk. Define terms

$$diag(x) = xx$$

and

$$l(x) = g(diag(x))$$

using our notational conventions; in other words, l is the term $\lambda x.g(xx)$. Let k be the term ll. Then we have

$$\begin{aligned} k &= (\lambda x.g(xx))(\lambda x.g(xx)) \\ &\triangleright\ g((\lambda x.g(xx))(\lambda x.g(xx))) \\ &= gk. \end{aligned}$$

If one takes

$$Y = \lambda g\ ((\lambda x.\ g(xx))(\lambda x.\ g(xx)))$$

then Yg and $g(Yg)$ reduce to a common term; so $Yg \equiv_\beta g(Yg)$. This is known as "Curry's combinator." If instead one takes

$$Y = (\lambda xg.\ g(xxg))(\lambda xg.\ g(xxg))$$

then in fact Yg reduces to $g(Yg)$, which is a stronger statement. This latter version of Y is known as "Turing's combinator."

3.5 Applications of the fixed-point theorem

The fixed-point theorem essentially lets us define partial computable functions in terms of their indices. Let us consider some applications.

3.5.1 Whimsical applications

For example, we can find an index e such that for every y,

$$\varphi_e(y) = e + y.$$

As another example, one can use the proof of the fixed-point theorem to design a program in Java or C++ that prints itself out.

3.5.2 An application in computability theory

Remember that if for each e, we let W_e be the domain of φ_e, then the sequence $W_0, W_1, W_2, \ldots$ enumerates the computably enumerable sets. Some of these sets are computable. One can ask if there is an algorithm which takes as input a value x, and, if W_x happens to be computable, returns an index for its characteristic function. The answer is "no," there is no such algorithm:

Theorem 3.5.1 *There is no partial computable function f with the following property: whenever W_e is computable, then $f(e)$ is defined and $\varphi_{f(e)}$ is its characteristic function.*

Proof. Let f be any computable function; we will construct an e such that W_e is computable, but $\varphi_{f(e)}$ is not its characteristic function. Using the fixed point theorem, we can find an index e such that

$$\varphi_e(y) \simeq \begin{cases} 0 & \text{if } y = 0 \text{ and } \varphi_{f(e)}(0) \downarrow = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

That is, e is obtained by applying the fixed-point theorem to the function defined by

$$g(x,y) \simeq \begin{cases} 0 & \text{if } y = 0 \text{ and } \varphi_{f(x)}(0) \downarrow = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Informally, we can see that g is partial computable, as follows: on input x and y, the algorithm first checks to see if y is equal to 0. If it is, the algorithm computes $f(x)$, and then uses the universal machine to compute $\varphi_{f(x)}(0)$. If this last computation halts and returns 0, the algorithm returns 0; otherwise, the algorithm doesn't halt.

But now notice that if $\varphi_{f(e)}(0)$ is defined and equal to 0, then $\varphi_e(y)$ is defined exactly when y is equal to 0, so $W_e = \{0\}$. If $\varphi_{f(e)}(0)$ is not defined, or is defined but not equal to 0, then $W_e = \emptyset$. Either way, $\varphi_{f(e)}$ is not the characteristic function of W_e, since it gives the wrong answer on input 0.

3.5.3 Defining functions using self-reference

It is generally useful to be able to define functions in terms of themselves. For example, given computable functions k, l, and m, the fixed-point lemma tells us that there is a partial computable function f satisfying the following equation for every y:

$$f(y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ f(m(y)) & \text{otherwise} \end{cases}$$

Again, more specifically, f is obtained by letting

$$g(x,y) \simeq \begin{cases} k(y) & \text{if } l(y) = 0 \\ \varphi_x(m(y)) & \text{otherwise} \end{cases}$$

and then using the fixed-point lemma to find an index e such that $\varphi_e(y) = g(e,y)$.

For a concrete example, the "greatest common divisor" function $gcd(u,v)$ can be defined by

$$gcd(u,v) \simeq \begin{cases} v & \text{if } 0 = 0 \\ gcd(v \bmod u, u) & \text{otherwise} \end{cases}$$

where $v \bmod u$ denotes the remainder of dividing v by u. An appeal to the fixed-point lemma shows that gcd is partial computable. (In fact, this can be put in the format above, letting y code the pair $\langle u,v \rangle$.) A subsequent induction on u then shows that, in fact, gcd is total.

Of course, one can cook up self-referential definitions that are much fancier than the examples just discussed. Most programming languages support definitions of functions in terms of themselves, one way or another. Note that this is a little bit less dramatic than being able to define a function in terms of an *index* for an algorithm computing the functions, which is what, in full generality, the fixed-point theorem lets you do.

3.5.4 Minimization, with lambda terms.

Now I can finally pay off an IOU. When it comes to the lambda calculus, we've shown the following:

- Every primitive recursive function is represented by a lambda term.
- There is a lambda term Y such that for any lambda term G, $YG \triangleright G(YG)$.

To show that every partial computable function is represented by some lambda term, I only need to show the following.

Lemma 3.5.2 *Suppose $f(x, y)$ is primitive recursive. Let g be defined by*

$$g(x) \simeq \mu y\ f(x, y).$$

Then g is represented by a lambda term.

Proof. The idea is roughly as follows. Given x, we will use the fixed-point lambda term Y to define a function $h_x(n)$ which searches for a y starting at n; then $g(x)$ is just $h_x(0)$. The function h_x can be expressed as the solution of a fixed-point equation:

$$h_x(n) \simeq \begin{cases} n & \text{if } f(x, n) = 0 \\ h_x(n+1) & \text{otherwise.} \end{cases}$$

Here are the details. Since f is primitive recursive, it is represented by some term F. Remember that we also have a lambda term D, such that $D(M, N, \overline{0}) \triangleright M$ and $D(M, N, \overline{1}) \triangleright N$. Fixing x for the moment, to represent h_x we want to find a term H (depending on x) satisfying

$$H(\overline{n}) \equiv D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})).$$

We can do this using the fixed-point term Y. First, let U be the term

$$\lambda h\ \lambda z\ D(z, (h(Sz)), F(x, z)),$$

and then let H be the term YU. Notice that the only free variable in H is x. Let us show that H satisfies the equation above.

By the definition of Y, we have

$$H = YU \equiv U(YU) = U(H).$$

In particular, for each natural number n, we have

$$\begin{array}{rcl} H(\overline{n}) & \equiv & U(H, \overline{n}) \\ & \triangleright & D(\overline{n}, H(S(\overline{n})), F(x, \overline{n})), \end{array}$$

as required. Notice that if you substitute a numeral $\overline{m}$ for x in the last line, the expression reduces to $\overline{n}$ if $F(\overline{m}, \overline{n})$ reduces to $\overline{0}$, and it reduces to $H(S(\overline{n}))$ if $F(\overline{m}, \overline{n})$ reduces to any other numeral.

To finish off the proof, let G be $\lambda x.H(\overline{0})$. Then G represents g; in other words, for every m, $G(\overline{m})$ reduces to reduces to $\overline{g(m)}$, if $g(m)$ is defined, and has no normal form otherwise.

Chapter 4

Incompleteness

4.1 Historical background

We have finally reached the "incompleteness" part of this course. In this section, I will briefly discuss historical developments that will help put the incompleteness theorems in context. In particular, I will give a very sketchy and inadequate overview of the history of mathematical logic; and then say a few words about the history of the foundations of mathematics.

The phrase "mathematical logic" is ambiguous. One can interpret the word "mathematical" as describing the subject matter, as in, "the logic of mathematics," denoting the principles of mathematical reasoning; or as describing the methods, as in "the mathematics of logic," denoting a mathematical study of the principles of reasoning. The account that follows involves mathematical logic in both senses, often at the same time.

The study of logic began, essentially, with Aristotle, who lived approximately 384–322 B.C. His *Categories*, *Prior analytics*, and *Posterior analytics* include systematic studies of the principles of scientific reasoning, including a thorough and systematic study of the syllogism.

Aristotle's logic dominated scholastic philosophy through the middle ages; indeed, as late as eighteenth century Kant maintained that Aristotle's logic was perfect and in no need of revision. But the theory of the syllogism is far too limited to model anything but the most superficial aspects of mathematical reasoning. A century earlier, Leibniz, a contemporary of Newton's, imagined a complete "calculus" for logical reasoning, and made some rudimentary steps towards designing such a calculus, essentially describing a version of propositional logic.

The nineteenth century was a watershed for logic. In 1854 George Boole

wrote *The Laws of Thought*, with a thorough algebraic study of propositional logic that is not far from modern presentations. In 1879 Gottlob Frege published his *Begriffsschrift* (Concept writing) which extends propositional logic with quantifiers and relations, and thus includes first-order logic. In fact, Frege's logical systems included higher-order logic as well, and more — enough more to be (as Russell showed in 1902) inconsistent. But setting aside the inconsistent axiom, Frege more or less invented modern logic singlehandedly, a startling achievement. Quantificational logic was also developed independently by algebraically-minded thinkers after Boole, including Peirce and Schröder.

Let us now turn to developments in the foundations of mathematics. Of course, since logic plays an important role in mathematics, there is a good deal of interaction with the developments I just described. For example, Frege developed his logic with the explicit purpose of showing that all of mathematics could be based solely on his logical framework; in particular, he wished to show that mathematics consists of a priori *analytic* truths instead of, as Kant had maintained, a priori *synthetic* ones.

Many take the birth of mathematics proper to have occurred with the Greeks. Euclid's *Elements*, written around 300 B.C., is already a mature representative of Greek mathematics, with its emphasis on rigor and precision. The definitions and proofs in Euclid's *Elements* survive more or less in tact in high school geometry textbooks today (to the extent that geometry is still taught in high schools). This model of mathematical reasoning has been held to be a paradigm for rigorous argumentation not only in mathematics but in branches of philosophy as well. (Spinoza even presented moral and religious arguments in the Euclidean style, which is strange to see!)

Calculus was invented by Newton and Leibniz in the seventeenth century. (A fierce priority dispute raged for centuries, but most scholars today hold that the two developments were for the most part independent.) Calculus involves reasoning about, for example, infinite sums of infinitely small quantities; these features fueled criticism by Bishop Berkeley, who argued that belief in God was no less rational than the mathematics of his time. The methods of calculus were widely used in the eighteenth century, for example by Leonhard Euler, who used calculations involving infinite sums with dramatic results.

In the nineteenth century, mathematicians tried to address Berkeley's criticisms by putting calculus on a firmer foundation. Efforts by Cauchy, Weierstrass, Bolzano, and others led to our contemporary definitions of limits, continuity, differentiation, and integration in terms of "epsilons and deltas," in other words, devoid of any reference to infinitesimals. Later in

the century, mathematicians tried to push further, and explain all aspects of calculus, including the real numbers themselves, in terms of the natural numbers. (Kronecker: "God created the whole numbers, all else is the work of man.") In 1872, Dedekind wrote "Continuity and the irrational numbers," where he showed how to "construct" the real numbers as sets of rational numbers (which, as you know, can be viewed as pairs of natural numbers); in 1888 he wrote "Was sind und was sollen die Zahlen" (roughly, "What are the natural numbers, and what should they be?") which aimed to explain the natural numbers in purely "logical" terms. In 1887 Kronecker wrote "Über den Zahlbegriff" ("On the concept of number") where he spoke of representing all mathematical object in terms of the integers; in 1889 Giuseppe Peano gave formal, symbolic axioms for the natural numbers.

The end of the nineteenth century also brought a new boldness in dealing with the infinite. Before then, infinitary objects and structures (like the set of natural numbers) were treated gingerly; "infinitely many" was understood as "as many as you want," and "approaches in the limit" was understood as "gets as close as you want." But Georg Cantor showed that it was impossible to take the infinite at face value. Work by Cantor, Dedekind, and others help to introduce the general set-theoretic understanding of mathematics that we discussed earlier in this course.

Which brings us to twentieth century developments in logic and foundations. In 1902 Russell discovered the paradox in Frege's logical system. In 1904 Zermelo proved Cantor's well-ordering principle, using the so-called "axiom of choice"; the legitimacy of this axiom prompted a good deal of debate. Between 1910 and 1913 the three volumes of Russell and Whitehead's *Principia Mathematica* appeared, extending the Fregean program of establishing mathematics on logical grounds. Unfortunately, Russell and Whitehead were forced to adopt two principles that seemed hard to justify as purely logical: an axiom of infinity and an axiom of "reducibility." In the 1900's Poincaré criticized the use of "impredicative definitions" in mathematics, and in the 1910's Brouwer began proposing to refound all of mathematics in an "intuitionistic" basis, which avoided the use of the law of the excluded middle ($p \vee \neg p$).

