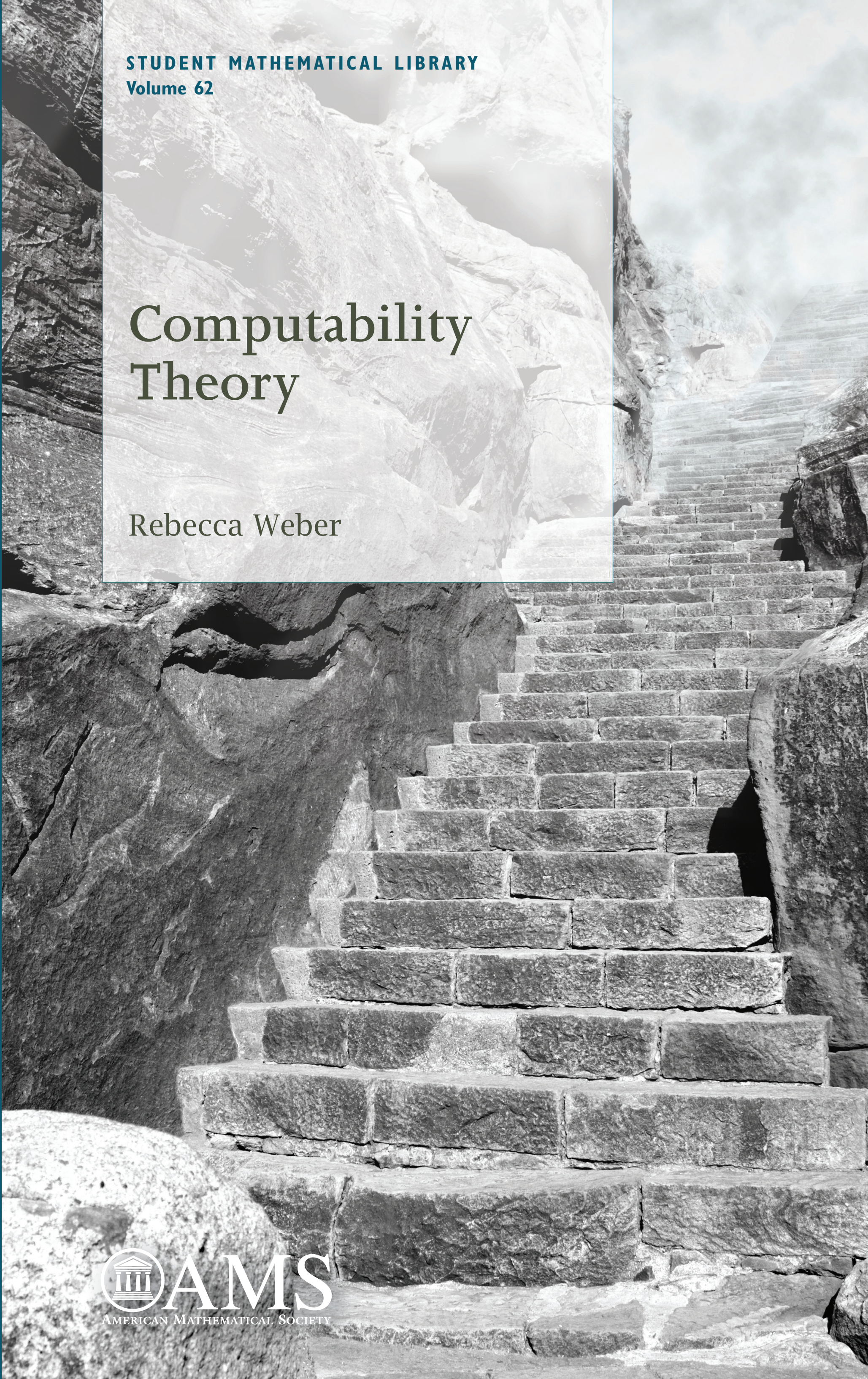
STUDENT MATHEMATICAL LIBRARY
Volume 62

Computability Theory

Rebecca Weber



AMS
American Mathematical Society

Computability Theory

STUDENT MATHEMATICAL LIBRARY
Volume 62

Computability Theory

Rebecca Weber

AMS
American Mathematical Society
Providence, Rhode Island

2000 *Mathematics Subject Classification*. Primary 03Dxx;
Secondary 68Qxx.

For additional information and updates on this book, visit
www.ams.org/bookpages/stml-62

Library of Congress Cataloging-in-Publication Data

Weber, Rebecca, 1977–
Computability theory / Rebecca Weber.
p. cm. — (Student mathematical library ; v. 62)
Includes bibliographical references and index.
ISBN 978-0-8218-7392-2 (alk. paper)
1. Recursion theory. 2. Computable functions. I. Title.

QA9.6.W43 2012
511.3′52—dc23

2011050912

Contents

Chapter 1

Introduction

The bird's-eye view of computability: what does it mean, how does it work, where did it come from?

1.1. Approach

If I can program my computer to execute a function, to take any input and give me the correct output, then that function should certainly be called computable. That set of functions is not very large, though; I am not a particularly skilled programmer and my computer is nothing spectacular. I might expand my definition to say if a wizardly programmer can program the best computer there is to execute a function, that function is computable. However, programming languages and environments improve with time, making it easier to program complicated functions, and computers increase in processing speed and amount of working memory.

The idea of "computable" as "programmable" provides excellent intuition, but the precise mathematical definition needs to be an idealized version that does not change with hardware advances. It should capture all of the functions that may be automated even in theory.

To that end, computability theory removes any restriction on time and memory except that a successful computation must use only finitely much of each. When that change is made, the rest of the details are fairly flexible: with sufficient memory, even the simple language of a programmable calculator is sufficient to implement all computable functions. An explicit definition is in §3.2.

Most computable functions, by this definition, are completely infeasible today, requiring centuries to run and more memory than currently exists on the planet. Why should we include them? One reason is that if we require feasibility, we have to draw a line between feasible and infeasible, and it is unclear where we ought to draw that line. A related reason is that if we do not require feasibility, and then we prove a function is noncomputable, the result is as strong as possible: no matter how much computing technology improves, the function can never be automated. A major historical example of such a proof is described in the next section, and more are in §4.5, from a variety of mathematical fields. These unsolvable problems are often simple statements about basic properties, so intuition may suggest they should be easy to compute.

A rigorous and general definition of computability also allows us to pin down concepts that are *necessarily* noncomputable. One example is randomness, which will be explored in more depth in §9.2. If a fair coin is flipped repeatedly, we do not expect to be able to win a lot of money betting on the outcome of each flip in order. We may get lucky sometimes, but in the long run whatever betting strategy we use should be essentially equivalent in success to betting heads every time: a fifty percent hit rate. That is an intuitive notion of what it means for the flip outcomes to be random. To turn it into mathematics we might represent betting strategies as functions, and say that a sequence of flips is random if no betting strategy is significantly better than choosing heads every time. However, for any given sequence of coin flips C, there is a function that bets correctly on every flip of C. If the class of functions considered to be betting strategies is unrestricted, no sequences will be random, but intuition also says *most* sequences will be random. One restriction we can choose is to require that the betting function be computable; this ties into the idea that the computable functions are those to which we have some kind of "access." If we could never hope to implement a function, it is difficult to argue that it allows us to predict coin flips.

I would like to highlight a bit of the mindset of computability theory here. Computability theorists pay close attention to *uniformity.* A nonuniform proof of computability proves some program exists to compute the function, and typically has some ad hoc component. A uniform proof explicitly builds such a program. A common example is showing explicitly that there is a program P that computes $f(n)$ provided $n \geq N$ for some fixed, finite N, and considering f computable despite the fact that P may completely fail at finding $f(n)$ for $0 \leq n < N$ (in computability theory, functions are on the natural numbers; see §3.4 for why that is not as stringent a restriction as it may seem). P is sufficient because for any given sequence of N numbers, there is a program Q that assigns those numbers in order as $f(0)$ through $f(N-1)$, and uses P to find the output for larger numbers. The function we want to compute must have *some* fixed sequence of outputs for the first N inputs, and hence via the appropriate Q it may be computed. It is computable, although we would need to magically (that is, nonuniformly) know the first N outputs to have the full program explicitly. Nonuniformity in isolation is not a problem, but it reduces the possible uses of the result. We will discuss uniformity from time to time as more examples come up.

Computability theorists more often work in the realm of the noncomputable than the computable, via approximations and partial computations. Programs that do not always give an output are seen regularly; on certain inputs such a program may just chug and chug and never finish. Of course, the program either gives an output or doesn't, and in many areas of mathematics we would be able to say "if the program halts, use the output in this way; if not, do this other computation." In fact we *could* do that in computability theory, but the question of whether any given program halts is a noncomputable one (see §4.1). Typically we want to complete our constructions using as little computational power as we can get away with, because that allows us to make stronger statements about the computability or noncomputability of the function we have built.

To succeed in such an environment, the construction must continue while computations are unfinished, working with incomplete and possibly incorrect assumptions. Mistakes will be made and need to be repaired. The standard means of dealing with this situation is a *priority argument* (§6.2), which breaks the goals of the construction into small pieces and orders them. A piece (*requirement*) that is earlier in the ordering has higher priority and gets to do what appears at the moment to satisfy its goal even if that is harmful to later requirements. When done correctly, each requirement can be met at a finite stage and cease harming the lower-priority requirements, and each requirement can recover from the damage it sustains from the finitely many higher-priority requirements. Satisfaction of requirements cascades irregularly from the beginning of the order down.

1.2. Some History

The more early work in computability theory you read, the more it seems its founding was an inevitability. There was a push to make mathematics formal and rigorous, and find mechanical methods to solve problems and determine truth or falsehood of statements; hence there was a lot of thought on the nature of mechanical methods and formality. For more on this early work, I recommend John Hopcroft's article "Turing Machines" [41] and two survey papers by Martin Davis [20, 21]. If you wish to go deeper philosophically (and broader mathematically), try van Heijenoort [86] and Webb [88].

David Hilbert gave an address in 1900 in which he listed problems he thought should direct mathematical effort as the new century began [37]. His tenth problem, paraphrased, was to find a procedure to determine whether any given multivariable polynomial equation with integer coefficients (a *Diophantine* equation) has an integer solution. Hilbert asked for "a process according to which it can be determined by a finite number of operations" whether such a solution exists. For the specific case of single-variable Diophantine equations, mathematicians already had the rational root test; for an equation of the form $a_nx^n + a_{n-1}x^{n-1} + \cdots + a_0 = 0$, any rational solution must be of the form $\pm\frac{r}{s}$ where r divides a_0 and s divides a_n. One may manually check each such fraction and determine whether any of them yields equality.

In 1910, the first volume of Alfred North Whitehead and Bertrand Russell's *Principia Mathematica* was published [89]. This ultimately three-volume work was an effort to develop all mathematics from a small common set of axioms, with full rigor at every step. Russell and Whitehead wanted to remove vagueness and paradox from mathematics; every step in their system could be checked mechanically.

It seems this might take all creativity out of mathematics, as all possible theorems would eventually be produced by the mechanical application of logical deduction rules to axioms and previously generated theorems. However, in 1931 Gödel showed it was impossible for such a system to produce all true mathematical statements [30]. He used the mechanical nature of the system, intended for rigor, and showed it allowed a system of formula encoding that gave access to self-reference: he produced a formula P that says "P has no proof." If P is true, it is unprovable. If the negation of P is true, P has a proof, since that is the assertion made by P's negation. Therefore, unless Russell and Whitehead's system is internally inconsistent, P must be true, and hence unprovable. We will see a proof of Gödel's Incompleteness Theorem via undecidable problems in §5.3.

Principia Mathematica was a grand undertaking and one might expect it to fail to fulfill all its authors' hopes. However, Hilbert's quest was doomed as well. The proof that there can be no procedure to determine whether an arbitrary Diophantine equation has integer roots was not completed until 1973 [61], but in 1936 Church made the first response suggesting that might be the case [14]. He wrote:

> There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function f of n positive integers, such that $f(x_1, \ldots, x_n) = 2$ is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving $x_1, \ldots, x_n$ as free variables. [footnote: The selection of the particular positive integer 2 instead of some other is, of course, accidental and nonessential.]
>
> ... The purpose of the present paper is to propose a definition of effective calculability which is thought to correspond satisfactorily to the somewhat vague

> intuitive notion in terms of which problems of this class are often stated, and to show, by means of an example, that not every problem of this class is solvable.

Church's major contribution here is the point that we need some *formal* notion of "finite process" to answer Hilbert. He proposes two options in this paper: the lambda calculus, due to him and Kleene, and recursive functions, defined originally by Gödel [30] (after a suggestion by Herbrand) and modified by Kleene. Shortly thereafter Kleene proposed what we now call the partial recursive functions [44]. It was not widely accepted at the time that any of these definitions was a good characterization of "effectively computable," however. It was not until Turing developed his Turing machine [85], which *was* accepted as a good characterization, and it was proved that Turing-computable functions, lambda-computable functions, and partial recursive functions are the same class, that the functional definitions were accepted. All three of these formalizations of computability are studied in Chapter 3. The idea that not all problems are solvable comes up in Chapter 4, along with many of the tools used in such proofs.

Both Gödel's Incompleteness Theorem and Church's unsolvability result treat the limitations of mechanical, or algorithmic, procedures in mathematics. As is common in mathematics, these new ideas and tools took on a life of their own beyond answering Hilbert or finding the major flaw in Russell and Whitehead's approach to mathematics. The new field became known as recursion theory or computability theory. Chapters 5–8 explore some of the additional topics and fundamental results of the area, and Chapter 9 contains a survey of some areas of current interest to computability theorists.

1.3. Notes on Use of the Text

My intent is that Chapter 2 will be covered on an as-needed basis, and I have tried to include references to it wherever applicable throughout the text. However, the vocabulary in §§2.1 and 2.2 is needed throughout, so they should be read first if unfamiliar. Returning to §1.1 after reading through Chapter 4 may be helpful as well.

The core material is in Chapters 3 through 7. In those, without losing continuity, §§3.7, 4.5, 5.3, 6.2, and 6.3 may be omitted; if §6.2 is omitted, §5.4 may also be.

Chapters 8 and 9 should be covered as interest warrants. There is no interdependence between sections of these chapters except that §§9.4 and 9.5 both draw on §9.3, and §9.1 leans lightly on §8.3.

1.4. Acknowledgements and References

These notes owe a great debt to a small library of logic books. For graduate- and research-level work I regularly refer to *Classical Recursion Theory* by P. G. Odifreddi [68], *Theory of Recursive Functions and Effective Computability* by H. Rogers [75], and *Recursively Enumerable Sets and Degrees* by R. I. Soare [82]. The material in here owes a great deal to those three texts. More recently, I have enjoyed A. Nies' book *Computability and Randomness* [67]. As you will see, M. Davis' *The Undecidable* [18] was a constant presence on my desk as well.

In how to present such material to undergraduates, I was influenced by such books as *Computability and Logic* by Boolos, Burgess, and Jeffrey [10], *Computability* by Cutland [17], *A Mathematical Introduction to Logic* by Enderton [25], *An Introduction to Formal Languages and Automata* by Linz [57], and *A Transition to Advanced Mathematics* by Smith, Eggen, and St. Andre [81].

I have talked to many people about bits and pieces of the book, getting clarifications and opinions on my exposition, but the lion's share of gratitude must go to Denis Hirschfeldt. Many thanks are due also to the students in the three offerings of Computability Theory I gave as I was writing this text, first as course notes and then with my eye to publishing a book. With luck, their questions as they learned the material and their comments on the text have translated into improved exposition.

Chapter 2

Background

This chapter covers a collection of topics that are not computability theory per se, but are needed for it. They are set apart so the rest of the text reads more smoothly. If you are not familiar with logical and set notation, read §§2.1 and 2.2 now. The rest should be covered as needed when they become relevant.

2.1. First-Order Logic

In this section we learn a vocabulary for expressing formulas, logical sentences. This is useful for brevity ($x < y$ is much shorter than "x is less than y," and the savings grow as the statement becomes more complicated) but also for clarity. Expressing a mathematical statement symbolically can make it more obvious what needs to be done with it, and however carefully words are used they may admit some ambiguity.

We use lowercase Greek letters (mostly φ, ψ, and θ) to represent formulas. The simplest formula is a single symbol (or assertion) which can be either true or false. There are several ways to modify formulas, which we'll step through one at a time.

The *conjunction* of formulas φ and ψ is written "φ and ψ," "$\varphi \wedge \psi$," or "φ & ψ." It is true when both φ and ψ are true, and false otherwise. Logically "and" and "but" are equivalent, and so are $\varphi \mathbin{\&} \psi$ and $\psi \mathbin{\&} \varphi$, though in natural language there are some differences in connotation.

The *disjunction* of φ and ψ is written "φ or ψ" or "$\varphi \vee \psi$." It is false when both φ and ψ are false, and true otherwise. That is, $\varphi \vee \psi$ is true when *at least one* of φ and ψ is true; it is *inclusive or*. English tends to use *exclusive or*, which is true only when exactly one of the clauses is true, though there are exceptions. One such: "Would you like sugar or cream in your coffee?" Again, $\varphi \vee \psi$ and $\psi \vee \varphi$ are equivalent.

The *negation* of φ is written "not(φ)," "not-φ," "$\neg\varphi$," or "$\sim\!\varphi$." It is true when φ is false and false when φ is true. The potential difference from natural language negation is that $\neg\varphi$ must cover all cases where φ fails to hold, and in natural language the scope of a negation is sometimes more limited. Note that $\neg\neg\varphi = \varphi$.

How does negation interact with conjunction and disjunction? $\varphi \mathbin{\&} \psi$ is false when φ, ψ, or both are false, and hence its negation is $(\neg\varphi) \vee (\neg\psi)$. $\varphi \vee \psi$ is false only when both φ and ψ are false, and so its negation is $(\neg\varphi)\&(\neg\psi)$. We might note in the latter case that this matches up with English's "neither...nor" construction. These two negation rules are called *De Morgan's Laws.*

Exercise 2.1.1. Simplify the following formulas.

(i) $\varphi \mathbin{\&} ((\neg\varphi) \vee \psi)$

(ii) $(\varphi \mathbin{\&} (\neg\psi) \mathbin{\&} \theta) \vee (\varphi \mathbin{\&} (\neg\psi) \mathbin{\&} (\neg\theta))$

(iii) $\neg((\varphi \mathbin{\&} \neg\psi) \mathbin{\&} \varphi)$

There are two classes of special formulas to highlight now. A *tautology* is always true; the classic example is $\varphi\vee(\neg\varphi)$ for any formula φ. A *contradiction* is always false; here the example is $\varphi \mathbin{\&} (\neg\varphi)$. You will sometimes see the former expression denoted T (or $\top$) and the latter $\bot$.

To say φ *implies* ψ ($\varphi \rightarrow \psi$ or $\varphi \Rightarrow \psi$) means whenever φ is true, so is ψ. We call φ the *antecedent*, or assumption, and ψ the *consequent*, or conclusion, of the implication. We also say φ is *sufficient* for ψ (since whenever we have φ we have ψ, though we may also have ψ when φ is false), and ψ is *necessary* for φ (since it is impossible to have φ without ψ). Clearly $\varphi \to \psi$ should be true when both formulas are true, and it should be false if φ is true but ψ is false. It is maybe not so clear what to do when φ is false; this is clarified by rephrasing implication as disjunction (which is often how it is defined in the first place). $\varphi \to \psi$ means either ψ holds or φ fails; i.e., $\psi \vee (\neg\varphi)$. The truth of that statement lines up with our assertions earlier, and gives truth values for when φ is false – namely, that the implication is true. Another way to look at this is to say $\varphi \to \psi$ is only false when *proven* false; i.e., when it has a true antecedent but a false consequent. From this it is clear that $\neg(\varphi \to \psi)$ is $\varphi \mathbin{\&} (\neg\psi)$.

There is an enormous difference between implication in natural language and implication in logic. Implication in natural language tends to connote causation, whereas the truth of $\varphi \to \psi$ need not give any connection at all between the meanings of φ and ψ. It could be that φ is a contradiction, or that ψ is a tautology. Also, in natural language we tend to dismiss implications as irrelevant or meaningless when the antecedent is false, whereas to have a full and consistent logical theory we cannot throw those cases out.

Example 2.1.2. The following are true implications:

- If fish live in the water, then earthworms live in the soil.
- If rabbits are aquamarine blue, then earthworms live in the soil.
- If rabbits are aquamarine blue, then birds drive cars.

The negation of the final statement is "Rabbits are aquamarine blue but birds do not drive cars."

The statement "If fish live in the water, then birds drive cars" is an example of a false implication.

Equivalence is two-way implication and indicated by a double-headed arrow: $\varphi \leftrightarrow \psi$ or $\varphi \Leftrightarrow \psi$. It is an abbreviation for $(\varphi \to \psi) \mathbin{\&} (\psi \to \varphi)$, and is true when φ and ψ are either both true or both false. Verbally we might say "φ if and only if ψ", which is often abbreviated to "φ iff ψ". In terms of just conjunction, disjunction, and negation, we may write equivalence as $(\varphi \mathbin{\&} \psi) \vee ((\neg\varphi) \mathbin{\&} (\neg\psi))$. Its negation is exclusive or, $(\varphi \vee \psi) \mathbin{\&} \neg(\varphi \mathbin{\&} \psi)$.

Exercise 2.1.3. Negate the following statements.

(i) 56894323 is a prime number.

(ii) If there is no coffee, I drink tea.

(iii) John watches but does not play.

(iv) I will buy the blue shirt or the green one.

Exercise 2.1.4. Write the following statements using standard logical symbols.

(i) φ if ψ.

(ii) φ only if ψ.

(iii) φ unless ψ.

As an aside, let us have a brief introduction to *truth tables*. These are nothing more than a way to organize information about logical statements. The leftmost columns are generally headed by the individual propositions, and under those headings occur all possible combinations of truth and falsehood. The remaining columns are headed by more complicated formulas that are built from the propositions, and the lower rows have T or F depending on the truth or falsehood of the header formula when the propositions have the true/false values in the beginning of that row. Truth tables aren't particularly relevant to our use for this material, so I'll leave you with an example and move on.

φ	ψ	$\neg\varphi$	$\neg\psi$	$\varphi \mathbin{\&} \psi$	$\varphi \vee \psi$	$\varphi \rightarrow \psi$	$\varphi \leftrightarrow \psi$
T	T	F	F	T	T	T	T
T	F	F	T	F	T	F	F
F	T	T	F	F	T	T	F
F	F	T	T	F	F	T	T

If we stop here, we have *propositional* (or *sentential*) logic. These formulas usually look something like $[A \vee (B \& C)] \rightarrow C$ and their truth or falsehood depends on the truth or falsehood of the assertions A, B, and C. We will continue on to *predicate* logic, which replaces these assertions with statements such as $(x < 0) \mathbin{\&} (x + 100 > 0)$, which will be true or false depending on the value substituted for the variable x. We will be able to turn those formulas into statements which are true or false inherently via *quantifiers.* Note that writing $\varphi(x)$ indicates the variable x appears in the formula φ, and does not technically forbid φ containing other variables.

The *existential* quantification $\exists x$ is read "there exists x." The formula $\exists x \varphi(x)$ is true if for some value n the unquantified formula $\varphi(n)$ is true. *Universal* quantification, on the other hand, is $\forall x \varphi(x)$ ("for all x, $\varphi(x)$ holds"), true when no matter what n we fill in for x, $\varphi(n)$ is true.

Quantifiers must have a specified set of values to range over, because the truth value of a formula may be different depending on this *domain of quantification.* For example, take the formula

$$(\forall x)(x \neq 0 \rightarrow (\exists y)(xy = 1)).$$

This asserts every nonzero x has a multiplicative inverse. If we are letting our quantifiers range over the real numbers (denoted $\mathbb{R}$) or the rational numbers ($\mathbb{Q}$), this statement is true, because the reciprocal of x is available to play the role of y. However, in the integers ($\mathbb{Z}$) or natural numbers ($\mathbb{N}$) this is false, because $1/x$ is only in the domain when x is ± 1.

Introducing quantification opens us up to two kinds of logical formulas. If all variables are quantified over (*bound* variables), then the formula is called a *sentence.* If there are variables that are not in the scope of any quantifier (*free* variables), the formula is called a *predicate.* The truth value of a predicate depends on what values are plugged in for the free variables; a sentence has a truth value, period. For example, $(\forall x)(\exists y)(x < y)$ is a sentence, and it is true in all our usual domains of quantification. The formula $x < y$ is a predicate, and it will be true or false depending on whether the specific values plugged in for x and y satisfy the inequality.

Exercise 2.1.5. Write the following statements as formulas, specifying the domain of quantification.

(i) 5 is prime.

(ii) For any number x, the square of x is nonnegative.

(iii) There is a smallest positive integer.

Exercise 2.1.6. Consider $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$. Over which domains of quantification are each of the following statements true?

(i) $(\forall x)(x \geq 0)$

(ii) $(\exists x)(5 < x < 6)$

(iii) $(\forall x)((x^2 = 2) \to (x = 5))$

(iv) $(\exists x)(x^2 - 1 = 0)$

(v) $(\exists x)(x^2 = 5)$

(vi) $(\exists x)(x^3 + 8 = 0)$

(vii) $(\exists x)(x^2 - 2 = 0)$

When working with multiple quantifiers the order of quantification can matter a great deal, as in the following formulas.

$$\varphi = (\forall x)(\exists y)(x \cdot x = y)$$

$$\psi = (\exists y)(\forall x)(x \cdot x = y)$$

φ says "every number has a square" and is true in our typical domains. ψ says "there is a number which is all other numbers' square" and is true only if your domain contains only 0 or only 1.

Exercise 2.1.7. Over the real numbers, which of the following statements are true? Over the natural numbers?

(i) $(\forall x)(\exists y)(x + y = 0)$

(ii) $(\exists y)(\forall x)(x + y = 0)$

(iii) $(\forall x)(\exists y)(x \leq y)$

(iv) $(\exists y)(\forall x)(x \leq y)$

(v) $(\exists x)(\forall y)(x < y^2)$

(vi) $(\forall y)(\exists x)(x < y^2)$

(vii) $(\forall x)(\exists y)(x \neq y \to x < y)$

(viii) $(\exists y)(\forall x)(x \neq y \to x < y)$

The order of operations when combining quantification with conjunction or disjunction can also make the difference between truth and falsehood.

Exercise 2.1.8. Over the real numbers, which of the following statements are true? Over the natural numbers?

(i) $(\forall x)(x \geq 0 \vee x \leq 0)$

(ii) $[(\forall x)(x \geq 0)] \vee [(\forall x)(x \leq 0)]$

(iii) $(\exists x)(x \leq 0 \ \& \ x \geq 5)$

(iv) $[(\exists x)(x \leq 0)] \ \& \ [(\exists x)(x \geq 5)]$

How does negation work for quantifiers? If $\exists x\varphi(x)$ fails, it means no matter what value we fill in for x the formula obtained is false; i.e., $\neg(\exists x\varphi(x)) \leftrightarrow \forall x(\neg\varphi(x))$. Likewise, $\neg(\forall x\varphi(x)) \leftrightarrow \exists x(\neg\varphi(x))$: if φ does not hold for all values of x, there must be an example for which it fails. If we have multiple quantifiers, the negation walks in one by one, flipping each quantifier and finally negating the predicate inside. For example:

$$\neg[(\exists x)(\forall y)(\forall z)(\exists w)\varphi(x,y,z,w)] \leftrightarrow (\forall x)(\exists y)(\exists z)(\forall w)(\neg\varphi(x,y,z,w)).$$

Exercise 2.1.9. Negate the following sentences.

(i) $(\forall x)(\exists y)(\forall z)((z < y) \rightarrow (z < x))$

(ii) $(\exists x)(\forall y)(\exists z)(xz = y)$

(iii) $(\forall x)(\forall y)(\forall z)(y = x \ \vee \ z = x \ \vee \ y = z)$
[Bonus: over what domains of quantification would this be true?]

A final notational comment: you will sometimes see the symbols $\exists^\infty$ and $\forall^\infty$. The former means "there exist infinitely many;" $\exists^\infty x\varphi(x)$ is shorthand for $\forall y\exists x(x > y \ \& \ \varphi(x))$ (no matter how far up we go, there are still examples of φ above us). The latter means "for all but finitely many;" $\forall^\infty x\varphi(x)$ is shorthand for $\exists y\forall x((x > y) \rightarrow \varphi(x))$ (we can get high enough up to bypass all the failed cases of φ). Somewhat common in predicate logic but less so in computability theory is $\exists!x$, which means "there exists a unique x." The sentence $(\exists!x)\varphi(x)$ expands into $(\exists x)(\forall y)(\varphi(x) \ \& \ (\varphi(y) \rightarrow (x = y)))$.

2.2. Sets

A *set* is a collection of objects. If x is a member, or *element*, of a set A, we write $x \in A$, and otherwise $x \notin A$. Two sets are *equal* if they have the same elements; if they have no elements in common they are called *disjoint*. The set A is a *subset* of a set B if all of the elements of A are also elements of B; this is denoted $A \subseteq B$. If we know that A is not equal to B, we may write $A \subset B$ or (to emphasize the non-equality) $A \subsetneq B$. The collection of all subsets of A is denoted $\mathcal{P}(A)$ and called the *power set* of A.

We may write a set using an explicit list of its elements, such as $\{\text{red, blue, green}\}$ or $\{5, 10, 15, \ldots\}$. When writing down sets, order does not matter and repetitions do not count, so $\{1, 2, 3\}$, $\{2, 3, 1\}$, and $\{1, 1, 2, 2, 3, 3\}$ are all representations of the same set. We may also use notation that may be familiar to you from calculus:

$$A = \{x : (\exists y)(y^2 = x)\}.$$

This is the set of all values we can fill in for x that make the logical predicate $(\exists y)(y^2 = x)$ true. The syntax of set definitions is often looser than in logical formulas generally, using commas to mean "and" and condensing clauses. For example, $\{(x, y) : x, y^2 \in \mathbb{Q}, x < y\}$ abbreviates $\{(x, y) : x \in \mathbb{Q} \;\&\; y^2 \in \mathbb{Q} \;\&\; x < y\}$.

We are always working within some fixed *universe*, a set which contains all of our sets. The domain of quantification is all elements of the universe, and hence the contents of the set A above will vary depending on what our universe is. If we are living in the integers it is the set of perfect squares; if we are living in the real numbers it is the set of all non-negative numbers.

Given two sets, we may obtain a third from them in several ways. First there is *union*: $A \cup B$ is the set containing all elements that appear in at least one of A and B. Next *intersection*: $A \cap B$ is the set containing all elements that appear in both A and B. We can subtract: $A - B$ contains all elements of A that are *not* also elements of B. You will often see $A \setminus B$ for set subtraction, but we will use ordinary minus because the slanted minus is sometimes given a different meaning in computability theory. Finally, we can take their *Cartesian product*: $A \times B$ consists of all ordered pairs that have their first entry an element of A and their second an element of B. We may take the product of more than two sets to get ordered triples, quadruples, quintuples, and in general *n-tuples*. If we take the Cartesian product of n copies of A, we may abbreviate $A \times A \times \ldots \times A$ as A^n. A generic ordered n-tuple from A^n will be written $(x_1, x_2, \ldots, x_n)$, where x_i are all elements of A.

Example 2.2.1. Let $A = \{x, y\}$ and $B = \{y, z\}$. Then $A \cup B = \{x, y, z\}$, $A \cap B = \{y\}$, $A - B = \{x\}$, $B - A = \{z\}$, $A \times B = \{(x, y), (x, z), (y, y), (y, z)\}$, and $\mathcal{P}(A) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$.

The sets we will use especially are $\emptyset$ and $\mathbb{N}$. The former is the *empty set*, the set with no elements. The latter is the *natural numbers*, the set $\{0, 1, 2, 3, \ldots\}$. In computability, we often use lowercase omega, ω, to denote the natural numbers, but in these notes we will be consistent with $\mathbb{N}$. On occasion we may also refer to $\mathbb{Z}$ (the integers), $\mathbb{Q}$ (the rational numbers), or $\mathbb{R}$ (the real numbers).

We will assume unless otherwise specified that our universe is $\mathbb{N}$; i.e., all of our sets are subsets of $\mathbb{N}$. When a universe is fixed we can define *complement*. The complement of A, denoted $\overline{A}$, is all the elements of $\mathbb{N}$ that are not in A; i.e., $\overline{A} = \mathbb{N} - A$.

Exercise 2.2.2. Convert the list or description of each of the following sets into notation using a logical predicate. Assume the domain of quantification is $\mathbb{N}$.

(i) $\{2, 4, 6, 8, 10, \ldots\}$

(ii) $\{4, 5, 6, 7, 8\}$

(iii) The set of numbers that are cubes.

(iv) The set of pairs of numbers such that one is twice the other (in either order).

(v) The intersection of the set of square numbers and the set of numbers that are divisible by 3.

(vi) [For this and the next two, you'll need to use $\in$ in your logical predicate.] $A \cup B$ for sets A and B.

(vii) $A \cap B$ for sets A and B.

(viii) $A - B$ for sets A and B.

Exercise 2.2.3. For each of the following sets, list (a) the elements of X, and (b) the elements of $\mathcal{P}(X)$.

(i) $X = \{1, 2\}$

(ii) $X = \{1, 2, \{1, 2\}\}$

(iii) $X = \{1, 2, \{1, 3\}\}$

Exercise 2.2.4. Work inside the finite universe $\{1, 2, \ldots, 10\}$. Define the following sets:

$$A = \{1, 3, 5, 7, 9\}$$
$$B = \{1, 2, 3, 4, 5\}$$
$$C = \{2, 4, 6, 8, 10\}$$
$$D = \{7, 9\}$$
$$E = \{4, 5, 6, 7\}$$

(i) Find all the subset relationships between pairs of the sets above.

(ii) Which pairs, if any, are disjoint?

(iii) Which pairs, if any, are complements?

(iv) Find the following unions and intersections: $A \cup B$, $A \cup D$, $B \cap D$, $B \cap E$.

We can also take unions and intersections of infinitely many sets. For sets A_i for $i \in \mathbb{N}$, these are defined as follows.

$$\bigcup_i A_i = \{x : (\exists i)(x \in A_i)\}$$

$$\bigcap_i A_i = \{x : (\forall i)(x \in A_i)\}$$

The i under the union or intersection symbol is also sometimes written "$i \in \mathbb{N}$."

Exercise 2.2.5. For $i \in \mathbb{N}$, let $A_i = \{0, 1, \ldots, i\}$ and let $B_i = \{0, i\}$. What are $\bigcup_i A_i$, $\bigcup_i B_i$, $\bigcap_i A_i$, and $\bigcap_i B_i$?

If two sets are given by descriptions instead of explicit lists, we must prove one set is a subset of another by taking an arbitrary element of the first set and showing it is also a member of the second set. For example, to show the set of people eligible for President of the United States is a subset of the set of people over 30, we might say: Consider a person in the first set. That person must meet the criteria listed in the U.S. Constitution, which includes being at least

35 years of age. Since 35 is more than 30, the person we chose is a member of the second set.

We can further show that this containment is proper, by demonstrating a member of the second set who is not a member of the first set. For example, a 40-year-old who is not a U.S. citizen.

Exercise 2.2.6. Prove that the set of squares of even numbers, $\{x : \exists y(x = (2y)^2)\}$, is a proper subset of the set of multiples of 4, $\{x : \exists y(x = 4y)\}$.

To prove two sets are equal, there are three options: show the criteria for membership on each side are the same, manipulate set operations until the expressions are the same, or show each side is a subset of the other side.

An extremely basic example of the first option is showing $\{x : \frac{x}{2}, \frac{x}{4} \in \mathbb{N}\} = \{x : (\exists y)(x = 4y)\}$. For the second, we have a bunch of *set identities*, including a set version of De Morgan's Laws.

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

We also have distribution laws.

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

To prove identities we have to turn to the first or third option.

Example 2.2.7. Prove that $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.

We work by showing each set is a subset of the other. Suppose first that $x \in A \cup (B \cap C)$. By definition of union, x must be in A or in $B \cap C$. If $x \in A$, then x is in both $A \cup B$ and $A \cup C$, and hence in their intersection. On the other hand, if $x \in B \cap C$, then x is in both B and C, and hence again in both $A \cup B$ and $A \cup C$.

Now suppose $x \in (A \cup B) \cap (A \cup C)$. Then x is in both unions, $A \cup B$ and $A \cup C$. If $x \in A$, then $x \in A \cup (B \cap C)$. If, however, $x \notin A$, then x must be in both B and C, and therefore in $B \cap C$. Again, we obtain $x \in A \cup (B \cap C)$.

Notice that in the $\subseteq$ direction we used two cases that could overlap, and did not worry whether we were in the overlap or not. In the $\supseteq$ direction, we could only assert $x \in B$ *and* $x \in C$ if we knew $x \notin A$ (although it is certainly possible for x to be in all three sets), so forbidding the first case was part of the second case.

Exercise 2.2.8. Using any of the three options listed above, as long as it is applicable, do the following.

(i) Prove intersection distributes over union (i.e., for all A, B, C, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$).

(ii) Prove de Morgan's Laws.

(iii) Prove that $A \cup B = (A - B) \cup (B - A) \cup (A \cap B)$ for any sets A and B.

Our final topic in the realm of sets is *cardinality.* The cardinality of a finite set is the number of elements in it. For example, the cardinality of the set of positive integer divisors of 6 is 4: $|\{1, 2, 3, 6\}| = 4$. When we get to infinite sets, cardinality separates them by "how infinite" they are. We'll get to its genuine definition in §2.3, but it is fine now and later to think of cardinality as a synonym for size. The way to tell whether set A is bigger than set B is to look for a one-to-one function from A into B. If no such function exists, then A is bigger than B, and we write $|B| < |A|$. The most important result is that $|A| < |\mathcal{P}(A)|$ for *any* set A (see §A.3).

If we know there is a one-to-one function from A into B but we don't know about the reverse direction, we write $|A| \leq |B|$. If we have injections both ways, $|A| = |B|$. It is a significant theorem of set theory that having injections from A to B and from B to A is equivalent to having a bijection between A and B; the fact that this requires work is a demonstration of the fact that things get weird when you work in the infinite world. Another key fact (for set theorists; not so much for us) is *trichotomy*: for any two sets A and B, exactly one of $|A| < |B|$, $|A| > |B|$, or $|A| = |B|$ is true.

For us, infinite cardinalities are divided into two categories. A set is *countably infinite* if it has the same cardinality as the natural numbers. The integers and the rational numbers are important examples of countably infinite sets. The term *countable* is used by some authors to mean "countably infinite," and by others to mean "finite or countably infinite," so you often have to rely on context. We will mean the latter, but will try to be explicit. To prove that a set is countably infinite, you must demonstrate it is in bijection with the natural numbers – that is, that you can count the objects of your set 1, 2, 3, 4, . . . , and not miss any. We'll come back to this in §3.4; for now you can look in the appendices to find Cantor's proofs that the rationals are countable and the reals are not (§A.3).

The rest of the infinite cardinalities are called *uncountable*, and for our purposes that's as fine-grained as it gets. The fundamental notions of computability theory live in the world of countable sets, and the only uncountable ones we get to are those which can be approximated in the countable world.

2.3. Relations

The following definition is not the most general case, but we'll start with it.

Definition 2.3.1. A *relation* $R(x, y)$ on a set A is a logical predicate that is true or false of each pair $(x, y) \in A^2$, never undefined.

We also think of relations as subsets of A^2 consisting of the pairs for which the relation is true. For example, in the set $A = \{1, 2, 3\}$, the relation $<$ consists of $\{(1, 2), (1, 3), (2, 3)\}$ and the relation $\leq$ is the union of $<$ with $\{(1, 1), (2, 2), (3, 3)\}$. Note that the order matters: although $1 < 2$, $2 \not< 1$, so $(2, 1)$ is not in $<$. The first definition shows you why these are called *relations*; we think of R as being true when the values filled in for x and y have some relationship to each other. The set-theoretic definition is generally more useful, however.

More generally, we may define *n-ary* relations on a set A as logical predicates that are true or false of any n-tuple (ordered set of n elements) of A, or alternatively as subsets of A^n. For $n = 1, 2, 3$ we refer to these relations as *unary*, *binary*, and *ternary*, respectively.

Exercise 2.3.2. Prove the two definitions of relation are equivalent. That is, prove that every logical predicate corresponds to a unique set, and vice-versa.

Exercise 2.3.3. Let $A = \{a, b, c, d, e\}$.

(i) What is the ternary relation R on A defined by $(x, y, z) \in R \Leftrightarrow$ (xyz is an English word)?

(ii) What is the unary relation on A which is true of elements of A that are vowels?

(iii) What is the complement of the relation in (ii)? We may describe it in two ways: as "the negation of the relation in (ii)," and how?

(iv) Define the 5-ary relation R by $(v, w, x, y, z) \in R \Leftrightarrow (v, w, x, y, z$ are all *distinct* elements of $A)$. How many elements does R contain?

(v) How many unary relations are possible on A? What other collection associated with A does the collection of all unary relations correspond to?

Exercise 2.3.4. How many n-ary relations are possible on an m-element set?