Strange days indeed! The program of reducing all of mathematics to logic is now referred to as "logicism," and is commonly viewed as having failed, due to the difficulties mentioned above. The program of developing mathematics in terms of intuitionistic mental constructions is called "intuitionism," and is viewed as posing overly severe restrictions on everyday mathematics. Around the turn of the century, David Hilbert, one of the most influential mathematicians of all time, was a strong supporter of the

new, abstract methods introduced by Cantor and Dedekind: "no one will drive us from the paradise that Cantor has created for us." At the same time, he was sensitive to foundational criticisms of these new methods (oddly enough, now called "classical"). He proposed a way of having one's cake and eating it too:

- Represent classical methods with formal axioms and rules.
- Use safe, "finitary" methods to prove that these formal deductive systems are consistent.

In 1931, Gödel proved the two "incompleteness theorems," which showed that this program could not succeed. In this chapter, we will discuss the incompleteness theorems in detail.

4.2 Background in logic

In this section I will discuss some of the prerequisites in logic needed to understand the incompleteness theorems. Most of the material I will discuss here is really a prerequisite for this course, so, hopefully, for many of you this will be mostly a review, and perhaps an indication of the basic facts that you may need to brush up on.

First, let us consider propositional logic. Remember that one starts with propositional variables $p, q, r, \ldots$ and builds formulas with connectives $\wedge, \vee, \neg, \rightarrow, \ldots$. Whether or not a formula comes out true depends on the assignment of truth values to the variables; the method of "truth tables" allows you to calculate the truth values of formulas under such assignments to variables.

Definition 4.2.1 *A propositional formula φ is* valid*, written $\models \varphi$, if φ is true under any truth assignment. More generally, if Γ is a set of formulas and φ is a formula, φ is a* semantic consequence *of Γ, written $\Gamma \models \varphi$, if φ is true under any truth assignment that makes every formula in Γ true.*

These notions of validity and logical consequence are *semantic*, in that the notion of "truth" is central to the definition. There are also *syntactic* notions, which involve describing a formal system of deduction. Assuming we have chosen an appropriate system, we have:

Definition 4.2.2 *A propositional formula φ is* provable*, written $\vdash \varphi$, if there is a formal derivation of φ. More generally, if Γ is a set of formulas*

and φ is a formula, φ is a deductive consequence *of Γ, written $\Gamma \vdash \varphi$, if there is a formal derivation of φ from hypotheses in Γ.*

It is an important fact the the two notions coincide:

Theorem 4.2.3 (soundness and completeness) *For any formula φ and any set of formulas Γ, $\Gamma \vdash \varphi$ if and only if $\Gamma \models \varphi$.*

Of the two directions, the forwards direction (soundness) is easier: it simply says that the rules of the proof system conform to the semantics. The other direction (completeness) is harder: it says that the proof system is strong enough to prove every valid formula. Completeness (at least in the slightly restricted form, "every valid formula is provable) was proved by Bernays and Post independently in the 1910's. Note, incidentally, that the truth-table method gives an explicit algorithm for determining whether or not a formula φ is valid.

The problem with propositional logic is that it is not very expressive. First-order logic is more attractive in this respect, because it models reasoning with relations and quantifiers. The setup is as follows. First, we start with a *language*, with is a specification of that basic symbols that will be used in constructing formulas:

- functions symbols $f_0, f_1, f_2, f_3, \ldots$
- relation symbols $r_0, r_1, r_2, r_3, \ldots$
- constant symbols $c_0, c_1, c_2, c_3, \ldots$

In addition, we always have variables x_0, x_1, x_2, ...; logical symbols $\wedge$, $\vee$, $\rightarrow$, $\neg$, $\forall$, $\exists$, $=$; and parentheses (,).

For a concrete example, we can specify that the "language of arithmetic" has

- a constant symbol, 0
- function symbols $+$, $\times$, $'$
- a relation symbol, $<$

I am following the textbook in using the $'$ symbol to denote the successor operation.

Given a language, L, one then defines the set of *terms*. These are things that "name" objects; for example, in the language of arithmetic

$$0, \quad x', \quad (x' + y) \times z, \quad (0'' + 0') \times 0''$$

are all terms. Strictly speaking, there should be more parentheses, and function symbols should all be written before the arguments (e.g. $+(x, y)$), but we will adopt the usual conventions for readability. I will typically use symbols r, s, t to range over terms, as in "let t be any term." Some terms, like the last one above, have no variables; they are said to be "closed."

Once one has specified the set of terms, one then defines the set of *formulas*. Do not confuse these with terms: terms *name* things, while formulas *say* things. I will use Greek letters like φ, ψ, and θ to range over formulas. Some examples are

$$x < y, \quad \forall x\, \exists z\, (x + y < z), \quad \forall x\, \forall y\, \exists z\, (x + y < z).$$

I am assuming you are comfortable reading these and understanding that they mean, informally. Note that variables can be free or bound in a formula: x and y are free in the first formula above, y is free in the second, and the third formula has no free variables. A formula without free variables is called a *sentence*.

The notation $\varphi[t/x]$ denotes the result of substituting the term t for the variable x in the formula φ. Here is a useful convention: if I introduce a formula as $\varphi(x)$, then $\varphi(t)$ later denotes $\varphi[t/x]$. I will also adopt the usual conventions on dropping parentheses: to supply missing parentheses, do $\neg$, $\forall$, and $\exists$ first; then $\wedge$ and $\vee$; then $\rightarrow$ and $\leftrightarrow$ last. For example

$$\forall x\, R(x) \rightarrow \neg S(z) \wedge T(u)$$

is read

$$((\forall x\, R(x)) \rightarrow ((\neg S(z)) \wedge T(u))).$$

One can extend both the semantic and syntactic notions above to first-order logic. On the semantic side, an interpretation of a language is called a *model*. Intuitively, it should be clear which of the following models satisfies the sentence $\forall x\, \exists y\, R(x, y)$:

- $\langle \mathbb{N}, < \rangle$, that is, the interpretation in which variables range over natural number and R is interpreted as the less-than relation
- $\langle \mathbb{Z}, < \rangle$
- $\langle \mathbb{N}, > \rangle$
- $\langle \mathbb{N}, | \rangle$
- $\langle \mathbb{N} \cup \{\omega\}, S \rangle$ where S is the usual ordering on the elements of $\mathbb{N}$, and $S(n, \omega)$ also holds for every n in $\mathbb{N}$

As above, a sentence φ is said to be valid if it is true in every model, and so on. One can also specify deductive systems for first-order logic. And it turns out that reasonable deductive systems are sound and complete for the semantics:

Theorem 4.2.4 (soundness and completeness) *For every set of sentences Γ and every sentence φ, $\Gamma \vdash \varphi$ if and only if $\Gamma \models \varphi$.*

Here too the hard direction is completeness; this was proved by Gödel in 1929. It may seem consummately confusing that he later proved *incompleteness* in 1931, but it is important to keep in mind that the two theorems say very different things.

- Gödel's completeness theorem says that the usual deductive systems are complete for the semantics "true in all models."
- Gödel's incompleteness theorem says that there is no effective, consistent proof system at all that is complete for the semantics "true of the natural numbers."

The relationship between syntactic and semantic notions of logical consequence are explored thoroughly in *Logic and Computation*, a companion course to this one. It turns out that in this course, we can for the most part set semantic issues aside, and focus on deduction. In particular, all we need to know is that there is an effective proof system that is sound and complete for first-order logic. I will clarify the notion of "effective" below, but, roughly, it means that the rules and axioms have to be presented in such a way that proofs are computationally verifiable.

Here is one such proof system, from the textbook. Take, as axioms, the following:

- Propositional axioms: any instance of a valid propositional formula. For example, if φ and ψ are any formulas, then $\varphi \wedge \psi \rightarrow \varphi$ is an axiom.
- Axioms involving quantifiers:
  - $\forall x\ \varphi(x) \rightarrow \varphi(t)$
  - $\varphi(t) \rightarrow \exists x\ \varphi(x)$
  - $\forall x\ (\varphi \rightarrow \psi(x)) \rightarrow (\varphi \rightarrow \forall x\ \psi(x))$, as long as x is not free in φ.
- Axioms for equality:
  - $\forall x\ (x = x)$

- $\forall x\ (x = y \to y = x)$
- $\forall x\ (x = y \wedge y = z \to z = y)$
- $\forall x_0, \ldots, x_k, y_0, \ldots, y_k\ (x_0 = y_0 \wedge \ldots \wedge x_k = y_k \wedge \varphi(x_0, \ldots, x_k) \to \varphi(y_0, \ldots, y_k))$.

Note that the first clause relies on the fact that the set of propositional validities is decidable. Note also that there are infinitely many axioms above; for example, the first quantifier axiom is really an infinite list of axioms, one for each formula φ. Finally, there are three rules that allow you to derive more theorems:

- Modus ponens: from φ and $\varphi \to \psi$ conclude ψ
- Generalization: from φ conclude $\forall x\ \varphi$
- From $\varphi \to \psi$ conclude $\exists x\ \varphi \to \psi$, if x is not free in ψ.

Incidentally, any sound and complete deductive system will satisfy what is known as the *deduction theorem*: if Γ is any set of sentences and φ and ψ are any sentences, then if $\Gamma \cup \{\varphi\} \vdash \psi$, then $\Gamma \vdash \varphi \to \psi$ (the converse is obvious). This is often useful. Since $\neg\varphi$ is logically equivalent to $\varphi \to \bot$, where $\bot$ is any contradiction, the deduction theorem implies that $\Gamma \cup \{\varphi\}$ is consistent if and only if $\Gamma \nvdash \neg\varphi$, and $\Gamma \cup \{\neg\varphi\}$ is consistent if and only if $\Gamma \nvdash \varphi$.

Where are we going with all this? We would like to bring computability into play; in other words, we would like to ask questions about the computability of various sets and relations having to do with formulas and proofs. So the first step is to choose numerical codings of

- terms,
- formulas, and
- proofs

in such a way that "straightforward" operations and questions are computable. You have already seen enough to know how such a coding should go. For example, one can code terms as follows:

- each variable x_i is coded as $\langle 0, i\rangle$
- each constant c_j is coded as $\langle 1, j\rangle$

- each compound term of the form $f_l(t_0, \ldots, t_k)$ is coded by the number $\langle 2, l, \#(t_0), \ldots, \#(t_k)\rangle$, where $\#(t_0), \ldots, \#(t_k)$ are the codes for $t_0, \ldots, t_k$, respectively.

One can do similar things for formulae, and then a proof is just a sequence of formulae satisfying certain restrictions. It is not difficult to choose coding such that the following, for example, are all computable (and, in fact, primitive recursive):

- the predicate "t is (codes) a term"
- the predicate "φ is (codes) a formula"
- the function of t, x, and φ, which returns the result of substituting t for x in φ
- the predicate "φ is an axiom of first-order logic"
- the predicate "d is a proof of φ" in first-order logic

Informally, all I am saying here is that objects can be defined in a programming language like Java or C++ in such a way that there are subroutines that carry out the computations above or determine whether or not the given property holds.

We can now bring logic and computability together, and inquire as to the computability of various sets and relations that arise in logic. For example:

1. For a given language L, is the set $\{\varphi \mid \vdash \varphi\}$ computable?
2. For a given language L and set of axioms Γ, is $\{\varphi \mid \Gamma \vdash \varphi\}$ computable?
3. Is there a computable set of axioms Γ such that $\{\varphi \mid \Gamma \vdash \varphi\}$ is the set of true sentences in the language of arithmetic? (Here "true" means "true of the natural numbers.")

The answer to 1 depends on the language, L. The set is always computably enumerable; but we will see that for most languages L it is not computable. (For example, it is not computable if L has any relation symbols that take two or more arguments, or if L has two function symbols.) Similarly, the answer to 2 depends on Γ, but we will see that for many interesting cases the answer is, again, "no." The shortest route to getting these answers is to use ideas from computability theory: under suitable conditions, we can reduce the halting problem to the sets above. Finally, we will see that the

answer to 3 is "no." In fact, it can't even be "described" in the language of arithmetic, a result due to Tarski.

The connections between logic and computability run deep. Using the ideas above, we can compute with formulas and proofs; conversely, we will see that we can reason about computation in formal axiomatic systems. It turns out that this back-and-forth relationship is very fruitful.

4.3 Representability in Q

Let us start by focusing on theories of arithmetic. We will describe a very minimal such theory called "Q" (or, sometimes, "Robinson's Q," after Raphael Robinson). We will say what it means for a function to be *representable* in Q, and then we will prove the following:

> A function is representable in Q if and only if it is computable.

For one thing, this provides us with yet another model of computability. But we will also use it to show that the set $\{\varphi \mid Q \vdash \varphi\}$ is not decidable, by reducing the halting problem to it. By the time we are done, we will have proved much stronger things than this; but this initial sketch gives a good sense of where we are headed.

First, let us define Q. The language of Q is the language of arithmetic, as described above; Q consists of the following axioms (to be used in conjunction with the other axioms and rules of first-order logic with equality):

1. $x' = y' \to x = y$
2. $0 \neq x'$
3. $x \neq 0 \to \exists y\ (x = y')$
4. $x + 0 = x$
5. $x + y' = (x + y)'$
6. $x \times 0 = 0$
7. $x \times y' = x \times y + x$
8. $x < y \leftrightarrow \exists z\ (z' + x = y)$

For each natural number n, define the numeral $\overline{n}$ to be the term $0^{\prime\cdots\prime}$ where there are n tick marks in all. (Note that the book does not take $<$ to be

a special symbol in the language, but, rather, takes it to be *defined* by the last axiom above.)

As a theory of arithmetic, Q is *extremely* weak; for example, you can't even prove very simple facts like $\forall x\ (x \neq x')$ or $\forall x, y\ (x + y = y + x)$. A stronger theory called *Peano arithmetic* is obtained by adding a schema of induction:

$$\varphi(0) \wedge (\forall x\ \varphi(x) \rightarrow \varphi(x')) \rightarrow \forall x\ \varphi(x)$$

where $\varphi(x)$ is any formula, possibly with free variables other than x. Using induction, one can do much better; in fact, it takes a good deal of work to find "natural" statements about the natural numbers that can't be proved in Peano arithmetic! But we will see that much of the reason that Q is so interesting is *because* it is so weak, in fact, just barely strong enough for the incompleteness theorem to hold; and also because it has a *finite* set of axioms.

Definition 4.3.1 *A function $f(x_0, \ldots, x_k)$ from the natural numbers to the natural numbers is said to be* representable in Q *if there is a formula $\varphi_f(x_0, \ldots, x_k, y)$ such that whenever $f(n_0, \ldots, n_k) = m$, Q proves*

- $\varphi_f(\overline{n}_0, \ldots, \overline{n}_k, \overline{m})$
- $\forall y\ (\varphi_f(\overline{n}_0, \ldots, \overline{n}_k, y) \rightarrow \overline{m} = y)$.

There are ways of stating the definition; for example, we could equivalently require that Q proves $\forall y\ (\varphi_f(\overline{n}_0, \ldots, \overline{n}_k, y) \leftrightarrow \overline{m} = y)$, where we can take $\theta \leftrightarrow \eta$ to abbreviate $(\theta \rightarrow \eta) \wedge (\eta \rightarrow \theta)$. The main theorem of this section is the following:

Theorem 4.3.2 *A function is representable in Q if and only if it is computable.*

There are two directions to proving the theorem. One of them is fairly straightforward.

Lemma 4.3.3 *Every function that is representable in Q is computable.*

Proof. All we need to know is that we can code terms, formulas, and proofs in such a way that the relation "d is a proof of φ in theory Q" is computable, as well as the function $\mathit{SubNumeral}(\varphi, n, v)$ which returns (a numerical code of) the result of substituting the numeral corresponding to n for the variable (coded by) v in the formula (coded by) φ. Assuming this, suppose

the function f is represented by $\varphi_f(x_0, \ldots, x_k, y)$. Then the algorithm for computing f is as follows: on input $n_0, \ldots, n_k$, search for a number m and a proof of the formula $\varphi_f(\overline{n}_0, \ldots, \overline{n}_k, \overline{m})$; when you find one, output m. In other words,

$$f(n_0, \ldots, n_k) = (\mu s(\text{“}(s)_0 \text{ is a proof of } \varphi(\overline{n}_0, \ldots, \overline{n}_k, \overline{(s)_1}) \text{ in } Q\text{”}))_1.$$

This completes the proof, modulo the (involved but routine) details of coding and defining the function and relation above. □

The other direction is more interesting, and requires more work. We will complete the proof as follows:

- We will define a set of (total) functions, C.
- We will show that C is the set of computable functions, i.e. our definition provides another characterization of computability.
- Then we will show that every function in C can be represented in Q.

Our proof will follow the proof in Chapter 22 of the textbook very closely. (The textbook takes C to include partial functions as well, but then implicitly restricts to the total ones later in the proof.)

Let C be the smallest set of functions containing

- 0,
- successor,
- addition,
- multiplication,
- projections, and
- the characteristic function for equality, $\chi_=$;

and closed under

- composition and
- unbounded search, applied to regular functions.

Remember this last restriction means simply that you can only use the μ operation when the result is total. Compare this to the definition of the *general recursive* functions: here we have added plus, times, and $\chi_=$, but we have dropped primitive recursion. Clearly everything in C is recursive, since plus, times, and $\chi_=$ are. We will show that the converse is also true; this amounts to saying that with the other stuff in C we can carry out primitive recursion.

To do so, we need to develop functions that handle sequences. (If we had exponentiation as well, our task would be easier.) When we had primitive recursion, we could define things like the "nth prime," and pick a fairly straightforward coding. But here we do not have primitive recursion, so we need to be more clever.

Lemma 4.3.4 *There is a function $\beta(d, i)$ in C such that for every sequence $a_0, \ldots, a_n$ there is a number d, such that for every i less than or equal to n, $\beta(d, i) = a_i$.*

Think of d as coding the sequence $\langle a_0, \ldots, a_n \rangle$, and $\beta(d, i)$ returning the ith element. The lemma is fairly minimal; it doesn't say we can concatenate sequences or append elements with functions in C, or even that we can *compute* d from $a_0, \ldots, a_n$ using functions in C. All it says is that there is a "decoding" function such that every sequence is "coded."

The use of the notation β is Gödel's. To repeat, the hard part of proving the lemma is defining a suitable β using the seemingly restricted resources in the definition of C. There are various ways to prove this lemma, but one of the cleanest is still Gödel's original method, which used a number-theoretic fact called the Chinese Remainder theorem. The details of the proof are interesting, but tangential to the main theme of the course; it is more important to understand what Lemma 4.3.4 says. I will, however, outline Gödel's proof for the sake of completeness.

Definition 4.3.5 *Two natural numbers a and b are* relatively prime *if their greatest common divisor is 1; in other words, they have no other divisors in common.*

Definition 4.3.6 *$a \equiv b \mod c$ means $c|(a - b)$, i.e. a and b have the same remainder when divided by c.*

Here is the *Chinese remainder theorem*:

Theorem 4.3.7 *Suppose $x_0, \ldots, x_n$ are (pairwise) relatively prime. Let $y_0, \ldots, y_n$ be any numbers. Then there is a number z such that*

$$\begin{aligned} z &\equiv y_0 \mod x_0 \\ z &\equiv y_1 \mod x_1 \\ &\vdots \\ z &\equiv y_n \mod x_n. \end{aligned}$$

I will not prove this theorem, but you can find the proof in many number theory textbooks. The proof is also outlined as exercise 1 on page 201 of the textbook.

Here is how we will use the Chinese remainder theorem: if $x_0, \ldots, x_n$ are bigger than $y_0, \ldots, y_n$ respectively, then we can take z to code the sequence $\langle y_0, \ldots, y_n \rangle$. To recover y_i, we need only divide z by x_i and take the remainder. To use this coding, we will need to find suitable values for $x_0, \ldots, x_n$.

A couple of observations will help us in this regard. Given $y_0, \ldots, y_n$, let

$$j = \max(n, y_0, \ldots, y_n) + 1,$$

and let

$$\begin{aligned} x_0 &= 1 + j! \\ x_1 &= 1 + 2 \cdot j! \\ x_2 &= 1 + 3 \cdot j! \\ &\vdots \\ x_n &= 1 + (n+1) \cdot j! \end{aligned}$$

Then two things are true:

1. $x_0, \ldots, x_n$ are relatively prime.
2. For each i, $y_i < x_i$.

To see that clause 1 is true, note that if p is a prime number and $p|x_i$ and $p|x_k$, then $p|1 + (i+1)j!$ and $p|1 + (k+1)j!$. But then p divides their difference,

$$(1 + (i+1)j!) - (1 + (k+1)j!) = (i-k)j!.$$

Since p divides $1 + (1+1)j!$, it can't divide $j!$ as well (otherwise, the first division would leave a remainder of 1). So p divides $i - k$. But $|i - k|$ is

at most n, and we have chosen $j > n$, so this implies that $p|j!$, again a contradiction. So there is no prime number dividing both x_i and x_k. Clause 2 is easy: we have $y_i < j < j! < x_i$.

Now let us prove the β function lemma. Remember that C is the smallest set containing 0, successor, plus, times, $\chi_=$, projections, and closed under composition and μ applied to regular functions. As usual, say a relation is in C if its characteristic function is. As before we can show that the relations in C are closed under boolean combinations and bounded quantification; for example:

- $not(x) = \chi_=(x, 0)$
- $\mu x \leq z\ R(x, y) = \mu x\ (R(x, y) \vee x = z)$
- $\exists x \leq z\ \ R(x, y) \equiv R(\mu x \leq z\ R(x, y), y)$

We can then show that all of the following are in C:

- The pairing function, $J(x, y) = \frac{1}{2}[(x + y)(x + y + 1)] + x$
- Projections
$$K(z) = \mu x \leq q\ (\exists y \leq z\ (z = J(x, y)))$$
and
$$L(z) = \mu y \leq q\ (\exists x \leq z\ (z = J(x, y))).$$
- $x < y$
- $x|y$
- The function $rem(x, y)$ which returns the remainder when y is divided by x

Now define
$$\beta^*(d_0, d_1, i) = rem(1 + (i + 1)d_1, d_0)$$
and
$$\beta(d, i) = \beta^*(K(d), L(d), i).$$
This is the function we need. Given $a_0, \ldots, a_n$, as above, let
$$j = \max(n, a_0, \ldots, a_n) + 1,$$
and let $d_1 = j!$. By the observations above, we know that $1 + d_1, 1 + 2d_1, \ldots, 1 + (n + 1)d_1$ are relatively prime and all are bigger than $a_0, \ldots, a_n$. By the Chinese remainder theorem there is a value d_0 such that for each i,
$$d_0 \equiv a_i \mod (1 + (i + 1)d_1)$$

and so (because d_1 is greater than a_i),

$$a_i = rem(1 + (i+1)d_1, d_0).$$

Let $d = J((d)_0, (d)_1)$. Then for each i from 0 to n, we have

$$\begin{aligned} \beta(d,i) &= \beta^*(d_0, d_1, i) \\ &= rem(1 + (i+1)d_1, d_0) \\ &= a_i \end{aligned}$$

which is what we need. This completes the proof of the β-function lemma.

Now we can show that C is closed under primitive recursion. Suppose $f(\vec{z})$ and $g(u, v, \vec{z})$ are both in C. Let $h(x, \vec{z})$ be the function defined by

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

We need to show that h is in C.

First, define an auxiliary function $\hat{h}(x, \vec{z})$ which returns the least number d such that d codes a sequence satisfying

- $(d)_0 = f(\vec{z})$, and
- for each $i < x$, $(d)_{i+1} = g(i, (d)_i, \vec{z})$,

where now $(d)_i$ is short for $\beta(d, i)$. In other words, $\hat{h}$ returns a sequence that begins $\langle h(0, \vec{z}), h(1, \vec{z}), \ldots, h(x, \vec{z})\rangle$. $\hat{h}$ is in C, because we can write it as

$$\hat{h}(x, z) = \mu d(\beta(d, 0) = f(\vec{z}) \wedge \forall i < x\ (\beta(d, i+1) = g(i, \beta(d, i), \vec{z}))).$$

But then we have

$$h(x, \vec{z}) = \beta(\hat{h}(x, \vec{z}), x),$$

so h is in C as well.