We tend to focus on binary relations, since most of our common, useful examples are binary: $<, \leq, =, \neq, \subset, \subseteq$. Binary relations may have certain properties:

- Reflexivity: $(\forall x)R(x, x)$
- Symmetry: $(\forall x, y)[R(x, y) \rightarrow R(y, x)]$
  i.e., $(\forall x, y)[(R(x, y) \ \& \ R(y, x)) \ \vee \ (\neg R(x, y) \ \& \ \neg R(y, x))]$
- Antisymmetry: $(\forall x, y)[(R(x, y) \ \& \ R(y, x)) \rightarrow x = y]$
- Transitivity: $(\forall x, y, z)[(R(x, y) \ \& \ R(y, z)) \rightarrow R(x, z)]$

I want to point out that reflexivity is a property of possession: R must have the reflexive pairs (the pairs (x, x)). Antisymmetry is, loosely, a property of nonpossession. Symmetry and transitivity, on the other hand, are *closure* properties: **if** R has certain pairs, **then** it must also have other pairs. Those conditions may be met either by adding in the pairs that are consequences of the pairs already present, or omitting the pairs that are requiring such additions. In particular, the empty relation is symmetric and transitive, though it is not reflexive.

Exercise 2.3.5. Is $=$ reflexive? Symmetric? Antisymmetric? Transitive? How about $\neq$?

Exercise 2.3.6. For finite relations we may check these properties by hand. Let $A = \{1, 2, 3, 4\}$.

(a) What is the smallest binary relation on A that is reflexive?

(b) Define the following binary relations on A.

$$R_1 = \{(2,3), (3,4), (4,2)\}$$

$$R_2 = \{(1,1), (1,2), (2,1), (2,2)\}$$

$$R_3 = \{(1,1), (1,2), (2,2), (2,3), (3,3), (3,4), (4,4)\}$$

For each of those relations, answer the following questions.

(i) Is the relation reflexive? Symmetric? Antisymmetric? Transitive?

(ii) If the relation is not reflexive, what is the smallest collection of pairs that needs to be added to make it reflexive?

(iii) If the relation is not symmetric, what is the smallest collection of pairs that needs to be added to make it symmetric?

(iv) If the relation is not transitive, what is the smallest collection of pairs that needs to be added to make it transitive?

(v) If the relation is not antisymmetric, what is the smallest collection of pairs that could be removed to make it antisymmetric? Is this answer unique?

Exercise 2.3.7. Let $A = \{1, 2, 3\}$. Define binary relations on A with the following combinations of properties or say why such a relation cannot exist. Can such a relation be nonempty?

(i) Reflexive and antisymmetric but neither symmetric nor transitive.

(ii) Symmetric but neither reflexive nor transitive.

(iii) Transitive but neither reflexive nor symmetric.

(iv) Symmetric and transitive but not reflexive.

(v) Both symmetric and antisymmetric.

(vi) Neither symmetric nor antisymmetric.

(vii) Reflexive and transitive but not symmetric.

(viii) Reflexive and symmetric but not transitive.

(ix) Symmetric, antisymmetric, and transitive.

(x) Reflexive, symmetric, and transitive.

(xi) None of reflexive, symmetric, or transitive.

Exercise 2.3.8. Suppose R and S are binary relations on A. For each of the following properties, if R and S possess the property, must $R \cup S$ possess it? $R \cap S$?

(i) Reflexivity

(ii) Symmetry

(iii) Antisymmetry

(iv) Transitivity

Exercise 2.3.9. Each of the following relations has a simpler description than the one given. Find such a description.

(i) R_- on $\mathcal{P}(\mathbb{N})$ where $R_-(A, B) \leftrightarrow A - B = \emptyset$.

(ii) $R_{(\cap)}$ on $\mathbb{R}$ where $R_{(\cap)}(x, y) \leftrightarrow (-\infty, x) \cap (y, \infty) = \emptyset$.

(iii) $R_{[\cap]}$ on $\mathbb{R}$ where $R_{[\cap]}(x, y) \leftrightarrow (-\infty, x] \cap [y, \infty) = \emptyset$.

(iv) $R_{(\cup)}$ on $\mathbb{R}$ where $R_{(\cup)}(x, y) \leftrightarrow (-\infty, x) \cup (y, \infty) = \mathbb{R}$.

(v) $R_{[\cup]}$ on $\mathbb{R}$ where $R_{[\cup]}(x, y) \leftrightarrow (-\infty, x] \cup [y, \infty) = \mathbb{R}$.

We may visualize a binary relation R on A as a directed graph. The elements of A are the vertices, or nodes, of the graph, and there is an arrow (directed edge) from vertex x to vertex y if and only if $R(x, y)$ holds. The four properties we have just been exploring may be stated as:

- Reflexivity: every vertex has a loop.
- Symmetry: any pair of vertices is either directly connected in both directions or not directly connected at all.
- Antisymmetry: any two vertices have at most one edge directly connecting them.
- Transitivity: if there is a path of edges from one vertex to another (always proceeding in the direction of the edge), there is an edge directly connecting them, in the same direction as the path.

Exercise 2.3.10. Properly speaking, transitivity just gives the graphical interpretation "for any vertices x, y, z, if there is an edge from x to y and an edge from y to z, there is an edge from x to z." Prove that this statement is equivalent to the (a priori more general) one given for transitivity above.

We will consider two subsets of these properties that define classes of relations which are of particular importance.

Definition 2.3.11. An *equivalence relation* is a binary relation that is reflexive, symmetric, and transitive.

The quintessential equivalence relation is equality, which is the relation consisting of only the reflexive pairs. What is special about an equivalence relation? We can take a *quotient structure* whose elements are *equivalence classes.*

Definition 2.3.12. Let R be an equivalence relation on A. The *equivalence class* of some $x \in A$ is the set $[x] = \{y \in A : R(x, y)\}$.

Exercise 2.3.13. Let R be an equivalence relation on A and let x, y be elements of A. Prove that either $[x] = [y]$ or $[x] \cap [y] = \emptyset$.

In short, an equivalence relation puts all the elements of the set into boxes so that each element is unambiguously assigned to a single box. All possible pairings from within each box are in the relation, and no pairings that draw from different boxes are in the relation. We can consider the boxes themselves as elements, getting a quotient structure.

Definition 2.3.14. Given a set A and an equivalence relation R on A, the *quotient of A by R*, A/R, is the set whose elements are the equivalence classes of A under R.

Now we can define cardinality more correctly. The cardinality of a set is the equivalence class it belongs to under the equivalence relation of bijectivity, so cardinalities are elements of the quotient of the collection of all sets under that relation.

Exercise 2.3.15. Let A be the set $\{1, 2, 3, 4, 5\}$, and let R be the binary relation on A that consists of the reflexive pairs together with $(1, 2)$, $(2, 1)$, $(3, 4)$, $(3, 5)$, $(4, 3)$, $(4, 5)$, $(5, 3)$, $(5, 4)$.

(i) Represent R as a graph.

(ii) How many elements does A/R have?

(iii) Write out the sets [1], [2], and [3].

Exercise 2.3.16. A *partition* of a set A is a collection of disjoint subsets of A with union equal to A. Prove that any partition of A determines an equivalence relation on A, and every equivalence relation on A determines a partition of A.

Exercise 2.3.17. Let $R(m, n)$ be the relation on $\mathbb{Z}$ that holds when $m - n$ is a multiple of 3.

(i) Prove that R is an equivalence relation.

(ii) What are the equivalence classes of 1, 2, and 3?

(iii) What are the equivalence classes of -1, -2, and -3?

(iv) Prove that $\mathbb{Z}/R$ has three elements.

Exercise 2.3.18. Let $R(m, n)$ be the relation on $\mathbb{N}$ that holds when $m - n$ is even.

(i) Prove that R is an equivalence relation.

(ii) What are the equivalence classes of R? Give a concise verbal description of each.

The two exercises above are examples of *modular arithmetic*, which is also sometimes called *clock-face arithmetic* because its most widespread use in day-to-day life is telling what time it will be some hours from now. This is a notion that is used only in $\mathbb{N}$ and $\mathbb{Z}$. The idea of modular arithmetic is that it is only the number's remainder upon division by a fixed value that matters. For clock-face arithmetic, that value is 12; we say we are working *modulo 12*, or just *mod 12*, and the equivalence classes are represented by the numbers 0 through 11 (in mathematics; 1 through 12 in usual life). The fact that if it is currently 7:00 then in eight hours it will be 3:00 would be written as the equation $7 + 8 = 3 \pmod{12}$, where $\equiv$ is sometimes used in place of the equals sign.

Exercise 2.3.19. (i) Exercises 2.3.17 and 2.3.18 consider equivalence relations that give rise to arithmetic mod k for some k. For each, what is the correct value of k?

(ii) Describe the equivalence relation on $\mathbb{Z}$ that gives rise to arithmetic mod 12.

(iii) Let m, n, and p be integers. Prove that

$$n = m \pmod{12} \implies n + p = m + p \pmod{12}.$$

This shows that addition of equivalence classes via representatives is well-defined.

Our second important class of relations is partial orders.

Definition 2.3.20. A *partial order* $\leq$ on a set A is a binary relation that is reflexive, antisymmetric, and transitive. A with $\leq$ is called a *partially ordered set*, or *poset*.

In a poset, given two nonequal elements of A, either one is strictly greater than the other or they are incomparable. If all pairs of elements are comparable, the relation is called a *total order* or *linear order* on A.

Example 2.3.21. Let $A = \{a, b, c, d, e\}$ and define $\leq$ on A as follows:

- $(\forall x \in A)(x \leq x)$
- $a \leq c$, $a \leq d$
- $b \leq d$, $b \leq e$

We could graph this as follows:

$$\begin{tikzcd}
c & & d & & e \\
& a \arrow[ul, dash] \arrow[ur, dash] & & b \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

Technically, there are arrowheads pointing up and each element has a loop, but for partial orders we often assume that.

In Example 2.3.21, the elements c and d have no *upper bound*: no single element that is greater than or equal to both of them. We might add elements f and g, augmenting the relation to include $c \leq f$, $d \leq f$, $d \leq g$, and $e \leq g$ (making the diagram vertically symmetric), plus the necessary pairs to obtain transitivity (e.g., $a \leq f$; this is called taking the *transitive closure*). Then f is an upper bound for c

and d, and it is the least upper bound: any element that is greater than or equal to both c and d is also greater than or equal to f. The elements f and g are upper bounds for the pair a and b, but not the least upper bound; that is d. A *lower bound* for a pair of elements is defined symmetrically: an element that is less than or equal to both of the given elements. The greatest lower bound is a lower bound x such that any other lower bound is less than or equal to x. Note that even when lower and upper bounds exist, there need not be greatest lower or least upper bounds. This can happen in two ways: two incomparable bounds (think of a V, where the lower point is below both upper points, but the upper points are incomparable), or an infinite ascending or descending sequence of elements that limits to an element not in the poset (such as an open interval in the rational numbers with irrational endpoints).

Example 2.3.22. $\mathcal{P}(\mathbb{N})$ ordered by subset inclusion is a partially ordered set.

It is easy to check the relation $\subseteq$ is reflexive, transitive, and antisymmetric. Not every pair of elements is comparable: for example, neither $\{1, 2, 3\}$ nor $\{4, 5, 6\}$ is a subset of the other. This poset actually has some very nice properties that not every poset has: it has a top element ($\mathbb{N}$) and a bottom element ($\emptyset$), and every pair of elements has both a least upper bound (here, the union) and a greatest lower bound (the intersection).

If we were to graph this, it would look like an infinitely-faceted diamond with points at the top and bottom.

Example 2.3.23. Along the same lines as Example 2.3.22, we can consider the power set of a finite set, and then we can graph the poset that results.

Let $A = \{a, b, c\}$. Denote the set $\{a\}$ by a and the set $\{b, c\}$ by $\hat{a}$, and likewise for the other three elements. The graph is the transitive closure of the graph shown in Figure 2.1.

Exercise 2.3.24. Prove that every pair of elements in a poset has at most one greatest lower bound and at most one least upper bound.

Exercise 2.3.25. How many partial orders are possible on a set of two elements? Three elements?

$$\begin{tikzcd}
& A \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\hat{c} \arrow[d, dash] \arrow[dr, dash] & \hat{b} \arrow[dl, dash] \arrow[dr, dash] & \hat{a} \arrow[dl, dash] \arrow[d, dash] \\
a \arrow[dr, dash] & b \arrow[d, dash] & c \arrow[dl, dash] \\
& \emptyset &
\end{tikzcd}$$

Figure 2.1. Subsets of $\{a, b, c\}$.

Our final note is to point out relations generalize functions. The function $f : A \to A$ may be written as a binary relation on A consisting of the pairs $(x, f(x))$. A binary relation R, conversely, represents a function whenever $[(x, y) \in R \mathbin{\&} (x, z) \in R] \to y = z$ (the vertical line rule for functions).[1] We can ramp this up even further to multi-variable functions, functions from A^n to A, by considering $(n+1)$-ary relations. The first n places represent the input and the last one the output. The advantage to this is consolidation; we can prove many things about functions by proving them for relations in general.

2.4. Bijection and Isomorphism

As mentioned, a function $f : A \to B$ may be thought of as a relation $G_f \subseteq A \times B$, the Cartesian product of its domain (also denoted $\operatorname{dom} f$) and codomain, though the property that every element of A appear in exactly one pair of G_f is not particularly natural to state in terms of relations. G_f is called the *graph* of f. Before we proceed recall also that the range of f, $\operatorname{rng} f$, is $\{f(x) : x \in A\} \subseteq B$.

When are two sets "the same?" How about two partial orders? The notion of *isomorphism* is, loosely, the idea that two mathematical structures may be essentially the same, even if cosmetically different. Recall that a function between two sets is *injective*, or *one-to-one*, if

[1]You might object that this does not require every element of A be in the domain of the function. We will not be concerned by that; see §3.1.2.

no two domain elements map to the same range element. It is *surjective*, or *onto*, if the codomain equals the range. Since no function can take a single domain element to multiple range elements, and by definition it must give an image (output) to every domain element (well... see §3.1.2), a function that is both one-to-one and onto uniquely pairs off the elements of the domain and codomain. Such functions are called *bijections.* Some authors will refer to a bijection as a "one-to-one correspondence;" they are also called *permutations.*

For sets, bijections are everything. It does not matter whether we use the letters A through Z or the numbers 1 through 26; a set of the same size is essentially the same set. Two sets (without structure) are *isomorphic* if there is a bijection between them. Two sets with additional structure may be bijective without being isomorphic.

When there are relations (including functions) on the elements, to say two structures are isomorphic we need a special kind of bijection: one that preserves the relations. For example, an isomorphism between two partial orders P and Q is a bijection $f : P \to Q$ such that for all $x, y \in P$, $x \leq_P y \leftrightarrow f(x) \leq_Q f(y)$. As a consequence, any logical statement about the relation will hold in one structure if and only if it holds in the other, such as $\forall x \exists y (x \leq y)$. Isomorphism must be defined separately for each kind of object, but it follows the same template each time: a bijection that preserves the structure.

Example 2.4.1. Let $A = \{\{1\}, \{1,2\}, \{1,2,3\}, \{1,2,3,4\}\}$ and $B = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$. A and B are in bijection, or *isomorphic as sets.* However, they are not isomorphic as partial orders, under the relation of subset inclusion. A is a linear order and B a diamond; A satisfies the statement $\forall x\ \forall y\ (x \subseteq y\ \vee\ y \subseteq x)$ and B satisfies its negation, $\exists x\ \exists y\ (\neg(x \subseteq y) \& \neg(y \subseteq x))$.

An isomorphism between a structure and itself is called an *automorphism.*

2.5. Recursion and Induction

Recursive definitions and proofs by induction are opposite sides of the same coin. Both have some specific starting point, and then a way to extend from there via a small set of operations. For induction, you might be proving some property P is true of all the natural numbers. To do so, you prove that P is true of 0, and then prove that *if* P is true of some $n \geq 0$, *then* P is also true of $n+1$. To recursively define a class of objects C, you give certain simple examples of objects in C, and then operations that combine or extend elements of C to give results still in C. They relate more deeply than just appearance, though. We'll tackle induction, then recursion, then induction again.

2.5.1. Induction on $\mathbb{N}$. The basic premise of induction is that if you can start, and once you start you know how to keep going, then you will get all the way to the end. If I can get on the ladder, and I know how to get from one rung to the next, I can get to the top of the ladder.

Definition 2.5.1. The principle of mathematical induction, basic form, says the following.

If S is a subset of the positive integers containing 1 such that $n \in S$ implies $n+1 \in S$ for all n, then S contains all of the positive integers. [We may need the beginning to be 0 or another value depending on context.]

In general you want to use induction to show that some property holds no matter what integer you feed it, or no matter what size finite set you are dealing with. The proofs always have a *base case*, the case of 1 (or wherever you're starting). Then they have the *inductive step*, the point where you assume the property holds for some unspecified n and then show it holds for $n+1$.

Example 2.5.2. Prove that for every positive integer n,

$$1+3+5+\ldots+(2n-1)=n^2.$$

Proof. Base case: For $n=1$, the equation is $1=1^2$, which is true. Inductive step: Assume that $1+3+5+\ldots+(2n-1)=n^2$ for some $n \geq 1$. To show that it holds for $n+1$, add $2(n+1)-1$ to each side, in the simplified form $2n+1$.

$$1+3+5+\ldots+(2n-1)+(2n+1)=n^2+2n+1=(n+1)^2$$

Since the equation above is that of the theorem, for $n+1$, by induction the equation holds for all n. $\square$

The format of the proof above is typical of inductive proofs of summation formulas: use the inductive hypothesis to simplify a portion of the next value's sum.

For the next example we need to know a *convex* polygon is one where all the corners point out. If you connect two corners of a convex polygon with a straight line segment, the segment will lie entirely within the polygon, cutting it into two smaller convex polygons.

As you get more comfortable with induction, you can write it in a more natural way, without segmenting off the base case and inductive step portions of the argument. We'll do that here. Notice the base case is not 0 or 1 for this proof.

Example 2.5.3. For $n > 2$, the sum of angle measures of the interior angles of a convex polygon of n vertices is $(n-2)\cdot 180^\circ$.

Proof. We work by induction. For $n = 3$, the polygon in question is a triangle, and it has interior angles which sum to $180^\circ = (3-2)\cdot 180^\circ$.

Assume the theorem holds for some $n \geq 3$ and consider a convex polygon with $n+1$ vertices. Let one of the vertices be named x, and pick a vertex y such that along the perimeter from x in one direction there is a single vertex between x and y, and in the opposite direction, $(n+1)-3 = n-2$ vertices. Join x and y by a new edge, dividing the original polygon into two polygons. The new polygons' interior angles together sum to the sum of the original polygon's interior angles. One of the new polygons has 3 vertices and the other has n vertices (x, y, and the $n-2$ vertices between them). The triangle has interior angle sum 180°, and by the inductive hypothesis the n-gon has interior angle sum $(n-2)\cdot 180^\circ$. The $n+1$-gon therefore has interior angle sum $180^\circ + (n-2)180^\circ = (n+1-2)\cdot 180^\circ$, as desired. □

Notice also in this example that we used the base case as part of the inductive step, since one of the two polygons was a triangle. This is not uncommon.

Exercise 2.5.4. Prove the following statements by induction.

(i) For every positive integer n,

$$1 + 4 + 7 + \ldots + (3n-2) = \frac{1}{2}n(3n-1).$$

(ii) For every positive integer n,

$$2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 2.$$

(iii) For every positive integer n, $\dfrac{n^3}{3} + \dfrac{n^5}{5} + \dfrac{7n}{15}$ is an integer.

(iv) For every positive integer n, $4^n - 1$ is divisible by 3.

(v) The sequence $a_0, a_1, a_2, \ldots$ defined by $a_0 = 0$, $a_{n+1} = \frac{a_n+1}{2}$ is bounded above by 1.

Exercise 2.5.5. Recall that, for a binary operation $*$ on a set A, associativity is defined as "for any x, y, z, $(x*y)*z = x*(y*z)$." Use induction to prove that for any collection of n elements from A put together with $*$, $n \geq 3$, any grouping of the elements that preserves order will give the same result.

Exercise 2.5.6. A *graph* consists of *vertices* and *edges*. Each edge has a vertex at each end (they may be the same vertex). Each vertex has a *degree*, which is the number of edge endpoints at that vertex (so if an edge connects two distinct vertices, it contributes 1 to each of their degrees, and if it is a loop on one vertex, it contributes 2 to that vertex's degree). It is possible to prove without induction that for a graph the sum of the degrees of the vertices is twice the number of edges. Find a proof of that fact using

(i) induction on the number of vertices;

(ii) induction on the number of edges.

Exercise 2.5.7. The *Towers of Hanoi* is a puzzle consisting of a board with three pegs sticking up out of it and a collection of disks that fit on the pegs, each with a different diameter. The disks are placed on a single peg in order of size (smallest on top), and the goal is to move the entire stack to a different peg. A move consists of removing the top disk from any peg and placing it on another peg; a disk may never be placed on top of a smaller disk.

Determine how many moves it requires to solve the puzzle when there are n disks, and prove your answer by induction.

2.5.2. Recursion. To define a class *recursively* means to define it via a set of basic objects and a set of rules, called *closure operators*, allowing you to extend the set of basic objects. We give some simple examples.

Example 2.5.8. The natural numbers may be defined recursively as follows:

- $0 \in \mathbb{N}$.
- If $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$.

Example 2.5.9. The *well-formed formulas* (wffs) in propositional logic are a recursively defined class.

- Any propositional symbol P, Q, R, etc., is a wff.
- If φ and ψ are wffs, so are the following:
  (i) $(\varphi \mathbin{\&} \psi)$
  (ii) $(\varphi \vee \psi)$
  (iii) $(\varphi \rightarrow \psi)$
  (iv) $(\varphi \leftrightarrow \psi)$
  (v) $(\neg\varphi)$

The important fact, which gives the strength of this method of definition, is that we may apply the closure operators repeatedly to get more and more complicated objects.

For example, $((A\&B) \vee ((P\&Q) \rightarrow (\neg A)))$ is a wff, as we can prove by giving a construction procedure for it. A, B, P, and Q are all basic wffs. We combine them into $(A\&B)$ and $(P\&Q)$ by operation (i), obtain $(\neg A)$ from (v), $((P\&Q) \rightarrow (\neg A))$ from (iii), and finally our original formula by (ii).

Exercise 2.5.10. (i) Prove that $((A \vee (B\&C)) \leftrightarrow C)$ is a wff.

(ii) Prove that $(P \rightarrow Q(\vee$ is *not* a wff.

Exercise 2.5.11. (i) Add a closure operator to the recursive definition of $\mathbb{N}$ to get a recursive definition of $\mathbb{Z}$.

(ii) Add a closure operator to the recursive definition of $\mathbb{Z}$ to get a recursive definition of $\mathbb{Q}$.

Exercise 2.5.12. Give two different recursive definitions of the set of all positive multiples of 5.

Exercise 2.5.13. Write a recursive definition of the rational functions in x, those functions which can be written as a fraction of two polynomials of x. Your basic objects should be x and all real numbers.

We may also define functions recursively. For that, we say what $f(0)$ is (or whatever our basic object is) and then define $f(n+1)$ in terms of $f(n)$. For example, $(n+1)! = (n+1)n!$, with $0! = 1$, is *factorial*, a recursively defined function you've probably seen before. We could write a recursive definition for addition of natural numbers as follows.

$$\begin{aligned} a(0,0) &= 0 \\ a(m+1,n) &= a(m,n)+1 \\ a(m,n+1) &= a(m,n)+1 \end{aligned}$$

This looks lumpy but is actually used in logic in order to minimize the number of operations that we take as fundamental: this definition of addition is all in terms of *successor*, the plus-one function.

Exercise 2.5.14. Write a recursive definition of $p(m,n) = m \cdot n$, on the natural numbers, in terms of addition.

Exercise 2.5.15. Write a recursive definition of $f(n) = 2^n 3^{2n+1}$, on the natural numbers.

Definition 3.3.1 gives a recursively defined set of functions in which one of the closure operators gives a function defined recursively from functions already in the set.

2.5.3. Induction Again.

Induction and recursion have a strong tie beyond just their resemblance. Proving a property holds of all members of a recursively defined class often requires induction. This use of induction is less codified than the induction on $\mathbb{N}$ we saw above. In fact, the induction in §2.5.1 is simply the induction that goes with the recursively defined set of natural numbers, as in Example 2.5.8.

To work generally, the base case of the inductive argument must match the basic objects of the recursive class. The inductive step comes from the closure operations that build up the rest of the class. You are showing the set of objects that have a certain property contains the basic objects of the class and is closed under the operations of the class, and hence must contain the entire class.

Example 2.5.16. Consider the class of wffs, defined in Example 2.5.9. We may prove by induction that for any wff φ, the number of positions where binary connective symbols occur in φ (that is, $\&, \vee, \rightarrow$, and $\leftrightarrow$) is one less than the number of positions where propositional symbols occur in φ.

Proof. For any propositional symbol, the number of propositional symbols is 1 and the number of binary connectives is 0.

Suppose by induction that $p_1 = c_1 + 1$ and $p_2 = c_2 + 1$ for p_1, p_2 the number of propositional symbols and c_1, c_2 the number of binary connectives in the wffs φ, ψ, respectively. The number of propositional symbols in $(\varphi Q \phi)$, for Q any of $\vee, \&, \rightarrow$, and $\leftrightarrow$, is $p_1 + p_2$, and the number of connective symbols is $c_1 + c_2 + 1$. By the inductive hypothesis we see that

$$p_1 + p_2 = c_1 + 1 + c_2 + 1 = (c_1 + c_2 + 1) + 1,$$

so the claim holds for $(\varphi Q \psi)$.

Finally, consider $(\neg\varphi)$. Here the number of binary connectives and propositional symbols has not changed, so the claim holds. $\square$

Exercise 2.5.17. The length of a wff is the number of symbols it contains, including parentheses. Suppose φ is a wff not containing negation (that is, it comes from the class defined as in Example 2.5.9 but without closure operation (v)). Prove by induction that for each such φ, there is some $k \geq 0$ such that the length of φ is $4k+1$ and the number of positions at which propositional symbols occur is $k+1$.

If we are careful, we can perform induction on $\mathbb{N}$ to get results about other recursively defined classes. For wffs, we might induct on the number of propositional symbols or the number of binary connectives, for instance.

Exercise 2.5.18. Recall from calculus that a function f is *continuous at a* if $f(a)$ is defined and equals $\lim_{x\to a} f(x)$. Recall also the *limit laws*, which may be summarized for our purposes as

$$\lim_{x\to a}(f(x)\square g(x)) = (\lim_{x\to a} f(x))\square(\lim_{x\to a} g(x)), \quad \square \in \{+, -, \cdot, /\},$$

as long as both limits on the right are defined and if $\square = /$ then $\lim_{x\to a} g(x) \neq 0$. Using those, the basic limits $\lim_{x\to a} x = a$ and

$\lim_{x\to a} c = c$ for all constants c, and your recursive definition from Exercise 2.5.13, prove that every rational function is continuous on its entire domain.

Exercise 2.5.19. Using the recursive definition of addition from the previous section ($a(0,0) = 0$; $a(m+1, n) = a(m, n+1) = a(m, n)+1$), prove that addition is commutative (i.e., for all m and n, $a(m, n) = a(n, m)$).

2.6. Some Notes on Proofs and Abstraction

2.6.1. Definitions. Definitions in mathematics are somewhat different from definitions in English. In natural language, the definition of a word is determined by the usage and may evolve. For example, "broadcasting" was originally just a way of sowing seed. Someone used it by analogy to mean spreading messages widely, and then it was adopted for radio and TV. For present-day speakers of English I doubt the original meaning is ever the first to come to mind.

In contrast, in mathematics we begin with the definition and assign a term to it as a shorthand. That term then denotes exactly the objects that fulfill the terms of the definition. To say something is "by definition impossible" has a rigorous meaning in mathematics: if it contradicts any of the properties of the definition, it cannot hold of an object to which we apply the term.

Mathematical definitions do not have the fluidity of natural language definitions. Sometimes mathematical terms are used to mean more than one thing, but that is a re-use of the term and not an evolution of the definition. Furthermore, mathematicians dislike that because it leads to ambiguity (exactly what is meant by this term in this context?), which defeats the purpose of mathematical terms in the first place: to serve as shorthand for specific lists of properties.

2.6.2. Proofs. There is no way to learn how to write proofs without actually writing them, but I hope you will refer back to this section from time to time. There are also a number of books available about learning to write proofs and solve mathematical problems that you can turn to for more thorough advice.

A proof is an object of convincing. It should be an explicit, specific, logically sound argument that walks step by step from the hypotheses to the conclusions. Avoid vagueness and leaps of deduction, and strip out irrelevant statements. Be careful to state what you are trying to prove in such a way that it does not appear you are asserting its truth prior to proving it. More broadly, make sure your steps are in the right order. Often, a good way to figure out how to prove something is to work backwards, in steps of "I get this as a conclusion if this other property holds; this property holds whenever the object is of this kind; and oh, everything that meets my hypothesis is an object of that kind!" However, the final proof should be written from hypothesis to kind of object to special property to conclusion.

True mathematical proofs are very verbal, bearing little to no resemblance to the two-column proofs of high school geometry. A proof which is just strings of symbols with only a few words is unlikely to be a good (or even understandable) proof. However, it can also be clumsy and expand proofs out of readability to avoid symbols altogether. For example, it is important for specificity to assign symbolic names to (arbitrary) numbers and other objects to which you will want to refer. Striking the symbol/word balance is a big step on the way to learning to write good proofs.

Make your proof self-contained except for explicit reference to definitions or previous results (i.e., don't assume your reader is so familiar with the theorems that you may use them without comment; instead say "by Theorem 2.5, . . ."). Be clear; sentence fragments and tortured grammar have no place in mathematical proofs. If a sentence seems strained, try rearranging it, possibly involving the neighboring sentences. Do not fear to edit: the goal is a readable proof that does not require too much back-and-forth to understand. There is a place for words like *would*, *could*, *should*, *might*, and *ought* in proofs, but they should be kept to a minimum. Most of the time the appropriate words are *has*, *will*, *does*, and *is*. This is especially important in proofs by contradiction. Since in such a proof you are assuming something that is not true, it may feel more natural to use the subjunctive, but that can make things unclear. You assume some hypothesis; given that hypothesis other statements *are* or *are not* true. Be bold and let the whole contraption go up in flames when it runs into the statement it contradicts.

Your audience is a person who is familiar with the underlying definitions used in the statement being proved, but not the statement itself. For instance, it could be yourself after you learned the definitions, but before you had begun work on the proof. You do not have to put every tiny painful step in the write-up, but be careful about what you assume of the reader's ability to fill in gaps. Your goal is to convince the reader of the truth of the statement, and that requires the reader to understand the proof. Along those lines, it is often helpful to insert small statements (I call it "foreshadowing" or "telegraphing") that let the reader know why you are doing what you are currently doing, and where you intend to go with it. In particular, when working by contradiction or induction, it is important to let the reader know at the beginning. More complicated proofs, the kinds that take several pages to complete, often benefit from an expository section at the beginning, that outlines the proof with a focus on the "why" of each step. A more technical portion that fills in all the details comes afterward.

You must keep to the definitions and other statements as they are written. In fact, a good strategy for finding a proof of a statement is first to unwrap the definitions involved. Be especially wary of mentally adding words like *only*, *for all*, *for every*, or *for some* which are not actually there. If you are asked to prove an implication it is likely the converse does not hold, so if you "prove" equivalence you will be in error. Statements that claim existence of an object satisfying certain hypotheses may be proved by producing an example, but if you are asked to prove something holds of *all* objects of some type, you cannot do so via a specific example. Instead, give a symbolic name to an arbitrary object and prove the property holds using only facts that are true for all objects of the given type. Similarly, the term *without loss of generality*, or WLOG, appears from time to time in proofs to indicate a simplifying but not restrictive assumption. If you use the term make sure the assumption truly does not restrict the cases. For example, one may assume without loss of generality that the coefficient of x in the equation of a plane is nonnegative, since if it is negative another equation for the same plane may be obtained by multiplying through by -1. It is *not* without loss of generality to assume the coefficient of x is 1, since the plane defined by $y + z = 0$, for example, has no equation with an x-coefficient of 1.

Exercise 2.6.1. Here are some proofs you can try that don't involve induction:

(i) $\neg(\forall m)(\forall n)(3m + 5n = 12)$ (over $\mathbb{N}$)

(ii) For any integer n, the number $n^2 + n + 1$ is odd.

(iii) If every even natural number greater than 2 is the sum of two primes, then every odd natural number greater than 5 is the sum of three primes.

(iv) For nonempty sets A and B, $A \times B = B \times A$ if and only if $A = B$.

(v) $\sqrt{2}$ is irrational. (Hint: work by contradiction, assuming there is a fraction of integers in least terms that equals $\sqrt{2}$.)

Chapter 3

Defining Computability

There are many ways we could try to get a handle on the concept of computability. We could think of all possible computer programs, or a class of functions defined in a way that feels more algebraic. Many definitions that seem to come from widely disparate viewpoints actually define the same collection of functions, which gives us some claim to calling that collection the *computable* functions (see §3.6).

3.1. Functions, Sets, and Sequences

We mention three aspects of functions important to computability before beginning.

3.1.1. Limits. Our functions take only whole-number values. Therefore, for $\lim_{n\to\infty} f(n)$ to exist, f must eventually be constant. If it changes values infinitely many times, the limit simply doesn't exist. In computability we typically abbreviate our limit notation, as well. It would be more common to see the limit above written as $\lim_n f(n)$.

3.1.2. Partiality. A function is only fully defined when both the rule associating domain elements with range elements *and* the domain itself are given. However, in calculus, we abuse this to give functions as algebraic formulas that calculate a range element from a domain element, without specifying their domains. By convention, the domain is all elements of $\mathbb{R}$ on which the function is defined. However, we treat these functions as though their domain is actually all of $\mathbb{R}$, and talk about, for example, values at which the function has a hole or vertical asymptote.

Here we take that mentality and make it official. In computability we use *partial functions* on $\mathbb{N}$, functions that take elements of some subset of $\mathbb{N}$ as inputs and produce elements of $\mathbb{N}$ as outputs. When applied to a collection of functions, "partial" means "partial or total," though "the partial function f" may generally be read as saying f's domain is a proper subset of $\mathbb{N}$.

Since our functions are only on the nonnegative integers, a straightforward function could be partial simply because on some inputs it gives a negative, fractional, or irrational output, such as division, subtraction, or square root. However, there is a second possibility that we will see is more problematic: the computational procedure implementing the function may go into an infinite loop on certain inputs and never stop running. We will argue we must work with this possibility in §3.3 and with Theorem 3.5.1. Note that a bijection or isomorphism (§2.4) will still have to be a total function.

If x is in the domain of f, we write $f(x)\downarrow$ and say the computation *halts*, *converges*, or *is defined.* We might specify halting when saying what the output of the function is, $f(x)\downarrow = y$, though there the $\downarrow$ is not necessary. When x is not in the domain of f we say the computation *diverges* or *is undefined* and write $f(x)\uparrow$.

For *total* functions f and g, we say $f = g$ if $(\forall x)(f(x) = g(x))$. When f and g may be partial, we require a little more: $f = g$ means

$$(\forall x)[(f(x)\downarrow \leftrightarrow g(x)\downarrow) \,\&\, (f(x)\downarrow = y \rightarrow g(x) = y)].$$

Some authors write this as $f \simeq g$ to distinguish it from equality for total functions and to highlight the fact that f and g might be partial.

Finally, when the function intended is clear, $f(x) = y$ may be written $x \mapsto y$.

3.1.3. Ones and Zeros. In computability, as in many fields of mathematics, we use certain terms and notation interchangeably even though technically they define different objects, because in some deep sense the objects aren't different at all. They might be referred to as not just isomorphic structures, but *naturally* isomorphic. We begin here with a definition.

Definition 3.1.1. For a set A, the *characteristic function* of A is the following total function.

$$\chi_A(n) = \begin{cases} 1 & n \in A \\ 0 & n \notin A \end{cases}$$

In the literature, χ_A is often represented simply by A, so, for instance, we can say $\varphi_e = A$ to mean $\varphi_e = \chi_A$ as well as saying $A(n)$ to mean $\chi_A(n)$, so $A(n) = 1$ is another way to say $n \in A$. Additionally, we may conflate the function and set with the binary sequence made of the outputs of the function in order of input size, which we will sometimes call the "characteristic sequence" of A.

Example 3.1.2. The sequence 1010101010... can represent

(i) The set of even numbers, $\{0, 2, 4, \dots\}$.

(ii) The function $f(n) = n \mod 2$, which is the characteristic function of the set of even numbers.

Exercise 3.1.3. Construct bijections between (i) and (ii), (ii) and (iii), and (i) and (iii) below, and prove they are bijections.

(i) The set $2^{\mathbb{N}}$, infinite binary sequences.

(ii) The set of total functions from $\mathbb{N}$ to $\{0, 1\}$.

(iii) The power set of $\mathbb{N}$.

Exercise 3.1.4. Construct bijections between (i) and (ii), (ii) and (iii), and (i) and (iii) below, and prove they are bijections.

(i) The set $2^{<\mathbb{N}}$, finite binary strings.

(ii) The set of finite subsets of $\mathbb{N}$.

(iii) $\mathbb{N}$.

3.2. Turing Machines

Our first rigorous definition of computation is due to Turing [85]. He thought about what people do when they calculate on paper, and abstracted those actions: reading symbols, writing symbols, holding some amount of information in their mind, and looking at different parts of the paper. They apply appropriate rules of calculation based on the information in their mind, which is influenced by what they have seen and done so far, and on what they are currently viewing.[1]

A *Turing machine* (TM) is an idealized computer which has a tape it can read from and write on, a head which does that reading and writing and which moves back and forth along the tape, and an internal state which may be changed based on what happens during the computation. It also has a list of computational rules that are applied based on the current internal state and symbol being read. There is a finite alphabet of symbols (e.g., capital and lowercase letters, numbers, and punctuation) and a finite list of internal states. Everything here is discrete: the tape is divided into squares, and the read/write head rests on an individual square and moves from square to square. The tape is a sheet of paper, cut apart into lines and attached end to end; the squares each hold one symbol as though the person is writing neatly in block letters and numbers. We specify Turing machines via quadruples $\langle a, b, c, d\rangle$, sets of instructions that are decoded as follows:

- a is the state the TM is currently in.
- b is the symbol the TM's head is currently reading.
- c is an instruction to the head to write or move.
- d is the state the TM is in at the end of the instruction's execution.

For example, $\langle q_3, 0, R, q_3\rangle$ means "if I am in state q_3 and currently reading a 0, move one square to the right and remain in state q_3." The instruction $\langle q_0, 1, 0, q_1\rangle$ means "if I am in state q_0 and reading a 1, overwrite that 1 with a 0 and change to state q_1." We typically use only numbers and punctuation for the alphabet, so L and R may unambiguously represent instructions to move one square to the

[1]Post [71] independently gave an essentially identical analysis of the computing process. He gives the same abstraction as Turing's tape, motion, and writing, but in place of internal states Post has ordered lists of rules that include instructions to jump from place to place on the list. However, he omitted details of the mathematical implementation and hence cannot be co-credited for Turing's theorems.

left or right, respectively. The symbol in position c may also be a blank ($*$), indicating the machine should erase whatever symbol it is reading. Any fixed a, b begin *at most* one quadruple of a given Turing machine. It is not necessary that there be any instruction at all; the computation may halt because of the absence of applicable instructions.

Any computation that gives an output must use only finitely many squares of the tape and finitely many steps of computation. However, since Turing machines represent idealized computers, we must allow them unlimited time and memory to perform their computations. This is not the same as allowing *infinite* time or memory. We simply can't bound them from the beginning; what if our bound were just one step short of completion or one square of tape too small? Therefore, a TM's tape is infinite, though any given computation uses only a finite length of it.

Since the symbols and states come from a finite list, the collection of instructions will always be finite. It does not matter how long the lists are; generally we stick to the symbols 0, 1, and blank ($*$) – or even just 1 and $*$ – but allow arbitrarily long lists of states, mostly because this is the mode that lends itself best to writing descriptions of machines. Note that some authors distinguish legal halting states from other states, and consider dead-ending in a non-halting state equivalent to entering an infinite loop. They may also require the read/write head to end up on a particular end of the tape contents. This is all to make proofs easier, and it does not reduce the power of the machines. For us, however, all states are legal halting states and the read/write head can end up anywhere.

Example 3.2.1. A very simple Turing machine zeroes out the original string, so $*011010*$ becomes $*000000*$.

We are allowed to set conditions on the form of input our machine can successfully work on, and the initial set-up of the machine. In general, we want to make the input as simple as possible and put the read/write head in a known location. Conditions that make this machine easier to write are: the input must be one consecutive string, and the read/write head must begin at the leftmost bit of the input. We also specify the start state is q_0.

Our desired computation is

> if we see a 0, move right
> if we see a 1, overwrite a 0 and move right
> halt at the end of the string.

Note that if the machine writes a 0, at the next step it will be scanning a 0, and so the first line takes care of the last part of the second. We can accomplish this with two quadruples:

$\langle q_0, 0, R, q_0\rangle$	already 0, bypass
$\langle q_0, 1, 0, q_0\rangle$	if 1, change to 0.

We don't need an instruction beginning $q_0, *$; the machine will halt when the read/write head walks off the right end of the input.

Example 3.2.2. We can modify the machine in Example 3.2.1 to create one that blanks the input tape, taking $*011010*$ to $********$. Set the same initial conditions as before.

Attempt 1:

$\langle q_0, 0, *, q_0\rangle$	blank 0s
$\langle q_0, 1, *, q_0\rangle$	blank 1s
$\langle q_0, *, R, q_0\rangle$	bypass blanks.

This is a perfectly valid Turing machine, and it does erase all the 1s and 0s, but it does not accomplish our goal: it never halts. Once the tape is blank, this machine continues walking right forever. This is where state comes in: we need to keep track of whether the blank we are viewing is an "in-progress blank" or the blank marking the end of the input. We need two states: one to indicate the symbol being viewed has not yet been dealt with, and one to indicate it has.

Real program:

$\langle q_0, 0, *, q_1\rangle$	blank 0s
$\langle q_0, 1, *, q_1\rangle$	blank 1s
$\langle q_1, *, R, q_0\rangle$	bypass blanks, provided we just created them.

This machine will halt when it arrives at a square that is already blank; there is no $q_0, *$ instruction. Exercise 3.2.3 continues this train of modification.

Exercise 3.2.3. Write a Turing machine to flip the bits of an input string, so $*011010*$ becomes $*100101*$.

Example 3.2.4. Next we'll write a Turing machine that adds 1 to an input in tally notation. For example, $*1111*$ should become $*11111*$.

Again, we specify that the input must be in tally notation and the TM be in state q_0 with read/write head at the leftmost 1. Our desired computation is

move right to first $*$
write 1
halt.

Therefore we write two instructions, letting halting happen because of an absence of relevant instructions.

$\langle q_0, 1, R, q_0\rangle$	move R as long as you see 1
$\langle q_0, *, 1, q_1\rangle$	when you see $*$, write 1 and change state.

Since we specified what tape content and head position we were writing a machine for, these are sufficient: we know the only time the machine will read a blank while in state q_0 is the first blank at the end of x.

Example 3.2.5. Our final example is to add 1 to a number given in binary notation, which is more complicated than Example 3.2.4.

We can choose to interpret the input with the leftmost digit as the smallest, specifying that this is the sort of input our TM is designed to handle. Under that interpretation $*011001*$ is 38. To get 39 we need only change that leading 0 to a 1: $\langle q_0, 0, 1, q_1\rangle$.

But what if instead we begin with 39, $*111001*$? We need the tape to end reading $*000101*$. A computation that takes care of both 38 and 39 is

if you see 0, write 1 and stop
if you see 1, write 0 and move right.

Eventually we will pass the 1s and find a 0 to change to 1. We can add to our previous quadruple to take care of writing 0s and moving.

$\langle q_0, 1, 0, q_2\rangle$	reading 1, write 0 and change state
$\langle q_2, 0, R, q_0\rangle$	move right and go back to start state.

We have to change state when we write 0 so the machine knows the 0 it is reading the next time around is the one it just wrote.

However, there's yet a third case we haven't yet accounted for: numbers like 31, represented as $*11111*$. Our current states fall off the edge of the world – we get to $*00000*$ and halt because no instruction begins with $q_0, *$. We *want* to write a 1 in this case, and we know the only way we get here is to have just executed the instruction $\langle q_2, 0, R, q_0\rangle$. Therefore we can take care of this third case simply by adding the quadruple $\langle q_0, *, 1, q_1\rangle$.

The full program:

$$\begin{aligned}
&\langle q_0, 0, 1, q_1\rangle\\
&\langle q_0, 1, 0, q_2\rangle\\
&\langle q_2, 0, R, q_0\rangle\\
&\langle q_0, *, 1, q_1\rangle
\end{aligned}$$

Exercise 3.2.6. Step through the binary addition program with the following tapes, where you may assume the read/write head begins at the leftmost non-blank square. Give the contents of the tape, position of read/write head, and current state of the machine for each step; see Exercise 3.2.10 for a suggested way to write that information.

(i) $*011*$

(ii) $*101*$

(iii) $*111*$

Exercise 3.2.7. Determine the function computed by the following Turing machine (indenting is for clarity only). The machine takes as input a binary number with least-value digit to the left and read/write head on the leftmost symbol of the input, and starts in state q_0.

$$\begin{aligned}
&\langle q_0, 0, R, q_1\rangle\\
&\langle q_0, 1, R, q_1\rangle\\
&\langle q_1, *, L, q_2\rangle\\
&\qquad\quad \langle q_2, 1, 0, q_2\rangle\\
&\langle q_1, 0, L, q_3\rangle\\
&\langle q_1, 1, L, q_3\rangle\\
&\langle q_3, 0, 1, q_4\rangle\\
&\qquad\quad \langle q_4, 1, R, q_3\rangle\\
&\langle q_3, 1, 0, q_5\rangle
\end{aligned}$$

$\langle q_5, 0, R, q_6\rangle$
$\langle q_6, *, L, q_7\rangle$
$\langle q_7, 0, *, q_7\rangle$

Exercise 3.2.8. Write a Turing machine to compute the function $f(x) = 4x$, where the input is given in

(i) tally notation.
(ii) binary notation.

Exercise 3.2.9. Write a Turing machine to compute the function $f(x) = x \mod 3$, where the input is given in

(i) tally notation.
(ii) binary notation.

Exercise 3.2.10. This exercise will walk you through writing an inverter, a Turing machine that reverses the input string, so $*0111001*$ becomes $*1001110*$.

Here is what ought to happen, where the arrow represents the read/write head's position.

- Instruction Block A

	$*$	0	1	1	$*$	$*$	$*$	$*$	
		$\uparrow$							

walk right to first blank
back up one square, read and erase symbol
walk right and write symbol

- Instruction Block B

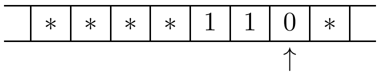
	$*$	0	1	$*$	1	$*$	$*$	$*$	
					$\uparrow$				

walk left past block of blanks
read and erase symbol
walk right past blanks and symbols
write symbol

Likewise:

	$*$	0	$*$	$*$	1	1	$*$	$*$	
						$\uparrow$			

- Instruction Block C



halt.

These three blocks of states, if written correctly, will allow the machine to deal with arbitrarily long symbol strings. Longer strings will result in more iterations of B, but A and C occur only once apiece.

Use state to remember which symbol to print and to figure out which block of symbols you're currently walking through. A complication is knowing when to stop: once the last symbol has been erased, how do you know not to walk leftward forever? Step one extra space left to see if what you just read was the last symbol and use state to account for a yes or no answer.

3.3. Partial Recursive Functions

Turing's definition of computability was far from the only competitor on the field. We will explore only one other in depth, but survey a few more in the next section. Before Turing's paper appeared, there had been a lot of work on recursive functions and on computable functions generally; in particular the question of whether they were the same had been addressed (see §3.3.2). Gödel (see [30], though the definition was used earlier) proposed a definition of recursive function, due partially to Herbrand, which was tidied up by Kleene [44] and retitled the *primitive recursive functions*.[2] Kleene also extended the definition to the *partial recursive functions*, Definition 3.3.9 below.

3.3.1. Primitive Recursive Functions. Since the definition of the primitive recursive functions can be a little opaque at first, we will state it and then discuss it.

Definition 3.3.1. The class of *primitive recursive functions* is the smallest class $\mathcal{C}$ of functions such that the following hold:

[2]The history here is difficult to untangle; Rózsa Péter – who appears to have coined the term "primitive recursion" in [69] – attributes the form of recursion laid out in Definition 3.3.1 (v) to Hilbert in [70]. Odifreddi gives the primitive recursive functions to Dedekind, Skolem, and Gödel (Definition I.1.6 in [68]). Ackermann [1] frames his function as an answer to a conjecture of Cantor.

(i) The successor function $S(x) = x + 1$ is in $\mathcal{C}$.

(ii) All constant functions $M^n_m(x_1, x_2, \ldots, x_n) = m$ for $n, m \in \mathbb{N}$ are in $\mathcal{C}$.

(iii) All projection (or identity) functions $P^n_i(x_1, x_2, \ldots, x_n) = x_i$ for $n \geq 1$, $1 \leq i \leq n$, are in $\mathcal{C}$.

(iv) (Composition, or substitution.) If $g_1, g_2, \ldots, g_m, h$ are in $\mathcal{C}$, then

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

is in $\mathcal{C}$, where the g_i are functions of n variables and h is a function of m variables.

(v) (Primitive recursion, or just recursion.) If $g, h \in \mathcal{C}$ and $n \geq 0$, then the function f defined below is in $\mathcal{C}$:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$$

where g is a function of n variables and h a function of $n + 2$ variables.

Cast in the terminology of §2.5.2, the first three items in the list are the basic objects of the class, and the last two items are the closure operators. Demonstrating that functions are primitive recursive can be complicated, as one must demonstrate how they are built from the ingredients above.

Example 3.3.2. The addition function, $f(x, y) = x + y$, is primitive recursive.

We can express addition recursively with $f(x, 0) = x$ and $f(x, y+1) = f(x, y) + 1$. The former is almost in proper primitive recursive form; let $f(x, 0) = P^1_1(x)$.

The latter needs to be in the form $f(x, y + 1) = h(x, y, f(x, y))$, so we want an h that spits out the successor of its third input. With an application of composition, we get $h(x, y, z) = S(P^3_3(x, y, z))$, and our derivation is complete.

Example 3.3.3. The modified subtraction function

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$

is primitive recursive. The first analysis is that we may define this recursively by $x \dot{-} 0 = x$, and $x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1$.

Therefore, we begin with just $x \dot{-} 1$, which is a function of one variable we will call $\mathrm{sub1}(x)$. The recursive step hardly warrants the name recursive: $\mathrm{sub1}(0) = 0 = M_0^0$, and $\mathrm{sub1}(y+1) = y = P_1^2(y, \mathrm{sub1}(y))$.

Now the full generality can be obtained using sub1 in the recursion, from the initial analysis. Letting $x \dot{-} y = f(x,y)$ for clarity, we need $f(x,0) = x = P_1^1(x)$ and $f(x, y+1) = \mathrm{sub1}(P_3^3(x, y, f(x,y)))$.

In the derivations for each of the exercises below you may use any functions that were earlier proved primitive recursive.

Exercise 3.3.4. Prove that the maximum and minimum functions, $\max(x,y)$ and $\min(x,y)$, are primitive recursive.

Exercise 3.3.5. Prove that the multiplication function, $g(x,y) = x \cdot y$, is primitive recursive.

Exercise 3.3.6. Prove that the factorial function, $n! = n \cdot (n-1) \cdot \ldots 2 \cdot 1$, is primitive recursive.

Exercise 3.3.7. Prove the exponential function x^y is a primitive recursive function of two variables.

Exercise 3.3.8. Consider a grid of streets, n east-west streets crossed by m north-south streets to make a rectangular map with nm intersections; each street reaches all the way across or up and down. If a pedestrian is to walk along streets from the northwest corner of this rectangle to the southeast corner, walking only east and south and changing direction only at corners, let $r(n,m)$ be the number of possible routes. Prove r is primitive recursive.

In fact, the primitive recursive functions form a very large class, including nearly all functions encountered in usual mathematical work, and perhaps have claim on the label "computable" by themselves. We will argue in the following sections that they are insufficient.

3.3.2. The Ackermann Function. The Ackermann function is the most common example of a (total) computable function that is not primitive recursive; in other words, it is evidence that something needs to be added to the closure schema of primitive recursive functions in order to fully capture the notion of computability, even if we require everything to be total.

To say "the Ackermann function" is actually misleading, as a number of different functions are given under this title and nearly all of them are significant simplifications of Ackermann's original function, which is the following, defined recursively for non-negative integers a, b, and n [1].

$$\begin{gathered}\varphi(a,b,0)=a+b\\ \varphi(a,0,n+1)=\alpha(a,n)\\ \varphi(a,b+1,n+1)=\varphi(a,\varphi(a,b,n+1),n)\end{gathered}$$

Here, $\alpha(a,n)=n$ if $n\in\{0,1\}$ and a otherwise.

Péter [70] simplified the definition to essentially the following function of only two variables, though her $m=0$ case is $2n+1$.

$$A(m,n)=\begin{cases} n+1 & \text{if } m=0\\ A(m-1,1) & \text{if } m>0 \text{ and } n=0\\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0\end{cases}$$

She makes it explicit that one question of interest is whether the uncountably many functions that are not primitive recursive are computable at all, and, in the event that there are more computable functions, whether they can specified using some form of recursion. The Ackermann function is a positive answer to both questions.

The proof this is not primitive recursive is technical, but the idea is simple. Here is what we get when we plug small integer values in for m.

$$\begin{aligned} A(0,n) &= n+1\\ A(1,n) &= n+2\\ A(2,n) &= 2n+3\\ A(3,n) &= 2^{n+3}-3\\ A(4,n) &= 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}-3\end{aligned}$$

The stack of 2s in the final equation is $n+3$ entries tall. That value grows incredibly fast: $A(4,2)$ is a 19729-digit number. In fact, the

Ackermann function grows faster than any primitive recursive function, meaning for every primitive recursive function $\theta(n)$, the following holds.

$$(\exists N)(\forall m)(m > N \Rightarrow A(m,m) > \theta(m))$$

The key is found in the stack of 2s in $A(4,n)$. Roughly, each iteration of exponentiation requires an application of primitive recursion. We can have only a finite number of applications of primitive recursion, fixed in the function definition, in any given primitive recursive function. However, as n increases, $A(4,n)$ requires more and more iterations of exponentiation, eventually surpassing any fixed number of applications of primitive recursion, no matter how large.

3.3.3. Partial Recursive Functions: Unbounded Search. To increase the computational power of our class of functions we add an additional closure scheme. This accommodates problems like the need for increasingly many applications of primitive recursion in the Ackermann function.

Definition 3.3.9. The class of *partial recursive functions* is the smallest class of functions containing (i), (ii), and (iii) from Definition 3.3.1 of the primitive recursive functions, and closed under (iv) and (v) from that definition as well as

(vi) (Unbounded search, minimization, or μ-recursion.) If $\bar{x} = x_1, \ldots, x_n$, $\theta(\bar{x}, y)$ is a partial recursive function of $n+1$ variables, and we define $\psi(\bar{x})$ to be the least y such that $\theta(\bar{x}, y) = 0$ and $\theta(\bar{x}, z)$ is defined for all $z < y$, then ψ is a partial recursive function of n variables.

Note that any primitive recursive function is also partial recursive.

One of the most important features of this closure scheme is that it introduces partiality; the primitive recursive functions are all total. A simple (if pointless) example would be to set

$$\theta(x,y) = \begin{cases} 0 & x \text{ even} \\ 1 & x \text{ odd} \end{cases}$$

so that the $\psi(x)$ obtained by unbounded search is 0 whenever x is even, and divergent whenever x is odd.

A function using unbounded search *can* be total, of course, and of course the Ackermann function requires unbounded search, despite being total.

Why should partiality be allowed? Unbounded search allows us to hunt through the natural numbers for conditions that may never hold, or, if they do, may hold arbitrarily far out along the number line. The primitive recursive functions require a sort of time limit set in advance. We can't include all conditions that hold only of unpredictably large numbers without also including conditions that fail to hold; this is true even restricting to conditions that may be finitely computably checked of any individual number. That is, from the modern perspective, real computers sometimes get caught in infinite loops. Another reason is that we can't "get at" just the total functions from the collection of all partial recursive functions. There's no way to single them out; this notion is made precise as Theorem 3.5.1.

The name μ-recursion comes from a common notation. The symbol μ, or μ-operator, is read "the least" and is used (from a purely formula-writing standpoint) in the same way that quantifiers are used. For example, $\mu x(x > 5)$ is read "the least x such that x is greater than five" and returns the value 6. In μ-notation, unbounded search gives the following function.

$$\psi(x_1, \ldots, x_n) = \mu y[\theta(x_1, \ldots, x_n, y) = 0 \ \& \ (\forall z < y)\theta(x_1, \ldots, x_n, z)\downarrow]$$

Example 3.3.10. Using unbounded search we can write a function to return $\sqrt{x}$ if x is a square number and diverge otherwise.

We will use the primitive recursive functions $+$, $\cdot$, and integer subtraction $\dot{-}$ (Example 3.3.3) without derivation. We would like the following.

$$\psi(x) = \mu y[(x \dot{-} (y \cdot y)) + ((y \cdot y) \dot{-} x) = 0]$$

To properly define the function in brackets requires some nested applications of composition, even taking the three arithmetic operators as given.

3.4. Coding and Countability

So far we've computed only with natural numbers. How could we define computation on domains outside of $\mathbb{N}$? If the desired domain is countable, we may be able to encode its members as natural numbers. For example, we could code $\mathbb{Z}$ into $\mathbb{N}$ by using the even natural numbers to represent nonnegative integers, and the odd to represent negative integers. Specifically, we can write the following computable function.

$$f(k) = \begin{cases} 2k & k \geq 0 \\ -2k-1 & k < 0 \end{cases}$$

We might also want a subset of $\mathbb{N}$ to serve in place of all of the natural numbers. For example, we might let every natural number be interpreted as its double. In terms of coding the even numbers E into $\mathbb{N}$, the function is now $g(k) = k/2$.

The important property we need to treat natural numbers as codes of elements from another set S is a bijection between S and $\mathbb{N}$ that is computable with computable inverse. Sets for which such bijections, or *coding functions*, exist are called *effectively countable.* The image of the element of S under this bijection is its *code*; the fact that the function is bijective ensures every natural number codes some unique element of S. Finally, the effectiveness of the function gives us its usefulness: any infinite countable set is in bijection with $\mathbb{N}$, but we can't use that bijection in a Turing machine unless it is computable.

Note that unlike countability in general, effective countability is not guaranteed for subsets of effectively countable sets. One example is in Theorem 3.5.1; more will appear in Chapter 5.

There are two ways to compute on coded input.

(1) The Turing machine can decode the input, perform the computation, and encode the answer.

(2) The Turing machine can compute on the encoded input directly, obtaining the encoded output.

Exercise 3.4.1. Consider $\mathbb{Z}$ encoded into $\mathbb{N}$ by f above. Write a function that takes $f(k)$ as input and outputs $f(2k)$ using approach 2 above. An algebraic expression will suffice.

Coding is often swept under the rug; in research papers one generally sees at most a comment to the effect of "we assume a coding of [our objects] as natural numbers is fixed." It is a vital component of computability theory, however, as it removes the need for separate definitions of *algorithm* for different kinds of objects.

To move into $\mathbb{N}^2$, the set of ordered pairs of natural numbers, there is a standard *pairing function* indicated by angle brackets.

$$\langle x, y\rangle := \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$$

For longer tuples we iterate, so for example $\langle x, y, z\rangle := \langle\langle x, y\rangle, z\rangle$. The pairing function and its iterations show that for every k, $\mathbb{N}^k$ is effectively countable. They also let us treat multivariable functions in the same way as single-input functions.

The pairing function is often given as a magic formula from on high, but it's quite easy to derive. You may be familiar with Cantor's proof that the rational numbers are the same size as the natural numbers, where he walks diagonally through the grid of integer-coordinate points in the first quadrant and skips any that have common factors (if not, see Appendix A.3). We can do essentially that now, though we won't skip anything.

Starting with the origin, we take each diagonal and walk down it from the top (see Figure 3.1). The number of pairs on a given diagonal is one more than the sum of the entries of each pair. The number of pairs above a given (x, y) on its own diagonal is x, so if we want to number pairs from 0, we let (x, y) map to

$$1 + 2 + \ldots + (x + y) + x,$$

where each term except the last corresponds to a diagonal below (x, y)'s diagonal. This sums to

$$\frac{(x + y + 1)(x + y)}{2} + x = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y).$$

$$\begin{tikzcd}
(0,3) \arrow[dr] & & & \\
(0,2) \arrow[dr] & (1,2) \arrow[dr] & & \\
(0,1) \arrow[dr] & (1,1) \arrow[dr] & (2,1) \arrow[dr] & \\
(0,0) \arrow[u] & (1,0) \arrow[uul] & (2,0) \arrow[uuull] & (3,0)
\end{tikzcd}$$

Figure 3.1. Order of pair counting for pairing function.

If you are unfamiliar with the formula for the summation of the integers 1 through n, you can find it in Appendix A.2; it is not needed beyond this point.

A coding function need not have an explicit formula like the polynomial for the pairing function. The rational numbers, $\mathbb{Q}$, may be coded similarly to $\mathbb{N}^2$, using their fractional representation. Let 0 map to 0, and then map the strictly positive numbers into $(\mathbb{N} - \{0\}) \times (\mathbb{N} - \{0\})$. Skipping any pairs (x, y) such that x/y is not in least terms, list pairs in the same order as in the pairing function. The code of a positive rational is the position in the list of its least-terms fractional representation, so 1 maps to 1, 1/2 maps to 2, and 2 maps to 3, the positions of $(1, 1)$, $(1, 2)$, and $(2, 1)$, respectively. To account for negative rational numbers, double all of these codes, and subtract 1 if the rational is negative; 0 still maps to 0, but 1 maps to 2 (-1 maps to 1) and 1/2 maps to 4 ($-1/2$ maps to 3).

The important set $\bigcup_{k\geq 0} \mathbb{N}^k$, finite tuples of any length, is also effectively countable (note $\mathbb{N}^0 = \{\emptyset\}$). The function

$$\tau : \bigcup_{k\geq 0} \mathbb{N}^k \to \mathbb{N}$$

given by $\tau(\emptyset) = 0$ and

$$\tau(a_1, \ldots, a_k) = 2^{a_1} + 2^{a_1+a_2+1} + 2^{a_1+a_2+a_3+2} + \ldots + 2^{a_1+\ldots+a_k+k-1}$$

demonstrates the effective countability. A singleton – that is, an element of $\mathbb{N}$ itself – is mapped to a number with binary representation using a single 1. An n-tuple maps to a number whose binary representation uses exactly n 1s.

Exercise 3.4.2. (i) Find the images under τ of the tuples $(0, 0)$, $(0, 0, 0)$, $(0, 1, 2)$, and $(2, 1, 0)$.

(ii) What is the purpose of summing subsequences of a_i and adding $1, 2, \ldots, k-1$ in the exponents? What tuples get confused if you perform only one of those actions?

(iii) Prove that τ is a bijection.

Exercise 3.4.3. Let A and B be effectively countable, infinite sets.

(i) If A and B are disjoint, prove that $A \cup B$ is effectively countable.

(ii) If A and B are not necessarily disjoint, prove that $A \cup B$ is effectively countable.

(iii) Prove that $A \cap B$ is effectively countable or finite.

Exercise 3.4.4. (i) Show that if a class A of objects is constructed recursively using a finite set of basic objects and a finite collection of computable closure operators (see §2.5), A is effectively countable.

(ii) Show that even if the sets of basic objects and rules in part (i) are infinite, as long as they are effectively countable, A is also effectively countable. §5.1 may be helpful.

Example 3.4.5. This example is vital for the rest of the text. The set of Turing machines is, in fact, effectively countable; the TMs may be coded as natural numbers. One way to code them is to first encode individual quadruples as single numbers, so the machine is represented by a finite set, and then encode finite subsets of $\mathbb{N}$ as single numbers, as in Exercise 3.1.4. Similarly to the rational numbers, we can encode a subset of the quadruples and then spread them out to "fit in" the rest. The symbols that may appear in the middle two slots of a quadruple are drawn from a fixed finite list, but the indices of the states that appear first and last may be arbitrarily large. Therefore, we start by encoding $\langle q_i, \cdot, \cdot, q_j\rangle$ as $\langle i, j\rangle$ by pairing, and then multiply those values by a large enough number to fit exactly the possibilities for the middle two slots. That value will depend on what symbols are allowed; if the symbol set is only $*$ and 1, it will be 8: there are 8 pairs with first element $*$ or 1 and second element $*$, 1, L, or R. Order the pairs in some fixed way; perhaps all pairs beginning with $*$ precede all pairs beginning with 1 and within those blocks they are ordered according to the list of second elements above. In that case, $\langle q_3, *, L, q_3\rangle$ maps to $8\langle 3,3\rangle + 2$ and $\langle q_3, 1, R, q_3\rangle$ to $8\langle 3,3\rangle + 7$; $\langle q_0, *, *, q_1\rangle$ maps to $8\langle 0,1\rangle + 0$.

Conversely, the natural number n will be read as $k + \ell$, where $0 \leq \ell \leq 7$ and k is a multiple of 8. The quotient $k/8$ is decoded into $\langle i, j\rangle$ and ℓ is used to find the symbol read and action taken.

It is important to note that these codes will include "junk machines," codes that may be interpreted as TMs but which give machines that don't do anything. There will also be codes that give machines that, while different, compute the same function – they are different implementations of the function. In fact, we can prove the Padding Lemma, Exercise 3.4.7, after a bit of vocabulary.

Definition 3.4.6. We call the code of a Turing machine its *index*, and say when we choose a particular coding that we *fix an enumeration* of the Turing machines (or, equivalently, the partial recursive functions). It is common to use φ for partial recursive functions; φ_e is the e^{th} machine/function in the enumeration, the machine with index e, and the machine that encodes to e.

Indices are often called *Gödel numbers*, because Gödel introduced the notion of coding formulas in order to prove incompleteness [30]. Coding formulas and sequences of formulas (such as proofs) as numbers allows the statement "there is no proof of me" to be expressed as, roughly, "no number codes a proof of the formula coded by this number," where the number in question at the end codes the given formula itself.

Exercise 3.4.7. (The Padding Lemma). Prove that given any index of a Turing machine M, there is a larger index which codes a machine that computes the same function as M.

We often use the indices simply as tags, to put an ordering on the functions, but it is often important to remember that the index *is* the function, in a very literal way. Given the index, it is primitive recursive to obtain the entire machine. Conversely, you will also see indices defined by description of the function of which they code an implementation. For example, letting e be such that

$$\varphi_e(x,y) = \begin{cases} x+y & y<2 \\ xy & 2 \leq y \leq 4 \\ x^y & y \geq 4 \end{cases}$$

is a perfectly good definition of e. In principle, we can write that function as a Turing machine, compute the machine's index, and use that value as e.

In fact, we can define functions in this way as well. An example of such a definition is "let $f(x)$ index the function with domain $\{x\}$." Given x, we can create a Turing machine that halts on x alone, and encode that as an index. That is the value we would like to call $f(x)$. The missing piece is *uniformity*: to claim f is a computable function, the process of creating the Turing machine from x must be independent of the value of x. That is, while it will create different machines from different values of x, there is one instruction sheet we can give to the person writing the machines that covers all the possibilities. Here, that is straightforward: have, say, $x+1$ states, x of which count 1s from 0 to x and halt if there are the right number, the last of which is the "nonhalting" state that is entered when the 1s run out too early or last too long. The value $f(x)$ is the index of the machine for x.

Exercise 3.4.8. Explicitly give a template for a Turing machine with domain $\{x\}$, with x a natural number. You are not required to have exactly $x+1$ states.

Another collection of objects commonly indexed is the finite sets, as in Exercise 3.1.4. The n^{th} finite set, or set corresponding to n in the bijection, is typically denoted D_n.

The key points of this section for our later work are the following:

(1) Via coding, we can treat any effectively countable set as though it were $\mathbb{N}$.

(2) A single natural number may mean a great variety of objects; Turing machines are set to interpret their input in a specific way.

(3) We can fix an enumeration of the Turing machines (equivalently, the partial computable functions); the index of a particular machine will *be* that machine in code. It is understood that the coding is fixed from the start so we are never trying to decode with the wrong bijection.

3.5. A Universal Turing Machine

From the enumeration of all Turing machines and the pairing function we can define a *universal* Turing machine U; that is, a machine that will emulate *every* other machine. Simply set

$$U(\langle e, x\rangle) = \varphi_e(x).$$

U decodes the single number it receives into a pair (e, x), decodes e into the appropriate set of quadruples, and uses x as input, acting according to the quadruples it decoded. This procedure is computable because decoding the pairing function, decoding Turing machines from indices, and executing quadruples on a given input are computable procedures. Turing [85] gives an explicit, full construction of a universal machine.

Note that of course there are infinitely many universal Turing machines, as there are for any program via padding, and that a universal machine exists for any collection of functions that may be indexed. Although we will use U and analogously defined universal machines for other indexings in this section, typically we simply refer to the individual indexed functions, using $\varphi_e(x)$ instead of $U(\langle e, x\rangle)$.

We are now in the position to demonstrate a very practical reason to allow partial functions in our definition of computability. Recall that by *total computable function* we mean a function from the class of partial computable functions which happens to be total. The following theorem also gives an example of a subset of the effectively countable set $\mathbb{N}$ that is not itself effectively countable: the indices of the total computable functions.

Theorem 3.5.1. *There is no computable indexing of the total computable functions.*

Proof. Suppose the contrary and let f_e denote the e^{th} function in an effective enumeration of all total computable functions. Let $u(\langle x, y\rangle)$ be a universal machine for the total computable functions; it is itself computable, and since $f_x(y)$ is defined for all x and y, u is also total. We define a new function as follows.

$$g(e) = u(\langle e, e\rangle) + 1$$

It is clear that g is total computable, since it is successor composed with the total computable function u. Hence g must have an index; that is, there must be some e' such that $g = f_{e'}$. However, $g(e') = u(\langle e', e'\rangle) + 1 = f_{e'}(e') + 1 \neq f_{e'}(e')$, which is a contradiction. Therefore no such indexing can exist. $\square$

As an aside, those who are familiar with Cantor's proof that the real numbers are uncountable will notice a distinct similarity (if not, see Appendix A.3). This is an example of a *diagonal argument*, where you accomplish something with respect to the e^{th} Turing machine using e. Of course you need not use literally e, as we will see in later chapters.

We have two choices, then, with regard to the collection of functions we call "computable:" to have them all be total, but fail to have an indexing of them, or to include partial functions and be able to enumerate them. We will see many proofs that rely completely on the existence of an indexing in order to work; this combined with the justifications in §3.3.3 weigh heavily on the side of allowing partial functions to be called computable.

We now return to the primitive recursive functions, and a way to break out of that class other than the Ackermann function.

Theorem 3.5.2. *There is a function f such that $f(\langle n, x\rangle)$ computes the n^{th} primitive recursive function on input x.*

Sketch of Proof. We may code the primitive recursive functions by their derivation from the basic functions through the application of closure operators. The basic functions and closure operators are given values, which are then combined via pairing in ways that indicate how closure operators are being applied to basic functions and functions appearing earlier in the derivation. This may be done computably, giving an indexing of the primitive recursive functions; call the n^{th} such function θ_n. Define $f(\langle n, x\rangle) = \theta_n(x)$; f operates analogously to U, as described at the beginning of the section. ☐

Corollary 3.5.3. *There is a function that is total recursive but not primitive recursive.*

Proof. From f as in Theorem 3.5.2, define $g(n) = f(\langle n, n\rangle)+1$. Since all primitive recursive functions are total and codings are bijective, f is total, and hence g is total. However, g is not primitive recursive, because for every n, it differs from the n^{th} primitive recursive function on input n. ☐

Notice that this is essentially identical to Theorem 3.5.1, but in a different setting and hence with a different conclusion.

Corollary 3.5.4. *The universal function defined in Theorem 3.5.2 is not primitive recursive.*

Proof. The function g from the proof of Corollary 3.5.3 may be written rigorously as

$$g(n) = S(f(\langle P_1^1(n), P_1^1(n)\rangle)),$$

which is a valid primitive recursive derivation, with three applications of composition, provided all the component functions are primitive recursive. Since g is not primitive recursive, one of the component functions must fail to be. As successor, the pairing function, and projection are all primitive recursive, f must be the weak link. ☐

Exercise 3.5.5. A function h *dominates* a function g if there is some $N \in \mathbb{N}$ such that $h(n) \geq g(n)$ for all $n \geq N$.

(i) Construct a function h that dominates all the primitive recursive functions $\{\theta_e\}_{e\in\mathbb{N}}$ (it is permitted that different θ_e require distinct values of N), and prove that it does. There are at least three natural ways to define h.

(ii) Is h primitive recursive? Prove or refute.

3.6. The Church-Turing Thesis

It is not obvious, but the class of Turing-computable functions and the class of partial recursive functions are the same. In fact, there are a large number of models of computation that give the same class of functions as TMs and partial recursive functions (see §3.7 for a sample). It is even possible to introduce nondeterminism into Turing machines without increasing their power! Because of this, we use the term *partial computable functions* to mean those that are partial recursive or, equivalently, computed by a Turing machine, and *total computable functions* to mean the partial computable functions that happen to be total. If a function is referred to as just computable, it is total computable.

To show partial recursive functions are Turing computable, one can explicitly construct machines that compute successor (e.g., Example 3.2.4), constants, and projections, and then show the Turing machines are closed under composition, primitive recursion, and unbounded search. Conversely, given a Turing machine, one can create a partial recursive function that emulates it, in the very strong sense that it mimics the contents of the TM's tape at every step of the computation. There is an excellent presentation of this in §8.1 of Boolos, Burgess, and Jeffrey [10] that we sketch.

The tape contents are viewed in two pieces, the spaces to the left of the read/write head as a binary number with the least digit rightmost, and the spaces from the read/write head and on to the right as a binary number with its least digit leftmost (the scanned square). Motion along the tape and rewriting are now arithmetic: if the read/write head moves left, the left binary number halves, rounded down, and the right binary number doubles and possibly adds one, depending on the contents of the new scanned square. The current status of the computation is coded as a triple: leftward tape contents, current state, rightward tape contents. Actions (motion and rewriting) are assigned numbers, which allows us to code the tuples of the Turing machine, as in §3.4. Finally, the acceptable halting configuration is standardized, and a single application of unbounded search finds the least step t such that the halting condition holds. The output of

$F(m, x)$, where m is the code for a Turing machine and x is the intended input, is the tape contents at the time t found by unbounded search, if such a t exists.

The coincidence of all these classes of functions, defined from very different points of view, may seem nothing short of miraculous. It is necessary, though, if each is correctly claiming to rigorously capture the notion of *computable*, and the fact that we do get the same class of functions is strong evidence that they do so. We can never actually *prove* we have captured the full, correct notion, because any proof requires formalization – the only equivalences we can prove are between different formal definitions. In his original paper [85], Turing does a thought experiment (an expansion of the first paragraph of §3.2) in which he breaks down the operations a human computer is capable of and shows a Turing machine can do each of them, but this is not a *proof*. However, it is compelling when set alongside the collection of disparate approaches that reach the same destination, and the fact that no one has found a disproof. The idea that we have captured the full and correct notion is called the **Church-Turing Thesis**: the computable functions are exactly the Turing-computable functions.

3.7. Other Definitions of Computability

Any programming language with the ability to do arithmetic, use variables, and execute loops of some kind, as well as get input and produce output, is as strong as a Turing machine if given unlimited memory and time to work with. Even a circa 1990 programmable calculator's constrained BASIC, if hooked up to an inexhaustible power source and unlimited memory, can compute anything a Turing machine can. Of course, the closer a programming language is to the absolute minimum required, the harder it is for humans to use it. The trade-off is usually that when you get further from the absolute minimum required, proofs of general properties get more difficult.

3.7.1. Nonstandard Turing Machines. We mentioned in §3.2 that the number of symbols a Turing machine is allowed to use, as long as it is finite, will not change the power of the machine. This is because even with just 1 and $*$, we can represent any finite collection of symbols on the tape by using different length blocks of 1s, separated by $*$s.

Exercise 3.7.1. Prove that a two-symbol Turing machine can simulate an n-symbol Turing machine, for any n.

Likewise, we said that requiring a machine to halt in particular states or end with the read/write head at a particular location (relative to the tape contents) did not reduce the power of the machines. This may be accomplished in a straightforward manner by the addition of extra states and instructions.

Our Turing machine, which we will refer to as *standard*, had a tape that was infinite in both directions. Drawing on §3.4 you can show it can make do with half of that.

Exercise 3.7.2. Prove that a Turing machine whose tape is only infinite to the right (i.e., has a left endpoint) can simulate a standard Turing machine.

More or less complicated coding, on the tape or in the states, gives all of the following as well. This is certainly an incomplete list of the changes we may make to the definition of Turing machine without changing the class of functions computed.

Exercise 3.7.3. Prove that each of the following augmented Turing machines can be simulated by a standard Turing machine.

(i) A TM with a "work tape" where the input is given and the output must be written, with no restrictions in between, as well as a "memory tape" which can hold any symbols at any time through the computation, and independent read/write heads for each of them.

(ii) A TM with a grid of symbol squares instead of a tape, and a read/write head that can move up or down as well as left or right.

(iii) A TM whose read/write head can move more than one square at a time to the right or left.

(iv) A TM where the action and state change depend not only on the square currently being scanned, but on its immediate neighbors as well.

(v) A TM with multiple read/write heads sharing a single tape.

Finally, we introduce the notion of *nondeterminism*, which may seem to introduce noncomputability.

Definition 3.7.4. A *nondeterministic Turing machine* has an infinite tape, a single read/write head that works from a finite list of symbols, and a finite list of internal states, exactly as a standard Turing machine. It is specified by a list of quadruples $\langle a, b, c, d\rangle$, where a and d are states, b is a symbol, and c is a symbol or the letter R or L, with no restriction on the quadruples in the list (note that it will still be finite, since there are only finitely many options for each position of the quadruple).

In particular, there may be multiple quadruples that start with the same state/symbol pair. When the machine gets to such a situation, it picks one such quadruple at random and continues, meaning there could be multiple paths to halting. If we want to define functions from this model of computation, we have to demand that every path that terminates results in the same output. For reasons that Proposition 5.2.4 will make clear, we often dispense with that and ask only on which inputs the nondeterministic machine halts at all. Call those inputs the machine's *domain.*[3]

Claim 3.7.5. *If T is a nondeterministic Turing machine, there is a standard (deterministic) Turing machine T' with the same domain.*

The idea behind the proof is that every time T' comes to a state/symbol pair that starts multiple quadruples of T, it clones its computation enough times to accommodate each of the quadruples in separate branches of its computation. It runs one step of each existing computation (plus all steps necessary to clone, if required) before moving on to another step of any of them, and halts whenever one of its branches halts. If we have required that T define a function, T'

[3]In computer science, we might refer to the domain as the *language* the machine *accepts.*

can output the same thing T' does in this halting branch, since the output will not depend on which halting branch T' finds first.

Exercise 3.7.6. Turn the idea above into a proof of Claim 3.7.5.

3.7.2. The Lambda Calculus. This is an important definition of computability due to Church and his students Kleene and Rosser. Those with an interest in computer science may know that the lambda calculus is the basis of *functional* programming languages such as Lisp and Scheme. A good reference for the lambda calculus is the programming languages book by Peter Kogge [45].

The lambda calculus is based entirely on *substitution*; a typical expression is

$$(\lambda x|E)A,$$

which means “replace every instance of x in E by A.”

Expressions are built recursively. We have a symbol set that consists of parentheses, $|$, λ, and an infinite collection of *identifiers*, generally represented by lowercase letters. An expression can be an identifier, a function, or a pair of expressions side-by-side, where a *function* is of the form $(\lambda\langle\text{identifier}\rangle|\langle\text{expression}\rangle)$. We will use capital letters to denote arbitrary lambda expressions. Formally everything should be thoroughly parenthesized, but understanding that evaluation always happens left to right (i.e., $E_1E_2E_3$ means $(E_1E_2)E_3$, and so on), we may often drop a lot of parentheses. In particular,

$$(\lambda xy|E)AB = (((\lambda x|(\lambda y|E)))A)B.$$

Identifiers are essentially variables, but are called identifiers because their values don’t change over time. We solve problems with lambda calculus by manipulating the form in which the variables appear, not their values. An identifier x *occurs free* in expression E if (1) $E = x$, (2) $E = (\lambda y|A)$, $y \neq x$, and x appears free in A, or (3) $E = AB$ and x appears free in either A or B. Otherwise x *occurs bound* (or does not occur). In $(\lambda x|E)$, only free occurrences of x are candidates for substitution, and no substitution is allowed that converts a free variable to a bound one. If that would be the result of substitution, we rename the problematic variable instead.

Here are the full substitution rules for $(\lambda x|E)A \to E'$. They are defined recursively, in cases matching those of the recursive definition of expression. When needed for clarity, the intermediate notation $[A/x]E$ is used to indicate the substitution $(\lambda x|E)A$ has been made.

(1) If $E = y$, an identifier, then if $y = x$, $E' = A$. Otherwise $E' = E$.

(2) If $E = BC$ for some expressions B, C, the substitutions are made within B and C: $E' = (([A/x]B)([A/x]C))$.

(3) If $E = (\lambda y|C)$ for some expression C and
   - (i) $y = x$, then $E' = E$.
   - (ii) $y \neq x$ where y does not occur free in A (i.e., substitution will not cause a free variable to become bound), then $E' = (\lambda y|[A/x]C)$.
   - (iii) $y \neq x$ where y does occur free in A, we apply the renaming rule: $E' = (\lambda z|[A/x]([z/y]C))$, where z is a symbol that does not occur free in A.

Example 3.7.7. Evaluate

$$(\lambda xy|yxx)(\lambda z|yz)(\lambda rs|rs).$$

Remember that formally this is

$$[(\lambda x|(\lambda y|yxx))(\lambda z|yz)](\lambda rs|rs).$$

The first instance of substitution should be for x, but this will bind what is currently a free instance of y, so we apply rule (3.iii) using identifier symbol a.

$$(\lambda y|y(\lambda z|az)(\lambda z|az))(\lambda rs|rs)$$

Next we make a straightforward substitution to get

$$(\lambda rs|rs)(\lambda z|az)(\lambda z|az),$$

which becomes $(\lambda z|az)(\lambda z|az)$ and finally $a(\lambda z|az)$.