We have shown that every computable function is in C. So all we have left to do is show that every function in C is representable in Q. In the end, we need to show how to assign to each k-ary function $f(x_0, \ldots, x_{k-1})$ in C a formula $\varphi_f(x_0, \ldots, x_{k-1}, y)$ that represents it. This is done in Chapter 22B of Epstein and Carnielli's textbook, and the proof that the assignment works involves 16 lemmas. I will run through this list, commenting on some of the proofs, but skipping many of the details.

To get off to a good start, however, let us go over the first lemma, Lemma 3 in the book, carefully.

Lemma 4.3.8 *Given natural numbers n and m, if $n \neq m$, then $Q \vdash \overline{n} \neq \overline{m}$.*

Proof. Use induction on n to show that for every m, if $n \neq m$, then $Q \vdash \overline{n} \neq \overline{m}$.

In the base case, $n = 0$. If m is not equal to 0, then $m = k + 1$ for some natural number k. We have an axiom that says $\forall x\ (0 \neq x')$. By a quantifier axiom, replacing x by $\overline{k}$, we can conclude $0 \neq \overline{k}'$. But $\overline{k}'$ is just $\overline{m}$.

In the induction step, we can assume the claim is true for n, and consider $n + 1$. Let m be any natural number. There are two possibilities: either $m = 0$ or for some k we have $m = k + 1$. The first case is handled as above. In the second case, suppose $n + 1 \neq k + 1$. Then $n \neq k$. By the induction hypothesis for n we have $Q \vdash \overline{n} \neq \overline{k}$. We have an axiom that says $\forall x, y\ (x' = y' \rightarrow x = y)$. Using a quantifier axiom, we have $\overline{n}' = \overline{k}' \rightarrow \overline{n} = \overline{k}$. Using propositional logic, we can conclude, in Q, $\overline{n} \neq \overline{k} \rightarrow \overline{n}' \neq \overline{k}'$. Using modus ponens, we can conclude $\overline{n}' \neq \overline{k}'$, which is what we want, since $\overline{k}'$ is $\overline{m}$. □

Note that the lemma does not say much: in essence it says that Q can prove that different numerals denote different objects. For example, Q proves $0'' \neq 0'''$. But showing that this holds in general requires some care. Note also that although we are using induction, it is induction *outside* of Q.

I will continue on through Lemma 12. At that point, we will be able to represent zero, successor, plus, times, and the characteristic function for equality, and projections. In each case, the appropriate representing function is entirely straightforward; for example, zero is represented by the formula

$$y = 0,$$

successor is represented by the formula

$$x_0' = y,$$

and plus is represented by the formula

$$x_0 + x_1 = y.$$

The work involves showing that Q can prove the relevant statements; for example, saying that plus is represented by the formula above involves showing that for every pair of natural numbers m and n, Q proves

$$\overline{n} + \overline{m} = \overline{n + m}$$

and

$$\forall y\ (\overline{n} + \overline{m} = y \to y = \overline{n+m}).$$

What about composition? Suppose h is defined by

$$h(x_0, \ldots, x_{l-1}) = f(g_0(x_0, \ldots, x_{l-1}), \ldots, g_{k-1}(x_0, \ldots, x_{l-1})).$$

where we have already found formulas $\varphi_f, \varphi_{g_0}, \ldots, \varphi_{g_{k-1}}$ representing the functions $f, g_0, \ldots, g_{k-1}$, respectively. Then we can define a formula φ_h representing h, by defining $\varphi_h(x_0, \ldots, x_{l-1}, y)$ to be

$$\exists z_0, \ldots, z_{k-1}\ (\varphi_{g_0}(x_0, \ldots, x_{l-1}, z_0) \wedge \ldots \wedge \varphi_{g_{k-1}}(x_0, \ldots, x_{l-1}, z_{k-1}) \wedge \varphi_f(z_0, \ldots, z_{k-1}, y)).$$

Lemma 12 shows that this works, for a simplified case.

Finally, let us consider unbounded search. Suppose $g(x, \vec{z})$ is regular and representable in Q, say by the formula $\varphi_g(x, \vec{z}, y)$. Let f be defined by $f(\vec{z}) = \mu x\ g(x, \vec{z})$. We would like to find a formula $\varphi_f(\vec{z}, y)$ representing f. Here is a natural choice:

$$\varphi_f(\vec{z}, y) \equiv \varphi_g(y, \vec{z}, 0) \wedge \forall w\ (w < z \to \neg\varphi_g(w, \vec{z}, 0)).$$

Lemma 18 in the textbook says that this works; it uses Lemmas 13–17. I will go over the statements of these lemmas. For example, here is Lemma 13:

Lemma 4.3.9 *For every variable x and every natural number n, Q proves $x' + \overline{n} = (x + \overline{n})'$.*

It is again worth mentioning that this is weaker than saying that Q proves $\forall x, y\ (x' + y = (x + y)')$ (which is false).

Proof. The proof is, as usual, by induction on n. In the base case, $n = 0$, we need to show that Q proves $x' + 0 = (x + 0)'$. But we have:

$x' + 0 = x'$ from axiom 4
$x + 0 = x$ from axiom 4
$(x + 0)' = x'$ an equalty axiom on line 2
$x' + 0 = (x + 0)'$ equality axioms and lines 1 and 3

In the induction step, we can assume that we have derived $x' + \overline{n} = (x + \overline{n})'$ in Q. Since $\overline{n+1}$ is $\overline{n}'$, we need to show that Q proves $x' + \overline{n}' = (x + \overline{n}')'$. The following chain of equalities can be derived in Q:

$$\begin{aligned} x' + \overline{n}' &= (x' + \overline{n})' && \text{axiom 5} \\ &= (x + \overline{n}')' && \text{from the inductive hypothesis} \end{aligned}$$

For a final example, here is Lemma 15:

Lemma 4.3.10 *1. Q proves $\neg(x < \overline{0})$.*

2. For every natural number n, Q proves

$$x < \overline{n+1} \rightarrow x = \overline{0} \vee \ldots x = \overline{n}.$$

Proof. Let us do 1 and part of 2, informally (i.e. only giving hints as to how to construct the formal derivation).

For part 1, by the definition of $<$, we need to prove $\neg\exists y\ (y' + x = 0)$ in Q, which is equivalent (using the axioms and rules of first-order logic) to $\forall y\ (y' + x \neq 0)$. Here is the idea: suppose $y' + x = 0$. If x is 0, we have $y' + 0 = 0$. But by axiom 4 of Q, we have $y' + 0 = y'$, and by axiom 2 we have $y' \neq 0$, a contradiction. So $\forall y\ (y' + x \neq 0)$. If x is not 0, by axiom 3 there is a z such that $x = z'$. But then we have $y' + z' = 0$. By axiom 5, we have $(y' + z)' = 0$, again contradicting axiom 2.

For part 2, use induction on n. Let us consider the base case, when $n = 0$. In that case, we need to show $x < \overline{1} \rightarrow x = \overline{0}$. Suppose $x < \overline{1}$. Then by the defining axiom for $<$, we have $\exists y\ ((y' + x) = 0')$. Suppose y has that property; so we have $y' + x = 0'$.

We need to show $x = 0$. By axiom 3, if x is not 0, it is equal to z' for some z. Then we have $y' + z' = 0'$. By axiom 5 of Q, we have $(y' + z)' = 0'$. By axiom 1, we have $y' + z = 0$. But this means, by definition, $z < 0$, contradicting part 1.

For the induction step, and more details, see the textbook. □

We have shown that the set of computable functions can be characterized as the set of functions representable in Q. In fact, the proof is more general. From the definition of representability, it is not hard to see that any theory extending Q (or in which one can interpret Q) can represent the computable functions; but, conversely, in any proof system in which the notion of proof is computable, every representable function is computable. So, for example, the set of computable functions can be characterized as the set of functions represented in Peano arithmetic, or even Zermelo Fraenkel set theory. As Gödel noted, this is somewhat surprising. We will see that when it comes to provability, questions are very sensitive to which theory you consider; roughly, the stronger the axioms, the more you can prove. But across a wide range of axiomatic theories, the representable functions are exactly the computable ones.

Let us say what it means for a *relation* to be representable.

Definition 4.3.11 *A relation $R(x_0, \ldots, x_k)$ on the natural numbers is* representable in Q *if there is a formula $\varphi_R(x_0, \ldots, x_k)$ such that whenever $R(n_0, \ldots, n_k)$ is true, Q proves $\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$, and whenever $R(n_0, \ldots, n_k)$ is false, Q proves $\neg\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$.*

Theorem 4.3.12 *A relation is representable in Q if and only if it is computable.*

Proof. For the forwards direction, suppose $R(x_0, \ldots, x_k)$ is represented by the formula $\varphi_R(x_0, \ldots, x_k)$. Here is an algorithm for computing R: on input $n_0, \ldots, n_k$, simultaneously search for a proof of $\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$ and a proof of $\neg\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$. By our hypothesis, the search is bound to find one of the other; if it is the first, report "yes," and otherwise, report "no."

In the other direction, suppose $R(x_0, \ldots, x_k)$ is computable. By definition, this means that the function $\chi_R(x_0, \ldots, x_k)$ is computable. By Theorem 4.3.2, χ_R is represented by a formula, say $\varphi_{\chi_R}(x_0, \ldots, x_k, y)$. Let $\varphi_R(x_0, \ldots, x_k)$ be the formula $\varphi_{\chi_R}(x_0, \ldots, x_k, \overline{1})$. Then for any $n_0, \ldots, n_k$, if $R(n_0, \ldots, n_k)$ is true, then $\chi_R(n_0, \ldots, n_k) = 1$, in which case Q proves $\varphi_{\chi_R}(\overline{n}_0, \ldots, \overline{n}_k, \overline{1})$, and so Q proves $\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$. On the other hand if $R(n_0, \ldots, n_k)$ is false, then $\chi_R(n_0, \ldots, n_k) = 0$. This means that Q proves $\varphi_{\chi_R}(\overline{n}_0, \ldots, \overline{n}_k, y) \to y = \overline{0}$. Since Q proves $\neg(\overline{0} = \overline{1})$, Q proves $\neg\varphi_{\chi_R}(\overline{n}_0, \ldots, \overline{n}_k, \overline{1})$, and so it proves $\neg\varphi_R(\overline{n}_0, \ldots, \overline{n}_k)$. □

4.4 The first incompleteness theorem

To recap, we have the following:

- a definition of what it means for a function to be representable in Q (Definition 4.3.1)
- a definition of what it means for a relation to be representable in Q (Definition 4.3.11)
- a theorem asserting that the representable functions of Q are exactly the computable ones (Theorem 4.3.2)
- a theorem asserting that the representable relations of Q are exactly the computable ones (Theorem 4.3.12)

With these two definitions and theorems in hand, we have opened the floodgates between logic and computability, and now we can use the work we

have done in computability theory to read off conclusions in logic. Rather than state the strongest results first, I will build up to them in stages.

A *theory* is a set of sentences that is deductively closed, that is, with the property that whenever T proves φ then φ is in T. It is probably best to think of a theory as being a collection of sentences, together with all the things that these sentences imply. From now on, I will use Q to refer to the *theory* consisting of the set of sentences derivable from the eight axioms in Section 4.3. Remember that we can code formula of Q as numbers; if φ is such a formula, let $\#(\varphi)$ denote the number coding φ. Modulo this coding, we can now ask whether various sets of formulas are computable or not.

Theorem 4.4.1 *Q is c.e. but not decidable. In fact, it is a complete c.e. set.*

Proof. It is not hard to see that Q is c.e., since it is the set of (codes for) sentences y such that there is a proof x of y in Q:

$$Q = \{y \mid \exists x \; Pr_Q(x, y)\}.$$

But we know that $Pr_Q(x, y)$ is computable (in fact, primitive recursive), and any set that can be written in the above form is c.e.

Saying that it is a complete c.e. set is equivalent to saying that $K \leq_m Q$, where $K = \{x \mid \varphi_x(x) \downarrow\}$. So let us show that K is reducible to Q. Since Kleene's predicate $T(e, x, s)$ is primitive recursive, it is representable in Q, say by φ_T. Then for every x, we have

$$\begin{aligned} x \in K &\rightarrow \exists s \; T(x, x, s) \\ &\rightarrow \exists s (Q \text{ proves } \varphi_T(\overline{x}, \overline{x}, \overline{s})) \\ &\rightarrow Q \text{ proves } \exists s \; \varphi_T(\overline{x}, \overline{x}, s). \end{aligned}$$

Conversely, if Q proves $\exists s \; \varphi_T(\overline{x}, \overline{x}, s)$, then, in fact, for some natural number n the formula $\varphi_T(\overline{x}, \overline{x}, \overline{n})$ must be true. Now, if $T(x, x, n)$ were false, Q would prove $\neg\varphi_T(\overline{x}, \overline{x}, \overline{n})$, since φ_T represents T. But then Q proves a false formula, which is a contradiction. So $T(x, x, n)$ must be true, which implies $\varphi_x(x) \downarrow$.

In short, we have that for every x, x is in K if and only if Q proves $\exists s \; T(\overline{x}, \overline{x}, s)$. So the function f which takes x to (a code for) the sentence $\exists s \; T(\overline{x}, \overline{x}, s)$ is a reduction of K to Q. □

The proof above relied on the fact that any sentence provable in Q is "true" of the natural numbers. The next definition and theorem strengthen

this theorem, by pinpointing just those aspects of "truth" that were needed in the proof above. Don't dwell on this theorem too long, though, because we will soon strengthen it even further. I am including it mainly for historical purposes: Gödel's original paper used the notion of ω-consistency, but his result was strengthened by replacing ω-consistency with ordinary consistency soon after.

Definition 4.4.2 *A theory T is ω-consistent if the following holds: if $\exists x\ \varphi(x)$ is any sentence and T proves $\neg\varphi(\overline{0}), \neg\varphi(\overline{1}), \neg\varphi(\overline{2}), \ldots$ then T does not prove $\exists x\ \varphi(x)$.*

Theorem 4.4.3 *Let T be any ω-consistent theory that includes Q. Then T is not decidable.*

Proof. If T includes Q, then T represents the computable functions and relations. We need only modify the previous proof. As above, if $x \in K$, then T proves $\exists s\ \varphi_T(\overline{x}, \overline{x}, s)$. Conversely, suppose T proves $\exists s\ \varphi_T(\overline{x}, \overline{x}, s)$. Then x must be in K: otherwise, there is no halting computation of machine x on input x; since φ_T represents Kleene's T relation, T proves $\neg\varphi_T(\overline{x}, \overline{x}, \overline{0})$, $\neg\varphi_T(\overline{x}, \overline{x}, \overline{1})$, ..., making T ω-inconsistent. □

We can do better. Remember that a theory is *consistent* if it does not prove φ and $\neg\varphi$ for any formula φ. Since anything follows from a contradiction, an inconsistent theory is trivial: every sentence is provable. Clearly, if a theory if ω-consistent, then it is consistent. But being consistent is a weaker requirement (i.e. there are theories that are consistent but not ω-consistent — we will see an example soon). So this theorem is stronger than the last:

Theorem 4.4.4 *Let T be any consistent theory that includes Q. Then T is not decidable.*

To prove this, first we need a lemma:

Lemma 4.4.5 *There is no "universal computable relation." That is, there is no binary computable relation $R(x, y)$, with the following property: whenever $S(y)$ is a unary computable relation, there is some k such that for every y, $S(y)$ is true if and only if $R(k, y)$ is true.*

Proof. Suppose $R(x, y)$ is a universal computable relation. Let $S(y)$ be the relation $\neg R(y, y)$. Since $S(y)$ is computable, for some k, $S(y)$ is equivalent

to $R(k, y)$. But then we have that $S(k)$ is equivalent to both $R(k, k)$ and $\neg R(k, k)$, which is a contradiction. □

Proof (of the theorem). Suppose T is a consistent, decidable extension of Q. We will obtain a contradiction by using T to define a universal computable relation.

Let $R(x, y)$ hold if and only if

x codes a formula $\theta(u)$, and T proves $\theta(\overline{y})$.

Since we are assuming that T is decidable, R is computable. Let us show that R is universal. If $S(y)$ is any computable relation, then it is representable in Q (and hence T) by a formula $\theta_S(u)$. Then for every n, we have

$$\begin{aligned} S(\overline{n}) &\rightarrow T \vdash \theta_S(\overline{n}) \\ &\rightarrow R(\#(\theta_S(u)), n) \end{aligned}$$

and

$$\begin{aligned} \neg S(\overline{n}) &\rightarrow T \vdash \neg\theta_S(\overline{n}) \\ &\rightarrow T \nvdash \theta_S(\overline{n}) \quad \text{(since } T \text{ is consistent)} \\ &\rightarrow \neg R(\#(\theta_S(u)), n). \end{aligned}$$

That is, for every y, $S(y)$ is true if and only if $R(\#(\theta_S(u)), y)$ is. So R is universal, and we have the contradiction we were looking for. □

Let "true arithmetic" be the theory $\{\varphi \mid \langle \mathbb{N}, 0, ', +, \times, < \rangle \models \varphi\}$, that is, the set of sentences in the language of arithmetic that are true in the standard interpretation.

Corollary 4.4.6 *True arithmetic is not decidable.*

In Section 4.9 we will state a stronger result, due to Tarski.

Remember that a theory is said to be *complete* if for every sentence φ, either φ or $\neg\varphi$ is provable. Hilbert initially wanted a complete, consistent set of axioms for arithmetic. We can now say something in that regard.

A theory T is said to be *computably axiomatizable* if it has a computable set of axioms A. (Saying that A is a set of axioms for T means $T = \{\varphi \mid A \vdash \varphi\}$.) Any "reasonable" axiomatization of the natural numbers will have this property. In particular, any theory with a finite set of axioms is computably axiomatizable. The phrase "effectively axiomatizable" is also commonly used.

Lemma 4.4.7 *Suppose T is computably axiomatizable. Then T is computably enumerable.*

Proof. Suppose A is a computable set of axioms for T. To determine if $\varphi \in T$, just search for a proof of φ from the axioms.

Put slightly differently, φ is in T if and only if there is a finite list of axioms $\psi_1, \ldots, \psi_k$ in A and a proof of $\psi_1 \wedge \ldots \wedge \psi_k \rightarrow \varphi$ in first-order logic. But we already know that any set with a definition of the form "there exists ... such that ..." is c.e., provided the second "..." is computable. □

Lemma 4.4.8 *Suppose a theory T is complete and computably axiomatizable. Then T is computable.*

Proof. Suppose T is complete and A is a computable set of axioms. If T is inconsistent, it is clearly computable. (Algorithm: "just say yes.") So we can assume that T is also consistent.

To decide whether or not a sentence φ is in T, simultaneously search for a proof of φ from A and a proof of $\neg\varphi$. Since T is complete, you are bound to find one or another; and since T is consistent, if you find a proof of $\neg\varphi$, there is no proof of φ.

Put in different terms, we already know that T is c.e.; so by a theorem we proved before, it suffices to show that the complement of T is c.e. But a formula φ is in $\overline{T}$ if and only if $\neg\varphi$ is in T; so $\overline{T} \leq_m T$. □

Theorem 4.4.9 *There is no complete, consistent, computably axiomatized extension of Q.*

Proof. We already know that there is no consistent, decidable extension of Q. But if T is complete and computably axiomatized, then it is decidable. □

We will take this to be our official statement of Gödel's first incompleteness theorem. It really is not that far from Gödel's original 1931 formulation. Aside from the more modern terminology, the key differences are this: Gödel has "ω-consistent" instead of "consistent"; and he could not say "computably axiomatized" in full generality, since the formal notion of computability was not in place yet. (The formal models of computability were developed over the next few years, in large part by Gödel, and in large

part to be able to characterize the kinds of theories that are susceptible to the Gödel phenomenon.)

The theorem says you can't have it all, namely, completeness, consistency, and computable axiomatizability. If you give up any one of these, though, you can have the other two: Q is consistent and computably axiomatized, but not complete; the inconsistent theory is complete, and computably axiomatized (say, by $\{0 \neq 0\}$), but not consistent; and the set of true sentence of arithmetic is complete and consistent, but it is not computably axiomatized.

But wait! We can still do more. Let Q' be the set of sentences whose *negations* are provable in Q, i.e. $Q' = \{\varphi \mid Q \vdash \neg\varphi\}$. Remember that disjoint sets A and B are said to be computably inseparable if there is no computable set C such that $A \subseteq C$ and $B \subseteq \overline{C}$.

Lemma 4.4.10 *Q and Q' are computably inseparable.*

Proof. Suppose C is a computable set such that $Q \subseteq C$ and $Q' \subseteq \overline{C}$. Let $R(x, y)$ be the relation

x codes a formula $\theta(u)$ and $\theta(\overline{y})$ is in C.

I will show that $R(x, y)$ is a universal computable relation, yielding a contradiction.

Suppose $S(y)$ is computable, represented by $\theta_S(u)$ in Q. Then

$$\begin{aligned} S(\overline{n}) &\rightarrow Q \vdash \theta_S(\overline{n}) \\ &\rightarrow \theta_S(\overline{n}) \in C \end{aligned}$$

and

$$\begin{aligned} \neg S(\overline{n}) &\rightarrow Q \vdash \neg\theta_S(\overline{n}) \\ &\rightarrow \theta_S(\overline{n}) \in Q' \\ &\rightarrow \theta_S(\overline{n}) \notin C \end{aligned}$$

So $S(y)$ is equivalent to $R(\#(\theta_S(\overline{u})), y)$. □

The following theorem says that not only is Q undecidable, but, in fact, any theory that does not disagree with Q is undecidable.

Theorem 4.4.11 *Let T be any theory in the language of arithmetic that is consistent with Q (i.e. $T \cup Q$ is consistent). Then T is undecidable.*

Proof. Remember that Q has a finite set of axioms, $\varphi_1, \ldots, \varphi_8$. We can even replace these by a single axiom, $\eta = \varphi_1 \wedge \ldots \wedge \varphi_8$.

Suppose T is a decidable theory consistent with Q. Let

$$C = \{\varphi \mid T \vdash \eta \to \varphi\}.$$

I claim C is a computable separation of Q and Q', a contradiction. First, if φ is in Q, then φ is provable from the axioms of Q; by the deduction theorem, there is a proof of $\eta \to \varphi$ in first-order logic. So φ is in C.

On the other hand, if φ is in Q', then there is a proof of $\eta \to \neg\varphi$ in first-order logic. If T also proves $\eta \to \varphi$, then T proves $\neg\eta$, in which case $T \cup Q$ is inconsistent. But we are assuming $T \cup Q$ is consistent, so T does not prove $\eta \to \varphi$, and so φ is not in C.

We've shown that if φ is in Q, then it is in C, and if φ is in Q', then it is in $\overline{C}$. So C is a computable separation, which is the contradiction we were looking for. □

This theorem is very powerful. For example, it implies:

Corollary 4.4.12 *First-order logic for the language of arithmetic (that is, the set $\{\varphi \mid \varphi$ is provable in first-order logic$\}$) is undecidable.*

Proof. First-order logic is the set of consequences of $\emptyset$, which is consistent with Q. □

We can strengthen these results even more. Informally, an interpretation of a language L_1 in another language L_2 involves defining the universe, relation symbols, and function symbols of L_1 with formulae in L_2. Though we won't take the time to do this, one can make this definition precise.

Theorem 4.4.13 *Suppose T is a theory in a language in which one can interpret the language of arithmetic, in such a way that T is consistent with the interpretation of Q. Then T is undecidable. If T proves the interpretation of the axioms of Q, then no consistent extension of T is decidable.*

The proof is just a small modification of the proof of the last theorem; one could use a counterexample to get a separation of Q and Q'. One can take *ZFC*, Zermelo Fraenkel set theory with the axiom of choice, to be an axiomatic foundation that is powerful enough to carry out a good deal of ordinary mathematics. In *ZFC* one can define the natural numbers, and via this interpretation, the axioms of Q are true. So we have

Corollary 4.4.14 *There is no decidable extension of ZFC.*

Corollary 4.4.15 *There is no complete, consistent, computably axiomatized extension of ZFC.*

The language of *ZFC* has only a single binary relation, $\in$. (In fact, you don't even need equality.) So we have

Corollary 4.4.16 *First-order logic for any language with a binary relation symbol is undecidable.*

This result extends to any language with two unary function symbols, since one can use these to simulate a binary relation symbol. The results just cited are tight: it turns out that first-order logic for a language with only *unary* relation symbols and at most one *unary* function symbol is decidable.