You can see this can rapidly get quite unfriendly to do by hand, but it is very congenial for computer programming. There are two great strengths to functional programming languages: all objects are of the same type (functions) and hence are handled the same way, and evaluation may often be done in parallel. In particular, if we have $(\lambda x_1 \dots x_n|E)A_1 \dots A_m$, where $m \leq n$, the sequential evaluation

$$(\lambda x_{m+1} \dots x_n|([A_m/x_m](\dots([A_2/x_2]([A_1/x_1]E))\dots)))$$

is equivalent to the *simultaneous* evaluation

$$(\lambda x_{m+1} \dots x_n|[A_1/x_1, A_2/x_2, \dots, A_m/x_m]E)$$

provided there are no naming conflicts. That is, alongside the restriction of not having any $x_{i+1}, \dots, x_n$ free in A_i (which would then bind a free variable, never allowed), we must know none of the $x_{m+1}, \dots x_n$ appear free in any A_i, $i \leq m$.

To start doing arithmetic, we need to be able to represent zero and the rest of the positive integers, at least implicitly (i.e., via a successor function). Lambda calculus “integers” are functions that take two arguments, the first a successor function and the second zero, and which (if given the correct inputs) return an expression which “equals” an integer.

$$\begin{array}{lll}
0: & (\lambda sz|z) & (\lambda sz|z)SZ = [S/s][Z/z]z = Z \\
1: & (\lambda sz|s(z)) & (\lambda sz|s(z))SZ = S(Z) \\
\vdots & & \\
K: & (\lambda sz|\underbrace{s(s\dots s}_{K \text{ times}}(z)\dots)) & KSZ = \underbrace{S(S\dots S}_{K \text{ times}}(Z)\dots)
\end{array}$$

Interpreting Z as zero and $S(E)$ as the successor of whatever integer is represented by E, these expressions yield the positive integers.

We can define successor as a lambda operator in general, as well as addition and multiplication. Successor is a function that acts on an integer K (given as a function) and returns a function that is designed to act on SZ and give $K + 1$. Likewise, multiplication and addition are functions that act on a pair of integers K, L, and return a function designed to act on SZ to give $K \cdot L$ or $K + L$, respectively.

$$\text{Successor}: S(x) = (\lambda xyz|y(xyz))$$

$$\text{Addition}: (\lambda wzyx|wy(zyx))$$

$$\text{Multiplication}: (\lambda wzy|w(zy))$$

To understand these it is best to step through an example.

Example 3.7.8. $2+3$.

In order to avoid variable clashes, where the general definition of lambda calculus integers above uses s and z, we'll use s and a in 2 and r and b in 3.

$$\begin{aligned}2+3 &= (\lambda wzyx|wy(zyx))(\lambda sa|s(s(a)))(\lambda rb|r(r(r(b))))\\ &= (\lambda yx|(\lambda sa|s(s(a)))y((\lambda rb|r(r(r(b))))yx))\\ &= (\lambda yx|(\lambda a|y(y(a)))(y(y(y(x)))))\\ &= (\lambda yx|y(y(y(y(y(x)))))) = 5\end{aligned}$$

Exercise 3.7.9. Evaluate $S(3)$.

Exercise 3.7.10. Evaluate $2 \cdot 3$.

Similarly we can define lambda expressions that execute "if… then… else" operations. In this context those are expressions P such that PQR returns Q if P is true and R if P is false. Additional Boolean operations are also useful and are defined as follows.

$$\begin{array}{cc}\text{true} : T = (\lambda xy|x) & \text{false} : F = (\lambda xy|y)\\ \text{and} : (\lambda zw|zwF) & \text{or} : (\lambda zw|zTw)\\ \multicolumn{2}{c}{\text{not} : (\lambda z|zFT)}\end{array}$$

Exercise 3.7.11. Work out the following operations.

(i) not T, not F

(ii) and TT, and TF, and FT, and FF

(iii) or TT, or TF, or FT, or FF

(iv) or(and TF)(not F)

The missing piece to understand how this can be equivalent to Turing machines is *recursion*, in the computer science sense: if A is a base case for R, then RA is simply evaluated, and if not, then RA reduces to RB, where B is simpler than A. This is our looping procedure; it requires R calling itself as a subfunction. To make expressions call themselves we first need to make them duplicate themselves. We begin with a magic function.

$$(\lambda x|xx)(\lambda x|xx)$$

Try doing the substitution called for. Next, when R is an expression wherein x does not occur free, evaluate

$$(\lambda x|R(xx))(\lambda x|R(xx)).$$

This is not general, however, and so we remove the hard-coding of R via another lambda operator. This gives us our second magic function, the *fixed point combinator* Y.

$$Y = (\lambda y|(\lambda x|y(xx))(\lambda x|y(xx)))$$

When Y is applied to some other expression R, the result is to layer Rs onto the front.

$$YR = R(YR) = R(R(YR)) = R(R(R(YR)))\dots$$

Finally, consider $(YR)A$ to get to our original goal. This evaluates to $R(YR)A$; if R is a function of two variables, it can test A and return the appropriate expression if A passes the test, throwing away the (YR) part, and if A fails the test it can use the (YR) to generate a new copy of R for the next step of the recursion. We omit any examples.

3.7.3. Unlimited Register Machines. Shepherdson and Sturgis defined universal register machines, or URMs [77], and Nigel Cutland simplified them to use as the main model of computation in his book *Computability* [17]. We give his definition here. URMs are easier to work with than Turing machines if you want to get into the guts of the model, while still basic enough that the proofs remain manageable. This should feel like a Turing machine made more human-friendly.

The URM has an unlimited memory in the form of *registers* R_i, each of which can hold a natural number denoted r_i. The machine has a *program* comprised of a finite list of *instructions*, and based on those instructions it may alter the contents of its registers. Note that a given computation will only be able to use finitely many of the registers, just as a Turing machine uses only finitely many spaces on its tape, but we cannot cap *how many* it will need in advance.

There are four kinds of instructions.

(i) Zero instructions: $Z(n)$ tells the URM to change the contents of R_n to 0.

(ii) Successor instructions: $S(n)$ tells the URM to increment (that is, increase by one) the contents of R_n.

(iii) Transfer instructions: $T(m, n)$ tells the URM to replace the contents of R_n with the contents of R_m. The contents of R_m are unchanged.

(iv) Jump instructions: $J(m, n, i)$ tells the URM to compare the contents of R_n and R_m. If $r_n = r_m$, it is to jump to the i^{th} instruction in its program and proceed from there; if $r_n \neq r_m$ it continues to the instruction following the jump instruction. This allows for looping. If there are fewer than i instructions in the program, the machine halts.

The machine will also halt if it has executed the final instruction of the program, and that instruction did not jump it back into the program. You can see where infinite loops might happen: $r_n = r_m$, the URM hits $J(m, n, i)$ and is bounced *backward* to the i^{th} instruction, and nothing between the i^{th} instruction and the instruction $J(m, n, i)$ either changes the contents of one of R_n or R_m or jumps the machine out of the loop.

A computation using the URM consists of a program and an *initial configuration*, that is, the initial contents of the registers.

Example 3.7.12. Using three registers, we can compute sums. The initial contents of the registers will be x, y, 0, 0, 0, $\ldots$, where we would like to compute $x + y$. The sum will ultimately be in the first register and the rest will be zero.

We have only successor to increase our values, so we'll apply it to x y-many times. The third register will keep track of how many times we've done it; once its contents equal y we want to stop incrementing x, zero the second and third registers, and halt. Since jump instructions only jump when the registers checked are equal, we have to be clever in their application.

	Instruction:	Explanation:
1.	$J(2,4,8)$	if $y = 0$, nothing to do
2.	$S(1)$	increment x
3.	$S(3)$	increment counter
4.	$J(2,3,6)$	jump out of loop if we're done
5.	$J(1,1,2)$	otherwise continue incrementing
6.	$Z(2)$	zero y register
7.	$Z(3)$	zero counter

Exercise 3.7.13. Write out all steps of the computation of $3 + 3$ using the program above, including the contents of the registers and the instruction number to be executed next.

Exercise 3.7.14. Write a URM program to compute products. Note that $x \cdot y$ is the sum of y copies of x, and iterate the addition instructions appropriately. Be careful to keep your counters for the inside and outside loops separate, and zero them whenever necessary.

Chapter 4

Working with Computable Functions

4.1. The Halting Problem

Is it possible to define a specific function that is not computable? Yes and no. We can't write down a finite procedure, because by the Church-Turing thesis, that leads to a computable function. As Rice's Theorem 4.4.8 will show, we can describe functions that are noncomputable, and via the indexing of all partial computable functions we can define a noncomputable function even more specifically.

Recall from §3.1.2 that we use arrows to denote halting behavior: $\varphi_e(n)\downarrow$ means φ_e halts on input n, and $\varphi_e(n)\uparrow$ means it does not.

Define the *halting function* as follows.

$$f(e) = \begin{cases} 1 & \text{if } \varphi_e(e)\downarrow \\ 0 & \text{if } \varphi_e(e)\uparrow \end{cases}$$

To explore the computability of f, define g.

$$g(e) = \begin{cases} \varphi_e(e) + 1 & \text{if } f(e) = 1 \\ 1 & \text{if } f(e) = 0 \end{cases}$$

This is somewhat analogous to removing a discontinuity in calculus via a piecewise definition, such as setting $h(1) = 3$ in $h(x) = [(x-1)(x+2)]/(x-1)$.

Certainly if $\varphi_e(e)\downarrow$, it is computable to find the output value, and computable to add 1. The use of f avoids attempting to compute outputs for divergent computations, and hence if f is computable, then so is g. However, g is not computable, and so the halting function (or *halting problem*, the question of determining for which values of e $\varphi_e(e)\downarrow$) is not computable. This is a key example, and we define the *halting set* as well.

$$K = \{e : \varphi_e(e)\downarrow\}$$

Exercise 4.1.1. Prove that g defined above is not computable. It is similar but not identical to Theorem 3.5.1 and Corollary 3.5.3.

Another function that is noncomputable and not too hard to define is the busy beaver function. This function has different specific definitions, but always refers to "the most done" by a halting Turing machine from a certain class, where "most" might refer to, say, writing on the tape or steps of computation. We will use the latter. Note that if you fix a finite collection of states, there are only finitely many Turing machines with states contained in that collection.

Exercise 4.1.2. Define $T(n)$ as the maximum value of s such that some Turing machine with states contained in $\{q_0, \dots, q_n\}$ halts after exactly s steps of computation.

(i) Show that T is not computable.

(ii) Show that there is no computable function B such that $B(n) \geq T(n)$ for all n.

§4.5 has more noncomputable functions related to Turing machine behavior.

4.2. The "Three Contradictions"

Assume that a collection of functions may be indexed as ψ_n, $n \in \mathbb{N}$. Define the function $g(n) = \psi_n(n)+1$. If $\{\psi_n\}_{n\in\mathbb{N}}$ is really an indexing, g is one of the ψ, and g is total, we have a contradiction: for some n', we must have $g(n') = g(n') + 1$. As a consequence, one of the three properties must fail.

(a) $\{\psi_n\}_{n\in\mathbb{N}}$ is not really an indexing: In the case of ψ_n listing the total computable functions, we know g would have to be a total computable function, so we conclude there is no indexing.

(b) g is not one of the ψ: In the case of ψ_n listing the primitive recursive functions, we know we have the indexing and g must be total, so we conclude it is not primitive recursive.

(c) g is not total: In the case of ψ_n listing the partial recursive functions, we know we have the indexing and that g must appear in that indexing, so we conclude that on any offending n' g must diverge.

Additionally, in §4.1, we have

(c′) Some auxiliary function being used as though it were computable is not: When we "plug the gaps" in the g of part (c) with the halting problem f, we still have the indexing, and force g to be total. If f is computable, g must appear in the indexing, but then we reach a contradiction as before, so we conclude f is not computable.

4.3. Parametrization

Parametrization means something different in computability theory than it does in calculus. Here it means the ability to push input parameters into the index of a function. This is the first place where it is important that the indexing of Turing machines be fixed, and we take major advantage of the fact that the index contains all the information we need to reconstruct the machine itself.

The simplest form of the *s-m-n* theorem, which is what we traditionally call the parametrization theorem, is the following.

Theorem 4.3.1. *There is a total computable function s^1_1 such that for all i, x, and y, $\varphi_i(x, y) = \varphi_{s^1_1(i,x)}(y)$.*

If you accept a loose description, this is very simple to prove: s^1_1 decodes i, fills x into the appropriate spots, and recodes the resulting algorithm. The key is that although the new algorithm depends on i and x, it does so *uniformly* – the method is the same regardless of the numbers.

As discussed after Exercise 3.4.7, a process is uniform in its inputs if it is like a choose-your-own-adventure book or a complicated flowchart: all possible paths from start to finish are already there, and the particular inputs just tell you which path you'll take this time. Uniformity allows for a single function or construction method or similar process to work for every instance, rather than needing a new one for each instance.

Exercise 4.3.2. Prove there is a computable function f such that $\varphi_{f(x)}(y) = 2\varphi_x(y)$ for all y. Hint: think of an appropriate function $\varphi_i(x, y)$.

The full version of the theorem allows more than one variable to be moved, and more than one to remain as input. More uses of both versions appear in sections to come. Incidentally, the superscript and subscript on this function bear only cosmetic similarity to those in the definition of the primitive recursive functions.

Theorem 4.3.3. *Given m, n, there is a primitive recursive one-to-one function s_n^m such that for all i, all n-tuples $\bar{x}$, and all m-tuples $\bar{y}$,*

$$\varphi_{s_n^m(i,\bar{x})}(\bar{y}) = \varphi_i(\bar{x}, \bar{y}).$$

The fact that you can force this function to be one-to-one follows from the Padding Lemma (Exercise 3.4.7). The proof of Theorem 4.3.3 is the same as for Theorem 4.3.1: each m, n pair has its own function, so s_n^m is built to interpret i as the code of a function of $m + n$ inputs. It decodes i, replaces each of the first n inputs with the appropriate x_j, and encodes the resulting function of m inputs.[1]

While it looks at first as though all this is doing is allowing you to computably incorporate data into an algorithm, the fact that the data could itself be a code of an algorithm means this is more than that; it is composition via indices. In particular, parametrization and the universal machine give us a way to translate operations on sets and functions to operations on indices.

[1]In computer programming, this process of reducing the number of arguments of a function is called *currying*, after logician Haskell Curry; when specific inputs are given to the s_n^m function it is called *partial evaluation* or *partial application*.

For example, suppose we want to find an index for $\varphi_x + \varphi_y$ uniformly in x and y. We can let $f(x, y, z) = \varphi_x(z) + \varphi_y(z)$ by letting it equal $U(\langle x, z\rangle) + U(\langle y, z\rangle)$, so everything that was either in input or index is now in input. Then the s-m-n theorem gives us a computable function $s(x, y)$ such that $\varphi_{s(x,y)}(z) = f(x, y, z)$ (f is φ_i some fixed i; $s(x, y) = s^1_2(i, x, y)$), so $s(x, y)$ is an index for $\varphi_x + \varphi_y$ as a (total computable) function of x and y.

Exercise 4.3.4. Find an index for the composition $\varphi_x \circ \varphi_y$ uniformly in x and y.

4.4. The Recursion Theorem

Kleene's recursion theorem, though provable in only a few lines, is probably the most conceptually challenging theorem in fundamental computability theory. It is extremely useful – vital, in fact – for a large number of proofs in the field. We will discuss this after meeting the theorem and some of its corollaries.

Recall that equality for partial functions is the assertion that when one diverges, so does the other, and when they converge it is to the same output value.

Theorem 4.4.1 (Recursion or Fixed-Point Theorem). *Suppose that f is a total computable function. Then there is a number n such that $\varphi_n = \varphi_{f(n)}$. Moreover, n is computable from an index for f.*

Proof. This is the "magical" proof of the theorem, which will be expanded later in the section. By the s-m-n theorem there is a total computable function $s(x)$ such that for all x and y

$$\varphi_{f(\varphi_x(x))}(y) = \varphi_{s(x)}(y).$$

Let m be any index such that φ_m computes the function s; note that s and hence m are computable from an index for f. Rewriting the statement above yields

$$\varphi_{f(\varphi_x(x))}(y) = \varphi_{\varphi_m(x)}(y).$$

Then, setting $x = m$ and letting $n = \varphi_m(m)$ (which is defined because s is total), we have

$$\varphi_{f(n)}(y) = \varphi_n(y)$$

as required. □

The recursion theorem gives a fixed point at the index level. It need not be the case that $n = f(n)$ – in fact generally that will be false – but n and $f(n)$ are codes for implementations of the same function.

Corollary 4.4.2. *There is some n such that $\varphi_n = \varphi_{n+1}$.*

Corollary 4.4.3. *If f is a total computable function, then there are arbitrarily large numbers n such that $\varphi_{f(n)} = \varphi_n$.*

Corollary 4.4.4. *If $f(x, y)$ is any partial computable function, there is an index e such that $\varphi_e(y) = f(e, y)$.*

Exercise 4.4.5. (i) Prove Corollary 4.4.3. Note that we might obtain a fixed point via another function suitably related to f.

(ii) Prove Corollary 4.4.4. It requires both the recursion theorem and the s-m-n theorem.

Exercise 4.4.6. Prove the following applications of Corollary 4.4.4:

(i) $(\exists n)(\varphi_n(x) = x + n)$

(ii) $(\exists n)(\varphi_n(x) = n)$

(iii) $(\exists n)(\operatorname{dom}\varphi_n = \{n\})$

(iv) $(\exists n)(\operatorname{dom}\varphi_n = \operatorname{rng}\varphi_n = \{n\})$

Exercise 4.4.7. Show that there exists some n such that $\varphi_n(x, y) = xn^y$.

The recursion theorem allows us to prove that a large collection of functions are noncomputable. The set $A \subseteq \mathbb{N}$ is an *index set* if it has the property that if $x \in A$ and $\varphi_x = \varphi_y$, then $y \in A$.

Theorem 4.4.8 (Rice's Theorem). *Suppose that A is an index set not equal to $\emptyset$ or $\mathbb{N}$. Then χ_A is not computable.*

Proof. We work by contradiction, supposing χ_A is computable. Set some $a \in A$ and $b \notin A$ and consider the following function.

$$f(x) = \begin{cases} a & \chi_A(x) = 0 \\ b & \chi_A(x) = 1 \end{cases}$$

Apply the recursion theorem. □

Exercise 4.4.9. Complete the proof of Rice's theorem.

Rice's theorem is a strong statement about our inability to pluck out particular partial computable functions. There is no individual function f for which it is computable to decide whether a given index gives an implementation of f, and in fact there is *no* collection of functions save the empty set and the collection of all functions for which the indices can be distinguished computably.

The recursion theorem also gives results about enumeration of Turing machines. In particular, the least index of each function cannot be listed in order.

Theorem 4.4.10. *Suppose that f is a total increasing function such that*

(i) *if $m \neq n$, then $\varphi_{f(m)} \neq \varphi_{f(n)}$.*

(ii) *$f(n)$ is the least index of the function $\varphi_{f(n)}$.*

Then f is not computable.

Proof. Suppose f satisfies the conditions of the theorem. By (i), f cannot be the identity, so since it is increasing there is some k such that for all $n \geq k$, $f(n) > n$. Therefore by (ii), $\varphi_{f(n)} \neq \varphi_n$ for every $n \geq k$. However, if f is computable, this violates Corollary 4.4.3. □

We now expand the proof of the recursion theorem. One can view an index-level fixed point (n such that $\varphi_n = \varphi_{f(n)}$) as what can be salvaged from a failed attempt at a literal fixed point (n such that $n = f(n)$).

Let $\delta(e)$ be the diagonal function $\varphi_e(e)$. For any partial computable function f, there is some $\hat{e}$ such that $f \circ \delta = \varphi_{\hat{e}}$ (infinitely many such $\hat{e}$, in fact). By definition of δ, $\delta(\hat{e})$ and $f \circ \delta(\hat{e})$ must be equal. However, that gives a literal fixed point for f, and functions certainly exist that have no literal fixed points. In such a case, δ must diverge on $\hat{e}$ (since the theorem assumes f is total), and we do not get our fixed point.

However, if we loosen our requirement to index-level fixed points, we can be a little more subtle, using the *s*-*m*-*n* theorem to define a total function that is "close enough" to δ. Let e be such that

$$\varphi_e(i,x) = \begin{cases} \varphi_{\varphi_i(i)}(x) & \varphi_i(i)\downarrow \\ \uparrow & \text{otherwise.} \end{cases}$$

We could be more concise and say $\varphi_e(i,x) = \varphi_{\varphi_i(i)}(x)$, with the understanding that divergence of the index is divergence of the whole computation. By the *s-m-n* theorem, this is $\varphi_{S_1^1(e,i)}(x)$; let $d(i) = S_1^1(e,i)$ for this e.

Whenever $\delta(i)\downarrow$ (i.e., $\varphi_i(i)\downarrow$), $d(i)$ and $\delta(i)$ index implementations of the same function. However, unlike δ, d is total: when $\delta(i)\uparrow$, $\varphi_{d(i)}(x)\uparrow$ for all x, but $d(i)$ itself is defined.

Let us repeat our original attempt at a fixed point with d in place of δ, now letting $\hat{e}$ be such that $\varphi_{\hat{e}} = f \circ d$. Consider the result of applying $f \circ d$ to $\hat{e}$.

$$\varphi_{f\circ d(\hat{e})}(x) = \varphi_{\varphi_{\hat{e}}(\hat{e})}(x) = \varphi_{\delta(\hat{e})}(x) = \varphi_{d(\hat{e})}(x)$$

The first equality is by choice of $\hat{e}$ and the second is by definition of δ. Notice that since $f \circ d$ and hence $\varphi_{\hat{e}}$ are total, $\delta(\hat{e})$ must converge. The definition of d gives the final equality. We may abbreviate and rewrite a bit to see we have shown

$$\varphi_{f(d(\hat{e}))}(x) = \varphi_{d(\hat{e})}(x),$$

or in other words, that $d(\hat{e})$ is an index-level fixed point for f.

For the theorem's "moreover," note that d was defined independently from f, so with an index for f we can find an index for $f \circ d$ and hence an index-level fixed point for f.

This is extraordinarily useful in constructions. Many of its uses can be summed up as building a Turing machine using the index of the finished machine. The construction will have a line early on like "We construct a partial computable function ψ and assume by the recursion theorem that we have an index e for ψ." This looks insane, but it is completely valid. The construction, which will be computable, is the function for which we seek a fixed point (at the index level). Computability theorists think of a construction as a program. It might have outside components – the statement of the theorem could say "For every function f of this type, ..." – and the construction's if/then statements will give different results depending on which particular f was in play, but such variations will be *uniform*, as described in §4.3. If we give the construction the input e to be interpreted as the index of a partial computable function, it can use e to produce e', which is an index of the function ψ it is trying to build. The recursion theorem says the construction will have a fixed point, some i such that i and i' both index the same function. Furthermore, this fixed point will be *computable* from an index for the construction itself, which by its uniformity has such a well-defined index. That last detail allows the claim to having the index of ψ from the beginning.

4.5. Unsolvability

The word *solvable* is a synonym of *computable* used in particular contexts. In general, it is used to describe the ability to compute a solution to a problem stated not as a function, but as an algebraic or combinatorial question. *Decidable* is another synonym used in the same contexts as solvable.

We have seen an undecidable problem: the halting problem K in §4.1. In this context the problem would be stated as "is there an algorithm to decide, for any e, whether the e^{th} Turing machine halts on input e?"

The celebrity examples are Diophantine equations and the word problem for groups; the latter will be discussed in §4.5.3. As discussed in §1.2, in 1900 Hilbert posed a list of problems and goals to drive mathematical development [37]. The tenth problem on the list asked for an algorithm to determine whether an arbitrary multivariable polynomial equation $P = 0$, where the coefficients of P are all integers, has a solution in integers. At the time, the idea there *may not be* any such algorithm did not occur. In 1970, after a lot of work by a number of mathematicians, Matiyasevich proved the problem is unsolvable [61]. The full proof and story are laid out in a paper by Davis [19].

The method is to show that every Turing machine may be somehow "encoded" in a Diophantine equation so that the equation has an integer solution if and only if the machine halts. The fact that we cannot always tell whether a Turing machine will halt shows we cannot always tell whether a Diophantine equation has an integer solution.

The encoding must be uniform, however, to show undecidability. We obtain a contradiction to the noncomputability of K only if the reduction can be expressed as a single procedure; then the composition of it and the equation-solving algorithm computes K. We omit further details but note that this is the main method to show a problem is undecidable: show you can encode the halting problem (or another problem previously shown to be undecidable) into it.

I recommend Martin Davis's entry in the Handbook of Mathematical Logic [8], on which this section draws heavily, for more on unsolvable combinatorial problems. In §5.3, with more tools and vocabulary, we'll look at mathematical logic.

4.5.1. Halting Problem Relatives. There are a number of problems stemming from the behavior of Turing machines during computation. The most straightforward in terms of undecidability is the *full halting set* $K_0 = \{\langle x, y\rangle : \varphi_x(y)\downarrow\}$. It is clear that if we can compute the characteristic function of K_0 we can also compute the halting problem.

To solve the following exercises, for arbitrary i, produce a Turing machine that has behavior dependent on whether $\varphi_i(i)$ halts or diverges.

Exercise 4.5.1. Show that there is no computable procedure to determine, given e, whether the e^{th} Turing machine halts when started on a blank tape.

Exercise 4.5.2. Show that there is no computable procedure to determine, given e, whether the e^{th} Turing machine, when started on a blank tape, prints a 0 at any time during its computation.

4.5.2. Index Sets. Rice's Theorem 4.4.8 can be viewed as a summary of a large number of undecidability results. It essentially says that determining possession of any nontrivial property of the partial computable functions is unsolvable. Noting that the domain of φ_e is typically denoted W_e, among the index sets we might consider are the following.

$$\text{Fin} = \{e : W_e \text{ is finite}\}$$

$$\text{Inf} = \mathbb{N} - \text{Fin}$$

$$\text{Tot} = \{e : W_e = \mathbb{N}\} = \{e : \varphi_e \text{ is total}\}$$

$$\text{Rec} = \{e : \chi_{W_e} \text{ is computable}\}$$

All of the sets above are not only noncomputable, but in fact are at a higher level of the noncomputability hierarchy than the Halting Set. This notion will be made precise in Chapter 7.

4.5.3. Production Systems. Many undecidability examples are combinatorial in nature, having to do with one's ability to take a string of symbols and transform it into some other string via some finitary procedures. In production systems these procedures are to replace certain subsequences of a string with other subsequences. We use the term *alphabet* for the set of all symbols used and *word* for a string of symbols from the alphabet. If A is an alphabet and w a word all of whose symbols are in A, we call w a word *on* A. We may abbreviate strings of the same symbol using superscripts, and we use λ to denote the empty string.

Example 4.5.3. The most general of productions allows us to replace strings anywhere in a given word, in any number of locations at once. Suppose we're working with the symbols a and b. We might have a rule that says "if a occurs at the beginning of a word, ab^2 in the middle somewhere, and ba at the end, replace them with b, b^2a, and a^2, respectively." We'd abbreviate that to

$$aPab^2Qba \to bPb^2aQa^2,$$

understanding that P and Q are unspecified, possibly empty, strings of a's and b's.

We may apply this production to any word that has the correct original features. For example, we could do the following.

$$a^3b^2a^3b^2a = a(a)ab^2(a^3b)ba \to b(a)b^2a(a^3b)a^2 = bab^2a^4ba^2$$

The parentheses are there for clarity, around the strings that are playing the roles of P and Q. The shortest word this production applies to is a^2b^3a, or $a\lambda ab^2\lambda ba$, which it converts to b^3a^3.

We usually want to restrict the kinds of productions we work with. For example, a *normal* production removes a nonempty sequence from the beginning of a word and adds a nonempty sequence to the end; e.g., $aP \to Pb$.

Definition 4.5.4. Let g, $\hat{g}$ be finite nonempty words. A *semi-Thue production* is a production of the form

$$PgQ \to P\hat{g}Q.$$

When it is understood that the production is semi-Thue we may write simply $g \to \hat{g}$.

Definition 4.5.5. A *semi-Thue system* is a (possibly infinite) collection of semi-Thue productions together with a single nonempty word a, called the *axiom* of the system. If a word w may be produced from a by a finite sequence of applications of productions of the system, then we call w a *theorem* of the system.

Our systems all have computable sets of productions and finite alphabets.

Example 4.5.6. Let the semi-Thue system S be the axiom ab^2ab together with the following productions.

$$a^2 \to bab$$

$$b \to b^3$$

$$aba \to b$$

$$b^2a \to ab$$

From ab^2ab we can get to ab^4ab, ab^2ab^3, or a^2b^2 via a single production application. From a^2b^2 we can get to bab^3 or a^2b^4. We can continue that way potentially indefinitely, generating theorems: it may be that eventually any production applied to any theorem we've already generated produces a theorem we've also already generated, but it is easy to create a semi-Thue system with an infinite list of theorems.

However, if you are presented with a word, how difficult is it to tell whether that word is a theorem of S or another semi-Thue system?

Exercise 4.5.7. Construct a semi-Thue system with infinitely many theorems.

Exercise 4.5.8. Suppose you are given a semi-Thue system S and a word w. If you know w is a theorem of S, describe an algorithm to find a sequence of production applications that generates w.

Exercise 4.5.9. (i) Write an algorithm to determine whether a given word w is a theorem of the semi-Thue system S. Exercise 4.5.8 may be helpful.

(ii) With no special assumptions on S, under what conditions will your algorithm halt?

In fact, given any Turing machine M, we can mimic it with a semi-Thue system S_M. The used portion of M's tape becomes a word, with an additional symbol indicating M's current state inserted just left of the currently scanned tape square. The productions of S_M follow naturally:

(i) Rewriting: if $\langle q_i, S_j, S_k, q_\ell\rangle$ is in M, add the production $q_iS_j \to q_\ell S_k$ to S_M.

(ii) Moving: if $\langle q_i, S_j, R, q_\ell\rangle$ is in M, add the production $q_iS_jS_k \to S_jq_\ell S_k$ to S_M for each S_k. Similarly for $\langle q_i, S_j, L, q_\ell\rangle$.

The axiom of S_M is the initial state followed by the initial contents of the tape, which we will denote m.

The goal is to have a particular word be a theorem of S_M if and only if M halts on the input m. To account for the fact that a Turing machine's tape is infinite but production system strings are finite, we add a special unused symbol (h) to the beginning and end of each word, and add productions to extend the string on each end as needed. We also add special state-like symbols q, q' that are switched into when we hit a dead end: For every state q_i and symbol S_j that do not begin any quadruple of M, add the production $q_iS_j \to qS_j$. Once we're in q we delete symbols to the right: for every symbol S_i, S_M contains $qS_i \to q$. When we hit the right end, switch into q': $qh \to q'h$. Finally, delete symbols to the left: $S_iq' \to q'$. Ultimately, if M halts on m, our production system with axiom hq_0mh will produce the theorem $hq'h$, and not otherwise.

We have "proved" the following theorem:

Theorem 4.5.10. *It is not possible in general to decide whether a word is a theorem of a semi-Thue system.*

Exercise 4.5.11. How does the mimicry of Turing machines by semi-Thue systems give us Theorem 4.5.10?

Exercise 4.5.12. Write a proof of Theorem 4.5.10. In particular, fill in the details of the symbol h, formally verify that the construction works, and include the explanation of Exercise 4.5.11.

Note that not every individual semi-Thue system has an unsolvable word problem. If the alphabet is small and the productions few and simple, it could be quite easy to algorithmically decide whether any given word is a theorem. In particular, any semi-Thue system with only finitely many theorems is decidable, or for which only finitely many of the words on the alphabet are not theorems.

Exercise 4.5.13. Construct a semi-Thue system with a decidable word problem such that infinitely many of the words on its alphabet are theorems, and infinitely many are non-theorems.

However, there are individual semi-Thue systems with undecidable word problems, such as the system built from a universal Turing machine by the method above.

One of the first proofs of unsolvability of a problem from classical mathematics was for *Thue systems*, due to Post [74] and Markov [59] independently in 1947. A Thue system S is a semi-Thue system closed under inverse productions: if $g \to \hat{g} \in S$, then $\hat{g} \to g \in S$ as well. The word problem is defined identically to those for semi-Thue systems: produce a procedure that, given a Thue system S and a word w, determines whether w is a theorem of S.

Exercise 4.5.14. Prove that undecidability of the word problem for Thue systems implies undecidability of the word problem for semi-Thue systems.

The *word problem for groups* is also closely related. For the rest of this subsection I assume some algebra, and later topology. Certain Thue systems correspond to presentations of groups by generators and relations. The alphabet is almost the generator set, and the productions are almost the relations. The generators of a group may be given as a, b, c with the understanding that inverse elements a^{-1}, b^{-1}, c^{-1} and relations $aa^{-1} = \lambda, a^{-1}a = \lambda$, etc., are also included. A Thue system is a *group system* if the alphabet may be numbered as $a_1, a_2, \ldots, a_{2n}$ in such a way that $a_{2i}a_{2i-1} \to \lambda$ and $a_{2i-1}a_{2i} \to \lambda$ are productions for $1 \leq i \leq n$. This gives a correspondence between half of the alphabet and the generators of the group, and the other half of the alphabet and the inverse generators (a Thue system that is not a group system corresponds to a semigroup). The fact that the system is Thue, not semi-Thue, gives the rest of the correspondence between relations and productions: the relation $ab = ba$, for instance, corresponds to the pair of productions $ab \to ba, ba \to ab$.

Groups are arguably the fundamental object of algebra, but it is easier to see the consequences of this result in algebraic topology. Every topological space has a group associated with it called the fundamental group, and the word problem (with axiom λ) in the setting of fundamental groups is the problem of determining whether any given closed loop is contractible to a point. In fact, the problem of determining whether there exists any word in the group that is not equivalent to the empty word is unsolvable, which in the context of topology is the question of whether a given space is simply connected. Of course, most standard examples of topological spaces have fundamental groups with solvable word problems, but the undecidability result says this basic piece of information is, in generality, impossible to know.

4.5.4. Post Correspondence. The Post correspondence problem is included because it is significant for theoretical computer science. Because of its simplicity, it is used often enough as a stand-in for the Halting Problem in proving undecidability to be known simply by the acronym PCP. For a computer science-style treatment see, for example, §4.7 of Gurari [34]. We will prove its undecidability as in Davis [8], differently from Post's original work [73].

Definition 4.5.15. A *Post correspondence system* consists of an alphabet A and a finite set of ordered pairs $\langle h_i, k_i \rangle$, $1 \leq i \leq m$, of words on A. A word u on A is called a *solution* of the system if, for some sequence $1 \le i_1, i_2, \ldots, i_n \le m$ (the i_j need not be distinct, and n may be any value ≥ 1), we have $u = h_{i_1} h_{i_2} \cdots h_{i_n} = k_{i_1} k_{i_2} \cdots k_{i_n}$.

That is, given two lists of m words, $\{h_1, \ldots, h_m\}$ and $\{k_1, \ldots, k_m\}$, we want to determine whether any concatenation of words from the h list is equal to the concatenation of the *corresponding* words from the k list. A solution is such a concatenation.

Example 4.5.16. The word *aaabbabaaaba* is a solution to the system

$$\{\langle a^2, a^3\rangle, \langle b, ab\rangle, \langle aba, ba\rangle, \langle ab^3, b^4\rangle, \langle ab^2a, b^2\rangle\},$$

as shown by the two *decompositions*

$$\begin{array}{c|c|c|c|c} aa & abba & b & aa & aba \\ aaa & bb & ab & aaa & ba \end{array}$$

In fact, the segments *aaabbab* and *aaaba* are individually solutions as well.

Given a semi-Thue system S and a word v, we can construct a Post correspondence system that has a solution if and only if v is a theorem of S. Then we can conclude the following.

Theorem 4.5.17. *There is no algorithm for determining whether a given arbitrary Post correspondence system has a solution.*

Proof. Let S be a semi-Thue system on alphabet $A = \{a_1, \ldots, a_n\}$ with axiom u, and let v be a word on A. We construct a Post correspondence system P such that P has a solution if and only if v is a theorem of S. The alphabet of P is

$$B = \{a_1, \ldots, a_n, a_1', \ldots, a_n', [,], \triangleright, \triangleright'\},$$

with $2n + 4$ symbols. For any word w on A, write w' for the word on B obtained from w by replacing each symbol s of w by s'.

Suppose the productions of S are $g_i \to \hat{g}_i$, $1 \le i \le k$, and assume these include the n identity productions $a_i \to a_i$, $1 \le i \le n$. Note this is without loss of generality as the identity productions do not change the set of theorems of S. However, we may now assert that v is a theorem of S if and only if we can write $u = u_1 \to u_2 \to \cdots \to u_m = v$ for some *odd* m.

Let P consist of the following pairs:

$$\langle [u \triangleright, [\rangle, \langle \triangleright, \triangleright' \rangle, \langle \triangleright', \triangleright \rangle, \langle], \triangleright' v] \rangle$$

$$\left.\begin{array}{c} \langle \hat{g}_j, g_j' \rangle \\ \langle \hat{g}_j', g_j \rangle \end{array}\right\} \text{ for } 1 \le j \le k$$

Suppose v is a theorem of S, and $u = u_1 \to u_2 \to \cdots \to u_m = v$, where m is odd, is a production sequence demonstrating that. Then the word

$$w = [u_1 \triangleright u_2' \triangleright' u_3 \triangleright \cdots \triangleright u_{m-1}' \triangleright' u_m]$$

is a solution of P, with the decompositions

$$\begin{array}{c|c|c|c|c|c|c} [u_1 \triangleright & u_2' & \triangleright' & u_3 & \triangleright & \cdots &] \\ [& u_1 & \triangleright & u_2' & \triangleright' & \cdots & \triangleright' u_m], \end{array}$$

where u_2' corresponds to u_1 by the concatenation of three pairs: we can write $u_1 = rg_js$, $u_2 = r\bar{g}_js$ for some $1 \le j \le k$. Then $u_2' = r'\hat{g}_j's'$ and the productions $r \to r$, $s \to s$, and $g_j \to \hat{g}_j$ give the ordered pairs showing the correspondence.

For the converse, we show that any solution is a derivation or concatenation of derivations of v from u. First notice that any solution must begin with [and end with]; we forced this by adding $'$ to the symbols in half of every pair. The symbol at the beginning of any solution must match, and the only pair for which the first symbol matches is $\langle [u\triangleright, [\rangle$. Likewise, the only symbol that ends both elements of a pair of P is].

Let w be a solution with only one], possibly obtained by truncating a given solution after the first]. Then

$$w = [u \triangleright \cdots \triangleright' v],$$

and our decompositions are forced at the ends to be the pairs $\langle [u\triangleright, [\rangle$, $\langle], \triangleright' v] \rangle$. This gives us an initial correspondence.

$$\begin{array}{c|c|c} [u \triangleright & \cdots \triangleright' v &] \\ [& u \triangleright \cdots & \triangleright' v] \end{array}$$

Since w is a solution, we must have u corresponding to some r' and v to some s', where $u \to r$ and $s \to v$. Then the $\triangleright$ and $\triangleright'$

must correspond to a $\triangleright'$ and $\triangleright$, respectively. If the $\triangleright$ and $\triangleright'$ do not correspond to each other, then we have the following.

$$\begin{array}{r|c|c|c|c|c|c}
[u\triangleright & r & \triangleright' & \cdots\triangleright s & \triangleright' & v &] \\
[& u & \triangleright & r\triangleright'\cdots & \triangleright & s & \triangleright' v]
\end{array}$$

Iterating this procedure, we see that w shows $u \to v$.

Hence, P has a solution if and only if v is a theorem of S; if we can always decide whether a Post correspondence problem has a solution, we have contradicted Theorem 4.5.10. □

As a final note, we point out that this undecidability result is for arbitrary Post correspondence systems, just as the undecidability of the previous subsection was for arbitrary semi-Thue systems. We may get decidability results by restricting the size of the alphabet or the number of pairs $\langle h_i, k_i\rangle$. If we restrict to alphabets with only one symbol but any number of pairs, then the Post correspondence problem is decidable. If we allow two symbols and any number of pairs, it is undecidable. If we restrict to only one pair or two pairs of words, the problem is decidable regardless of the number of symbols [24], and at 7 pairs it is undecidable [62]. Between three and six pairs inclusive, the question is still open. The *bounded Post correspondence problem*, where the alphabet and number of sequences are finite and the number of pairs making up the solution string is bounded by a value no larger than the total number of sequences (repeats still allowed), is decidable but NP-complete [16].

Chapter 5

Computing and Enumerating Sets

We've talked about computability for functions; now we generalize to sets. First, we address a practical matter.

5.1. Dovetailing

Suppose we have a partial function $f : \mathbb{N} \to \mathbb{N}$, and we would like to know what it does. If we knew f were total, we could find $f(0)$, then $f(1)$, then $f(2)$, and so on. However, since f is partial, at some point we're going to get hung up and not find an output. This could even happen at $f(0)$, and then we would know nothing. In order to do much of anything with partial functions we need a way to bypass this problem and obtain the outputs that do exist.

The procedure used is called *dovetailing*, a term that comes from carpentry. A dovetailed joint is made by notching the end of each board so they can interlock (the notches and tabs are trapezoidal, reminiscent of the tail of a bird seen from above). In computability, we interleave portions of the computations; that is, we gradually fold in more steps of more computations. It is directly analogous to the ordering that gives the pairing function (see Figure 3.1).

It is easiest to imagine this process in terms of Turing machines, which clearly have step-by-step procedures. We run one step of the computation of $f(0)$. If it halts, then we know $f(0)$. Either way, we run two steps of the computation of $f(1)$, and if necessary, two steps of the computation of $f(0)$. Step (or *stage*) n of this procedure is to run the computations of $f(0)$ through $f(n-1)$ each for n steps, minus any we've already seen halt (though since they only add finitely many steps, there's no harm in including them[1]). Since every computation that halts must halt in finitely many stages, each element of f's domain will eventually give its output. Any collection of computations we can index can be dovetailed.