One more bit of trivia. We know that the set of sentences in the language $0, S, +, \times, <$ true in the standard model. In fact, one can define $<$ in terms of the other symbols, and then one can define $+$ in terms of $\times$ and S. So the set of true sentences in the language $0, S, \times$ is undecidable. On the other hand, Presberger has shown that the set of sentences in the language $0, S, +$ true in the language of arithmetic is decidable. The procedure is computationally infeasible, however.

4.5 The fixed-point lemma

The approach we have taken to proving the first incompleteness theorem differs from Gödel's. The more ways you know how to prove a theorem, the better you understand it; and Gödel's proof is no less interesting than the one we have already seen. So it is worthwhile to consider his methods.

The idea behind Gödel's proof can be found in the Epimenides paradox. Epimenides, a Cretin, asserted that all Cretins are liars; a more direct form of the paradox is the assertion "this sentence is false." Essentially, by replacing truth with provability, Gödel was able to formalize a sentence which, in essence, asserts "this sentence is not provable." Assuming ω-consistency, Gödel was able to show that this sentence is neither provable nor refutable from the system of axioms he was considering.

The first challenge is to understand how one can construct a sentence that refers to itself. For every formula φ in the language of Q, let $\ulcorner\varphi\urcorner$ denote the numeral corresponding to $\#(\varphi)$. Think about what this means: φ is a formula in the language of Q, $\#(\varphi)$ is a natural number, and $\ulcorner\varphi\urcorner$ is a *term* in the language of Q. So every formula φ in the language of Q has a *name*,

$\ulcorner\varphi\urcorner$, which is a term in the language of Q; this provides us with a conceptual framework in which formulas in the language of Q can "say" things about other formulas. The following lemma is known as Gödel's fixed-point lemma.

Lemma 4.5.1 *Let T be any theory extending Q, and let $\psi(x)$ be any formula with free variable x. Then there is a sentence φ such that T proves $\varphi \leftrightarrow \psi(\ulcorner\varphi\urcorner)$.*

The lemma asserts that given any property $\psi(x)$, there is a sentence φ that asserts "$\psi(x)$ is true of me."

How can we construct such a sentence? Consider the following version of the Epimenides paradox, due to Quine:

> "Yields falsehood when preceded by its quotation" yields falsehood when preceded by its quotation.

This sentence is not directly self-referential. It simply makes an assertion about the syntactic objects between quotes, and, in doing so, it is on par with sentences like

- "Robert" is a nice name.
- "I ran." is a short sentence.
- "Has three words" has three words.

But what happens when one takes the phrase "yields falsehood when preceded by its quotation," and precedes it with a quoted version of itself? Then one has the original sentence! In short, the sentence asserts that it is false.

To get the more general assertion in the fixed-point lemma, take the "self-substitution" of a syntactic phrase with a special symbol X to be the result of substituting of quoted version of the whole phrase for X. For example, the self-substitution of "X is a nice sentence" is " 'X is a nice sentence' is a nice sentence." The sentence we want is roughly

> "X when self-substituted has property ψ" when self-substituted has property ψ.

If you take the quoted part of the sentence, and "self-substitute" it, you get the original sentence. So we have found a clever way of saying "This sentence has property ψ."

We can formalize this. Let $diag(y)$ be the computable (in fact, primitive recursive) function that does the following: if y is the code of a formula

$\psi(x)$, $diag(y)$ returns a code of $\psi(\ulcorner\psi(x)\urcorner)$. If diag were a function symbol in T representing the function $diag$, we could take φ to be the formula $\psi(\text{diag}(\ulcorner\psi(\text{diag}(x))\urcorner))$. Notice that

$$\begin{aligned} diag(\#(\psi(\text{diag}(x)))) &= \#(\psi(\text{diag}(\ulcorner\psi(\text{diag}(x))\urcorner)) \\ &= \#(\varphi). \end{aligned}$$

Assuming T can prove

$$\text{diag}(\ulcorner\psi(\text{diag}(x))\urcorner) = \ulcorner\varphi\urcorner,$$

it can prove $\psi(\text{diag}(\ulcorner\psi(\text{diag}(x))\urcorner)) \leftrightarrow \psi(\ulcorner\varphi\urcorner)$. But the left hand side is, by definition, φ.

In general, diag will not be a function symbol of T. But since T extends Q, the function $diag$ will be *represented* in T by some formula $\theta_{diag}(x, y)$. So instead of writing $\psi(\text{diag}(x))$ we will have to write $\exists y\ (\theta_{diag}(x, y) \wedge \psi(y))$. Otherwise, the proof sketched above goes through.

Proof of the fixed point lemma. Given $\psi(x)$, let $\eta(x)$ be the formula $\exists y\ (\theta_{diag}(x, y) \wedge \psi(y))$ and let φ be the formula $\eta(\ulcorner\eta(x)\urcorner)$.

Since θ_{diag} represents $diag$, T can prove

$$\forall y\ (\theta_{diag}(\ulcorner\eta(x)\urcorner, y) \leftrightarrow y = \overline{diag(\ulcorner\eta(x)\urcorner)}).$$

But by definition, $diag(\#(\eta(x))) = \#(\eta(\ulcorner\eta(x)\urcorner)) = \#(\varphi)$, so T can prove

$$\forall y\ (\theta_{diag}(\ulcorner\eta(x)\urcorner, y) \leftrightarrow y = \ulcorner\varphi\urcorner).$$

Going back to the definition of $\eta(x)$, we see $\eta(\ulcorner\eta(x)\urcorner)$ is just the formula

$$\exists y\ (\theta_{diag}(\ulcorner\eta(x)\urcorner, y) \wedge \psi(y)).$$

Using the last two sentences and ordinary first-order logic, one can then prove

$$\eta(\ulcorner\eta(x)\urcorner) \leftrightarrow \psi(\ulcorner\varphi\urcorner).$$

But the left-hand side is just φ. □

You should compare this to the proof of the fixed-point lemma in computability theory. The difference is that here we want to define a *statement* in terms of itself, whereas there we wanted to define a *function* in terms of itself; this difference aside, it is really the same idea.

4.6 The first incompleteness theorem, revisited

We can now describe Gödel's original proof of the first incompleteness theorem. Let T be any computably axiomatized theory in a language extending the language of arithmetic, such that T includes the axioms of Q. This means that, in particular, T represents computable functions and relations.

We have argued that, given a reasonable coding of formulas and proofs as numbers, the relation $Pr_T(x, y)$ is computable, where $Pr_T(x, y)$ holds if and only if x is a proof of formula y in T. In fact, for the particular theory that Gödel had in mind, Gödel was able to show that this relation is primitive recursive, using the list of 45 functions and relations in his paper. The 45th relation, xBy, is just $Pr_T(x, y)$ for his particular choice of T. Remember that where Gödel uses the word "recursive" in his paper, we would now use the phrase "primitive recursive."

Since $Pr_T(x, y)$ is computable, it is representable in T. I will use $\mathrm{Pr}_T(x, y)$ to refer to the formula that represents it. Let $\mathrm{Prov}_T(y)$ be the formula $\exists x\ \mathrm{Pr}_T(x, y)$. This describes the 46th relation, $Bew(y)$, on Gödel's list. As Gödel notes, this is the only relation that "cannot be asserted to be recursive." What he probably meant is this: from the definition, it is not clear that it is computable; and later developments, in fact, show that it isn't.

We can now prove the following.

Theorem 4.6.1 *Let T be any ω-consistent, computably axiomatized theory extending Q. Then T is not complete.*

Proof. Let T be any computably axiomatized theory containing Q, and let $\mathrm{Prov}_T(y)$ be the formula we described above. By the fixed-point lemma, there is a formula φ such that T proves

$$\varphi \leftrightarrow \neg \mathrm{Prov}_T(\ulcorner \varphi \urcorner). \tag{4.1}$$

Note that φ says, in essence, "I am not provable."

I claim that

1. If T is consistent, T doesn't prove φ
2. If T is ω-consistent, T doesn't prove $\neg\varphi$.

This means that if T is ω-consistent, it is incomplete, since it proves neither φ nor $\neg\varphi$. Let us take each claim in turn.

Suppose T proves φ. Then there *is* a proof, and so, for some number m, the relation $Pr_T(m, \ulcorner \varphi \urcorner)$ holds. But then T proves the sentence

$\mathrm{Pr}_T(\overline{m}, \ulcorner\varphi\urcorner)$. So T proves $\exists x\ \mathrm{Pr}_T(x, \ulcorner\varphi\urcorner)$, which is, by definition, $\mathrm{Prov}_T(\ulcorner\varphi\urcorner)$. By the equivalence (4.1), T proves $\neg\varphi$. We have shown that if T proves φ, then it also proves $\neg\varphi$, and hence it is inconsistent.

For the second claim, let us show that if T proves $\neg\varphi$, then it is ω-inconsistent. Suppose T proves $\neg\varphi$. If T is inconsistent, it is ω-inconsistent, and we are done. Otherwise, T is consistent, so it does not prove φ. Since there is no proof of φ in T, T proves

$$\neg\mathrm{Pr}_T(\overline{0}, \ulcorner\varphi\urcorner), \neg\mathrm{Pr}_T(\overline{1}, \ulcorner\varphi\urcorner), \neg\mathrm{Pr}_T(\overline{2}, \ulcorner\varphi\urcorner), \ldots$$

On the other hand, by equivalence (4.1) $\neg\varphi$ is equivalent to $\exists x\ \mathrm{Pr}_T(x, \ulcorner\varphi\urcorner)$. So T is ω-inconsistent. □

Recall that we have proved a stronger theorem, "replacing ω-consistent" with "consistent."

Theorem 4.6.2 *Let T be any* consistent*, computably axiomatized theory extending Q. Then T is not complete.*

Can we modify Gödel's proof, to get this stronger result? The answer is "yes," using a trick discovered by Rosser. Let $not(x)$ be the primitive recursive function which does the following: if x is the code of a formula φ, $not(x)$ is a code of $\neg\varphi$. To simplify matters, assume T has a function symbol n*ot* such that for any formula φ, T proves $\mathrm{n}ot(\ulcorner\varphi\urcorner) = \ulcorner\neg\varphi\urcorner$. This is not a major assumption; since $not(x)$ is computable, it is represented in T by some formula $\theta_{not}(x, y)$, and we could eliminate the reference to the function symbol in the same way that we avoided using a function symbol diag in the proof of the fixed-point lemma.

Rosser's trick is to use a "modified" provability predicate $\mathrm{Prov}'_T(y)$, defined to be

$$\exists x\ (\mathrm{Pr}_T(x, y) \wedge \forall z\ (z < x \rightarrow \neg\mathrm{Pr}_T(z, \mathrm{not}(y)))).$$

Roughly, $\mathrm{Prov}'_T(y)$ says "there is a proof of y in T, and there is no shorter proof of the negation of y." (You might find it convenient to read $\mathrm{Prov}'_T(y)$ as "y is shmovable.") Assuming T is consistent, $\mathrm{Prov}'_T(y)$ is true of the same numbers as $\mathrm{Prov}_T(y)$; but from the point of view of *provability* in T (and we now know that there is a difference between truth and provability!) the two have different properties.

By the fixed-point lemma, there is a formula φ such that T proves

$$\varphi \leftrightarrow \neg\mathrm{Prov}'_T(\ulcorner\varphi\urcorner).$$

In contrast to the proof above, here I claim that if T is consistent, T doesn't prove φ, and T doesn't prove $\neg\varphi$. (In other words, we don't need the assumption of ω-consistency.) I will ask you to verify this on your own (it is a good exercise).

You should compare this to our previous proof, which did not explicitly exhibit a statement φ that is independent of T; you have to dig to extract it from the argument. The Gödel-Rosser methods therefore have the advantage of making the independent statement perfectly clear.

At this point, it is worthwhile to spend some time with Gödel's 1931 paper. The introduction sketches the ideas we have just discussed. Even if you just skim through the paper, it is easy to see what is going on at each stage: first Gödel describes the formal system P (syntax, axioms, proof rules); then he defines the primitive recursive functions and relations; then he shows that xBy is primitive recursive, and argues that the primitive recursive functions and relations are represented in P. He then goes on to prove the incompleteness theorem, as above. In section 3, he shows that one can take the unprovable assertion to be a sentence in the language of arithmetic. This is the origin of the β-lemma, which is what we also used to handle sequences in showing that the recursive functions are representable in Q. Gödel doesn't go so far to isolate a minimal set of axioms that suffice, but we now know that Q will do the trick. Finally, in Section 4, he sketches a proof of the second incompleteness theorem, which we will discuss next.

4.7 The second incompleteness theorem

Hilbert asked for a complete axiomatization of the natural numbers, and we have shown that that this is unattainable. A more important goal of Hilbert's, the centerpiece of his program for the justification of modern ("classical") mathematics, was to find finitary consistency proofs for formal systems representing classical reasoning. With regard to Hilbert's program, then, Gödel's second incompleteness theorem was a much bigger blow.

The second incompleteness theorem can be stated in vague terms, like the first incompleteness theorem. Roughly speaking, then, it says that no sufficiently strong theory of arithmetic can prove its own consistency. For the proof I will sketch below, we will have to take "sufficiently strong" to include a little bit more than Q. The textbook states the results in terms of Peano arithmetic, which is more than enough; in fact, one can prove even stronger versions of the incompleteness theorem for Q, but it involves more work.

Peano arithmetic, or PA, is the theory extending Q with induction axioms for all formulas. In other words, one adds to Q axioms of the form

$$\varphi(0) \wedge \forall x\, (\varphi(x) \to \varphi(x+1)) \to \forall x\, \varphi(x)$$

for every formula φ. Notice that this is really a *schema*, which is to say, infinitely many axioms (and it turns out that PA is *not* finitely axiomatizable). But since one can effectively determine whether or not a string of symbols is an instance of an induction axiom, the set of axioms for PA is computable. PA is a much more robust theory than Q. For example, one can easily prove that addition and multiplication are commutative, using induction in the usual way. In fact, most finitary number-theoretic and combinatorial arguments can be carried out in PA.

Since PA is computably axiomatized, the provability predicate $Pr_{PA}(x, y)$ is computable and hence represented in Q (and so, in PA). As before, I will take $\mathrm{Pr}_{PA}(x, y)$ to denote the formula representing the relation. Let $\mathrm{Prov}_{PA}(y)$ be the formula $\exists x\, Pr_{PA}(x, y)$, which, intuitively says, "y is provable from the axioms of PA." The reason we need a little bit more than the axioms of Q is we need to know that the theory we are using is strong enough to prove a few basic facts about this provability predicate. In fact, what we need are the following facts:

1. If $PA \vdash \varphi$, then $PA \vdash \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner)$
2. For every formula φ and ψ, $PA \vdash \mathrm{Prov}_{PA}(\ulcorner \varphi \to \psi \urcorner) \to (\mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \to \mathrm{Prov}_{PA}(\ulcorner \psi \urcorner))$
3. For every formula φ, $PA \vdash \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \to \mathrm{Prov}_{PA}(\ulcorner \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \urcorner)$.

The only way to verify that these three properties hold is to describe the formula $\mathrm{Prov}_{PA}(y)$ carefully and use the axioms of PA to describe the relevant formal proofs. Clauses 1 and 2 are easy; it is really clause 3 that requires work. (Think about what kind of work it entails...) Carrying out the details would be tedious and uninteresting, so here I will ask you to take it on faith the PA has the three properties listed above. A reasonable choice of $\mathrm{Prov}_{PA}(y)$ will also satisfy

4. If PA proves $\mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner)$, then PA proves φ.

But we will not need this fact.

(Incidentally, notice that Gödel was lazy in the same way we are being now. At the end of the 1931 paper, he sketches the proof of the second incompleteness theorem, and promises the details in a later paper. He never

got around to it; since everyone who understood the argument believed that it could be carried out, he did not need to fill in the details.)

How can we express the assertion that PA doesn't prove its own consistency? Saying PA is inconsistent amounts to saying that PA proves $0 = 1$. So we can take Con_{PA} to be the formula $\neg \mathrm{Prov}_{PA}(\ulcorner 0 = 1 \urcorner)$, and then the following theorem does the job:

Theorem 4.7.1 *Assuming PA is consistent, then PA does not prove* Con_{PA}.

It is important to note that the theorem depends on the particular representation of Con_{PA} (i.e. the particular representation of $\mathrm{Prov}_{PA}(y)$). All we will use is that the representation of $\mathrm{Prov}_{PA}(y)$ has the three properties above, so the theorem generalizes to any theory with a provability predicate having these properties.

It is informative to read Gödel's sketch of an argument, since the theorem follows like a good punch line. It goes like this. Let φ be the Gödel sentence that we constructed in the last section. We have shown "If PA is consistent, then PA does not prove φ." If we formalize this *in* PA, we have a proof of

$$\mathrm{C}on_{PA} \rightarrow \neg \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner).$$

Now suppose PA proves $\mathrm{C}on_{PA}$. Then it proves $\neg \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner)$. But since φ is a Gödel sentence, this is equivalent to φ. So PA proves φ.

But: we know that if PA is consistent, it doesn't prove φ! So if PA is consistent, it can't prove $\mathrm{C}on_{PA}$.

To make the argument more precise, we will let φ be the Gödel sentence and use properties 1–3 above to show that PA proves $\mathrm{C}on_{PA} \rightarrow \varphi$. This will show that PA doesn't prove $\mathrm{C}on_{PA}$. Here is a sketch of the proof, in PA:

$\varphi \rightarrow \neg \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner)$	since φ is a Gödel sentence
$\mathrm{Prov}_{PA}(\ulcorner \varphi \rightarrow \neg \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \urcorner)$	by 1
$\mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \rightarrow \mathrm{Prov}_{PA}(\ulcorner \neg \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \urcorner)$	by 2
$\mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \rightarrow \mathrm{Prov}_{PA}(\ulcorner \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \rightarrow 0 = 1 \urcorner)$	by 1 and 2
$\mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \rightarrow \mathrm{Prov}_{PA}(\ulcorner \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \urcorner)$	by 3
$\mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \rightarrow \mathrm{Prov}_{PA}(\ulcorner 0 = 1 \urcorner)$	using 1 and 2
$\mathrm{C}on_{PA} \rightarrow \neg \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner)$	by contraposition
$\mathrm{C}on_{PA} \rightarrow \varphi$	since φ is a Gödel sentence

The move from the third to the fourth line uses the fact that $\neg \mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner)$ is equivalent to $\mathrm{Prov}_{PA}(\ulcorner \varphi \urcorner) \rightarrow 0 = 1$ in PA. The more abstract version of the incompleteness theorem is as follows:

Theorem 4.7.2 *Let T be any theory extending Q and let $Prov_T(y)$ be any formula satisfying 1–3 for T. Then if T is consistent, then T does not prove $\neg Prov_T(\ulcorner 0 = 1 \urcorner)$.*

The moral of the story is that no "reasonable" consistent theory for mathematics can prove its own consistency. Suppose T is a theory of mathematics that includes Q and Hilbert's "finitary" reasoning (whatever that may be). Then, the whole of T cannot prove the consistency of T, and so, a fortiori, the finitary fragment can't prove the consistency of T either. In that sense, there cannot be a finitary consistency proof for "all of mathematics."

There is some leeway in interpreting the term finitary, and Gödel, in the 1931 paper, grants the possibility that something we may consider "finitary" may lie outside the kinds of mathematics Hilbert wanted to formalize. But Gödel was being charitable; today, it is hard to see how we might find something that can reasonably be called finitary but is not formalizable in, say, *ZFC*.

4.8 Löb's theorem

In this section, we will consider a fun application of the fixed-point lemma. We now know that any "reasonable" theory of arithmetic is incomplete, which is to say, there are sentences φ that are neither provable nor refutable in the theory. One can ask whether, in general, a theory can prove "If I can prove φ, then it must be true." The answer is that, in general, it can't. More precisely, we have:

Theorem 4.8.1 *Let T be any theory extending Q, and suppose $Prov_T(y)$ is a formula satisfying conditions 1–3 from the previous section. If T proves $Prov_T(\ulcorner \varphi \urcorner) \to \varphi$, then in fact T proves φ.*

Put differently, if φ is not provable in T, T can't prove $Prov_T(\ulcorner \varphi \urcorner) \to \varphi$. This is known as Löb's theorem.

The heuristic for the proof of Löb's theorem is a clever proof that Santa Claus exists. (If you don't like that conclusion, you are free to substitute any other conclusion you would like.) Here it is:

1. Let X be the sentence, "If X is true, then Santa Claus exists."
2. Suppose X is true.
3. Then what is says is true; i.e. if X is true, then Santa Claus exists.

4. Since we are assuming X is true, we can conclude that Santa Claus exists.

5. So, we have shown: "If X is true, then Santa Claus exists."

6. But this is just the statement X. So we have shown that X is true.

7. But then, by the argument above, Santa Claus exists.

A formalization of this idea, replacing "is true" with "is provable," yields the proof of Löb's theorem. Suppose φ is a sentence such that T proves $Prov_T(\ulcorner\varphi\urcorner) \to \varphi$. Let $\psi(y)$ be the formula $Prov_T(y) \to \varphi$, and used the fixed point theorem to find a sentence θ such that T proves $\theta \leftrightarrow \psi(\ulcorner\theta\urcorner)$. Then each of the following is provable in T:

$\theta \to (Prov_T(\ulcorner\theta\urcorner) \to \varphi)$	
$Prov_T(\ulcorner\theta \to (Prov_T(\ulcorner\theta\urcorner) \to \varphi)\urcorner)$	by 1
$Prov_T(\ulcorner\theta\urcorner) \to Prov_T(\ulcorner Prov_T(\ulcorner\theta\urcorner) \to \varphi\urcorner)$	using 2
$Prov_T(\ulcorner\theta\urcorner) \to (Prov_T(\ulcorner Prov_T(\ulcorner\theta\urcorner)\urcorner) \to Prov_T(\ulcorner\varphi\urcorner))$	using 2
$Prov_T(\ulcorner\theta\urcorner) \to Prov_T(\ulcorner Prov_T(\ulcorner\theta\urcorner)\urcorner)$	by 3
$Prov_T(\ulcorner\theta\urcorner) \to Prov_T(\ulcorner\varphi\urcorner)$	
$Prov_T(\ulcorner\varphi\urcorner) \to \varphi$	by assumption
$Prov_T(\ulcorner\theta\urcorner) \to \varphi$	
θ	def of θ
$Prov_T(\ulcorner\theta\urcorner)$	by 1
φ	

This completes the proof of Löb's theorem.

With Löb's theorem in hand, there is a short proof of the first incompleteness theorem (for theories having a provability predicate solving 1–3): if a theory proves $Prov_T(\ulcorner 0 = 1\urcorner) \to 0 = 1$, it proves $0 = 1$.

Here is a good exercise, to make sure you are attuned to the subtleties involved. Let T be an effectively axiomatized theory, and let Prov be a provability predicate for T. Consider the following four statements:

1. If $T \vdash \varphi$, then $T \vdash \mathrm{Prov}(\ulcorner\varphi\urcorner)$.

2. $T \vdash \varphi \to \mathrm{Prov}(\ulcorner\varphi\urcorner)$.

3. If $T \vdash \mathrm{Prov}(\ulcorner\varphi\urcorner)$, then $T \vdash \varphi$.

4. $T \vdash \mathrm{Prov}(\ulcorner\varphi\urcorner) \to \varphi$

Under what conditions are each of these statements true?

4.9 The undefinability of truth

Now, for the moment, we will set aside the notion of *proof* and consider the notion of *definability*. This notion depends on having a formal semantics for the language of arithmetic, and we have not covered semantic notions for this course. But the intuitions are not difficult. We have described a set of formulas and sentences in the language of arithmetic. The "intended interpretation" is to read such sentences as making assertions about the natural numbers, and such an assertion can be true or false. In this section I will take $\mathcal{N}$ to be the structure $\langle \mathbb{N}, 0, ', +, \times, < \rangle$, and I will write $\mathcal{N} \models \varphi$ for the assertion "φ is true in the standard interpretation."

Definition 4.9.1 *A relation $R(x_1, \ldots, x_k)$ of natural numbers is* definable *in $\mathcal{N}$ if and only if there is a formula $\varphi(x_1, \ldots, x_k)$ in the language of arithmetic such that for every $n_1, \ldots, n_k$, $R(n_1, \ldots, n_k)$ if and only if $\mathcal{N} \models \varphi(\overline{n}_1, \ldots, \overline{n}_k)$.*

Put differently, a relation is definable in in $\mathcal{N}$ if and only if it is representable in the theory *Arith*, where $Arith = \{\varphi \mid \mathcal{N} \models \varphi\}$ is the set of true sentences of arithmetic. (If this is not immediately clear to you, you should go back and check the definitions and convince yourself that this is the case.)