The state of the computation after s steps of computation is denoted with a subscript s: $f_s(n)$. We may use our halting and diverging notation: $f_s(n)\downarrow$ or $f_s(n)\uparrow$. Note that $f_s(n)\uparrow$ does not imply $f(n)\uparrow$; it could be that we simply need to run more steps. Likewise, $f_s(n)\downarrow$ does not mean $f_{s-1}(n)\uparrow$; it means convergence has happened at some step no later than s.

If we are drawing from an indexed list of functions, the stage may share the subscript with the index: $\varphi_{e,s}(n)$. Sometimes the stage number is put into brackets at the end of the function notation, as $\varphi_e(n)[s]$; this will be useful when more than just the function is approximated, as in §6.1. In this case the up or down arrow goes after everything: $\varphi_e(n)[s]\downarrow$.

5.2. Computing and Enumerating

Recall that the characteristic function of a set A (Definition 3.1.1), denoted χ_A or simply A, is the function outputting 1 when the input is a member of A and 0 otherwise. It is total, but not necessarily computable.

Definition 5.2.1. A set is *computable* (or *recursive*) if its characteristic function is computable.

[1]Likewise, we could record our stopping point and just run one more step of each computation plus the first step of an additional computation each time instead of starting from the beginning, but there is no harm in starting over each time. Remember that *computable* does not imply *feasible*. As another side note, this procedure would cause $f(i)$ to have its $(n-i+1)^{st}$ step run at stage n, making the dovetailing truly diagonal.

The word *effective* is often used as a synonym for computable and recursive, but only in the context of procedures (you might say a given construction is effective instead of saying it is recursive or computable; it would be strange to say a set is effective). Note, however, that in the literature they are not always exactly synonymous! Exercise 5.2.21 introduces the notion of *computably inseparable* sets. While *recursively inseparable* is an equivalent term, the *effectively inseparable* sets are a different collection.

A computable characteristic function is simply a computable procedure that will answer "is n in A?" for any n, correctly and in finite time.

Claim 5.2.2. (i) *The complement of a computable set is computable.*

(ii) *Any finite set is computable.*

Proof. (i) Simply note $\chi_{\overline{A}} = 1 - \chi_A$, so the functions are both computable or both noncomputable.

(ii) A finite set may be "hard-coded" into a Turing machine, so the machine has instructions which essentially say "if the input is one of these numbers, output 1; else output 0."

□

Part (ii) is the heart of most nonuniformity. Any finite amount of information is computable, so we may assume it without violating the computability of a procedure. However, to make such an assumption infinitely many times is typically not computable. This is discussed again at the end of §5.4.

Rice's Theorem 4.4.8 gives a large number of noncomputable sets: all nontrivial index sets. Not all noncomputable sets are created equal, of course, and in particular we pluck out sets that are *computably approximable*, in the following sense.

Definition 5.2.3. A set is *computably enumerable* (or *recursively enumerable*, abbreviated as *c.e.* or *r.e.*) if there is a computable procedure to list its elements (possibly out of order and with repeats).

That definition is perhaps a little nebulous. Here are some additional characterizations:

Proposition 5.2.4. *Given a set A, the following are equivalent.*

(i) *A is c.e.*

(ii) *A is the domain of a partial computable function.*

(iii) *A is the range of a partial computable function.*

(iv) *$A = \emptyset$ or A is the range of a total computable function.*

(v) *There is a total computable function $f(x, s)$ such that for every x, $f(x, 0) = 0$, there is at most one s such that $f(x, s+1) \neq f(x, s)$, and $\lim_s f(x, s) = \chi_A(x)$.*

(vi) *There is a computable sequence of finite sets A_s, $s \in \mathbb{N}$, such that for all s, $A_s \subseteq A_{s+1}$, and $A = \bigcup_s A_s$.*

Notice that property (iv) is almost effective countability, as in §3.4, but not quite.

Proof. The proofs that (ii), (iii), and (iv) imply (i) are essentially all the same. Dovetail all the $\varphi_e(x)$ computations, and whenever you see one converge, enumerate the preimage or the image involved depending on which case you're in. This is a computable procedure so the set produced will be computably enumerable.

(i)⇒(ii): Given A c.e., we define the following.

$$\psi(n) = \begin{cases} 1 & n \in A \\ \uparrow & n \notin A \end{cases}$$

We must show this is partial computable. To compute $\psi(n)$, begin enumerating A, a computable procedure. If n ever shows up, at that point output 1. Otherwise the computation never converges.

(i)⇒(iii): Again, given A c.e., note that we can think of its elements as having an order assigned to them by the enumeration: the first to be enumerated, the second to be enumerated, etc. (This will in general be different from their order by size.) Define the function using that:

$$\varphi(n) = (n+1)^{st} \text{ element to be enumerated in } A.$$

(We use $n+1$ to give 0 an image; this is not important here but we shall use it in the next part of the proof.) If A is finite, the enumeration will cease adding new elements and φ will be undefined from some point on.

(i)$\Rightarrow$(iv): Suppose we have a nonempty c.e. set A. If A is infinite, the function φ from the previous paragraph is total, and A is its range. If A is finite, it is actually computable, so we may define

$$\hat{\varphi}(n) = \begin{cases} \varphi(n) & n < |A| \\ \varphi(0) & n \geq |A|. \end{cases}$$

$\square$

Exercise 5.2.5. Prove the equivalence of part (v) with the rest of Proposition 5.2.4. Advice: prove (i) $\Leftrightarrow$ (v) directly, and remember that in defining f you simply need to give a procedure that takes an arbitrary pair x, s and computes an answer in finite time.

Exercise 5.2.6. Prove the equivalence of part (vi) with the rest of Proposition 5.2.4. Again, prove (i) $\Leftrightarrow$ (vi) directly, but this is simpler than Exercise 5.2.5. The sequence A_s is computable if there is a total computable function $f(s)$ giving the code of A_s for every s.

Exercise 5.2.7. Prove that every infinite c.e. set is the range of a *one to one* total computable function. This closes the gap in Proposition 5.2.4 (iv) with effective countability. Note that an enumeration of a set is allowed to list elements multiple times.

Every computable set is computably enumerable, but the reverse is not true. For example, we've seen that the halting set

$$K = \{e : \varphi_e(e)\downarrow\}$$

is c.e. (it is the domain of the diagonal function $\delta(e) = \varphi_e(e)$) but is not computable. What's the difference? Intuitively speaking, it's the waiting. If A is being enumerated and we have not yet seen 5, we do not know if that is because 5 is not an element of A or because it's going to be enumerated later. If we knew how long we had to wait before a number would be enumerated, and if it hadn't by then it never would be, then A would actually be computable: To find $\chi_A(n)$, enumerate A until you have waited the prescribed time. If n hasn't shown up in the enumeration by then, it's not in A, so output 0. If it has shown up, output 1. In the context of effective countability, to determine whether n is in the set, you can compute $f(0)$, $f(1)$, $f(2)$, etc., for f the decoding function, but you will only get an answer if some $f(k)$ actually is equal to n.

Conversely, we can show a c.e. set is computable if we can cap the wait time.

Exercise 5.2.8. Prove that an infinite set is computable if and only if it can be computably enumerated in increasing order (that is, it is the range of a *monotone* total computable function).

Exercise 5.2.9. Prove that if A is c.e., A is computable if and only if $\overline{A}$ is c.e.

As in §5.1, we use subscripts to denote the state of affairs at a finite stage. In this case the stage-s version of A is the finite set A_s from Proposition 5.2.4 (vi). In fact, formally, we computably build finite sets $A_0 \subseteq A_1 \subseteq A_2 \subseteq \ldots$, and then define A to be $\bigcup_s A_s$.

It is straightforward to see that there are infinitely many sets that are not even c.e., much less computable. It is traditional to denote the domain of φ_e by W_e (and hence the stage-s approximation by $W_{e,s}$). The c.e. (including computable) sets are all listed out in the enumeration $W_0, W_1, W_2, \ldots$, which is a countable collection of sets. However, the power set of $\mathbb{N}$, which is the set of all sets of natural numbers, is uncountable. Therefore, in fact, there are not just infinitely many, but uncountably many sets that are not computably enumerable. These include all of the index set examples given in §4.5.

Exercise 5.2.10. Use Exercise 5.2.9 and the enumeration of c.e. sets, $\{W_e\}_{e\in\mathbb{N}}$, to give an alternate proof of the noncomputability of K.

Exercise 5.2.11. Prove that if A is computable and $B \subseteq A$ is c.e., then B is computable if and only if $A - B$ is c.e. Prove that if A is only c.e., $B \subseteq A$ c.e., we cannot conclude that B is computable even if $A - B$ is *computable*.

Exercise 5.2.12. Show that a function $f : \mathbb{N} \to \mathbb{N}$ is partial computable if and only if its graph $G = \{\langle x, y\rangle : f(x) = y\}$ is a computably enumerable set.

Exercise 5.2.13. Prove the *reduction property*: given any two c.e. sets A, B, there are c.e. sets $\hat{A} \subseteq A$, $\hat{B} \subseteq B$ such that $\hat{A} \cap \hat{B} = \emptyset$ and $\hat{A} \cup \hat{B} = A \cup B$.

Exercise 5.2.14. Prove that the set $\{\langle e, x, s\rangle : \varphi_{e,s}(x)\downarrow\}$ is computable.

Exercise 5.2.15. Prove that the set $\{\langle e, x\rangle : x \in W_e\}$ is c.e.; that is, prove that the c.e. sets are *uniformly enumerable.*

Exercise 5.2.16. Prove that the collection $\{(A_n, B_n)\}_{n\in\mathbb{N}}$ of all pairs of disjoint c.e. sets is uniformly enumerable.

Suggestion: enumerate triples $\langle n, i, x\rangle$, where n gives the pair of sets, $i \in \{0, 1\}$, and x is in A if $i = 0$ and B if $i = 1$. Alternatively, show that reduction, as in Exercise 5.2.13, may be done uniformly.

"All pairs of disjoint c.e. sets" means all possible pairings of c.e. sets such that the sets are disjoint. Note that as with the c.e. sets individually, the enumeration will contain repeats.

Exercise 5.2.17. Show that any infinite c.e. set contains an infinite computable subset.

Exercise 5.2.18. Show that any infinite set contains a noncomputable subset.

Exercise 5.2.19. Prove that if A and B are both computable (respectively, c.e.), then the following sets are also computable (c.e.).

(i) $A \cup B$

(ii) $A \cap B$

(iii) $A \oplus B := \{2n : n \in A\} \cup \{2n + 1 : n \in B\}$, the *disjoint union* or *join.*

Exercise 5.2.20. Show that if $A\oplus B$, as defined above, is computable (respectively, c.e.), then A and B are both computable (c.e.).

Exercise 5.2.21. Two c.e. sets A, B are *computably separable* if there is a computable set C that contains A and is disjoint from B. They are *computably inseparable* otherwise.

(i) Let $A = \{x : \varphi_x(x)\downarrow= 0\}$ and $B = \{x : \varphi_x(x)\downarrow= 1\}$. Show that A and B are computably inseparable.

(ii) Let $\{(A_n, B_n)\}_{n\in\mathbb{N}}$ be the enumeration of all disjoint pairs of c.e. sets as in Exercise 5.2.16. Let $x \in A$ iff $x \in A_x$ and $x \in B$ iff $x \in B_x$, and show that A and B are computably inseparable. Hint: What if C were one of the B_n?

Exercise 5.2.22. (i) Show that if A is computably enumerable, the union $B = \bigcup_{e\in A} W_e$ is computably enumerable.

(ii) If A is computable, is B computable?

(iii) Can you make any claims about $C = \bigcap_{e\in A} W_e$ given the computability or enumerability of A?

Exercise 5.2.23. (i) Let X be a computable set, and define

$$A = \{n \in \mathbb{N} : (\exists m \in \mathbb{N})(\langle n, m\rangle \in X)\}.$$

$$B = \{n \in \mathbb{N} : (\forall m \in \mathbb{N})(\langle n, m\rangle \in X)\}.$$

Show that A is c.e. and B is co-c.e. (i.e., $\overline{B}$ is c.e.).

(ii) Let X and A be as above and assume A is noncomputable. Prove that for every total computable function f, there is some $n \in A$ such that $(\forall m)(\langle n, m\rangle \in X \;\Rightarrow\; m > f(n))$.

Exercise 5.2.24. Recall from §4.5 that the index set of all total functions is

$$\text{Tot} = \{e : W_e = \mathbb{N}\} = \{e : \varphi_e \text{ is total}\}.$$

Prove that Tot is not computably enumerable.

Hint: Reread Theorem 5.2.4 (iv) and §4.2.

5.3. Aside: Enumeration and Incompleteness

Gödel's incompleteness theorem [30] is often summarized as "there are true but unprovable statements." This is an imprecise formulation, seemingly in conflict with his completeness theorem, which could be summarized as "a statement is true if and only if it is provable." The meaning of "true" in the two statements is different, and in this section, going light on details, we will elucidate the difference and discuss these theorems' relationship to the existence of noncomputable sets. For more details of incompleteness, I recommend Murawski [65];

more details of first-order logic appear in §9.3. This section arguably belongs in §4.5 but we can do more with it with §5.2 behind us.

A *language* $\mathcal{L}$ is a collection of symbols representing constants, functions, and relations (the latter two come with a defined *arity*, or number of inputs). Formulas in $\mathcal{L}$ may use those symbols as well as variables and the logical symbols we met in §2.1 (e.g., $\neg$, $\forall$).

We have two directions to go from here. One is *syntactic*, where we discuss logical deductions (proofs) of sentences in $\mathcal{L}$. The other is *semantic*, where we define $\mathcal{L}$-structures, each of which is a set of elements together with *interpretations* of the constant, function, and relation symbols of $\mathcal{L}$.

The important fact about deduction for our discussion is that it is computable to determine whether a sequence of characters is a formula in $\mathcal{L}$, and whether a sequence of formulas is a legitimate proof in $\mathcal{L}$. In fact, the proofs are an effectively countable set, so we can computably enumerate all the formulas that are provable in $\mathcal{L}$ by taking each proof in turn and outputting its conclusion.[2] The *soundness theorem* says that any formula that appears on this list must be true under the interpretation of any $\mathcal{L}$-structure.

Call a formula *valid* if its interpretation in any $\mathcal{L}$-structure is always true. A priori the set of valid formulas may properly contain the provable formulas, but Gödel's completeness theorem says they are equal. Completeness plus soundness is properly written "a formula is valid if and only if it is provable."

For any fixed language including at least equality and another binary relation symbol, the *validity problem* (or provability problem) is undecidable. That is, the set of valid formulas is c.e. but not computable. We will give the proof of a version of this after some more definitions.

In any language $\mathcal{L}$ we may take a set T of $\mathcal{L}$-sentences as axioms, premises from which to prove other sentences. An $\mathcal{L}$-structure $\mathcal{M}$ is a *model* of T if all of the sentences of T are true when interpreted in $\mathcal{M}$. Completeness still holds: a sentence is true in all models of T if and only if it is provable from T. A key observation is that it is possible

[2]Technically, we need to require the language be computable, and not consist of, say, relations R_i with arity $f(i)$ for a noncomputable function f.

to have an $\mathcal{L}$-sentence φ such that neither φ nor $\neg\varphi$ is provable from T; such a sentence is called *independent* (of T).

One significant set of axioms is *Robinson arithmetic*, in the language $(0, S, +, \cdot, =, <)$, where 0 is a constant symbol, S a unary function symbol, $+$ and $\cdot$ binary function symbols, and $=$ and $<$ binary relation symbols. The axioms, described in §9.3.1, are designed to make these work as they do in $\mathbb{N}$, with S the successor function.

The *standard model* of Robinson arithmetic is $\mathcal{N}$, the structure with universe $\mathbb{N}$ and standard interpretations of the language symbols. However, it is possible to have other models of Robinson arithmetic; these models will all look like $\mathcal{N}$ at first but will have additional *nonstandard* elements that are larger than every successor of 0.

Robinson arithmetic has independent sentences. We can prove this, as Church [15] and Turing [85] did, by showing that the provability problem is undecidable. The set of provable sentences P is still c.e., because the axioms of Robinson arithmetic are computable. We may enumerate the set R of refutable sentences by listing $\neg\varphi$ whenever a new φ shows up in P. These sets are disjoint, and independent sentences exist if and only if P and R are not complements, which we will show by proving that their union is not computable.

For f a partial computable function and any number x, we may code the statement $(\exists y)(f(x) = y)$ (uniformly in f and x) as a statement of Robinson arithmetic, which in fact will still have the shape $(\exists y)\theta_f(x, y)$, where θ_f is quantifier-free. Now, if $f(x)$ really is defined, there is some natural number n such that $\theta_f(x, n)$ is true. It can be shown that every quantifier-free statement is provable or refutable in Robinson arithmetic, which means $\theta_f(x, n)$ and thus $(\exists y)\theta_f(x, y)$ are provable. Conversely, if $(\exists y)\theta_f(x, y)$ is provable, then it is true in all models; in particular, it is true in $\mathcal{N}$. Hence there is some natural number n such that $\theta_f(x, n)$ holds and $f(x)$ actually converges.

The discussion above shows that if $x \in \operatorname{dom} f$, the sentence $(\exists y)\theta_f(x, y)$ is in P. If $x \notin \operatorname{dom} f$, it is possible that $(\exists y)\theta_f(x, y)$ is in R or that it is in neither P nor R. Suppose for a contradiction that we may computably determine whether a logical statement is in $P \cup R$. To compute the halting problem, we simply ask whether

$(\exists y)\theta_f(x,y)$ is in $P \cup R$ for x the code of f. If no, we know $f(x)$ is undefined. If yes, we enumerate P and R until $(\exists y)\theta_f(x,y)$ shows up in one or the other. Since the halting problem is noncomputable, $P \cup R$ must be noncomputable. In particular, $P \cup R$ cannot be all logical statements, and there must be statements that are neither provable nor refutable: the axioms are *incomplete*.

So why is incompleteness phrased as "true but unprovable" sentences? Gödel showed a sentence exists that is not valid but is true in $\mathcal{N}$. His version was a formalization of $\psi =$ "there is no proof of ψ." Since $\neg\psi =$ "there is a proof of ψ," we get a contradiction if ψ is false. Therefore ψ must be true, but it is not deducible from the axioms of Robinson arithmetic. How do we resolve this with Gödel's completeness theorem?

The problem is nonstandard models, which we will now thoroughly anthropomorphize. A model in which ψ is false (which must exist, by completeness) believes it has a proof of ψ, but if you ask for the code of the proof, it will produce a nonstandard number. This is exactly analogous to claiming a computation halts, but requires $\infty + 5$ steps to do so. In "reality," a.k.a. the standard model, ψ is true and a computation either halts in finite time or never halts at all. That is the sense of "true" in the "true but unprovable" formulation.

This gives results for more than just Robinson arithmetic, however. Any computable, consistent extension of Robinson arithmetic is also incomplete, where "extension" means adding more axioms. More computable machinery cannot help solve an undecidable problem or erase the capability for self-reference that Gödel used.

5.4. Enumerating Noncomputable Sets

In order for a set A to be noncomputable, its characteristic function must be nonequal to every total computable function. If we do not care whether A is computably enumerable, we may decide the inequalities in any number of ways. If we would like A to be c.e., it is best to work diagonally, making A such that $\chi_A(e) \neq \varphi_e(e)$ (though of course it need not be literally diagonal).

A brief description would be "Let $\chi_A(e) = 1$ if $\varphi_e(e) = 0$, and otherwise let it be 0." For ease of generalization we'll expand this out into an explicit procedure that enumerates A as we learn more about the $\varphi_e(e)$ computations.

We can think of this definition as an infinite collection of *requirements*.

$$R_e : \chi_A(e) \neq \varphi_e(e)$$

We win each individual requirement if either $\varphi_e(e)\uparrow$, or $\varphi_e(e)\downarrow$ but is different from $\chi_A(e)$. We must also make sure A is c.e., which is a single requirement that permeates the construction (a *global* requirement).

To make A c.e., we put elements into it but never take them out, and we ensure every step of the construction is computable. The construction itself is the computable procedure that enumerates A.

Meeting each R_e will be *local*; none of the requirements will interact with any others. We dovetail the computations in question as in §5.1, so we will eventually see the end of any convergent computation. If $\varphi_e(e)\downarrow = 0$ at stage s we put e into A at that stage. If we never see that, meaning $\varphi_e(e)\uparrow$ or $\varphi_e(e)\downarrow \neq 0$, we keep e out; that's the whole construction.

Exercise 5.4.1. Prove that the procedure above produces a noncomputable c.e. set.

A particular kind of noncomputable set that is often used is a simple set.

Definition 5.4.2. A c.e. set A is *simple* if $\overline{A}$ is infinite but contains no infinite c.e. subsets. $\overline{A}$ is called *immune*.

If W_e is infinite, it must have nonempty intersection with A, but there still has to be enough outside of A that $\overline{A}$ is infinite. Note that sets with an infinite or finite complement are often called *coinfinite* or *cofinite*, respectively.

Exercise 5.4.3. (i) Prove that if A is simple, it is not computable.

(ii) Prove that if A is simple and W_e is infinite, $A \cap W_e$ must be infinite (not just nonempty).

Exercise 5.4.4. Prove that a coinfinite c.e. set is simple if and only if it is not contained in any coinfinite computable set.

Exercise 5.4.5. Prove that if A and B are simple, $A \cap B$ is simple and $A \cup B$ is either simple or cofinite.

We now discuss the construction of a simple set. This perhaps seems technical, but it is the most common way to force a set to be noncomputable in modern constructions (we often want to construct sets with certain properties and use construction "modules" to do so; the simplicity module is the most common for noncomputability, because it is easier to combine with other modules than the previous method).

As before, we have an infinite collection of local requirements.

$$R_e : (|W_e| = \infty) \Rightarrow (A \cap W_e \neq \emptyset)$$

Additionally we have two global requirements.

$$A \text{ is c.e.}$$
$$|\overline{A}| = \infty$$

As before, to make sure A is c.e., we will enumerate it during the construction and make sure that every step of the construction is computable.

To meet R_e while maintaining the size of $\overline{A}$, we look for $n > 2e$ such that $n \in W_e$. When we find one, we enumerate n into A. Then we stop looking for elements of W_e to put into A (the requirement R_e is *satisfied*).

Since W_e may be finite, we have to dovetail the search as in §5.1, so at stage s we look at $W_{e,s}$ for $e < s$ such that $W_{e,s} \cap A_s = \emptyset$.

Why does this work?

- As discussed before, A is c.e. because the construction is computable, and numbers are only put into A, never taken out.
- $\overline{A}$ is infinite because only the k-many requirements $R_0, \ldots,$ R_{k-1} are allowed to put numbers below $2k$ into A for any k. Each enumerates at most one, leaving at least k of those numbers in $\overline{A}$.

- For each W_e that is infinite, there must be some element $x > 2e$ in W_e. Eventually s is big enough that (a) we are considering W_e, and (b) such an x is in $W_{e,s}$. At that point we will put x into A and R_e will be satisfied forever after.

One thing to note: we cannot tell whether any given W_e will be finite or infinite. Although for a single e, or finitely many, we may assume we know whether W_e is finite or infinite, that assumption is nonuniform, as shown by Rice's theorem (in particular, the index set Fin). In fact, even if we know W_e is finite, we won't know when its enumeration finishes without knowing its size. Because of this, we may act on behalf of some finite sets W_e unnecessarily. That's okay, though, because we set up the $2e$ safeguard to make sure we never put so much into A that $\overline{A}$ becomes finite, and that would be the only way that extra elements in A could hurt the construction.

More complicated requirement-based constructions are explored in §6.2.

Chapter 6

Turing Reduction and Post's Problem

6.1. Reducibility of Sets

In the last two chapters, we have used the idea of computing the solution to a problem or the outcome of a function via a different problem or function. Here we make that notion of *relative computability* precise. In §4.5, we showed that certain problems are "at least as noncomputable" as the halting problem by showing that access to a decision procedure for the given problem would allow us to compute the halting problem or, in other words, by *reducing* the halting problem to the given problem. We begin by formalizing the notion of "access."

Definition 6.1.1. An *oracle Turing machine with oracle A* is a Turing machine that is allowed to ask a finite number of questions of the form "is n in A?" during the course of a computation.

The restriction to only finitely many questions is so the computation remains finite. We may think of oracle machines as computers with CD drives. We pop the CD of A into the drive, and the machine can look up finitely many bits from the CD during its computation on input n. Another way to think of it would be a Turing machine with an additional internal tape that is read-only and pre-printed with the characteristic function of A. That perhaps clarifies how we might define oracle Turing machines formally and code them, as well as making very concrete the fact that only finitely many questions may be asked of the oracle.

The number and kind of questions the machine asks may vary with not only the input value, but also the answers it gets along the way; i.e., with the oracle. However, once again we must have uniformity: every index must code a function that is well defined regardless of the oracle. For example, perhaps we would like to define the characteristic function of the set of all numbers that have factorizations consisting entirely of elements from the halting set K. Although our function must be defined for any oracle, it is not required that it be a characteristic function at all if the oracle is not K, much less a characteristic function of the specified sort. However, it is no more difficult, and in fact probably easier, to define a function that always gives the characteristic function of the set of products of elements of the oracle.

$$f^X(n) = \begin{cases} 1 & n \text{ factors into elements of } X \\ 0 & \text{otherwise} \end{cases}$$

To implement f as an oracle Turing machine, the procedure must be defined independently of n and X. One way to do so would be the following.

- Given input n, ask if $n \in X$.
  - If so, halt with output 1.
- Test divisibility of n by 2, 3, ..., $n/2$.
  - If n is divisible by k, ask whether k and n/k are in X.
    - If so, halt with output 1.
    - If $k \in X$ but $n/k \notin X$, test divisibility of n/k by 2, 3, ..., $n/(2k)$.
      - If n/k is divisible by ℓ, ask whether ℓ and $n/(k\ell)$ are in X.

$$\vdots$$

- Halt with output 0.

Going back to §4.5, to show that the word problem for semi-Thue systems is unsolvable, we produced a computable, uniform procedure to convert the question "does $\varphi_e(e)$ halt?" into one of the form "is W a theorem of S?" The oracle computation here is of the halting problem with an oracle for the word problem. The oracle Turing machine makes the conversion, queries the oracle for the answer to the particular word problem produced, and gives that as the answer to the original question.

We denote oracles by superscript: M^A for a machine, φ^A for a function. This is where we start needing the "brackets" notation from §5.1, because we consider the stage-s approximation of both the oracle and the computation, and sometimes even the input: $\varphi_{e,s}^{A_s}(n_s)$ abbreviates to $\varphi_e^A(n)[s]$. Finite binary strings may be used as oracles, treated as initial segments of characteristic sequences (see §3.1.3). If σ is the oracle, any oracle query about elements $\geq |\sigma|$ is unanswered, and the computation diverges.

Exercise 6.1.2. In the following, let σ range over all finite binary strings, and let e, s, x, y range over $\mathbb{N}$.

(i) Prove the set $\{\langle \sigma, e, x, s, y\rangle : \varphi_{e,s}^{\sigma}(x)\downarrow = y\}$ is computable.

(ii) Prove the set $\{\langle \sigma, e, x, y\rangle : \varphi_e^{\sigma}(x)\downarrow = y\}$ is computably enumerable.

When working with oracle computations we need to know how changes in the oracle affect the computation – or really, when we can be sure that changes *won't* affect the computation. Since each computation asks only finitely many questions of the oracle, we can associate it with a value called the *use* of the computation.

Definition 6.1.3. If $\varphi_{e,s}^A(x)\downarrow$, the *use* of the computation is

$$u(A; e, x, s) = 1 + \max\{n : \text{"}n \in A\text{" asked during computation}\}.$$

For halting computations, $u(A; e, x)$ is $u(A; e, x, s)$ for any stage s by which the computation has halted.

The use of divergent computations may be defined or undefined; commonly it is defined to 0 for the stage-bounded version and undefined for the unbounded version. The 1+ is to make the use line up with the definition of *restriction*: $A \upharpoonright n$ is $A \cap \{0, 1, \ldots, n-1\}$. The definition ensures that $A \upharpoonright u(A; e, x, s)$ includes all values about which A is queried during the computation.

For finite binary strings σ, τ with length $|\sigma| \leq |\tau|$ and infinite binary sequence A, $\sigma \subseteq \tau$ and $\sigma \subset A$ mean $\tau \upharpoonright |\sigma| = \sigma = A \upharpoonright |\sigma|$.

Proposition 6.1.4 (Use Principle). (i) $\varphi_e^A(x) = y \ \Rightarrow \ (\exists s)(\exists n)(\varphi_{e,s}^{A\upharpoonright n}(x) = y)$. *In fact, for a viable s, any $n \geq u(A; e, x, s)$ will work.*

(ii) *For σ, τ finite binary strings and A an infinite binary sequence,*

$$\varphi_{e,s}^{\sigma}(x) = y \ \Rightarrow \ (\forall t \geq s)(\forall \tau \supseteq \sigma)(\varphi_{e,t}^{\tau}(x) = y)$$

and

$$\varphi_{e,s}^{\sigma}(x) = y \ \Rightarrow \ (\forall t \geq s)(\forall A \supset \sigma)(\varphi_{e,t}^{A}(x) = y).$$

As a consequence of (ii), letting $u = u(A, e, x)$, if $A \upharpoonright u = B \upharpoonright u$, then $u(B, e, x) = u$ and $\varphi_e^A(x) = \varphi_e^B(x)$. In words, if A and B agree up to the largest element $\varphi_e(x)$ asks about when computing relative to A, then in fact on input x there is no difference between computing relative to A and relative to B because, by uniformity, φ_e is following the same path in its "ask about 5: if yes, then ask about 10; if no, then ask about 8" flowchart for computation. If B differs from A up to the use with oracle A, then both the use and the output with oracle B could be different.

The use principle is the basis for working with oracle computations. It tells us that, in constructions, if we want to preserve a computation $\varphi_e^A(x)$ while still allowing enumeration into A, we need only prevent enumeration of numbers $\leq u(A, e, x)$. Any others will leave the computation unharmed.

Definition 6.1.5. (i) A set A is *Turing reducible* to a set B, written $A \leq_T B$, if for some e, $\varphi_e^B = \chi_A$. We also say A is B-computable, computable with oracle B, or computable from B.

(ii) A and B are *Turing equivalent*, $A \equiv_T B$, if $A \leq_T B$ and $B \leq_T A$.

(iii) A and B are *Turing incomparable*, $A \perp_T B$, if $A \not\leq_T B$ and $B \not\leq_T A$.

This definition may also be made with functions. To match it to the above, using a function f as an oracle actually means using its (coded) graph $\{\langle x, y\rangle : f(x) = y\}$. Note that $f = \varphi_e^A$ means only that when φ_e is given A as its oracle, it computes f. The function f need not involve e or A at all.

Definitions for Exercises 6.1.6 and 6.1.13 may be found in §2.3.

Exercise 6.1.6. (i) Prove that $\leq_T$ is a *preorder* on $\mathcal{P}(\mathbb{N})$; that is, it is a reflexive, transitive relation.

(ii) In fact, $\leq_T$ is uniformly transitive: prove there is a function k such that for all i, e, A, B, C, if $\chi_C = \varphi_e^B$ and $\chi_B = \varphi_i^A$, then $\chi_C = \varphi_{k(e,i)}^A$. Work at the same level of (non)rigor as in the s-m-n theorem (4.3.1).

(iii) Prove that $\equiv_T$ is an equivalence relation on $\mathcal{P}(\mathbb{N})$.

Exercise 6.1.7. Argue that $u(A; e, x, s)$ is an A-computable function of e, x, and s.

Exercise 6.1.8. Prove $A \equiv_T \overline{A}$.

Exercise 6.1.9. (i) Prove that if A is computable, then $A \leq_T B$ for all sets B.

(ii) Prove that if A is computable and $B \leq_T A$, then B is computable.

Exercise 6.1.10. Prove that if A is c.e. but noncomputable, there is some non-c.e. $B \leq_T A$.

Exercise 6.1.11. Given sets A and B, prove that $B \leq_T A$ if and only if there exist computable functions f and g such that

$$x \in B \Leftrightarrow (\exists\sigma)[\sigma \in W_{f(x)} \ \&\ \sigma \subset A]$$
$$x \notin B \Leftrightarrow (\exists\sigma)[\sigma \in W_{g(x)} \ \&\ \sigma \subset A],$$

where σ ranges over all finite binary strings, as in the use principle 6.1.4. Technically the code of σ is the element of W, as in Exercise 3.1.4, but you may ignore that detail. The discussion after Exercise 3.4.7 may be useful.

Note that Turing reduction and subset inclusion have no relationship to each other. Neither implies the other, and they do not associate in a fixed direction: you may have $A \leq_T B$ with $A \subseteq B$, $B \subseteq A$, or neither.

Sets that are closely related, as A and $\overline{A}$ are, are often also Turing equivalent. The following is another such close relationship.

Definition 6.1.12. The *symmetric difference* of two sets A and B is

$$A \bigtriangleup B = (A \cap \overline{B}) \cup (\overline{A} \cap B).$$

If $|A \bigtriangleup B| < \infty$ we write $A =^* B$ and say A and B are *equal modulo finite difference.* We let A^* denote A's equivalence class modulo finite difference and write $A \subseteq^* B$ when $A \cap \overline{B}$ is finite.

Exercise 6.1.13. Prove $=^*$ is an equivalence relation on $\mathcal{P}(\mathbb{N})$.

Exercise 6.1.14. (i) Prove that if $A =^* B$, then $A \equiv_T B$.

(ii) Prove that $A \equiv_T B$ does not imply $A =^* B$.

On the opposite end of the c.e. sets from the computable sets are the (*Turing*) *complete* sets, those sets that are c.e. and that compute all other c.e. sets. Recall that the halting set is $K = \{e : \varphi_e(e)\downarrow\}$.

Theorem 6.1.15 (Post [72]). *K is Turing complete.*

Proof. We argued in §5.2 that K is a c.e. set. Given A, we construct a computable function f such that $x \in A \Leftrightarrow f(x) \in K$. Let e be such that $A = W_e$, and define the function $\psi(x, y)$ to equal 0 if $\varphi_e(x)\downarrow$, and diverge otherwise. Since φ_e is partial computable, so is ψ, so it is $\varphi_i(x, y)$ for some i. By Theorem 4.3.1, there is a total computable function s^1_1 such that $\varphi_{s^1_1(i,x)}(y) = \varphi_i(x, y)$ for all x and y. However, since i is fixed, we may view $s^1_1(i, x)$ as a (computable) function of one variable, $f(x)$, which gives the reduction.

$$x \in A \;\Rightarrow\; \varphi_e(x)\downarrow \Rightarrow\; \forall y(\varphi_{f(x)}(y) = 0) \;\Rightarrow\; \varphi_{f(x)}(f(x))\downarrow \Rightarrow\; f(x) \in K$$

$$x \notin A \;\Rightarrow\; \varphi_e(x)\uparrow \Rightarrow\; \forall y(\varphi_{f(x)}(y)\uparrow) \;\Rightarrow\; \varphi_{f(x)}(f(x))\uparrow \Rightarrow\; f(x) \notin K$$

□

Exercise 6.1.16. Prove that the *weak jump* $H = \{e : W_e \neq \emptyset\}$ is a Turing-complete c.e. set. For explanation of the name see Theorem 7.1.17 and the discussion after.

Exercise 6.1.17. (i) Prove that the set $\{\langle x, y\rangle : y \in \operatorname{rng} \varphi_x\}$ is a Turing-complete c.e. set.

(ii) Is the set $\{x : y_0 \in \operatorname{rng} \varphi_x\}$ for fixed y_0 always Turing complete? Prove that it is or give a y_0 for which it is not.

(iii) Is the set $\{y : y \in \operatorname{rng} \varphi_{x_0}\}$ for fixed x_0 always Turing complete? Prove that it is or give an x_0 for which it is not.

In §4.5 we explored some noncomputable sets. Many of them are computably enumerable: the set of theorems of a production system, the set of solutions to a Post correspondence system, the index set of machines that print a 0. The only ones that are not c.e. are the index sets that live at yet a higher level of the computability hierarchy. In full generality every one of the noncomputable c.e. sets from §4.5 is equivalent to the halting problem, and no one has found a way to reduce the generality to make the set weaker than the halting problem without it becoming computable.

This prompted Emil Post to ask the following question [72].

Question 6.1.18 (Post's Problem)**.** *Is there a computably enumerable set A such that A is noncomputable and incomplete?*

The assumption that A is c.e. tells us $\emptyset \leq_T A \leq_T K$, and the question is whether $A \not\equiv_T \emptyset$ and $A \not\equiv_T K$ are possible simultaneously. The answer is yes, though it took a while to get there. One way to solve Post's Problem is by producing incomparable sets, though that construction is left as Exercise 6.2.3.

Exercise 6.1.19. Prove that if A and B are Turing incomparable c.e. sets, then neither one can be complete or computable.

We will see a different solution in the next section.

6.2. Finite Injury Priority Arguments

Suppose we have an infinite collection $\{R_e\}_{e\in\mathbb{N}}$ of requirements to meet while constructing a set A. We've seen this in the noncomputable set constructions of §5.4. However, suppose further that these requirements may interact with each other, and to each other's detriment. For an extremely simplified example, suppose R_6 wants to put even numbers into A and R_8 wants there to be no even numbers in A. Then if R_6 puts 2 into A, R_8 will take it back out, and R_6 will try again with 2 or some other even number, and again R_8 will take it back out. We'll go round and round and neither requirement will end up satisfied (in fact, in this example, A may not even be well-defined).

In this example the requirements are actually set directly in opposition. At the other end of the spectrum, we can have requirements that are completely independent from each other and still have to worry about *injury* to a requirement. The reason is that information is parceled out slowly, stage by stage, since we're working with enumerations rather than full, pre-known characteristic functions. Our information is at best not *known* to be correct and complete, and at worst is actually incomplete, misleading, or outright wrong. Therefore we will make mistakes acting on it. However, we can't wait to act, because what we're waiting for might never happen, and not acting is almost certainly not correct either. For example, in the simple set construction, there is no waiting until we determine whether a set is finite. We can't ever know if we've seen all the elements of the set, so we have to act as soon as we see a chance (a large-enough number). The "mistake" we make there is putting additional elements into the set that we didn't have to. We eliminate the damage from that mistake by putting a lower bound on the size of the elements we can enumerate. In this more complicated construction, we will make mistakes that actually cause damage, but we will also set up the construction in such a way that the damage can be survived.

The key to getting the requirements to play nicely together is *priority.* We put the requirements into a list and allow each to injure only requirements further down the list. In our situation above, R_6 would be allowed to injure R_8, but not vice-versa.

The kind of priority arguments we will look at in this section are *finite-injury priority arguments.* That means each requirement only breaks the ones below it a finite number of times. We show every requirement can recover from finitely-much injury, and so after the finite collection of requirements earlier in the list than R_e have finished causing injury, R_e can act to satisfy itself and remain satisfied forever. [The proofs, therefore, are induction arguments.]

Let's work through a different version of the simple set construction. Recall a c.e. set A is *simple* if $\overline{A}$ is infinite but contains no infinite c.e. subset.

Theorem 6.2.1. *There is a simple set.*

Proof. We will construct A to be simple via meeting the following two sets of requirements:

$$R_e : (|W_e| = \infty) \Rightarrow (A \cap W_e \neq \emptyset),$$

$$Q_e : (\exists n > e)(n \in \overline{A}).$$

The construction will guarantee that A is computably enumerable. It is clear, as discussed in §5.4, that meeting all R_e will guarantee that $\overline{A}$ contains no infinite c.e. subsets.

To see that meeting all Q_e guarantees that $\overline{A}$ is infinite, consider a specific Q_e. If it is met, there is some $n > e$ in $\overline{A}$. Now consider Q_n. If it is met, there is some $n' > n$ in $\overline{A}$; we may continue in this way. Thus satisfying all Q_e requires infinitely many elements in $\overline{A}$.

We first discuss meeting each requirement in isolation. As long as $W_{e,s} \cap A_s = \emptyset$, R_e looks for an element of $W_{e,s+1}$ that it may put into A. Q_e chooses a *marker* $n_e > e$ and prevents its enumeration into A. As long as R_e is allowed to enumerate an element of W_e into A and Q_e is able to keep its marker out of A, the requirements are satisfied.

The Q requirements are often called *negative* because they want to keep elements out of A, and the R requirements are called *positive* because they want to enumerate elements into A. Some Q requirements will prevent R requirements from enumerating, and some R requirements will enumerate against Q requirement wishes. The priority ordering is as follows.

$$R_0, Q_0, R_1, Q_1, R_2, Q_2, \ldots$$

Requirements earlier in the list have higher priority (are stronger).

Under priority, each R_e requirement may enumerate anything from W_e into A except for the elements prohibited by $Q_0, Q_1, \ldots, Q_{e-1}$. Thus R_e might injure $Q_{e'}$ for some $e' > e$ by enumerating its chosen value $n_{e'}$ into A. This will cause the negative requirements from that point on to choose new n_e values larger than the number R_e enumerated into A. We therefore refer to $n_{e,s}$, the value of the marker n_e at stage s.

One further definition will streamline the construction. We say a requirement R_e *requires attention* at stage $s+1$ if $W_{e,s} \cap A_s = \emptyset$ (so R_e is unsatisfied) and there is some $x \in W_{e,s+1}$ such that $x \neq n_{k,s}$ for all $k < e$. Such an x is called a *witness* and is a viable element for satisfying R_e at stage $s+1$.

Construction:

Stage 0: $A_0 = \emptyset$. Each Q_e chooses value $n_{e,0} = e+1$.

Stage $s+1$: If any R_e, $e \leq s$, requires attention, choose the least such e and the least witness x for that e and let $A_{s+1} = A_s \cup \{x\}$; if x is $n_{k,s}$ for any $k \geq e$, let $n_{k',s+1} = n_{k'+1,s}$ for all $k' \geq k$. Note that this preserves the ordering $n_e < n_{e+1}$ for all e.

If no R_e requires attention, let $A_{s+1} = A_s$ and $n_{e,s+1} = n_{e,s}$ for all e. In either case, move on to stage $s+1$.

Verification:

Lemma 1. Each R_e acts at most once.

Proof. Once $W_e \cap A \neq \emptyset$, R_e will never act again. It is clear this will require at most one enumeration on the part of R_e. ⊣

Lemma 2. For each e, $n_e = \lim_s n_{e,s}$ exists. That is, the markers eventually stop shifting to the right. Moreover, for all e, $n_e \notin A$, so Q_e is met.

Proof. A marker moves only when the current value of it or one of its predecessor markers is enumerated into A. Therefore, the value of n_e can change only when some R_k for $k \leq e$ acts. By Lemma 1, each such requirement acts at most once. Therefore the value of n_e will change at most $e+1$ times during the construction, and afterward will remain fixed. Each n_e is in $\overline{A}$ because enumeration of a marker's value into A causes that marker to be moved to a different value. ⊣

Lemma 3. If W_e is infinite, then $W_e \cap A$ is nonempty, so R_e is met.

Proof. Suppose W_e is infinite and let s be a stage at which all requirements R_k for $k < e$ have stopped acting, which exists by Lemma 1. This means also, as in the proof of Lemma 2, that all markers n_k for

$k < e$ have attained their final values. As there are only finitely many such markers, and all others may be disregarded by R_e, at stage s a finite list of disallowed numbers is fixed. Since W_e is infinite, it must contain numbers not on that list, and if R_e has not yet been satisfied by stage s at the next stage at which W_e contains an element not on the list, R_e will have the opportunity to act and will be satisfied. $\dashv$

Lemma 2 shows each Q_e is satisfied, and Lemma 3 shows that each R_e is satisfied. A is c.e. because the construction is computable and never removes elements from A. Combined with the discussion preceding the construction, this completes the proof of the theorem. $\square$

The point of a priority argument is that we want to accomplish something infinitary, such as creating nonempty intersection with every infinite c.e. set, but we cannot do it all at once. The construction breaks that infinitary goal into finite pieces and organizes them so that if, in satisfying one, we might hurt another, we know how and how much.

In the construction above, we moved markers only if there was actual injury to a Q requirement, and then each marker was simply shifted down to the value of the next marker as of the injury. It would be more common to move markers whenever a higher-priority requirement enumerated into A, and to move them a great distance. In this scenario, only Q_e for $e \leq s$ would have defined markers at stage s. If R_e enumerated into A at s, the markers for $Q_{\hat{e}}$, $e \leq \hat{e} \leq s$, would all be shifted. Q_e's marker would go to the first number larger than every number used so far in the construction, and the rest would be assigned values increasing from there. In a proof as simple as Theorem 6.2.1, it is just as easy to have cases where markers are moved and cases where they are not, but in more complicated constructions it is often easier to be able to say "these markers always get moved when this requirement acts." In the proof of Theorem 6.2.2 we will take this more general approach.

In Theorem 6.2.2, however, we will also do something that seems more specific: we want to make sure two functions are nonequal, and we will choose a specific input value (n_e) on which to ensure the inequality. We could simply look for a difference in the inputs up to e, or up to the current stage, but looking at a specific place streamlines the construction and doesn't make it any more difficult to verify.

Finally, in both theorems, the lemmas are arranged so that we prove the construction settles down before proving the requirements are met. This is also common. The typical approach to a priority argument is first to determine what the requirements are, then how each would operate in an injury-free environment, and lastly how injury impacts them and how to control it. The initial lemmas show that injury is manageable, so that the later lemmas can essentially discuss requirements as though they were in an injury-free situation.

We now construct a set that gives a positive answer to Post's problem, Question 6.1.18.

Theorem 6.2.2. *There is a c.e. set A such that A is noncomputable and incomplete.*

We will construct a simple set A together with an auxiliary c.e. set B that is not computed by A, to show that A is incomplete.

Again, as in the previous constructions, the fact that A and B are c.e. will be implicit in the construction, by making sure the construction is computable and never removing elements from A or B.

Proof of Theorem 6.2.2. We build A simple and B c.e. such that $B \not\leq_T A$. The requirements are the following:

$$R_e : (|W_e| = \infty) \Rightarrow (A \cap W_e \cap \{x \in \mathbb{N} : x > 2e\} \neq \emptyset),$$

$$Q_e : \varphi_e^A \neq \chi_B.$$

A and B will be c.e. by construction.

The method for satisfying an R_e requirement in isolation is the same as in our original work on the simple set construction in §5.4.

We win a Q_e requirement if φ_e^A is not total, or we can produce a witness n such that $\varphi_e^A(n) \neq \chi_B(n)$. The incompleteness of information we have in this construction is that we can never know whether a computation diverges, or whether we need to let it chug along for a few more stages; this part is very much like the first construction of a noncomputable set in §5.4. Each Q_e will again pick a witness n_e, as in Theorem 6.2.1, but this time it will keep n_e out of B unless it sees $\varphi_e^A(n_e)\downarrow= 0$ (meaning φ_e^A and χ_B agree on n_e). In that case, Q_e puts the witness into B. The difference between this and §5.4 is that, with an oracle, the computation might not "stay halted." As A changes, the behavior of φ_e^A may also change. Therefore, Q_e tries to preserve its computation by restricting enumeration into A: it wants to keep any new elements $\leq u(A, e, n_e)$ out of A (recall Definition 6.1.3 and Proposition 6.1.4).

The priority ordering is as in Theorem 6.2.1.

$$R_0, Q_0, R_1, Q_1, R_2, Q_2, \ldots$$

Each R_e must obey restraints set by Q_k for $k < e$, but may injure later Q requirements by enumerating into A below the restraint of that Q, after Q has enumerated its witness into B. In that case, Q must choose a later witness and start over.

Again, we make a streamlining definition: R_e *requires attention* at stage s if $W_{e,s} \cap A_s = \emptyset$ and there is some $x \in W_{e,s}$ such that $x > 2e$ and x is also greater than any restraints in place from Q_k for $k < e$. Q_e *requires attention* at stage s if $n_{e,s}$ is defined, and $\varphi_e^A(n_e)[s]\downarrow= 0$ but $n_{e,s} \notin B_s$.

Construction:

$A_0 = B_0 = \emptyset$.

Stage s: Ask if any R_e or Q_e with $e < s$ requires attention. If so, take the highest-priority one that does and act to satisfy it:

- If Q_e, put $n_{e,s}$ into B_{s+1} and restrain A up to $u(A_s, e, n_{e,s}, s)$.
- If R_e, put the least applicable x into A_{s+1}. Cancel the witnesses (and restraints, if any were set) of requirements Q_k for $e \leq k \leq s$ and give them new witnesses $n_{k,s+1}$, distinct unused large numbers, increasing in k.

If no requirement needs attention, do nothing.

In either case, let any $n_{e,s+1}$ that was not specifically defined be equal to $n_{e,s}$ and set $n_{s,s}$ to be the least number not yet used in the construction. Restraints hold until they are cancelled by injury.

Verification:

Lemma 1. Each R_e requirement acts at most once.

Proof. Clear. $\dashv$

Lemma 2. For every e, $n_e = \lim_s n_{e,s}$ is defined.

Proof. As before, this lemma follows entirely from Lemma 1: since every R requirement acts at most once, the finite collection of them preceding Q_e in the priority list will all be finished acting at some finite stage. After that stage, whatever witness n_e is in place is the permanent value. $\dashv$

Lemma 3. Every Q_e requirement is met. Moreover, each Q_e either has no restraint from some stage s on or it has a permanent finite restraint that is unchanged from some stage s on.

Proof. Consider some fixed Q_e. By Lemma 2, n_e eventually reaches its final value, and by Lemma 1, all higher-priority positive requirements eventually stop acting. Thus, after some stage, Q_e will never be injured again and will have a single witness n_e to worry about. By induction, assume all higher-priority Q_e requirements have stopped acting. There are two cases:

Case 1: $\varphi_e^A(n_e)$ never converges to 0. That is, it either never converges or it converges to some value other than 0. In this case, the correct action is to keep n_e out of B, which Q_e does by default. Q_e is the only requirement that could put n_e into B, since witnesses for different requirements are always distinct values, so n_e will remain out of B. In this case, Q_e never sets a restraint for the permanent value of n_e, and any restraints it set on behalf of earlier witnesses are cancelled. Q_e will never act again.

Case 2: $\varphi_e^A(n_e)\downarrow= 0$. Suppose the final value of n_e is assigned to Q_e at stage s (so we know additionally that by stage s all higher-priority R_e requirements have stopped acting), and that at stage $s' \geq s$ all higher-priority Q requirements have stopped acting. Then at the first stage $s'' \geq s'$ such that $\varphi_e^A(n_e)[s'']\downarrow= 0$, Q_e will be the highest-priority requirement needing attention and will set restraint on A up to $u(A_{s''}, e, n_e, s'')$ and enumerate n_e into B. As the only positive requirements that might still act are bound to obey that restraint, it will never be violated and thus the computation $\varphi_e^A(n_e)\downarrow=$

0 is permanent and unequal to $\chi_B(n_e)$. Likewise, the restraint set is permanent and finite, and Q_e will never act again. $\dashv$

Lemma 4. Every R_e requirement is met.

Proof. Consider some fixed R_e. By Lemma 3, let s be a stage by which each requirement Q_k, $k < e$, has set its permanent restraint r_k or has had its restraint canceled and never thereafter sets a new one. By Lemma 1, let s also be large enough that all higher-priority R requirements that will ever act have already done so. Let $r = \max\{2e, r_k : k < e\}$; note that this is a finite value. If W_e is finite, then R_e is met automatically. If W_e is infinite, then it must contain a number $x > r$. If R_e is not already satisfied by stage s, then at the first stage thereafter in which such an x enters W_e, R_e will be the highest-priority requirement needing attention and will be able to enumerate x into A, making it permanently satisfied. $\dashv$

This completes the proof of the theorem. $\square$

Exercise 6.2.3. Construct a different solution to Post's Problem by a finite injury priority construction of two Turing incomparable c.e. sets A and B. Use two versions of requirement Q_e from Theorem 6.2.2. This is the approach of the original proof of the c.e. version of Post's problem, due independently to Friedberg [27] and Muchnik [64], though your proof will not look much like theirs.

Exercise 6.2.4. Using a finite injury priority construction, build a computable linear ordering L isomorphic to the natural numbers $\mathbb{N}$ such that the successor relation of L is not computable. That is, take $\mathbb{N}$ and reorder it by $\leq_L$ such that the ordering is total and has a least element, and so that there is a total computable function f such that $f(x, y) = 1$ if and only if $x \leq_L y$, but no total computable function g such that $g(x, y) = 1$ if and only if y is the successor of x. The computability of the ordering will be implicit in the construction: place s into the ordering at stage s, so you can determine whether $x \leq_L y$ by running the construction until stage $\max\{x, y\}$. For the remainder, satisfy the following requirements:

$$P_e : \varphi_e \text{ total } \Rightarrow (\exists x, y)[\varphi_e(x, y) = 1 \text{ but } (\exists z)(x \leq_L z \leq_L y)],$$

$$Q_x : \text{there are only finitely many elements } L\text{-below } x.$$

6.3. Notes on Approximation

So far, our approximations have all been enumerative processes: sets gain elements one by one, or functions gradually give results for various input values. There are other ways to get information about noncomputable sets; being c.e. is actually quite strong. The weakest condition on a set computable from K is simply to be computable from K, or Δ^0_2 (see §7.2.1). For A to be Δ^0_2 means there is a computable function $g(x, s)$ such that for each x, $g(x, 0) = 0$, $\lim_{s\to\infty} g(x, s) = \chi_A(x)$, and the number of times g "changes its mind" is finite (compare part (v) of Theorem 5.2.4).

$$(\forall x)\,(|\{s : g(x, s) \neq g(x, s+1)\}| < \infty)$$

This is not a definition but a theorem, of course, and you can see its proof in §8.1. In the context of a finite injury priority argument, we must be able to cope with injury caused by the removal of elements we thought were in the set as well as additional elements we hadn't counted on. Restraint on the set being constructed is now on both addition and removal of elements. We also no longer know only one change will be made; we only know the changes to each element's status will be finite, so *eventually* the approximation will be correct.

Exercise 6.3.1. Using a finite injury priority argument, build a bi-immune Δ^0_2 set A. That is, build A such that A and $\overline{A}$ both intersect every infinite c.e. set. Meet the following requirements:

$$Q_e : |W_e| = \infty \Rightarrow (\exists n)(n \in W_e \cap A),$$

$$R_e : |W_e| = \infty \Rightarrow (\exists n)(n \in W_e - A).$$

Q_e puts things in, R_e takes them out. Remember, since A is merely Δ^0_2, these "things" can be the same numbers, provided they only seesaw finitely many times apiece.

In between c.e. and Δ^0_2 there are various families of approximability. For a given computable function $f(x)$, a set is *f-c.e.* if it has a Δ^0_2 approximation g such that the number of mind changes of g on x is bounded by $f(x)$. If f is the identity, we call the set *id-c.e.*, and when f is a constant function, we may say *n-c.e.* for the appropriate value of n, where the 1-c.e. sets are simply the c.e. sets. When n is

2 ($f(x) = 2$ for all x), we furthermore may say *d.c.e.*, where the *d* stands for *difference.*

Exercise 6.3.2. Is there such a thing as a 0-c.e. set? If not, why not? If so, describe such an object.

Exercise 6.3.3. Prove that A is 2-c.e. if and only if for some $e, \hat{e}$, $A = W_e - W_{\hat{e}}$. This is the source of the term d.c.e.

Exercise 6.3.4. Prove that the intersection of two d.c.e. sets is d.c.e.

Exercise 6.3.5. Show that, for each $n \geq 0$, the set $\{e : |W_e| = n\}$ is d.c.e.

Exercise 6.3.6. Show that the Cartesian product $K \times \overline{K}$ and join $K \oplus \overline{K}$ are each d.c.e.

An approximation useful in the study of randomness (see §9.2) is *left computable enumerability.* In a left-c.e. approximation, it is always okay to put elements in, but only okay to take x out if you have put in something smaller than x. This is more natural in the context of characteristic functions viewed as infinite binary sequences. If you think of the sequence given by χ_A as an infinite binary expansion of a number between 0 and 1, then it is left-c.e. if we can approximate it so that the numerical value of the approximation is always increasing.

Chapter 7

Two Hierarchies of Sets

7.1. Turing Degrees and Relativization

Recall from Exercise 6.1.6 that Turing equivalence is an equivalence relation on $\mathcal{P}(\mathbb{N})$, the power set of the natural numbers. As in §2.3, we may define a quotient structure.

Definition 7.1.1. (i) The *Turing degrees* (or *degrees of unsolvability*) are the quotient of $\mathcal{P}(\mathbb{N})$ by Turing equivalence.

(ii) For a set $A \subseteq \mathbb{N}$, the *degree of* A is the equivalence class of A under Turing equivalence. This is often denoted $\deg(A)$, $\boldsymbol{a}$, or $\underset{\sim}{a}$ on the chalkboard.

The Turing degrees are partially ordered by Turing reducibility, meaning $\deg(A) \leq \deg(B)$ iff $A \leq_T B$. This is well defined (i.e., not dependent on the choice of degree representative A, B) by definition of Turing equivalence and the fact that it is an equivalence relation.

Exercise 7.1.2. Prove the following.

(i) The least Turing degree is $\deg(\emptyset)$ (also denoted $\mathbf{0}, \boldsymbol{\emptyset}$); it is the degree of all computable sets.

(ii) Every pair of degrees $\deg(A), \deg(B)$ has a least upper bound; moreover, that l.u.b. is $\deg(A \oplus B)$ (defined in Exercise 5.2.19).

(iii) For all sets A, $\deg(A) = \deg(\overline{A})$.

Note that for part (ii) you must show not only that $\deg(A \oplus B) \geq \deg(A), \deg(B)$, but also that any degree $\geq \deg(A), \deg(B)$ is also $\geq \deg(A \oplus B)$.

It is *not* the case that every pair of degrees has a greatest lower bound. The least upper bound of a pair of sets is often called their *join* and the greatest lower bound, should it exist, their *meet.*

Exercise 7.1.3. Prove that every infinite c.e. set contains subsets of every Turing degree.

Part (iii) of Exercise 7.1.2 explains the following definition.

Definition 7.1.4. A degree is called *c.e.* if it contains a c.e. set.

The maximum c.e. degree is $\deg(K)$, the degree of the halting set, which follows from Theorem 6.1.15.

Exercise 7.1.5. The "full halting set," often called K_0, is the set $\{\langle x, y\rangle : \varphi_x(y)\downarrow\}$. Prove that for any c.e. degree $\boldsymbol{d}$ there is a computable set A such that $A \cap K_0$ is in $\boldsymbol{d}$.

Exercise 7.1.6. Prove that each Turing degree contains only countably many sets.

Corollary 7.1.7. *There are uncountably many Turing degrees.*

Exercise 7.1.8. Using only material from the book to this point, prove that there is no maximal Turing degree.

Relativization means fixing some set A and always working with A as your oracle: working *relative to A.* Some examples follow.

Theorem 7.1.9 (Relativized *s-m-n* Theorem)**.** *For every $m, n \geq 1$ there exists a one-to-one computable function s_n^m of $m+1$ variables so that for all sets $A \subseteq \mathbb{N}$ and for all i, n-tuples $\bar{x}$, and m-tuples $\bar{y}$,*

$$\varphi^A_{s_n^m(i,\bar{x})}(\bar{y}) = \varphi_i^A(\bar{x}, \bar{y}).$$

Two important points: 1) this is a poor example of relativization, though it is important for *using* relativization. This is because 2) s_n^m is not just A-computable, it is computable. Because of the uniformity of oracle machine definitions, the proof here is essentially the same as for the original version; the only difference is, with oracle machines, the exact enumeration of φ_e is different.

Here's a better example:

Definition 7.1.10. The set B is *computably enumerable in* the set A if, for some e, $B = \operatorname{dom} \varphi_e^A$, also written $B = W_e^A$.

Exercise 7.1.11. Prove that the relation "B is c.e. in A" is not transitive.

Exercise 7.1.12. Prove that $B \leq_T A$ if and only if B and $\overline{B}$ are both c.e. in A.

Exercise 7.1.13. Prove that B is c.e. in A if and only if B is c.e. in $\overline{A}$.

Exercise 7.1.14. State the relativization of the equivalent definitions of c.e. in Theorem 5.2.4. These are also equivalent in the relativized case.

Exercise 7.1.15. State the relativization of Exercises 5.2.11, 5.2.17, and 5.2.18.

Exercise 7.1.16. State the relativization of the definition of computable inseparability and the examples of computably inseparable sets from Exercise 5.2.21.

Here's a very important example of relativization.

Theorem 7.1.17. *The set $A' = \{e : \varphi_e^A(e)\downarrow\}$ is c.e. in A but not A-computable.*

Again, by uniformity, the proof is essentially the same as for the original theorem. This is the halting set relativized to A, otherwise known as the *jump* (or *Turing jump*) of A and read "A-prime" or "A-jump". The original halting set is often denoted $\emptyset'$ or $0'$, and therefore the Turing degree of the complete sets is denoted $\boldsymbol{\emptyset'}$ or $\mathbf{0'}$ (this is never ambiguous, whereas K could easily be). Iteration of the jump is indicated by adding additional primes (for small numbers of iterations only) or using a number in parentheses: A'' is the second jump of A, a.k.a. $(A')'$, but for the fourth jump we would write $A^{(4)}$ rather than A''''.

Exercise 7.1.18. Give an alternate solution to Exercise 7.1.8.

Exercise 7.1.19. Show that the jump is not only c.e. in the original set, it is uniformly c.e. in that set. That is, show $(\exists e)(\forall A)(A' = W_e^A)$.

The jump of a set is always strictly Turing-above the original set, and computes it. Jump never inverts the order of Turing reducibility, but it may collapse it.

Proposition 7.1.20. (i) *If $B \leq_T A$, then $B' \leq_T A'$.*

(ii) *There exist sets A, B such that $B \lneq_T A$ but $B' \equiv_T A'$.*

One example of part (ii) is noncomputable *low* sets, sets A such that $A' \equiv_T \emptyset'$. Such a set is constructed with the aid of Theorem 8.1.1 and is discussed there.

Exercise 7.1.21. Prove part (i) of Proposition 7.1.20.

Exercise 7.1.22. Show that although the jump operator is not one-to-one on degrees, it is on sets: prove $A' = B' \Rightarrow A = B$.

Exercise 7.1.23. Recall the join of two sets from Exercise 5.2.19. Show that for any sets A and B, $A' \oplus B' \leq_T (A \oplus B)'$.

Exercise 7.1.24. While the proof of part (ii) of Proposition 7.1.20 is not yet within our reach, a modification is. The *omega jump* of A, $A^{(\omega)}$, is the join of all $A^{(n)}$ for $n \in \mathbb{N}$: $A^{(\omega)} = \{\langle x, n\rangle : x \in A^{(n)}\}$. Show the following.

(i) For all n, $A^{(n)} \lneq_T A^{(\omega)}$.

(ii) $A \leq_T B \Rightarrow A^{(\omega)} \leq_T B^{(\omega)}$.

(iii) There exist A, B such that $A \lneq_T B$ but $A^{(\omega)} \equiv_T B^{(\omega)}$.

Exercise 7.1.25. The weak jump relativized to A is $H^A = \{e : W_e^A \neq \emptyset\}$. In Exercise 6.1.16 you proved that the unrelativized weak jump is a Turing-complete c.e. set, which means for any A, $H^A \geq_T \emptyset'$. Prove that if $H^A \leq_T \emptyset'$, $A \leq_T \emptyset'$ as well. Such A are called *semi-low*. This proof will be more similar to Theorem 6.1.15 and Exercise 6.1.16 than to the more recent proofs.

7.2. The Arithmetical Hierarchy

The degrees $\mathbf{0}, \mathbf{0}', \ldots, \mathbf{0}^{(n)}, \ldots$ are, in some sense, the "spine" of the Turing degrees, in part because starting with the least degree and moving upward by iterating the jump is the most natural and non-arbitrary way to find a sequence of strictly increasing Turing degrees.

Additionally, however, those degrees line up with the number of quantifiers we need to write a logical formula. The *arithmetical hierarchy* is a way of categorizing sets according to how complicated the logical predicate representing them has to be.

Definition 7.2.1. (i) A set B is in Σ_0 (equivalently, Π_0) if it is computable.

(ii) A set B is in Σ_1 if there is a computable relation $R(x, y)$ such that

$$x \in B \iff (\exists y)(R(x, y)).$$

(iii) A set B is in Π_1 if there is a computable relation $R(x, y)$ such that

$$x \in B \iff (\forall y)(R(x, y)).$$

(iv) For $n \geq 1$, a set B is in Σ_n if there is a computable relation $R(x, y_1, \ldots, y_n)$ such that

$$x \in B \iff (\exists y_1)(\forall y_2)(\exists y_3) \ldots (Q y_n)(R(x, y_1, \ldots, y_n)),$$

where the quantifiers alternate and hence $Q = \exists$ if n is odd, and $Q = \forall$ if n is even.

(v) For $n \geq 1$, a set B is in Π_n if there is a computable relation $R(x, y_1, \ldots, y_n)$ such that

$$x \in B \iff (\forall y_1)(\exists y_2)(\forall y_3) \ldots (Q y_n)(R(x, y_1, \ldots, y_n)),$$

where the quantifiers alternate and hence $Q = \forall$ if n is odd, and $Q = \exists$ if n is even.

(vi) B is in Δ_n if it is in both Σ_n and Π_n.

(vii) B is *arithmetical* if, for some n, B is in $\Sigma_n \cup \Pi_n$.

We often say "B is Σ_2" instead of "B is in Σ_2." These definitions relativize to A by allowing the relation to be A-computable instead of just computable, and in that case we tack an A superscript onto the Greek letter: $\Sigma_n^A, \Pi_n^A, \Delta_n^A$.

I should note here that these are more correctly written as Σ_n^0 and the like, with oracles indicated as $\Sigma_n^{0,A}$. The superscript 0 indicates that all the quantifiers have domain $\mathbb{N}$. If we put a 1 in the superscript, we would be allowing quantifiers that range over sets as well, and would obtain the *analytical hierarchy.* That's outside the scope of this course, but it is good to note that not every set is arithmetical.[1]

Exercise 7.2.2. Prove the following.

(i) If B is in Σ_n or Π_n, then B is in Σ_m *and* Π_m for all $m > n$.

(ii) B is in Σ_n if and only if $\overline{B}$ is in Π_n.

(iii) B is c.e. if and only if it is in Σ_1.

(iv) B is computable if and only if it is in Δ_1 (i.e., $\Delta_0 = \Delta_1$).

(v) The union and intersection of two Σ_n sets (respectively, Π_n, Δ_n sets) are Σ_n (Π_n, Δ_n).

(vi) The complement of any Δ_n set is Δ_n.

That this actually is a hierarchy, and not a lot of names for the same collection of sets, needs to be proven.

Proposition 7.2.3. *For any $n \geq 1$, there is a Σ_n set that is not Π_n.*

Proof. The Σ_1 sets are effectively countable by the fact that they correspond exactly to the c.e. sets, and the Π_1 sets as a consequence. From those enumerations we may create enumerations of the Σ_n and Π_n sets for any larger n ($n = 0$ falls apart because the computable sets are not effectively enumerable; see Example 7.3.2). This gives us a universal Σ_n set S, analogous to the universal Turing machine and itself Σ_n, such that $\langle e, x\rangle \in S$ if and only if the e^{th} Σ_n set contains x.

From S, define $P := \{x : \langle x, x\rangle \in S\}$. P is also Σ_n, but it is not Π_n. If it were, by part (ii) of Exercise 7.2.2, $\overline{P}$ would be Σ_n. However,

[1]One relatively simple example: a binary relation on $\mathbb{N}$ is a *well order* if it is a linear order such that every subset of $\mathbb{N}$ has a least element under this ordering. $\mathbb{N}$ with the usual ordering is a well order, but $\mathbb{Q}$ and even $\mathbb{Q} \cap [0, 1]$ are not - even though the latter has a least element, any open subinterval does not. The property of being a linear order is arithmetical, but the additional property to be a well order is not: there is, in essence, no better way to say it than "for every subset S, there is an element of S less than or equal to all elements of S," which is a Π_1^1 sentence.

then $\overline{P}$ is the $\hat{e}^{th}$ Σ_n set for some $\hat{e}$. We have $\hat{e} \in \overline{P} \Leftrightarrow \langle \hat{e}, \hat{e} \rangle \in S$ on the one hand, but $\hat{e} \in \overline{P} \Leftrightarrow \hat{e} \notin P \Leftrightarrow \langle \hat{e}, \hat{e} \rangle \notin S$, a contradiction.

Likewise, the complement $\overline{P}$ is Π_n but not Σ_n. $\square$

Exercise 7.2.4. Prove that, for any $n \geq 1$, there is a Δ_{n+1} set that is neither Π_n nor Σ_n. Hint: combine P and $\overline{P}$ as in Proposition 7.2.3 and use parts (i) and (v) of Exercise 7.2.2.

A drawing of the lower levels of the arithmetical hierarchy appears in Figure 7.1 at the end of the chapter.

You may be wondering why only one of each kind of quantifier is allowed consecutively in the definitions. We may collapse like quantifiers into a single quantifier. For example, $(\exists x_1)(\exists x_2)(R(y, x_1, x_2))$ may be rewritten $(\exists N)[(\exists x_1 < N)(\exists x_2 < N)(R(y, x_1, x_2))]$, which is still Σ_1 because the part in square brackets is still computable. Since the x_i quantifiers are bounded, the procedure may check each possible pair of values in turn and be guaranteed to halt. It bears mentioning in particular that adding an existential quantifier to the beginning of a Σ_n formula keeps it Σ_n, and likewise for universal quantifiers and Π_n formulas.

The strong connection to Turing degree seen in Exercise 7.2.2 continues as we move up the scale of complexity.

Definition 7.2.5. A set A is Σ_n-*complete* if it is in Σ_n and, for every $B \in \Sigma_n$, there is a total computable one-to-one function f such that $x \in B \Leftrightarrow f(x) \in A$ (we say B is 1-*reducible* to A). Π_n-completeness is defined analogously.

The function f need not be surjective and in general will not be; though f allows us to compute A if we know the contents of B, we can only enumerate subsets of B and $\overline{B}$ from A and f.

Exercise 7.2.6. Prove X is Σ_n-complete if and only if $\overline{X}$ is Π_n-complete.

Note that because s_n^m from Theorem 4.3.1 is 1-1 and computable, the proof that K is Turing-complete (Theorem 6.1.15) shows it is Σ_1-complete as well. This generalizes to iterations of the halting set.

Theorem 7.2.7. *For every $n > 0$, $\emptyset^{(n)}$ is Σ_n-complete and $\overline{\emptyset^{(n)}}$ is Π_n-complete.*

A lot more sets Turing-reduce to $\emptyset'$ than 1-reduce to it.

Exercise 7.2.8. Prove that $A \leq_T \emptyset'$ if and only if A is Δ_2.

In fact, the following also hold. Theorem 7.2.7 combined with the proposition below is known as Post's theorem.

Proposition 7.2.9. (i) *$B \in \Sigma_{n+1} \iff B$ is c.e. in some Π_n set $\iff B$ is c.e. in some Σ_n set.*

(ii) *$B \in \Sigma_{n+1} \iff B$ is c.e. in $\emptyset^{(n)}$.*

(iii) *$B \in \Delta_{n+1} \iff B \leq_T \emptyset^{(n)}$.*

This strong tie between enumeration and existential quantifiers should make sense – after all, you're waiting for some input to do what you care about. If it happens, it will happen in finite time (the relation on the inside is computable), but you don't know how many inputs you'll have to check or how many steps you'll have to wait, just that if the relation holds at all, you'll find an appropriate input eventually.

Exercise 7.2.10. Compare and contrast sets that are (a) Turing-complete, (b) Σ_1-complete, (c) Π_1-complete, and (d) of degree $\boldsymbol{\emptyset'}$. What implications hold among these properties? Which properties are mutually exclusive? When property (i) implies (j) but they are not equivalent, is there a straightforward condition P such that (i) is equivalent to [(j) & P]?

Theorem 7.2.7 and Proposition 7.2.9 both relativize. For Theorem 7.2.7, the relativized version starts with "for every $n > 0$ and every set A, $A^{(n)}$ is Σ_n^A-complete" (recall that to relativize the arithmetical hierarchy, we allow the central relation to be A-computable rather than requiring it to be computable). The others relativize similarly.

One more exercise, for practice.

Exercise 7.2.11. (i) Prove A is Σ_2 if and only if there is a total computable function $g(x,s)$ with codomain $\{0,1\}$ such that

$$x \in A \Leftrightarrow \lim_s g(x,s) = 1.$$

(ii) Without appealing to the complement, prove A is Π_2 if and only if there is a total computable function $g(x,s)$ with codomain $\{0,1\}$ such that

$$x \in A \Leftrightarrow \lim_s g(x,s) \neq 0.$$

Hint for building g, when A is $\{x : (\exists y)(\forall z)R(x,y,z)\}$, or with quantifiers reversed: in (i), for each x, think about the smallest y such that so far it appears $(\forall z)R(x,y,z)$; define $g(x,s)$ depending on whether that has changed at s. For (ii), think about the largest y such that, for every $y' \leq y$, it appears that $(\exists z)R(x,y',z)$ holds. Define g based on whether y has changed at s.

7.3. Index Sets and Arithmetical Completeness

We have seen that $\emptyset^{(n)}$ and $\overline{\emptyset^{(n)}}$ are Σ_n- and Π_n-complete, respectively. Index sets give a wealth of additional examples. Most of the following were listed in §4.5.

$$\begin{aligned}
\text{Fin} &= \{e : |W_e| < \infty\}\\
\text{Inf} &= \{e : |W_e| = \infty\}\\
\text{Tot} &= \{e : \varphi_e \text{ is total}\}\\
\text{Con} &= \{e : \varphi_e \text{ is total and constant}\}\\
\text{Rec} &= \{e : W_e \text{ is computable}\}
\end{aligned}$$

Fin and Inf are complements, so by proving a completeness result for one of them, we have proved it for the other. In fact, we will prove that Fin is Σ_2-complete and that Inf, Tot, and Con are Π_2-complete. Rec is Σ_3-complete but we will prove only that it is Σ_3. All of the natural subcollections of Turing machines I am aware of have index sets complete at some level of the arithmetical hierarchy. This is perhaps unsurprising once you know the work that goes into producing a set strictly between $\emptyset$ and $\emptyset'$ in degree.

To prove a set is of a given arithmetic complexity, we must give a defining formula with the required alternation of quantifiers, being careful to quantify only over numbers, not sets, and give a computable central relation. In particular, when enumerating sets or running Turing machines, everything must be stage-bounded.

Example 7.3.1. Fin is Σ_2.

$$\text{Fin} = \{e : |W_e| < \infty\} = \{e : (\exists N)(\forall n)(\forall s)\,(n > N \Rightarrow \varphi_{e,s}(n)\uparrow)\}$$

This also shows that Inf is Π_2.

Example 7.3.2. Rec is Σ_3.

Rec can't be c.e., because if it were, we could diagonalize out of it. However, we might not guess it is as high up as Σ_3 without writing out a formula.

Rec $= \{e : W_e$ is computable$\}$, so the leading existential quantifier says that there must be an index $\hat{e}$ giving W_e's characteristic function. For $\varphi_{\hat{e}}$ to be that function, it must be total and have output 1 on n if $n \in W_e$ and output 0 otherwise. Going from the "not in W_e" side is hard, though, because we only have an enumeration, so we work by making membership in W_e equivalent to an output of 1, with the additional condition that $\varphi_{\hat{e}}$ is total with codomain $\{0, 1\}$.

For every n, if n appears in W_e, at some stage we must have $\varphi_{\hat{e}}(n)\downarrow= 1$, but not necessarily at the same stage. Likewise halting with output 1 says n must appear in W_e eventually, but not at a specific stage.

$$\begin{aligned}\text{Rec} = \quad & \{e : (\exists\hat{e})(\forall n, s)(\exists t)[(\varphi_{\hat{e},t}(n)\downarrow \in \{0,1\}\ \&\\ & (n \in W_{e,s} \Rightarrow \varphi_{\hat{e},t}(n) = 1)\ \&\ (\varphi_{\hat{e},s}(n) = 1 \Rightarrow n \in W_{e,t})]\}\end{aligned}$$

Exercise 7.3.3. Show that Tot and Con are Π_2.

Exercise 7.3.4. Recall A and B are *computably separable* if there is some computable set $C \supseteq A$ such that $C \cap B = \emptyset$. Show that

$$\text{Sep} = \{\langle x, y\rangle : W_x, W_y \text{ are computably separable}\}$$

is a Σ_3 set.

Exercise 7.3.5. (i) Prove that $\{\langle x, y\rangle : W_x = W_y\}$ is Π_2.

(ii) Recall Definition 6.1.12 of the symmetric difference of two sets. Prove that $\{\langle x, y\rangle : W_x =^* W_y\}$ is Σ_3.

Example 7.3.6. Fin is Σ_2-complete, and Con, Tot, and Inf are Π_2-complete.

To show that Fin is Σ_2-complete, since we have already shown it is Σ_2, for any given Σ_2 set A, we must produce a computable 1-1 function f such that $x \in A \Leftrightarrow f(x) \in \text{Fin}$. For some computable relation R, $x \in A \Leftrightarrow (\exists y)(\forall z)R(x, y, z)$, by definition of being Σ_2. It turns out to be more useful to take the complement: $x \in \overline{A} \Leftrightarrow (\forall y)(\exists z)\neg R(x, y, z)$, because we can then "cap off" the leading universal quantifier at higher and higher points, looking to see if there is a z for each of the finitely many y. This allows us to define a partial computable function.

$$\psi(x, w) = \begin{cases} 0 & (\forall y \leq w)(\exists z)\neg R(x, y, z) \\ \uparrow & \text{otherwise} \end{cases}$$

This is partial computable despite the unbounded existential quantifier because we can dovetail the (finitely many) searches for a z to match each $y \leq w$. If each y has such a z we will eventually find it, and if not, it is simply a divergent unbounded search. As usual, ψ is some φ_e, we can use *s-m-n* to push x into the index, e is fixed by A and we end up with a 1-1 total computable f such that $\varphi_{f(x)}(w) = \psi(x, w)$. To finish the proof we show that f demonstrates the Σ_2 completeness of Fin.

If $x \in \overline{A}$, all y have a matching z, every w gives a convergent search, and $\varphi_{f(x)}$ is the constant 0 function. If $x \in A$, there is some y that has no matching z, and $\varphi_{f(x)}$ will diverge on all w from the least such y on. That is,

$$x \in A \ \Rightarrow\ W_{f(x)} \text{ finite } \Rightarrow\ f(x) \in \text{Fin, and}$$

$$x \in \overline{A} \ \Rightarrow\ (\forall w)(\varphi_{f(x)}(w) = 0) \ \Rightarrow\ f(x) \in \text{Con} \subset \text{Tot} \subset \text{Inf} = \overline{\text{Fin}}.$$

Note that this shows Π_2 completeness for Con, Tot, and Inf only in conjunction with the previous proofs that those sets are each Π_2.

Exercise 7.3.7. Show that $\{\langle x, y\rangle : W_x = W_y\}$ is Π_2-complete by showing Tot 1-reduces to it.

$$\begin{tikzcd}
\emptyset''\text{-c.e. }(\mathrm{Rec}, \mathrm{Sep}) & \Sigma_3 \arrow[dr, dash] & & \Pi_3 \arrow[dl, dash] & \emptyset''\text{-co-c.e.} \\
& & \Delta_3 \arrow[dl, dash] \arrow[dr, dash] & & \text{computable from } \emptyset'' \\
\emptyset'\text{-c.e. }(\mathrm{Fin}) & \Sigma_2 \arrow[dr, dash] & & \Pi_2 \arrow[dl, dash] & \emptyset'\text{-co-c.e. }(\mathrm{Inf}, \mathrm{Tot}, \mathrm{Con}) \\
& & \Delta_2 \arrow[dl, dash] \arrow[dr, dash] & & \text{computable from } \emptyset' \\
\text{c.e.} & \Sigma_1 \arrow[dr, dash] & & \Pi_1 \arrow[dl, dash] & \text{co-c.e.} \\
& & \Delta_1 & = \Delta_0 = \Pi_0 = \Sigma_0; \text{computable} &
\end{tikzcd}$$

Figure 7.1. A picture of the arithmetical hierarchy. Each set is contained in those directly above it and those above it to which it is connected by lines.

Chapter 8

Further Tools and Results

Where do we go from here? This chapter gives some ideas and results in computability theory that continue the work so far; Chapter 9 takes a little leap to survey some current areas of research.

8.1. The Limit Lemma

Proposition 5.2.4 part (v) gave a characterization of computably enumerable sets in terms of computable approximations to their characteristic functions: the approximation had to start at 0 and could change to 1 at most once per input. We know that not every set reducible to $\emptyset'$ is c.e.; for example, there are noncomputable co-c.e. sets and sets that are the join of a c.e. set and a co-c.e. set, both noncomputable, and hence neither c.e. nor co-c.e. themselves. We may ask whether this can be characterized in terms of value changes of approximations, and the following theorem shows it can.

Theorem 8.1.1 (Limit Lemma [78]). *For any function f, $f \leq_T B'$ if and only if there is a B-computable function $g(x,s)$ such that $f(x) = \lim_s g(x,s)$.*

In particular, $f \leq_T \emptyset'$ iff there is a computable function g that limits to f. Since not all sets reducible to $\emptyset'$ are c.e., this is often the most useful approach. As discussed in §3.1.1, for the limit to exist, g may change value (for any given x) only finitely many times. However, the number of changes may be arbitrarily large.

One significant use of this theorem is in the construction of low sets. Recall that A is *low* if $A' \equiv_T \emptyset'$. If A is c.e., it suffices to show $A' \leq_T \emptyset'$, because if $\emptyset \leq_T A$, we always have $\emptyset' \leq_T A'$.

In the construction of a low set, we work by making sure that if the computation $\varphi_e^A(e)[s]$ converges infinitely many times (i.e., for infinitely many stages s), then it converges. That is, it either eventually forever diverges or eventually forever converges. This guarantees the limit of the following function always exists.

$$g(e,s) = \begin{cases} 0 & \text{if } \varphi_e^A(e)[s]\uparrow \\ 1 & \text{if } \varphi_e^A(e)[s]\downarrow \end{cases}$$

The function g is computable provided the construction is computable (so that A_s may be obtained from s), and if $\lim_s g(x,s)$ exists, it has to match $\chi_{A'}(x)$. Hence by forcing the limit to exist, the Limit Lemma gives $A' \leq_T \emptyset'$.

To prove the theorem we need an auxiliary function.