Lemma 4.9.2 *Every computable relation is definable in $\mathcal{N}$.*

Proof. It is easy to check that the formula representing a relation in Q defines the same relation in $\mathcal{N}$. □

Now one can ask, is "definable in $\mathcal{N}$" the same as "representable in Q"? The answer is no. For example:

Lemma 4.9.3 *Every c.e. set is definable in $\mathcal{N}$.*

Proof. Suppose S is the range of φ_e, i.e.

$$S = \{x \mid \exists y\, T(\overline{e}, x, y)\}.$$

Let θ_T define T in $\mathcal{N}$. Then

$$S = \{x \mid \mathcal{N} \models \exists y\, \theta_T(\overline{e}, \overline{x}, y)\},$$

so $\exists y\, \theta_T(\overline{e}, x, y)$ defines S is $\mathcal{N}$. □

Corollary 4.9.4 *Q is definable in arithmetic.*

So, more sets are definable in $\mathcal{N}$. For example, it is not hard to see that complements of c.e. sets are also definable. The sets of numbers definable in $\mathcal{N}$ are called, appropriately, the "arithmetically definable sets," or, more simply, the "arithmetic sets." I will draw a picture on the board.

What about *Arith* itself? Is it definable in arithmetic? That is: is the set $\{\ulcorner\varphi\urcorner \mid \mathcal{N} \models \varphi\}$ definable in arithmetic? Tarski's theorem answers this in the negative.

Theorem 4.9.5 *The set of true statements of arithmetic is not definable in arithmetic.*

Proof. Suppose $\theta(x)$ defined it. By the fixed-point lemma, there is a formula φ such that Q proves $\varphi \leftrightarrow \neg\theta(\ulcorner\varphi\urcorner)$, and hence $\mathcal{N} \models \varphi \leftrightarrow \neg\theta(\ulcorner\varphi\urcorner)$. But then $\mathcal{N} \models \varphi$ if and only if $\mathcal{N} \models \neg\theta(\ulcorner\varphi\urcorner)$, which contradicts the fact that $\theta(y)$ is suppose to define the set of true statements of arithmetic. □

Tarski applied this analysis to a more general philosophical notion of truth. Given any language L, Tarski argued that an adequate notion of truth for L would have to satisfy, for each sentence X,

> 'X' is true if and only if X.

Tarski's oft-quoted example, for English, is the sentence

> 'Snow is white' is true if and only if snow is white.

However, for any language strong enough to represent the diagonal function, and any linguistic predicate $T(x)$, and can construct a sentence X satisfying "X if and only if not $T(\text{'}X\text{'})$." Given that we do not want a truth predicate to declare some sentences to be both true and false, Tarski concluded that one cannot specify a truth predicate for all sentences in a language without, somehow, stepping outside the bounds of the language. In other words, a the truth predicate for a language cannot be defined in the language itself.

Chapter 5

Undecidability

In Section 2, we saw that a many natural questions about computation are undecidable. Indeed, Rice's theorem tells us that any general question about programs that depends only on the function computed and not the program itself is undecidable. This includes questions like: "is the function computed by this program total?" "does it halt on input 0"? "Does it ever output an odd number?" In Section 4, we saw that many questions arising in the fields of logic and metamathematics are similarly undecidable: "Is sentence φ provable from the axioms of Q?" "Is sentence φ provable in pure logic?" "Is sentence φ a true statement about the natural numbers?" (Keep in mind that when one says that a certain question is algorithmically undecidable, one really means that a parameterized *class* of questions is undecidable. It does not make sense to ask whether or not a *single* question, like "Does machine 143 halt on input 0," is decidable; the answer is presumed to be either yes or no!)

One of the most exciting aspects of the field of computability is that undecidability extends well beyond questions related to logic and computation. Since the seminal work 1930's, many natural questions have been shown to be undecidable, in fields such as combinatorics, algebra, number theory, linguistics, and so on. A general method for showing that a problem is undecidable is to show that the halting problem is reducible to it; or, iteratively, to show that something you have previously shown to be undecidable is reducible to it. Most of the theory of undecidability has developed along these lines, and in many cases the appropriate reduction is far from obvious.

To give you a sense of the field, below I will present some examples of undecidable problems, and in class I will present some of the easier proofs.

Most of the examples I will discuss are in the handout I have given you, taken from Lewis and Papadimitriou's book, *Elements of the Theory of Computation*. Hilbert's 10th problem is discussed in an appendix to Martin Davis' book *Computability and Unsolvability*.

5.1 Combinatorial problems

A *Thue system* is a finite set of unordered pairs of strings,

$$\{u_0, v_0\}, \{u_1, v_1\}, \ldots, \{u_n, v_n\}$$

over some fixed (finite) alphabet. Given such a system and strings x and y, say x *can be transformed to* y *in one step* if there is a pair $\{u, v\}$ in the system such that y can be obtained from x be replacing some substring u in x by v; in other words, for some (possibly empty) strings s and t, x is sut and y is svt. Say that x and y are *equivalent* if x can be transformed to y in finitely many steps. For example, given the Thue system

$$\{abba, bba\}, \{aab, ba\}, \{bb, aa\}$$

try transforming $aabbab$ to $bbba$.

The following question is undecidable:

- Given a Thue system and strings x and y, is x equivalent to y?

In fact, there is even a single Thue system and a string x such that the following question is undecidable:

- Given a string y, is x equivalent to y?

A *correspondence system* is a finite set of *ordered* pairs of strings,

$$\langle u_0, v_0\rangle, \langle u_1, v_1\rangle, \ldots, \langle u_n, v_n\rangle.$$

A *match* is a sequence of pairs $\langle x_i, y_i\rangle$ from the list above (with duplicates allowed) such that

$$x_0 x_1 \ldots x_k = y_0 y_1 \ldots y_k.$$

For example, try finding a match for the following system:

$$\langle a, ab\rangle, \langle b, bbb\rangle, \langle b, a\rangle, \langle abb, b\rangle$$

The following question is undecidable:

- Given a correspondence system, is there a match?

This is known as *Post's correspondence problem*.

Finally, let us consider the *Tiling problem*. By *a set of tiles* I mean a finite set of square tiles, each of which comes an orientation ("this end up"), and each edge of which has a *color*. Given a finite set of tiles, a *tiling of the quarter plane* is a map from $\mathbb{N} \times \mathbb{N}$ to the set of tiles, such that adjacent edges share the same colors. The following question is undecidable:

- Given a finite set of tiles, is there a tiling of the quarter plane?

(Those of you who took 80-310/610 may recall that, by compactness, this is equivalent to asking "are there arbitrarily large finite tilings?" i.e. the question, "are there tilings of $\{0, \ldots, n\} \times \{0, \ldots, n\}$ for arbitrarily large n?")

There is nothing special about the quarter plane; one can just as well ask about tilings of the whole plane, though restricting to the quarter plane makes the proof of undecidability a little shorter. One can also show that there are finite sets of tiles for which it *is* possible to tile the plane, but such that there is no algorithm for deciding *which* tile should go in position i, j. (These are called "uncomputable tilings.") People have even tried to determine the smallest number of tiles for which this can happen. I do not know what is currently the best record.

Note that the set of Thue systems with equivalent pairs, the set of correspondence systems having a matching, and the set of tilings that do *not* tile the plane are all *computably enumerable*. Given a Thue system and a pair, we can search for a derivation showing the pair to be equivalent; given a correspondence system, we can search for a matching; and given a tiling, we can search for an n such that $n \times n$ is not tile-able. In fact, the proofs that they are not computable show that they are complete computably enumerable sets.

5.2 Problems in linguistics

Linguists are fond of studying *grammars*, which is to say, rules for producing sentences. A grammar consists of:

- A set of *symbols* V
- A subset of V called the *terminal symbols*
- A nonterminal *start symbol* in V

- A set of *rules*, i.e. pairs $\langle u, v \rangle$, where u is a string of symbols with at least one nonterminal symbol, and v is a string of symbols.

You can think of the symbols as denoting grammatical elements, and the terminal symbols as denoted basic elements like words or phrases. In the example below, you can think of Se as standing for "sentence," Su a standing for "subject," Pr as standing for "predicate," and so on.

$$\begin{aligned}
Se &\to Su\ Pr \\
Su &\to Art\ N \\
Art &\to \text{the} \\
Art &\to \text{a} \\
N &\to \text{dog} \\
N &\to \text{boy} \\
N &\to \text{ball} \\
Pr &\to VI \\
Pr &\to VT\ Su \\
VI &\to \text{flies} \\
VI &\to \text{falls} \\
VT &\to \text{kicks} \\
VT &\to \text{throws}
\end{aligned}$$

In the general setup, there may be more than one symbol on the left side; such grammars are called "unrestricted," or "context sensitive," because you can think of the extra symbols on the left as specifying the context in which a substitution can occur. For example, you could have rules

$$\begin{aligned}
Pr &\to Pr \text{ and } Pr \\
\text{and } Pr &\to \text{and his } Pr
\end{aligned}$$

indicating that one can replace "Pr" by "his Pr" only in the context of a preceding "and". (These are lame examples; I am not a linguist!) The *language* generated by the grammar is the set of strings of nonterminal symbols that one can obtain by applying the rules. For example, for the language above, "The boy throws the ball" is in the language generated by the grammar above.

For a more computational example, the following grammar generates strings of a's, b's, and c's, with the same number of each:

$$\begin{aligned} S &\to \emptyset \\ S &\to ABCS \\ AB &\to BA \\ BA &\to AB \\ AC &\to CA \\ CA &\to AC \\ BC &\to CB \\ CB &\to BC \\ A &\to a \\ B &\to b \\ C &\to c \end{aligned}$$

It is less obvious that one can code Turing machine computations into grammars, but the handout shows that this is the case. As a result, the following questions are undecidable:

- Given a grammar G and a string w, is w in the language generated by G?
- Given grammars G_1 and G_2, do they generate the same language?
- Given a grammar G, is *anything* in the language generated by G?

The first question shows that in general it is not possible to *parse* an unrestricted grammar, which is obviously an undesirable feature for some formal languages, such as programming languages. For that very reason, computer scientists are interested in more restricted classes of grammars, for which one can parse computably, and even reasonably efficiently. For example, if in every rule $\langle u, v\rangle$ has to consist of a single nonterminal symbol, one has the *context free grammars*, and these can be parsed. But still the following questions are undecidable:

- Given context free grammars G_1 and G_2, is there any string that is simultaneously in both languages?
- Given a context free grammar G, does every string in the language of G have a unique parse tree? (In other words, is the grammar *unambiguous*?)

5.3 Hilbert's 10th problem

A *Diophantine equation* is an equation between two terms built up from finitely many variables, plus, and times; for example:

$$xxyz = xy + xy + xy + yz.$$

Since $xy + xy + xy = 3xy$ and $xx = x^2$, we can rewrite the above as

$$x^2yz = 3xy + yz,$$

and, more generally, think of Diophantine equations as equations between multivariable polynomials with integer coefficients. (Terms with negative coefficients can always be moved to the other side of the equality). A *solution* to a Diophantine equation is an assignment of integer values to the variables that makes the equation true. Can you decide which of the following equations have integer solutions?

- $15x + 6y = 19$
- $x^2 = 48$
- $2x^2 - y^2 = 56$

As one of the 23 problems presented to the international congress of mathematicians in 1900, David Hilbert asked for an algorithm that, given a Diophantine equation, determines whether or not there is a solution; or for a proof that there is no such algorithm. Hilbert was prescient to have even considered the possibility that this problem is unsolvable, more than 30 years before the theory of computability was born!

Notice that the set of Diophantine equations with integer solutions is *computably enumerable*; given an equation, just search systematically for a solution. The question is as to whether this set is *computable*, and for a long time it was not clear which way the answer would go. In the 1950's and 1960's, Hilary Putnam, Martin Davis, and Julia Robinson made significant progress towards answering the question; and Yuri Matjasevic finally finished it off in 1970, providing the final lemma needed to showing that the problem is unsolvable. The theorem is usually called "the MRDP theorem" in their honor.