Definition 8.1.2. Suppose $g(x,s)$ converges to $f(x)$. A *modulus* (*of convergence*) for g is a function $m(x)$ such that for all $s \geq m(x)$, $g(x,s) = f(x)$. The *least modulus* is the function $m(x) = (\mu s)(\forall t \geq s)[g(x,t) = f(x)]$.

Exercise 8.1.3. Notation is as in Definition 8.1.2.

(i) Prove that the least modulus is computable in any modulus.

(ii) Prove that f is computable from g together with any modulus m for g.

In general, we cannot turn the reducibility of (ii) around, but for functions of c.e. degree there will be some modulus computable from f (and hence the least modulus will also be computable from f).

Theorem 8.1.4 (Modulus Lemma). *If B is c.e. and $f \leq_T B$, then there is a computable function $g(x,s)$ such that $\lim_s g(x,s) = f(x)$ for all x and a modulus m for g that is computable from B.*

Proof. Let B be c.e. and let $f = \varphi_e^B$. Define the following functions:

$$g(x,s) = \begin{cases} \varphi_e^B(x)[s] & \text{if } \varphi_e^B(x)[s]\downarrow, \\ 0 & \text{otherwise,} \end{cases}$$

$$m(x) = (\mu s)(\exists z \leq s)[\varphi_e^{B \restriction z}(x)[s]\downarrow \ \&\ B_s \restriction z = B \restriction z].$$

Clearly, g is computable; m is B-computable because the clauses inside are B-computable, the quantifier on z is bounded and hence does not increase complexity, and the unbounded search always halts, by choice of e. The second clause furthermore gives the desired property, that m is a modulus, because B is c.e., and hence once the approximation B_s matches B, it will never change to differ from B. ☐

Proof of Theorem 8.1.1. ($\Rightarrow$) Suppose $f \leq_T B'$. We know B' is c.e. in B, so $g(x,s)$ exists and is B-recursive by the Modulus Lemma 8.1.4 relativized to B.

($\Leftarrow$) Suppose the B-computable function $g(x,s)$ limits to $f(x)$. Define the following finite sets.

$$B_x = \{s : (\exists t)[s \leq t \ \&\ g(x,t) \neq g(x,t+1)]\}$$

If we let $C = \{\langle s,x\rangle : s \in B_x\}$ (also denoted $\oplus_x B_x$), then C is Σ_1^B and hence c.e. in B; therefore, $C \leq_T B'$. Additionally, given x, it is computable from C (and hence from B') to find the least modulus $m(x) = (\mu s)[s \notin B_s]$. Hence $f \leq_T m \oplus B \leq_T C \oplus B \leq_T B'$. ☐

We can actually distinguish the c.e. degrees from the non-c.e. Δ_2 degrees via properties of the modulus.

Corollary 8.1.5. *A function f has c.e. degree iff f is the limit of a computable function $g(x,s)$ that has a modulus $m \leq_T f$.*

Exercise 8.1.6. Prove Corollary 8.1.5. For ($\Rightarrow$) apply the Modulus Lemma; for ($\Leftarrow$) use C from the proof of the Limit Lemma.

Exercise 8.1.7. (i) Relativize the definition of simple set (Definition 5.4.2) to oracle $\emptyset'$.

(ii) What degree-theoretic conclusion can you make about a set A that meets your definition in (i)?

(iii) Could a computable procedure construct a set that meets your definition in (i)? That is, could one computably create a stage-wise approximation that in the limit gives the correct set? If so, explain how the requirement would be written and met. If not, explain why and say what oracle would be required to compute the construction.

Exercise 8.1.8. Work out the priority construction of a simple low set, with simplicity requirements R_e as usual and lowness requirements

$$Q_e : (\exists^\infty s)(\varphi_{e,s}^{A_s}(e)\downarrow) \Rightarrow \varphi_e^A(e)\downarrow .$$

8.2. The Arslanov Completeness Criterion

This is a result that can be viewed as the flip side of the recursion theorem. Every total computable function has an index-level fixed point, so a total function with no index-level fixed point must be noncomputable. The result below strengthens that to find a fixed-point-type characterization of Turing completeness. We need an extension of the recursion theorem, due to Kleene.

Theorem 8.2.1 (Recursion Theorem with Parameters)**.** *If $f(x, y)$ is a computable function, then there is a computable function $h(y)$ such that $\varphi_{h(y)} = \varphi_{f(h(y),y)}$ for all y.*

Proof. Define a computable function d as follows.

$$\varphi_{d(x,y)}(z) = \begin{cases} \varphi_{\varphi_x(x,y)}(z) & \text{if } \varphi_x(x,y)\downarrow, \\ \uparrow & \text{otherwise.} \end{cases}$$

Choose v such that $\varphi_v(x, y) = f(d(x, y), y)$. Then $h(y) = d(v, y)$ is a fixed point, since unpacking the definitions of h, d and v (and then repacking h), we see that

$$\varphi_{h(y)} = \varphi_{d(v,y)} = \varphi_{\varphi_v(v,y)} = \varphi_{f(d(v,y),y)} = \varphi_{f(h(y),y)}.$$

□

In fact, we may replace the total function $f(x, y)$ with a partial function $\psi(x, y)$ and have total computable h such that whenever $\psi(h(y), y)$ is defined, $h(y)$ is a fixed point. The proof is identical to the proof of Theorem 8.2.1. Note that the parametrized version implies the original version; if in the original you wanted a fixed point for the total computable function $g(x)$, define $f(x, y) = g(x)$.

The kind of fixed point for which avoidance is equivalent to completeness is a domain-level fixed point, where x and $f(x)$ index the same c.e. set.

Theorem 8.2.2 (Arslanov Completeness Criterion [5,6])**.** *A c.e. set A is complete if and only if there is a function $f \leq_T A$ such that $W_{f(x)} \neq W_x$ for all x.*

Proof. ($\Rightarrow$) Assume we are given a Turing-complete c.e. set A. In Exercise 6.1.16 you proved that the weak jump $H = \{e : W_e \neq \emptyset\}$ is also complete, and hence Turing-equivalent to A. Define f by

$$W_{f(x)} = \begin{cases} \emptyset & x \in H, \\ \{0\} & x \notin H. \end{cases}$$

It is clear $f \leq_T H \equiv_T A$, and that f satisfies the right-hand side of the theorem.

($\Leftarrow$) Let A be c.e., and assume $f \leq_T A$ is such that $(\forall x)[W_{f(x)} \neq W_x]$. By the Modulus Lemma 8.1.4, there is a computable function $g(x, s)$ that limits to f and such that g has a modulus $m \leq_T f$ (and hence $m \leq_T A$). Define the partial function $\theta(x) = (\mu s)[x \in \emptyset'_s]$. When $x \notin \emptyset'$, $\theta(x)$ diverges; otherwise it gives the stage of x's enumeration into $\emptyset'$.

We show $\emptyset' \leq_T A$ by showing that A can compute a function at least as large as θ, and therefore knows when to stop waiting for x to enter $\emptyset'$. Define the following partial function.

$$\psi(y, x) = \begin{cases} g(y, \theta(x)) & x \in \emptyset', \\ \uparrow & x \notin \emptyset'. \end{cases}$$

By Theorem 8.2.1, let h be a computable function giving a fixed point for ψ, with the following consequence.

$$W_{h(x)} = \begin{cases} W_{g(h(x), \theta(x))} & x \in \emptyset', \\ \emptyset & x \notin \emptyset'. \end{cases}$$

Note that this is not technically the h from the theorem; there is no guarantee in Theorem 8.2.1 that $h(x)$ will index the empty function when $\psi(h(x), x)\uparrow$, but we can force that computably by not allowing $\varphi_{h(x)}$ to converge on any input until x enters $\emptyset'$.

For the rest, recall that $m(x)$ gives a stage by which $g(x, s)$ has achieved its final value, $f(x)$. Now if $x \in \emptyset'$ and $\theta(x) \geq m(h(x))$, then $g(h(x), \theta(x)) = f(h(x))$ and $W_{f(h(x))} = W_{h(x)}$, contrary to assumption on f. Therefore, if $x \in \emptyset'$, we must have $\theta(x) < m(h(x))$, meaning

$$x \in \emptyset' \iff x \in \emptyset'_{m(h(x))}.$$

Since $m \leq_T A$ and h is computable, A can determine whether $x \in \emptyset'$ by enumerating $\emptyset'$ through stage $m(h(x))$, and so $\emptyset' \leq_T A$. $\square$

Corollary 8.2.3. *Given a c.e. degree $\boldsymbol{a}$, $\boldsymbol{a} < \mathbf{0}'$ if and only if for every function $f \in \boldsymbol{a}$ there exists n such that $W_n = W_{f(n)}$.*

The condition of being computably enumerable is necessary – there is a Δ^0_2 degree strictly below $\emptyset'$ such that some f reducible to that degree has the property $(\forall e)[W_e \neq W_{f(e)}]$. What else can be said about fixed points? We might look at $*$-fixed points; that is, n such that $W_n =^* W_{f(n)}$ (see Definition 6.1.12). These are also called *almost fixed points.* Weaker still are *Turing fixed points*, n such that $W_n \equiv_T W_{f(n)}$.

As a catalogue:

- Any total function $f \leq_T \emptyset'$ has an almost fixed point.
- A Σ^0_2 set $A \geq_T \emptyset'$ is Turing-equivalent to $\emptyset''$ if and only if there is some $f \leq_T A$ such that f has no almost fixed points.
- Any total function $f \leq_T \emptyset''$ has a Turing fixed point.

In fact, there is a whole hierarchy of fixed-point completeness criteria. We can define equivalence relations $\sim_\alpha$ for $\alpha \in \mathbb{N}$ as follows:

(i) $A \sim_0 B$ if $A = B$,

(ii) $A \sim_1 B$ if $A =^* B$,

(iii) $A \sim_2 B$ if $A \equiv_T B$,

(iv) $A \sim_{n+2} B$ if $A^{(n)} \equiv_T B^{(n)}$ for $n \in \mathbb{N}$.

Now completeness at higher levels of complexity may be defined in terms of computing a function that has no $\sim_\alpha$-fixed points.

Theorem 8.2.4 (Generalized Completeness Criterion [5, 6, 42])**.** *Fix $\alpha \in \mathbb{N}$. Suppose $\emptyset^{(\alpha)} \leq_T A$ and A is c.e. in $\emptyset^{(\alpha)}$. Then*

$$A \equiv_T \emptyset^{(\alpha+1)} \iff (\exists f \leq_T A)(\forall x)[W_{f(x)} \not\sim_\alpha W_x].$$

Notice this gives us a third hierarchy of sets that matches up with the Turing degrees and arithmetical hierarchy at iterations of the halting problem.

Exercise 8.2.5. The set B is *productive* if there is a partial computable function ψ, called a productive function for B, with the following property.[1]

$$(\forall x)\,[W_x \subseteq B \;\Rightarrow\; [\psi(x)\!\downarrow \;\&\; \psi(x) \in B - W_x]]$$

A c.e. set A is *creative* if its complement is productive. Use Theorem 8.2.2 to show that any creative set A is complete. Hint: $W_{f(x)} = \{\psi(x)\}$ is a good start, but remember the consequent of the productive function implication may hold even if the antecedent fails.

Exercise 8.2.6. A set A is *effectively simple* if it is simple (Definition 5.4.2), and furthermore there is a computable function g such that

$$(\forall e)[W_e \subseteq \overline{A} \;\Rightarrow\; |W_e| \leq g(e)].$$

Use Theorem 8.2.2 to show that any effectively simple set A is complete. Hint: what if $W_{f(x)}$ is always a subset of $\overline{A}$?

8.3. $\mathcal{E}$ Modulo Finite Difference

Recall from 6.1.12 that $A =^* B$ means that A and B differ only by finitely many elements, and $=^*$ is an equivalence relation that implies Turing equivalence. When $A =^* B$, we often treat A and B as interchangeable, and say we are working *modulo finite difference.* The usefulness of working modulo finite difference is that it gives you wiggle room in constructions – as long as eventually you're putting in exactly the elements you want to be, it doesn't matter if you mess up a little at the beginning and end up with a set that is not equal to what you want, but is $*$-equal.

[1]This definition was inspired by Gödel's incompleteness theorem, as in §5.3. If B is the set of sentences true in the standard model of arithmetic and W is the set of sentences provable from the axioms of arithmetic, we can computably produce a sentence that is not in W but is in B.

Nearly all of the properties of sets we care about are *closed under* finite difference; that is, if A has the property and $B =^* A$, then B has the property as well. For example, Exercise 6.1.14 showed that each Turing degree is closed under finite difference. To see that Turing completeness is closed under finite difference requires a bit of extra work:

Exercise 8.3.1. Prove that the property of being computably enumerable is closed under finite difference.

Simplicity, Definition 5.4.2, is another example.

Exercise 8.3.2. Prove that simplicity is closed under finite difference.

This is what allows injury arguments to work. In §6.2 we saw constructions where requirements were banned from enumerating anything smaller than a certain value. The fact that the properties we were trying to achieve are closed under finite difference means a threshold like that, provided it does stop at a finite point, is not a barrier to succeeding in the construction.

The structure of the c.e. sets modulo finite difference has been the object of much study. We usually use $\mathcal{E}$ to denote the c.e. sets and $\mathcal{E}^*$ to denote the quotient structure $\mathcal{E}/{=^*}$. The letters $\mathcal{N}$ and $\mathcal{R}$ are used to denote the collection of all subsets of $\mathbb{N}$ and of the computable sets, respectively. Unlike when we work with degrees, the lattice-theoretic operations we'd like to perform are defined everywhere.

Definition 8.3.3. $\mathcal{E}$, $\mathcal{R}$, and $\mathcal{N}$ are all *lattices*, partially ordered sets where every pair of elements has a least upper bound (join) and a greatest lower bound (meet). The ordering in each case is subset inclusion. The *join* of two sets A and B is $A \vee B := A \cup B$; their *meet* is $A \wedge B := A \cap B$. In each case, these operations distribute over each other, making all three *distributive lattices*. Moreover, all three lattices have least and greatest element (not required to be a lattice): the least element in each case is $\emptyset$ and the greatest is $\mathbb{N}$. A set A is *complemented* if there is some B in the lattice such that $A \vee B$ is the greatest element and $A \wedge B$ is the least element; the lattice is called complemented if all of its elements are. A complemented, distributive lattice with (distinct) least and greatest element is called a *Boolean algebra.*

Exercise 8.3.4. (i) What is the least number of elements a Boolean algebra may have?

(ii) Show that $\mathcal{N}$ and $\mathcal{R}$ are Boolean algebras but $\mathcal{E}$ is not.

(iii) Characterize the complemented elements of $\mathcal{E}$.

Definition 8.3.5. A property P is *definable* in a language (a set of relation, function, and constant symbols) if, using only the symbols in the language and standard logical symbols, one may write a formula with one free variable such that an object has property P if and only if when filled in for the free variable it makes the formula true. Likewise we may define n-ary relations (properties of tuples of n objects) using formulas of n free variables.

For example, the least element of a lattice is definable in the language $L = \{\leq\}$ (where we interpret $\leq$ as whatever ordering relation we're actually using; here it is $\subseteq$) by the formula $\psi(y) = (\forall x)[y \leq x]$. $\psi(y)$ holds if and only if y is less than or equal to all elements of the lattice, which is exactly the definition of least element.

Exercise 8.3.6. Let the language $L = \{\leq\}$ be fixed.

(i) Show that the greatest element is definable in L.

(ii) Show that meet and join are definable (via formulas with three free variables) in L.

Definition 8.3.7. An *automorphism* of a lattice $\mathcal{L}$ is a bijective function from $\mathcal{L}$ to $\mathcal{L}$ that preserves the partial order.

Exercise 8.3.8. (i) Show that automorphisms preserve meets and joins.

(ii) Show that a permutation of $\mathbb{N}$ induces an automorphism of $\mathcal{N}$.

Definition 8.3.9. Given a lattice $\mathcal{L}$, a class $X \subseteq \mathcal{L}$ is *invariant* (under automorphisms) if, for any $x \in \mathcal{L}$ and automorphism f of $\mathcal{L}$, $f(x) \in X \iff x \in X$. X is an *orbit* if it is invariant and *transitive*: for any $x, y \in X$, there is an automorphism f of $\mathcal{L}$ such that $f(x) = y$.

Exercise 8.3.10. What sort of structure (relative to automorphisms) must an invariant class that is not an orbit have?

Definition 8.3.11. A property P of c.e. sets is *lattice-theoretic* (*l.t.*) in $\mathcal{E}$ if it is invariant under all automorphisms of $\mathcal{E}$. P is *elementary lattice theoretic* (*e.l.t.*) if there is a formula of one free variable in the language $\mathcal{L} = \{\leq, \vee, \wedge, 0, 1\}$[2] that defines the class of sets with property P in $\mathcal{E}$, where $\leq, 0, 1$ are interpreted as $\subseteq, \emptyset, \mathbb{N}$, respectively.

Exercise 8.3.12. Show that a definable property P is preserved by automorphisms; that is, that e.l.t. implies l.t.

The definition and exercise above still hold when we switch from $\mathcal{E}$ to $\mathcal{E}^*$. Here's where the additional usefulness of working modulo finite difference comes in. As mentioned before, we are almost always worried about properties that are closed under finite difference.

Exercise 8.3.13. Using part (iii) of Exercise 8.3.4, show that the property of being finite is definable in $\mathcal{E}$.

Exercise 8.3.14. From Exercise 8.3.13, suppose $F(X)$ is a formula in the language $\{\subseteq\}$ that is true of $X \in \mathcal{E}$ if and only if X is finite. Prove that the relation $X =^* Y$ is also definable in $\{\subseteq\}$.

Exercise 8.3.15. Using Exercise 8.3.14, show that any property P closed under finite differences is e.l.t. in $\mathcal{E}$ if and only if it is e.l.t. in $\mathcal{E}^*$.

To show something is not e.l.t., one would likely show it is not l.t. by constructing an automorphism under which it is not invariant. Automorphisms are easier to construct in $\mathcal{E}^*$ than in $\mathcal{E}$, and by the agreement of definability between those two structures we can get results about $\mathcal{E}$ from $\mathcal{E}^*$. There will be more on this in §9.1.

[2]Though meet, join, and least and greatest element are definable from the partial order, we often include them in the language to simplify writing formulas.

Chapter 9

Areas of Research

In this chapter I try to give you a taste of various areas of computability theory in which research is currently active. Actually, only §9.1 discusses "pure" computability theory; the others are independent areas that intersect significantly with computability.

9.1. Computably Enumerable Sets and Degrees

The Turing degrees are a partially ordered set under $\leq_T$, as we know, and we also know any pair of degrees has a least upper bound and *not* every pair of degrees has a greatest lower bound (a *meet*). What else can be said about this poset?

Definition 9.1.1. $\mathcal{D}$ is the partial ordering of the Turing degrees, and $\mathcal{D}(\leq \boldsymbol{a})$ is the partial order of degrees less than or equal to $\boldsymbol{a}$. $\mathcal{R}$ is the partial order of the c.e. Turing degrees (so $\mathcal{R} \subset \mathcal{D}$).[1]

Question 9.1.2. *Is there a nonidentity automorphism of $\mathcal{D}$? Of $\mathcal{R}$?*

It is also still open whether there is a "natural" solution to Post's problem. The decidability problems we stated all gave rise to sets that are complete, and to get something noncomputable and incomplete we resorted to a finite-injury priority argument. Is there an intermediate set that arises naturally from, say, a decision problem?

[1]It is unfortunate that this repeats the notation for the computable sets in §8.3.

I think of classes of Turing degrees as being picked out by two kinds of definitions:

- *Computability-theoretic* definitions are of the form "A degree $\boldsymbol{d}$ is (name) if it contains a set (with some computability-theoretic property)."
- *Lattice-theoretic* definitions are properties defined by predicates that use basic logical symbols (&, $\vee$, $\neg$, $\rightarrow$, $\exists$, $\forall$) plus the partial order relation. Their format is somewhat less uniform but could be summed up as "A degree $\boldsymbol{d}$ is (name) if (it sits above a certain sublattice/it sits below a certain sublattice/there is another degree with a specified lattice relationship to it)."

Computability-theoretic definitions include c.e., simple, low and high degrees. The lattice-theoretic definitions are the kind we saw in §8.3. The definitions for least element, greatest element, join, and meet from that section carry over into this setting. They work for any partial order, since they are defined using only logical symbols and the ordering, though they may be empty in some partial orders (such as greatest element in $\mathcal{D}$). Sometimes the two kinds of definition line up: every c.e. degree except $\boldsymbol{\emptyset}$ contains a simple set, so simplicity in $\mathcal{R}$ is equivalent to the lattice-theoretic definition "not the least element." To say a property is *definable in the lattice* means this latter kind of definition.

For the following lattice-theoretic definition, recall that $\boldsymbol{\emptyset}$ and $\boldsymbol{\emptyset'}$ are definable in $\mathcal{R}$ as least and greatest element, respectively.

Definition 9.1.3. A degree $\boldsymbol{a} \in \mathcal{R}$ is *cuppable* if $(\exists \boldsymbol{b})(\boldsymbol{b} \neq \boldsymbol{\emptyset'} \ \&\ \boldsymbol{a} \vee \boldsymbol{b} = \boldsymbol{\emptyset'})$. The degree $\boldsymbol{a}$ is *cappable* if $(\exists \boldsymbol{c})(\boldsymbol{c} \neq \boldsymbol{\emptyset} \ \&\ \boldsymbol{a} \wedge \boldsymbol{b} = \boldsymbol{\emptyset})$.

Another aspect of lattices that helps distinguish them from each other is their substructures.

Definition 9.1.4. A poset L is *embeddable* into another poset M if there is a one-to-one order-preserving function from L into M.

In Example 2.3.23 we had a lattice with eight elements; call it $\mathcal{L}$. We can embed the four-element diamond lattice into $\mathcal{L}$ in many different ways, where in some embeddings the least element of the diamond maps to the least element of $\mathcal{L}$, in some the greatest element of the diamond maps to the greatest element of $\mathcal{L}$, and in some both happen together.

We have various directions to travel:

- Given a computability-theoretic degree definition, is there an equivalent lattice-theoretic definition?
- Given a lattice-theoretic definition, does it correspond to anything in these particular lattices or is it empty? If the former, is there an equivalent computability-theoretic definition?
- What lattices embed into $\mathcal{D}$ and $\mathcal{R}$? Can we embed preserving lattice properties such as least and greatest element?

On the second topic, let's return to cuppable and cappable degrees. Both do exist, and in fact they relate to a computability-theoretic property. First, a definition.

Definition 9.1.5 (Maass [58])**.** A coinfinite c.e. set A is *promptly simple* if there is a computable function p and a computable enumeration of A such that, for every e, if W_e is infinite, there are s and x such that x enters W_e at stage s and x enters A no later than stage $p(s)$.

As usual, a degree is promptly simple if it contains a promptly simple set. The following result shows that prompt simplicity is definable in $\mathcal{R}$.

Theorem 9.1.6 (Ambos-Spies, Jockusch, Shore, and Soare [2])**.** *The promptly simple degrees are exactly the non-cappable degrees.*

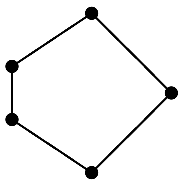
Remember that a set is *low* if its Turing jump is equivalent to $\emptyset'$. A c.e. degree is *low cuppable* if it is cuppable with a low c.e. degree as its cupping partner.

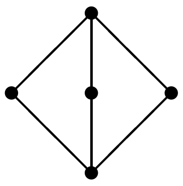
Theorem 9.1.7 ([2])**.** *The non-cappable degrees are exactly the low cuppable degrees. Furthermore, every degree either caps or low cups, though none do both.*

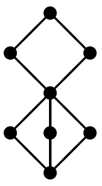
It is possible for a degree to cap and cup, just not cap and low cup. It is also, as we will see below, not possible for a degree to cap and cup via the same partner degree.

A pair of c.e. degrees that are capping partners in $\mathcal{R}$ is also called a *minimal pair*; there will be other degrees between each individual degree and $\mathbf{\emptyset}$ but nothing that is below both of them and above $\mathbf{\emptyset}$. This definition is purely lattice theoretic: we do not have to know where the poset $\mathcal{R}$ comes from to understand it. However, showing that minimal pairs exist in $\mathcal{R}$ – that the lattice-theoretic definition is not empty – is accomplished via a priority argument, building two noncomputable c.e. sets in such a way that anything computable from both of them is simply computable.

To discuss embeddability, we define some simple lattices, all with least and greatest elements. The *diamond* lattice is a four-element lattice with two incomparable intermediate elements. The *pentagon*, N_5, has two intermediate elements that are comparable and one that is not. The *1-3-1*, M_3, has three incomparable intermediate elements. Finally, S_8 is a diamond on top of a 1-3-1.


N_5


M_3


S_8

An important distinction between these lattices is *distributivity.* A lattice is distributive if $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$; i.e., meet and join distribute over each other. The diamond is distributive, but neither the pentagon nor the 1-3-1, and hence S_8, are distributive. In fact, the non-distributive lattices are exactly those that contain the pentagon or the 1-3-1 as a sublattice.

All finite distributive lattices embed into $\mathcal{R}$ preserving least element, as do the pentagon and 1-3-1, but not S_8. Where the embeddability cutoff lies is open. The lattices listed above also embed preserving greatest element instead of least element, but it is an open question as to whether a lattice embeds into $\mathcal{R}$ preserving greatest element if and only if it embeds preserving least element.

It is not always possible to embed preserving both least and greatest element. The *Lachlan non-diamond theorem* [53] shows this for the diamond lattice and $\mathcal{R}$. This is what tells us a c.e. degree cannot cup and cap with the same partner degree, because such a pair would then form the center of a diamond with least element $\boldsymbol{\emptyset}$ and greatest element $\boldsymbol{\emptyset'}$.

For more on embedding, see the survey by Lempp, Lerman, and Solomon [54].

Recall from §8.3 that we may also discuss the lattice $\mathcal{E}$ of c.e. *sets* under $\subseteq$. The greatest element is $\mathbb{N}$ and the least is $\emptyset$. Every pair of sets has a defined meet and join, given by intersection and union.

From join we have the idea of *splitting*, producing a pair of disjoint c.e. sets that union to the given set. Friedberg splitting is a useful example.

Theorem 9.1.8 (Friedberg Splitting [28])**.** *If B is noncomputable and c.e. there is a splitting of B into c.e., noncomputable A_0, A_1 such that if W is c.e. and $W - B$ is non-c.e., then $W - A_i$ is also non-c.e. for $i = 0, 1$ (this implies the A_i are noncomputable by setting $W = \mathbb{N}$).*

Note that if $W - B$ is not c.e., W must have an infinite intersection not only with B, but with each of A_0 and A_1. If $W \cap A_0$ were finite, say, then $W - A_0 =^* W$, and equality modulo finite difference preserves enumerability (Exercise 8.3.1). Therefore, anything that takes a big bite out of B does so in such a way that it takes a big bite out of *both* of the splitting sets. That's what makes this theorem useful; the splitting is very much "down the middle" of B.

Splitting questions in general ask "if B has a certain computability-theoretic property, can we split it into two sets that both have that same property?" or, "Can every c.e. set be split into two sets with a given property?" For example, consider the following definition.

Definition 9.1.9. A c.e. set B is *nowhere simple* if, for every c.e. C such that $C - B$ is infinite, there is some infinite c.e. set $W \subseteq C - B$.

Theorem 9.1.10 (Shore [79])**.** *Every c.e. set can be split into two nowhere simple sets.*

Finally, we may explore the interplay between sets and degrees. Most work in this area is on $\mathcal{E}$ and $\mathcal{R}$. This is a different connection between computability theory and lattice theory than Theorem 9.1.6, since the lattice is that of the c.e. sets under inclusion.

The main topic here is *degree invariance*, which is a correspondence between invariant classes of sets, as in Definition 8.3.9, and collections of degrees. Given Question 9.1.2, discussing invariance for degrees via automorphisms of degrees is problematic. Degree invariance in the following sense is a variation that we can work with.

Definition 9.1.11. A collection of c.e. Turing degrees $\mathcal{C}$ is *invariant* over $\mathcal{E}$ if there exists a collection of c.e. sets $\mathcal{S}$ such that

(i) for every $\boldsymbol{d} \in \mathcal{C}$, there is some $W \in \mathcal{S}$ such that $\deg(W) = \boldsymbol{d}$.

(ii) for every $W \in \mathcal{S}$, $\deg(W) \in \mathcal{C}$.

(iii) $\mathcal{S}$ is invariant under automorphisms of $\mathcal{E}$.

Degree invariance is one way to argue for the naturality of a collection of degrees, that it is less imposed on the c.e. sets and more drawn out from them. We include just one example.

Definition 9.1.12. A coinfinite c.e. set W is *maximal* in $\mathcal{E}$ if, for any c.e. set Z such that $W \subseteq Z \subseteq \mathbb{N}$, either $W =^* Z$ or $Z =^* \mathbb{N}$.

Anything between a maximal set and $\mathbb{N}$ is either essentially the maximal set or essentially all of $\mathbb{N}$. Maximal sets put an end to Post's program to find a noncomputable set which had such a small complement, was so close to being all of $\mathbb{N}$ without being cofinite and hence computable, that it would have to be Turing incomplete (see [68], Chapter III). A maximal set has the smallest possible complement from a c.e. set point of view, but not all maximal sets are incomplete.

However, all maximal sets are high, and every high degree contains a maximal set, as shown by Martin [60]. Since $=^*$ is definable in $\mathcal{E}$, maximality is as well, and so the maximal sets are invariant under automorphisms of $\mathcal{E}$. Hence the high degrees are invariant.

9.2. Randomness

With a fair coin, any one sequence of heads and tails is just as likely to be obtained as any other sequence of the same length. However, our intuition is that a sequence of all heads or all tails, presented as the outcome of an unseen sequence of coin flips, smells fishy. It's just too special, too *nonrandom*. Here we'll present one way to quantify that intuitive idea of randomness and briefly explore some of the consequences and applications.

We will discuss randomness for infinite binary sequences. There are three main approaches to randomness.

(I) Compression: is there a short description of the sequence?

(II) Betting: can you get unboundedly rich by betting on the bits of the sequence?

(III) Statistics: does the sequence have any "special" properties?

Intuitively, the answer to each of those should be "no" if the sequence is to be considered random: a random sequence should be incompressible, unpredictable, and typical. There are different ways to turn these approaches into actual mathematical tests. The most fundamental are the following, in the same order as above.

(I) Kolmogorov complexity: How long an input does a prefix-free Turing machine need in order to produce the first n bits of the sequence as output? If it's always approximately n or more, the sequence is random. (Prefix-free TMs will be defined shortly.)

(II) Martingales: Take a function that represents the capital you hold after betting double-or-nothing on successive bits of a sequence (so the inputs are finite strings and the outputs are non-negative real numbers). Take only such functions that are computably approximable from below. Those are the c.e. martingales; if every such function has bounded output on inputs that are initial segments of your sequence, then the sequence is random.

(III) Martin-Löf tests: A computable sequence of c.e. sets $\{U_n\}$ such that the measure of U_n is bounded by 2^{-n} will have intersection of measure zero; this measure-zero set represents statistical "specialness." If your sequence is outside every such measure zero set, then it is random. (Measure will also be defined shortly.)

The nice thing about the implementations above is that they coincide [76]: A sequence is random according to Kolmogorov complexity if and only if it is random according to c.e. martingales if and only if it is random according to Martin-Löf tests. We call such a sequence *1-random.*

The changes made to these approaches to implement randomness in a different way tend to be restricting or expanding the collection of Turing machines, betting functions, or tests. We might take away the requirement that the machine be prefix-free, or we might allow it a particular oracle. In the opposite direction, we could require our machines not only to be prefix-free, but to obey some other restriction as well. We could allow our martingales to be more complicated than "computably approximable" or we could require they actually be computable. Finally, we could require our test sets have measure equal to 2^{-n} or simply require their measure limit to zero with no restriction on the individual sets, and we could play with the complexity of the individual sets and the sequence. The question of the "correct" cutoff, the one that defines "true randomness," is partly a philosophical one; we want a level high enough to avoid nonrandom behavior (if the cutoff is set too low, we could have a sequence that is called random but always has at least as many 1s as 0s in any initial segment, for instance), but otherwise as low as possible, since it does not seem reasonable to claim a string is compressible when, say, oracle $\emptyset''$ is required for the compression.

What does one do with this concept?

- Prove random sequences exist.
- Look at computability-theoretic properties of random sequences, considering them as sets.
- Compare different definitions of randomness.

- Consider *relative randomness*: if I know this sequence, does it help me bet on/compress/zero in on this other sequence?
- Look for sequences to which every random sequence is relatively random. Prove noncomputable examples exist.
- Extend the definition to other realms, such as sets of sequences.

We will explore only the Kolmogorov complexity approach. Good references for randomness are the books by Downey and Hirschfeldt [22], Nies [67], and Li and Vitányi [56]. For historical reading I suggest Ambos-Spies and Kučera [4], section 1.9 of Li and Vitányi [56], and Volchan [87].

9.2.1. Notation and Basics. For readability, some earlier definitions are reprinted here, along with new ones.

$2^{\mathbb{N}}$ is the collection of all infinite binary sequences and $2^{<\mathbb{N}}$ the collection of all finite binary strings. If you read computer science papers, you may see $\{0,1\}^*$ for $2^{<\mathbb{N}}$. The empty string is denoted λ, Λ, or $\langle\rangle$. 1^n is the string of n 1s and likewise for 0, and if σ and τ are strings, $\sigma\tau$ and $\sigma^\frown\tau$ both mean their concatenation. The notation $\sigma \subseteq \tau$ means σ is a (possibly non-proper) initial segment of τ or, in other words, that there is some string ρ (possibly equal to λ) such that $\sigma\rho = \tau$. Restriction of a string or sequence X to its length-n initial segment is denoted $X \upharpoonright n$.

In this field we tend to use n and the binary expansion of n interchangeably, so we would say the length of n, denoted $|n|$, is $\log n$ (all of our logarithms have base 2). If we are working with a string σ, then $|\sigma|$ is simply the number of bits in σ.

Infinite binary sequences, which we may clearly associate with sets, are also often referred to as *reals*, because we may view them as binary expansions of numbers between 0 and 1. This is not a bijection; any number with a terminating decimal expansion will pair with two sequences, one ending 1000. . . and one ending 0111. . . . Some intuition about numbers carries over to randomness; rational numbers will correspond to computable and hence nonrandom sequences. However, $\pi - 3$ will also give a computable binary sequence, because we can compute its decimal expansion to any number of significant digits. It is in the cloud of anonymous transcendental numbers that the randoms live.

The real number intuition also helps with the topology. An *interval* is $[\sigma] = \{X : \sigma \subset X\}$, for any finite binary string σ. On the real line, it is $[0.\sigma00\ldots, 0.\sigma11\ldots]$. The *open sets* are countable unions of intervals; any interval (and hence finite union of intervals) is also closed, as the bracket notation implies. *Measure* is a way to assign a numerical value to a set to line up with some intuitive notion of size. We use the *coin-toss probability measure*; the measure of an interval $[\sigma]$ is $\mu([\sigma]) = 2^{-|\sigma|}$. It is the probability of landing inside $[\sigma]$ if you produce an infinite binary string by a sequence of coin flips from a fair coin. Intervals defined by longer strings have smaller measure; the sum of the measure of the intervals generated by all strings of a fixed length is 1, the measure of the whole space. The measure of the union of a pairwise-disjoint collection of intervals is the sum of the measure of the intervals.

9.2.2. Kolmogorov Complexity. Prefix-free Turing machines were suggested by Levin [55, 91] and later Chaitin [11] as the best way to approach the compressibility of strings. We will discuss the rationale after the definition and some results.

Definition 9.2.1. A Turing machine M is *prefix-free* if, for every pair of distinct strings $\sigma, \tau \in 2^{<\mathbb{N}}$ such that $\sigma \subset \tau$, M halts on at most one of σ, τ. Such a machine is generally taken to be *self-delimiting*, meaning the read/write head has only one-way movement; this does not restrict the class of functions computed by the machines.

What that means is that no string in the domain of M is a proper initial segment (or *prefix*) of any other string. Halting is therefore not contingent on knowing whether you've reached the end of the string: if you don't halt with the first n bits of input, either there is more input to be had or you will never halt.

Fortunately, there is a universal prefix-free machine. It can be taken to receive $1^e0\sigma$ and interpret that as "run the e^{th} prefix-free machine on input σ."[2] Call such a machine U. It is prefix-free because, in order to have $\sigma \subset \tau$, we must have $\sigma = 1^e 0\sigma'$ and $\tau = 1^e 0\tau'$ with $\sigma' \subset \tau'$, and since machine e is prefix-free, this cannot happen. We make the following definition.

Definition 9.2.2. The *prefix-free Kolmogorov complexity* of a string σ is

$$K(\sigma) = \min\{|\tau| : U(\tau) = \sigma\}.$$

Certainly if we hard-code a string into an input we can output any amount of it with just the constant cost of the program that says "print out the string that's listed here." We may have to do some additional work to put it into a prefix-free form, but this tells us the complexity of a string will have an upper bound related to the string's length. We say a string is random if we can't get much below that upper bound.

Definition 9.2.3. (i) A finite binary string σ is *random* if $K(\sigma) \geq |\sigma|$.

(ii) An infinite binary sequence X is 1-*random* if all of its initial segments are random, up to a constant. That is, $(\exists c)(\forall n)\, K(X \upharpoonright n) \geq n - c$.

Randomness for finite strings is problematic, because it depends on the enumeration of prefix-free Turing machines used in the definition of U. Randomness for infinite sequences is well-defined, however, because all differences in enumeration are swallowed up by the constant term.

9.2.3. The Size of K and Kraft's Inequality. To decide what it means to be incompressible, we needed to know something about the size of K. What upper bound can we assert about it, in terms of the

[2]Of course this assumes that the prefix-free machines can be enumerated. They can, by taking the enumeration of all Turing machines and modifying the machines that turn out to be non-prefix-free (compare Exercise 5.2.16). We work stagewise, with M being the given machine and P being the one we're building. At stage s, run M for s steps on the first s binary strings. If M is not prefix-free, then at some finite stage s^* M will halt on a string comparable to one on which M previously halted. Through stage $s^* - 1$ we let P exactly mimic M, and when (if) we see stage s^*, we define P to diverge on all remaining strings (including the one which witnessed that M was not prefix-free). If M is prefix-free, P will mimic it exactly.

length of σ? The following is part of a larger, more technical theorem, which I have trimmed in half. The proof uses a technique common in showing an upper bound on complexity: construct a specific machine that compresses the string by the desired amount.

Theorem 9.2.4 (Chaitin, [11]). *There is a c such that, for every σ of length n,*

$$K(\sigma) \leq n + K(n) + c.$$

Proof. Consider a prefix-free Turing machine T that computes $T(\tau\sigma) = \sigma$ for any τ and σ such that the universal prefix-free machine U gives $U(\tau) = |\sigma|$. Since T is prefix-free it has an index e in the enumeration of all prefix-free machines, and hence $U(1^{|e|}0e\tau\sigma) = \sigma$. That description has length $2|e| + |\tau| + |\sigma|$ or (if τ is as short as possible) $|\sigma| + K(|\sigma|) + 2|e|$, where e does not depend on σ, and its length certainly bounds the size of $K(\sigma)$. □

This upper bound leads to recursive further bounds:

$$K(\sigma) \leq n + |n| + ||n|| + |||n||| + \dots.$$

Why do we not use this upper bound in our definition of randomness? Because there are no infinite string X and value c such that for all n, $K(X \upharpoonright n) \geq n + K(n) - c$. We could kludge by saying "for infinitely many n" instead of for all, but that's unsatisfying and difficult to work with. And, of course, the definition we gave for randomness is the one that lines up with Martin-Löf tests and martingales.

A (very rough) lower bound on K comes from the Kraft Inequality, a very useful tool in randomness. For a set to be prefix-free, there must be a lot of binary strings missing. Thus we would expect the length of these strings to grow rapidly, and they do.

Theorem 9.2.5 (Kraft Inequality, [49]). *Let $\ell_1, \ell_2, \dots$ be a finite or infinite sequence of natural numbers. There is a prefix-free set of binary strings of length $\ell_1, \ell_2, \dots$ if and only if*

$$\sum_n 2^{-\ell_n} \leq 1.$$

Proof. First, suppose we have a prefix-free set of finite binary strings σ_i with lengths ℓ_i. Consider

$$\mu\left(\bigcup_i [\sigma_i]\right).$$

Certainly this is bounded by 1, and since the set is prefix-free, the intervals are disjoint. Hence the measure of their union is the sum of their measures, and the inequality holds.

Now suppose we have a set of values $\ell_1, \ell_2, \ldots$ such that the inequality holds. We are not working effectively, so we may assume the set is nondecreasing. To find a prefix-free set of binary strings that have these values as their lengths, we carve up the complete binary tree, taking the leftmost string of length ℓ_i that is incomparable to the previously-chosen strings. [For example, if our sequence of values began 3, 4, 7, we would choose 000, 0010, 0011000.] Every binary string of length ℓ corresponds to an interval of size exactly $2^{-\ell}$, so by the inequality there will always be enough measure left to fit the necessary strings, and by our selection procedure all the remaining measure will be concentrated on the right and thus usable. □

This tells us that $K(\sigma)$ has to grow significantly faster than length (overall). The set of programs giving the strings σ is the prefix-free set here, and if those programs have length approximately $|\sigma|$, the sum $\sum_n 2^{-\ell_n}$ is essentially $2^{-1} + 2^{-1} + 2^{-2} + 2^{-2} + 2^{-2} + 2^{-2} + \ldots$, which diverges.

In the proof, we assumed the lengths we were given were in increasing order. In the effectivized version of Theorem 9.2.5 (Theorem 9.2.6, called the Kraft-Chaitin or KC Theorem), we can be given the required lengths of strings in any order and still create a prefix-free set with those lengths. In the proof of Theorem 9.2.5, if the string lengths are given out of order and misbehave enough, our procedure could take bites of varying sizes out of the tree so that when we get to length ℓ_n, although there is at least $2^{-\ell_n}$ measure unused in the tree, it is not all in one piece.

Theorem 9.2.6 (Chaitin [11,12], Levin [55]). *Let $\ell_1, \ell_2, \ldots$ be a collection of values (in no particular order, possibly with repeats, possibly* *finite) such that* $\sum_i 2^{-\ell_i} \leq 1$. *Then from the sequence* ℓ_i *we can effectively compute a prefix-free set* A *with members* σ_i *of length* ℓ_i.

Proof. We present the proof in Downey and Hirschfeldt [22]. Assume that we have selected strings σ_i, $i \leq n$, such that $|\sigma_i| = \ell_i$. By induction, suppose also that we have a string $x[n] = 0.x_1x_2\dots x_m = 1 - \sum_{j\leq n} 2^{-\ell_j}$, and that for every $k \leq m$ such that $x_k = 1$, there is a string $\tau_k \in 2^{<\mathbb{N}}$ of length k incomparable to all σ_j such that $j \leq n$ and all τ_j such that $j < k$ and $x_j = 1$.

Note that since $x[n]$ is the measure of the unchosen portion of $2^{<\mathbb{N}}$, the fact that there are strings of lengths corresponding to the positions of 1s in $x[n]$ means the remaining measure is concentrated into intervals of size at least as large as $2^{-\ell_{n+1}}$ for any ℓ_{n+1} which would allow satisfaction of the Kraft Inequality. Note also that the τ_k are unique and among them they cover the unchosen portion of $2^{<\mathbb{N}}$.

Now we select a string to correspond to ℓ_{n+1}. If $x_{\ell_{n+1}} = 1$, let $\sigma_{n+1} = \tau_{\ell_{n+1}}$ and let $x[n+1]$ be $x[n]$ but with $x_{\ell_{n+1}} = 0$; all τ_k for $k \neq \ell_{n+1}$ remain the same. If $x_{\ell_{n+1}} = 0$, find the largest $j < \ell_{n+1}$ such that $x_j = 1$. For the leftmost string τ of length ℓ_{n+1} extending τ_j, let $\sigma_{n+1} = \tau$. Let $x[n+1] = x[n] - 2^{\ell_{n+1}}$. As a result, in $x[n+1]$, $x_j = 0$, all of the x_k for $j < k \leq \ell_{n+1}$ are 1, and the remaining places of $x[n+1]$ are the same as in $x[n]$. Since τ was chosen to be leftmost in the cone τ_j, there will be strings of lengths $j+1, \dots, \ell_{n+1}$ to be assigned as $\tau_{j+1}, \dots, \tau_{\ell_{n+1}}$ (namely, $\tau_{j+i} = \tau_j 0^{i-1} 1$), as required to continue the induction. □

One way to think of this is as a way to build prefix-free machines by enumerating a list of pairs of lengths and strings, with the intention that the string is described by an input of the specified length.

Theorem 9.2.7 (KC, restated). *Suppose we are effectively given a set of pairs* $\langle n_k, \sigma_k\rangle_{k\in\mathbb{N}}$ *such that* $\sum_k 2^{-n_k} \leq 1$. *Then we can computably build a prefix-free machine* M *and a collection of strings (descriptions)* τ_k *such that* $|\tau_k| = n_k$ *and* $M(\tau_k) = \sigma_k$.

The KC Theorem allows us to implicitly build machines by enumerating "axioms" $\langle n_k, \sigma_k\rangle$ and arguing that the set $\{n_k\}_{k\in\mathbb{N}}$ satisfies the Kraft Inequality. On input τ, the machine enumerates axioms while performing KC until such a time as τ is chosen to be an element of the prefix-free set, corresponding to some $\langle n_k, \sigma_k \rangle$. At that point (if it ever comes), the machine halts and outputs σ_k. This greatly expands the power of the proof technique used in Theorem 9.2.4.

9.2.4. Berry's Paradox, Formalized. *Berry's paradox* is often given as "the smallest number that cannot be described in fewer than thirteen words." That twelve-word phrase is (apparently) a description of the number, giving a self-contradictory statement. Philosophically, the resolution is in disallowing that as a description, since the claimed description references all descriptions (it is *impredicative*). If we formalize *description* in the sense of Kolmogorov complexity, however, we obtain meaningful results, including a proof of Gödel's incompleteness theorem. Durand and Zvonkin [23] give a very clear explanation of this material.[3]

First, let the function $t(n)$ be defined as $\max\{m \in \mathbb{N} : K(m) < n\}$. This exists because the number of possible descriptions of length $< n$ is finite. For all $x > t(n)$, $K(x) \geq n$; in particular, this holds of $t(n) + 1$. If t is a computable function, however, we need only n and some constant-size programming to get $t(n) + 1$, leading to a contradiction for sufficiently large n: $K(t(n) + 1) \leq K(n) + c_1 \leq \log n + K(\log n) + c_2$, where c_1 and c_2 do not depend on n. In fact, $t(n)$ grows faster than any computable function.

We may ramp this up to incompleteness by considering the provability of $K(x) \geq m$. We start with an axiomatizable sound theory T, which for our purposes here is simply a system for computably enumerating provable logical sentences (for details, see §§5.3 and 9.3). Suppose toward a contradiction that for such a T, $(\forall m)(\exists x)[(K(x) \geq m)$ is provable]. This gives an algorithm to find x from m, as follows: enumerate all theorems of T, and return x as soon as one of the form $K(x) \geq m$ is found. But then again, $K(x) \leq K(m) + c_1 \leq \log m + K(\log m) + c_2$ gives a contradiction for sufficiently large m.

This gives an entire collection of unprovable statements, namely $K(x) \geq m$ for any sufficiently large m. The function t and the provability discussion formalize Berry's Paradox in two ways. For t we

[3]Note that they use plain complexity (§9.2.6) but denote it by K.

use "the smallest integer n such that $K(n) > m$;" this does not give a description in the sense of Kolmogorov complexity, so there is no paradox to t existing (it simply can't be computable, or it *would* give a Kolmogorov description). However, we then changed it to "the integer n corresponding to the first theorem of the form $K(n) \geq m$ in the enumeration of theorems." That description did land us in the realm of Kolmogorov complexity and hence paradox, with the conclusion that for some m no such n exists.

Another result of formalizing Berry's Paradox in the manner of t above is the following. Recall that a *simple* set is a c.e. set A such that $\overline{A}$ is infinite but contains no infinite c.e. subsets.

Theorem 9.2.8 (Kolmogorov [46, 47])**.** *The set of nonrandom numbers is simple.*

Proof. The set for which we need to prove simplicity is $A = \{x : K(x) < |x|\}$. If we dovetail the computations of a universal prefix-free Turing machine U, for any nonrandom x, we will eventually see U output x on an input of length $< |x|$. At that point we can put x into A, so A is c.e. We know the set of random numbers is infinite, so $\overline{A}$ is infinite.

We now show that every infinite c.e. set W_e contains a nonrandom element. Let i be such that $\varphi_i(\langle e, n\rangle)$ is the n^{th} element enumerated into W_e, $x_{e,n}$, if $|W_e| \geq n$, and undefined otherwise. Let $h(e, n)$ be the string that instructs U to emulate $\varphi_i(\langle e, n\rangle)$; h is a total computable function. Note that $h(e, n)$ is a description of $x_{e,n}$.

Set $t(n) = \max_{e \leq n} h(e, n)$; t is also total computable. For any index e, $t(n)$ will take $h(e, n)$ into account on all but finitely many values of n. We will use t to define a subset of W_e such that for some n, h gives a short description of the n^{th} element of the subset. Since that number is also in W_e itself, W_e contains a nonrandom element.

Given W_e, enumerate a subset Y so that the n^{th} element of Y, y_n, is greater than $t(n)$. If W_e is infinite, for any n it will contain elements larger than $t(n)$. When we see such an element enumerated we can put it into Y, which will therefore be infinite.

Since Y is c.e., it is $W_{\hat{e}}$ for some $\hat{e}$. However, by the choice of $y_n > t(n)$ and the fact that for almost all n, $t(n) \geq h(\hat{e}, n)$, we know there is some n such that $x_{\hat{e},n} = y_n > t(n) \geq h(\hat{e}, n)$. For that n, $h(\hat{e}, n)$ gives a short description of $x_{\hat{e},n}$, so $x_{\hat{e},n}$ is a nonrandom element of $W_{\hat{e}} = Y$ and hence of W_e. $\square$

This result gives incompleteness for many theories simultaneously.

Corollary 9.2.9. *There exists a computably enumerable set B with infinite complement such that for all axiomatizable sound theories T there are only finitely many n such that $n \notin B$ is true and provable in T.*

Proof. Let B be the set of nonrandom numbers. Since T is axiomatizable we can enumerate the set D of elements provably in the complement of B. However, since B is simple, D must be finite. $\square$

Kolmogorov complexity may also be used to prove Gödel's second incompleteness theorem (see §9.5), but on that topic I will merely point you to the paper by Kritchman and Raz [50].

9.2.5. Halting Probability. Just as the halting problem is a fairly explicit noncomputable set, we may use halting to give a fairly explicit random number. This number is the *halting probability*, typically called (Chaitin's) Ω.

Before we define Ω, a brief discussion of degree. Every degree $\geq \boldsymbol{\emptyset'}$ contains a random set, and for every degree $\boldsymbol{d}$ there is a degree $\boldsymbol{d_1} \geq \boldsymbol{d}$ that contains a random. However, unless $\boldsymbol{d} \geq \boldsymbol{\emptyset'}$, there will also be some degree $\boldsymbol{d_2} \geq \boldsymbol{d}$ that does not contain a random.

No computably enumerable set is random. It is clear that any computable set cannot be random, so we need only consider noncomputable and hence infinite c.e. sets. From a betting perspective, this allows us to wait to place bets until we see a new 1 in the sequence ahead of where we have thus far placed bets. At that time we can bet evenly on 0 and 1 for the bits up to that new 1, neither gaining nor losing money, and then place all of our money on 1. Since we can thereby double our money infinitely many times, the sequence cannot be random.

However, it is possible for a set of c.e. degree to be random, as long as that degree is $\mathbf{\emptyset'}$. It is even possible for the set itself to be left-c.e., as defined in §6.3.[4] Ω is such a random. For U a universal prefix-free Turing machine, Ω is defined as a real:

$$\Omega = \sum\{2^{-|\sigma|} : \sigma \in 2^{<\mathbb{N}} \ \& \ U(\sigma)\!\downarrow\}.$$

In §6.3, left-c.e. sequences were defined as those possessing an approximation that increases in value, and here that is built into the definition: as we see computations halt, we add the measure of the input string to the current value of Ω.

Since U must halt on some σ, $\Omega > 0$. By the Kraft Inequality 9.2.5, $\Omega \leq 1$. In fact, U's domain is a subset of $\{1^e0\tau : e \in \omega\}$, and for e the index of a Turing machine with empty domain $U(1^e0\tau)\!\uparrow$ for all τ. Therefore $\Omega < 1$.

Claim 9.2.10. *Let σ be a binary string of length at most n. From $\Omega \upharpoonright n$ we may effectively determine whether $U(\sigma)\!\downarrow$.*

Proof. Observe that $\Omega \upharpoonright n \leq \Omega < \Omega \upharpoonright n + 2^{-n}$. We dovetail the computations of U on all inputs and approximate Ω by Ω_s, which begins as zero.

If $U(\tau)\!\downarrow$ at stage s, let $\Omega_s = \Omega_{s-1} + 2^{-|\tau|}$. Eventually we will see $\Omega_s \geq \Omega \upharpoonright n$. If σ is not among the programs for which we have already seen halting, it will never halt, as it would add at least 2^{-n} to Ω_s, making it larger than Ω. □

Chaitin's Ω has some philosophical interest as "the number of Wisdom" (Bennett and Gardner [9]): if you know $\Omega_{1:10000}$ and have an axiomatizable mathematical theory expressible in 10,000 or fewer bits, you can find whether its statements are true, false, or independent. This includes Goldbach's Conjecture, the Riemann Hypothesis, and most other conjectures in mathematics which would be refutable with finite counterexamples – the programs looking for such counterexamples will or will not halt, and Ω knows their behavior.

However, the function $t(n)$, giving the amount of time needed to find all halting programs of length less than n from $\Omega \upharpoonright n$, grows

[4] If you read the literature, be aware that in randomness a left-c.e. real is sometimes referred to as just a c.e. real.

faster than all computable functions, so knowing Ω gives no practical help.

Before leaving Ω we should prove it is random.

Claim 9.2.11. *Ω is K-random; that is, $(\exists c)(\forall n)\ K(\Omega \upharpoonright n) \geq n - c$.*

Proof. We demonstrate that there is a computable function φ such that $K(\varphi(\Omega \upharpoonright n)) > n$. The difference in K-complexity between σ and $f(\sigma)$ for any computable f is only a constant, so this will prove the claim.

From Claim 9.2.10 it follows that from $\Omega \upharpoonright n$, one may calculate all programs σ of length at most n on which U halts. There will be, therefore, some τ_n that is not yet computed by any such σ, and therefore such that $K(\tau_n) > n$. Let $\varphi(\Omega \upharpoonright n) = \tau_n$. □

9.2.6. Why Prefix-Free? We could define the complexity of σ as the minimum length input that produces σ when given to the standard universal Turing machine, rather than the universal prefix-free Turing machine. That is the *plain Kolmogorov complexity* of σ, denoted $C(\sigma)$.[5] However, as a standard for complexity it has some problems, even at the level of finite strings.

The first undesirable property of C is non-subadditivity: for any c there are x and y such that $C(\langle x, y\rangle) > C(x) + C(y) + c$. K, on the other hand, *is* subadditive, because with K we can concatenate descriptions of x and y, and the machine will be able to tell them apart: the machine can read until it halts, assume that is the end of x's description, and then read again until it halts to obtain y's description. Some constant-size code to specify that action and how to encode the x and y that result, and we have $\langle x, y\rangle$.

The second undesirable property is nonmonotonicity on prefixes: the complexity of a substring may be greater than the complexity of the whole string. For example, a power of 2 has very low complexity, so that if $n = 2^k$ then $C(1^n) \leq \log\log n + c$ (i.e., a description of k, which is no more than $\log k$ in size, plus some machinery to take

[5]Historically there has been some variation in randomness notation. The older the paper, the more likely you are to see K for plain complexity, and either H or KP for prefix-free complexity. For a time, prefix-free complexity was also unfortunately referred to as "prefix complexity."

powers and print 1s). However, once k is big enough, there will be numbers smaller than n that have much higher complexity because they have no nice concise description in terms of powers of smaller numbers or similar. For such a number m, $C(1^m)$ would be higher than $C(1^n)$ even though 1^m is a proper initial segment of 1^n.

The underlying problem is that $C(\sigma)$ contains information about the length of σ (that is, n) as well as the pattern of bits. For most n, about $\log n$ of the bits of the shortest description of σ will be used to determine n. What that means is that for simple strings of the same length n, any distinction between the pattern complexity of the two strings will be lost to the domination of the complexity of n.

Another way of looking at it is that C allows you to compress a binary sequence using a ternary alphabet: 0, 1, and "end of string." That's not a fair measure of compressibility, and as stated above, it leads to some technical as well as philosophical problems. The main practical argument for K over C, though, is that K gives the definition that lines up with the characterizations of randomness in terms of Martin-Löf tests and martingales.

9.2.7. Relative Randomness and *K*-Triviality. Our final topic is a way to compare sets to each other, more finely grained than saying both, one, or neither is random. For example, the bit-flip of a random sequence is random, but if we are given the original sequence as an oracle, its bit-flip can be produced by a constant-size program. Therefore no sequence's bit-flip is random *relative to* the original sequence.

Definition 9.2.12. (i) The *prefix-free Kolmogorov complexity of σ relative to A* is $K^A(\sigma) = \min\{|\tau| : U^A(\tau) = \sigma\}$.

(ii) A set or sequence B is *A-random* (or 1-A-random) if

$$(\exists c)(\forall n)[K^A(B \upharpoonright n) \geq n - c].$$

It should be clear that if B is nonrandom, it is also non-A-random for every A. Adding an oracle can never *increase* the randomness of another string; it can only derandomize. That is, if RAND is the set of all 1-random reals and RAND^A is the set of all A-random reals, then for any A, $\text{RAND}^A \subseteq \text{RAND}$. The question is then for which A

equality holds; certainly for any computable A it does, but are there others? Hence we have the following definition, a priori perhaps a duplication of "computable."

Definition 9.2.13. A set A is *low for random* if $\mathrm{RAND}^A = \mathrm{RAND}$.

A low for random set clearly cannot itself be random, because any sequence derandomizes itself and its infinite subsequences. The term "low" is by analogy with ordinary computability theory, where A is low if the halting problem relativized to A is unchanged in degree from the nonrelativized halting problem. As there exist noncomputable low sets, Kučera and Terwijn [52] have shown that there exist noncomputable low for random sets.

A low for random sequence is one that K cannot distinguish from a computable sequence, with respect to its usefulness as an oracle. Another way for K to distinguish between sequences is their initial segment complexity. Hence we have the following definition.

Definition 9.2.14. A real X is *K-trivial* if the prefix-free complexity of its length-n initial segments is bounded by the complexity of n; that is, $(\exists c)(\forall n)(K(X \upharpoonright n) \leq K(n) + c)$.

The question is, again, whether there are any noncomputable K-trivial reals. Certainly all computable reals X are such that $K(X \upharpoonright n) \leq K(n) + c$; the constant term holds the function that generates the initial segments of X, and then getting an initial segment is as simple as specifying the length you want.

Theorem 9.2.15 (Zambella [90], after Solovay [83])**.** *There is a noncomputable c.e. set A such that $(\exists c)(\forall n)(K(A \upharpoonright n) \leq K(n) + c)$.*

The truly remarkable thing is that these are the same class of reals: a real is low for random if and only if it is K-trivial. The proof is extremely difficult, involving work by Gács [29], Hirschfeldt, Nies, and Stephan [38], Kučera [51], and Nies [66].

9.3. Some Model Theory

Both computable model theory (§9.4) and reverse mathematics (§9.5) involve model theory, another area of mathematical logic. Classes on propositional and predicate logic primarily cover elements of model theory. This section repeats some definitions from §5.3, though with more detail and examples.

A *language* $\mathcal{L}$ is a collection of symbols representing constants, functions, and relations (the latter two come with a defined *arity*, or number of inputs). Formulas in $\mathcal{L}$ may use those symbols as well as variables and the logical symbols we met in §2.1 (e.g., $\neg$, $\forall$). A *structure* for that language (or $\mathcal{L}$-structure) is a collection of elements, called the *universe*, along with an *interpretation* for each relation. For example, $\mathbb{N}$ with the usual equality and ordering is a structure for the language $(=,<)$; we would denote it $(\mathbb{N},=^{\mathbb{N}},<^{\mathbb{N}})$. There are many possible structures for any given language, even after you take the quotient of the collection of structures by the equivalence relation of isomorphism.

To get to a *model*, we add axioms, a collection of logical sentences assumed true (recall sentences are formulas with no free variables, so that they have a truth value). These sentences may use all of the standard logical symbols, as well as variables and any symbol from the language. A set of sentences closed under logical deduction is a *theory*.[6] By convention, we may specify a theory by the axioms that generate it, with the understanding that the theory is the closure of the given set. A structure for the language is a model of the theory if, when the language is interpreted as the structure specifies and the domain of quantification is the universe of the structure, all the sentences in the theory are true. This may greatly restrict the number of structures we can have; in fact there are theories for which there is only one model with a countable universe, up to isomorphism (such theories are called *countably categorical*).

An isomorphism between $\mathcal{L}$-structures $\mathcal{A} = (A, c^{\mathcal{A}}, f^{\mathcal{A}}, R^{\mathcal{A}})$ and $\mathcal{B} = (B, c^{\mathcal{B}}, f^{\mathcal{B}}, R^{\mathcal{B}})$ is a bijection $F : A \to B$ such that $F(c^{\mathcal{A}}) = c^{\mathcal{B}}$, and for any tuple $\overline{a}$ of the appropriate arity, if $F(a_i) = b_i$ and $F(k) = \ell$, then $f^{\mathcal{A}}(\overline{a}) = k \iff f^{\mathcal{B}}(\overline{b}) = \ell$ and $R^{\mathcal{A}}(\overline{a}) \iff R^{\mathcal{B}}(\overline{b})$. This definition extends to other languages as expected. When the

[6]This is standard, but not universal; some texts use *theory* to refer to any set of sentences.

isomorphism is between a structure and itself, it is called an *automorphism.*

Let us consider the example $\mathcal{L} = (0, 1, =, <, +, \cdot)$, where 0 and 1 are constant symbols, $=$ and $<$ are binary relations, and $+$ and $\cdot$ are binary functions. We can create many structures for $\mathcal{L}$; let's look at a few that have countable universes.

(i) $\mathbb{N}$, with the usual meanings for all of these symbols;

(ii) $\mathbb{Q}$, with the usual meanings for all of these symbols;

(iii) $\mathbb{N}$, with the usual meanings for everything except $=$ and $<$; $=$ interpreted as equality modulo 12 (so $12 = 0$), and $n < m$ true if $n \pmod{12} < m \pmod{12}$ in the usual ordering (so $12 < 1$);

(iv) $\mathbb{N}$, with 0 and 1 interpreted as usual, $=$ interpreted as nonequality, $<$ interpreted as $>$, and $+$ and $\cdot$ interpreted as $\cdot$ and exponentiation, respectively.

The point of (iv) is to show that we do not have to abide by the conventional uses of the symbols. However, it is likely to make most logicians squirm, because it is standard to make $=$ a special symbol that may only be interpreted as genuine equality. We will soldier on, however, and consider some axioms on $\mathcal{L}$.

(I) $\neg(0 = 1)$;

(II) $(\forall x)(\forall y)(x < y + 1 \rightarrow (x < y \ \vee \ x = y))$;

(III) $(\forall x)(\neg(x < 0))$;

(IV) $(\forall x)(\forall y)(\exists z)((\neg(y = 0) \ \& \ \neg(x = 0)) \rightarrow x \cdot z = y)$.

Axiom I is true in structures (i), (ii), and (iii), but not (iv): 0 and 1 are nonequal, but in (iv) that is exactly the interpreted meaning of the symbol $=$. Axiom II is true in structures (i) and (iii). It is false in (ii), as shown by $x = 2$ and $y = 1.5$. Axiom II is also true in structure (iv), where in conventional terms it says if $x > y \cdot 1$, then $x > y$ or $x \neq y$.

Axiom III is clearly true in structures (i) and (iii) and false in (ii) and (iv) (where in the latter it asserts that no number is positive). Axiom IV asserts (in structures (i)–(iii)) that any nonzero number is divisible by any other nonzero number. It is false in structure (i) and true in (ii); it is false in (iii) but it takes maybe a bit more thought to see it. An example of axiom IV's failure in structure (iii) is $x = 2$ and $y = 3$: no multiple of x will be odd, but all members of y's equivalence class modulo 12 are odd. Axiom IV also fails in structure (iv), where it says in conventional terms that if x and y are both zero, there is a power to which one can raise x to get something not equal to y.

We could say structure (i) is a model for the set $\mathcal{S}$ of axioms I, II, and III. If we call the model $\mathcal{M}$, we denote this as $\mathcal{M} \models \mathcal{S}$. However, structure (i) is not a model for the last axiom; call it φ: $\mathcal{M} \not\models \varphi$. This sense of truth is referred to as *semantic*, and semantic truth is complete: for all sentences φ and models $\mathcal{M}$ over the same language, either $\mathcal{M} \models \varphi$ or $\mathcal{M} \models \neg\varphi$. The set $\{\varphi : \mathcal{M} \models \varphi\}$ is called the *theory of* $\mathcal{M}$, $\mathrm{Th}(\mathcal{M})$.

There is a *syntactic* notion of truth as well, which is membership in the theory itself, or provability from the axioms that generate it. In this setting, not every sentence is true or false. If φ is provable from axioms $\mathcal{S}$, we write $\mathcal{S} \vdash \varphi$. If for every $\mathcal{L}$-sentence φ, either $\mathcal{S} \vdash \varphi$ or $\mathcal{S} \vdash \neg\varphi$, $\mathcal{S}$ (or the theory it generates) is called *complete.* When neither φ nor $\neg\varphi$ is provable, φ is called *independent.*

The syntactic and semantic sides are connected by the soundness and completeness theorems. Soundness says that anything provable from a set of axioms $\mathcal{S}$ will be true in all models of $\mathcal{S}$: for all $\mathcal{M} \models \mathcal{S}$, $\mathcal{S} \vdash \varphi \quad\rightarrow\quad \mathcal{M} \models \varphi$. Essentially, this says logic works the same way in the models as in the syntax, and it means that we may use a theory and a set of axioms that generate it interchangeably, in terms of models. The converse implication is Gödel's completeness theorem, which is a different sense of completeness than completeness of a theory. It says that if φ is true in every model of $\mathcal{S}$, then $\mathcal{S} \vdash \varphi$. The closure of a set of axioms under logical deduction is exactly the truths on which models of the axioms all agree; on every independent sentence there will be models in disagreement.

A practical consequence of completeness and soundness is that if you want to show φ follows from $\mathcal{S}$, you may give a logical deduction, and if you want to show $\mathcal{S} \not\vdash \varphi$ (which is *not* the same as $\mathcal{S} \vdash \neg\varphi$), you may construct a model of $\mathcal{S}$ in which φ is false. These are typically the easier methods.

Thus far we have spoken only of *first-order* theories, where the domain of quantification is always the universe of objects, not anything more complicated such as sets of objects. In practice we might want to quantify over both elements and sets of elements, which puts us in the realm of *second-order logic*. The discussion above carries over to second-order logic, but models now consist of a number universe M and a set universe that is a subset of $\mathcal{P}(M)$, as well as interpretations of all of the language symbols. A prime example is $\mathbb{N}$ together with $\mathcal{P}(\mathbb{N})$, but in computability theory we often can restrict the subsets included and still get a model of our theory. More on this after a useful example.

9.3.1. Arithmetic. *Peano arithmetic* and *Robinson arithmetic* are important examples of first-order logical theories; the language in each case is $(0, S, +, \cdot, =, <)$, where 0 is a constant, S is a unary function, $+$ and $\cdot$ are binary functions, and $=$ and $<$ are binary relations. Hájek and Pudlák [35] includes a thorough discussion of these two theories.

The axioms of Robinson arithmetic, denoted Q, PA^-, or P^-, are the axioms of basic arithmetic. They say $+$ and $\cdot$ are associative and commutative and $\cdot$ distributes over $+$; S is one-to-one and its range is the entire universe except 0; $+$ and $\cdot$ interact as expected with 0 and S; $=$ is equality;[7] and $<$ is a total order with $x < y$ holding if and only if there is a nonzero z such that $x + z = y$.[8]

The axioms of Peano arithmetic, or PA, are PA^- together with an infinite collection of induction axioms, referred to collectively as the *induction schema*: For each formula $\varphi(x)$ of the language,

$$[\varphi(0) \,\&\, (\forall n)(\varphi(n) \rightarrow \varphi(S(n)))] \;\rightarrow\; (\forall n)\varphi(n)$$

is an axiom.

The *standard model* is $\mathcal{N}$, with universe $\mathbb{N}$ and the usual interpretations of all of the symbols; $\mathrm{Th}(\mathcal{N})$ is often called *true arithmetic*. It is a model of PA. However, we can also have *nonstandard models* that are not isomorphic to the standard model. Because $<$ is a total order with 0 at the bottom, and 0 is also the only element that is

[7]Typically this goes without saying.

[8]Notice we don't actually need $<$ because it is definable from $+$ and 0. Some authors will omit it from the language and some will include $\leq$ instead.

not successor to anything, every model $\mathcal{M}$ begins with $0^{\mathcal{M}}$ and continues with successors $S^{\mathcal{M}}(\ldots(S^{\mathcal{M}}(0^{\mathcal{M}}))\ldots)$. Nonstandard models have additional elements that are larger than all of the successors of 0, and they come in blocks isomorphic to the integers. However, the structure is complicated, because adding or multiplying two nonstandard elements, even from the same block, will propel you into a higher block. One nonstandard model of PA$^-$ may be thought of as the polynomials with integer coefficients and positive leading coefficient, together with the zero polynomial. The constant polynomials are the standard natural numbers, and thereafter each block is given by the terms of nonzero degree, with the constant term, ranging over all of $\mathbb{Z}$, giving the particular element of the block. After the standard numbers you have the $x + a$ block, the $2x + a$ block, and, after some time, the $x^2 + x + a$ block, etc. The correct interpretation of $+$ and $\cdot$ is given simply by the usual arithmetic of polynomials.

There is no computable nonstandard model of PA (Theorem 9.4.2), which shows that the model described above must have a failure of induction for some formula. Induction in a nonstandard model is a very strong statement: the antecedent requires only that the formula be true of successors of 0, which is to say standard numbers, but the conclusion is that the formula holds for every element of the model, standard or not. Viewed in that way, it is not surprising that nonstandard models of PA are difficult to construct.

9.4. Computable Model Theory

The area of computable model theory applies our questions of computability or levels of noncomputability to structures of model theory. The source for this section is primarily Ash and Knight's *Computable Structures and the Hyperarithmetical Hierarchy* [7]; Volume 1 of the *Handbook of Recursive Mathematics* [26] and a survey article by Harizanov [36] are also good references. In this section, all structures will be countable; in fact, one frequently assumes that every structure has universe $\mathbb{N}$, and we will follow this convention.

The degree of a countable structure is the least upper bound of the degrees of the functions and relations of the language as interpreted in that structure. We can ask many questions, including the following:

- What degrees are possible for models of a theory?
- What degrees are possible for models of a theory within a particular isomorphism type (equivalence class under isomorphism)?
- Given a degree $\boldsymbol{d}$, can we construct a theory with no models of degree $\boldsymbol{d}$?
- What happens if we restrict to computable models and isomorphisms? Do we get more or fewer isomorphism types, for example?

One major example is models of Peano arithmetic. As mentioned in §9.3, in a nonstandard model the induction axiom of PA is very strong; it is often restated in the following form.

Proposition 9.4.1 (Overspill). *If $\mathcal{M} \models PA$ is nonstandard and $\varphi(x)$ is a formula that holds for all finite elements of M, then $\varphi(x)$ also holds of some infinite element.*

Theorem 9.4.2 (Tennenbaum [84]). *If $\mathcal{M} \models PA$ is nonstandard, it is not computable.*

Proof. Let X and Y be computably inseparable c.e. sets (see Exercise 5.2.21). There are natural formulas that mean $x \in X_s$, $y \in Y_s$, as well as $p_n|u$ (the n^{th} prime divides u). Let $\psi(x,u)$ say

$$\forall y([(\exists s \leq x\ (y \in X_s)) \rightarrow p_y|u]\ \&\ [(\exists s \leq x\ (y \in Y_s)) \rightarrow p_y \not| u]).$$

For all finite c, $\mathcal{M} \models \exists u \psi(c,u)$, because the product of all primes corresponding to elements of X_c is such a u.

By Overspill, Proposition 9.4.1, there is an infinite c' such that $\mathcal{M} \models \exists u \psi(c',u)$. For d such that $\mathcal{M} \models \psi(c',d)$, let $Z = \{m \in \mathbb{N} : \mathcal{M} \models p_m|d\}$. Z is a separator for X and Y, and Z is computable from $\mathcal{M}$. Since X and Y are computably inseparable, Z and hence $\mathcal{M}$ are noncomputable. $\square$

For the following theorem we need a definition.

Definition 9.4.3. A *trivial structure* is one in which there is a finite set of elements $\overline{a}$ such that any permutation of the universe that fixes $\overline{a}$ pointwise is an automorphism.

For example, $\mathbb{N}$ with unary relations that are all either empty or the entire universe and only finitely many constants is trivial. Any permutation of $\mathbb{N}$ that preserves the constants will also preserve the relations, and hence be an automorphism.

Theorem 9.4.4 (Solovay, Marker, Knight [48]). *Suppose $\mathcal{A}$ is a nontrivial structure. If $\mathcal{A} \leq_T X$, there exists a structure $\mathcal{B}$ isomorphic to $\mathcal{A}$ via F such that $\mathcal{B} \equiv_T X$, and in fact $F \oplus \mathcal{A} \equiv_T X$.*

The proof uses F to code X into $\mathcal{B}$. For example, suppose $\mathcal{A}$ is a linear order, with universe $\{a_0, a_1, \ldots\}$. We create $\mathcal{B}$ with universe $\{b_0, b_1, \ldots\}$ and let F map $\{a_{2n}, a_{2n+1}\}$ to $\{b_{2n}, b_{2n+1}\}$ in some order. If $n \in X$, b_{2n+1} is the image of the $<^{\mathcal{A}}$-larger element, and if $n \notin X$, b_{2n} is the image of the $<^{\mathcal{A}}$-larger element. The interpretation $<^{\mathcal{B}}$ is exactly as necessary to make F an isomorphism. $\mathcal{B}$ computes X because $n \in X \leftrightarrow b_{2n} <^{\mathcal{B}} b_{2n+1}$.

Corollary 9.4.5. *Peano arithmetic has standard models in all Turing degrees.*

This follows from the fact that the standard model $\mathcal{N}$ is computable and nontrivial. Linear orders are a useful example for a number of our questions.

Theorem 9.4.6 (Miller [63]). *There is a linear order $\mathcal{A}$ that has no computable copy, but such that for all noncomputable $X \leq_T \emptyset'$, there is a copy of $\mathcal{A}$ computable from X.*

Note that whether we can remove $X \leq_T \emptyset'$ from the hypothesis is an open question. There exist noncomputable Turing degrees "off to the side:" not above any noncomputable Δ^0_2 degree.

A *computably categorical* structure is a computable $\mathcal{A}$ such that if $\mathcal{B}$ is computable and isomorphic to $\mathcal{A}$, the isomorphism may be chosen to be computable. Exercise 6.2.4 built a linear order on universe $\mathbb{N}$ such that the successor relation is not computable. However, the ordering itself is computable, so as a linear order the structure is computable. In the standard ordering on $\mathbb{N}$, however, the successor relation *is* computable, so these are two isomorphic computable orderings that cannot be computably isomorphic. The discrete linear order with one endpoint is not computably categorical.

Dense linear orders, such as $\mathbb{Q}$, however, are computably categorical. The non-computable construction of an isomorphism between two DLOs is called a *back and forth* argument. Start by mapping the endpoints, if any, appropriately. Then, working between $\mathcal{A}$ and $\mathcal{B}$, select an unmapped element from A and find an unmapped match to it in B, with the correct inequality relationships to previously-chosen elements. Then select an unmapped element of B and match it to an unmapped element of A. Trading off in which ordering you select your element ensures the map is onto in each direction; using only unused elements ensures it is one-to-one. Density and lack of endpoints guarantee finding a match. It is not too difficult to fill in the details of this construction and to show that it can be done computably whenever the DLOs $\mathcal{A}$ and $\mathcal{B}$ are computable (nonuniformly knowing which elements are the endpoints, if needed), proving that DLOs are computably categorical.

If a structure is not computably categorical, we can ask how many equivalence classes its computable copies have under computable isomorphism. This is called the *computable dimension* of the structure, and any finite value may be realized (Goncharov [31,32]). However, in linear orders, the only computable dimensions possible are infinity or 1 (categoricity), and the latter occurs if and only if the order has only finitely many successor pairs (Dzgoev-Goncharov [33]). This kind of analysis of the relationship between the structure of a mathematical object and its computability-theoretic properties is one of the main themes in computable model theory and computable mathematics in general.

9.5. Reverse Mathematics

You have likely noticed that computability and the related areas we've discussed involve a lot of hierarchies: Turing degrees, the arithmetic hierarchy, computation of kinds of fixed points, relative randomness. These all classify sets according to some complexity property. The program of reverse mathematics is also an effort to classify objects according to complexity, but here the objects are theorems of ordinary mathematics, and the complexity is the strength of the mathematical tools that are required to prove them.

The very big picture comes from Gödel's incompleteness theorems. We have explored the first one, which states that any computable, consistent theory fails to prove some statement true in the standard model. The second one, however, is more relevant here, and says one such unprovable statement is the assertion of the theory's own consistency. This gives rise to something referred to as the *Gödel hierarchy*, where a theory T is less than another theory Q if Q includes the statement of T's consistency.

Now, what does this have to do with ordinary mathematics? The logical axioms we consider, giving rise to these theories T, Q, and so forth, are about induction, how arithmetic works, what sets we can assert to exist (*comprehension*), and other fundamental notions. If we don't have all of the tools of standard mathematics, we may not be able to prove a theorem that we know is true in the "real world." The classification achieved by reverse mathematics is finding cutoff points where theorems go from nonprovable to provable, called their *proof-theoretic strength* or *consistency strength*.

The main reference for reverse mathematics is Steve Simpson's book *Subsystems of Second-Order Arithmetic* [80]; this section is drawn from that book and notes from lectures by Simpson and Jeff Hirst. The material on the arithmetic hierarchy in §7.2 will be useful for this section; also note some of the theorems of ordinary mathematics mentioned below will require vocabulary from analysis or algebra that is not defined here.

9.5.1. Second-Order Arithmetic. We work in the world of *second-order arithmetic*, or Z_2. This was mentioned in §9.3 but we expand it here. The language of Z_2 is *two-sorted*; the variables of Z_2 come in two kinds: number variables intended to range over $\mathbb{N}$ and set variables intended to range over $\mathcal{P}(\mathbb{N})$. The constants are 0 and 1, the functions are $+$ and $\cdot$, and the relations are $=$, $<$, and $\in$, where the first two are relations on $\mathbb{N} \times \mathbb{N}$ and the third on $\mathbb{N} \times \mathcal{P}(\mathbb{N})$. Conventionally we also use $\leq$, numerals 2 and up, superscripts to indicate exponentiation, and shorthand negations like $\notin$; these are all simply abbreviations and are definable in the original language. As explained in §9.3, a model $\mathcal{M}$ of Z_2 (or a subsystem thereof) will specify not only the universe M of the number variables, but also the universe $\mathcal{S}$

for the set variables, where $\mathcal{S} \subseteq \mathcal{P}(M)$. M and $\mathcal{S}$ are called the first and second order parts of $\mathcal{M}$, respectively.

The axioms of second-order arithmetic are as follows, with universal quantification as needed to make them sentences:

(1) basic arithmetic:
   (a) $n + 1 \neq 0$, $\neg(m < 0)$
   (b) $m < n{+}1 \leftrightarrow (m < n \vee m = n)$, $m{+}1 = n{+}1 \rightarrow m = n$
   (c) $m + 0 = m$, $m \cdot 0 = 0$
   (d) $m + (n + 1) = (m + n) + 1$, $m \cdot (n + 1) = (m \cdot n) + m$.

(2) induction: $(0 \in X)$ & $(\forall n)(n \in X \rightarrow n + 1 \in X)) \rightarrow (\forall n)(n \in X)$.

(3) comprehension (set existence): $(\exists X)(\forall n)(n \in X \leftrightarrow \varphi(n))$ for each formula $\varphi(n)$ in which X does not occur freely.

The last two axioms together say that for any second-order formula $\varphi(n)$,

$$(\varphi(0) \mathbin{\&} (\forall n)(\varphi(n) \rightarrow \varphi(n + 1))) \rightarrow (\forall n)\varphi(n). \tag{9.1}$$

We create subsystems of Z_2 by restricting comprehension and induction. Restricting comprehension has the effect of also restricting induction, because the induction axiom as written works with sets (for which we must be able to assert existence), so we may or may not want to treat induction separately.

How do we choose restrictions? For comprehension, we might limit φ to certain levels of complexity, such as being recursive (Δ^0_1) or arithmetical (with only number quantifiers, no set quantifiers; i.e., Σ^0_n or Π^0_n for some n). The former gives us the comprehension axiom for RCA_0, and the latter for ACA_0, both described below. We may similarly bound the complexity of the formulas for which the induction statement (9.1) holds.

9.5.2. Recursive Comprehension. The base system, in which reverse mathematics proofs are carried out, is called RCA_0, which stands for *recursive comprehension axiom.* RCA_0 contains all of the basic arithmetic axioms from Z_2, as well as restricted comprehension and induction. The comprehension axiom of Z_2 is limited to Δ^0_1 formulas φ; intuitively this means that all computable sets exist, though it is slightly more precise than that.[9] RCA_0 also has Σ^0_1 induction, which is as (9.1) above but where φ must be Σ^0_1. This is more than would be obtained by restricting set-based induction (axiom 2 of Z_2) via recursive comprehension, which is done because allowing only Δ^0_1-induction gives a system that is too weak to work with easily.[10]

Roughly speaking, anything we can do computably can be done in RCA_0. This includes coding finite sequences (such as pairs that might represent rational numbers) and finite sets as numbers, and coding functions and real numbers as sets. Within RCA_0 we have access to all total computable functions (i.e., all primitive recursive functions plus whatever we get from unbounded search when it always halts). For those who have had algebra, RCA_0 can prove that $(\mathbb{N}, +, \cdot, 0, 1, <)$ is a commutative ordered semiring with cancellation, $(\mathbb{Z}, +, \cdot, 0, 1, <)$ is an ordered integral domain that is Euclidean, and $(\mathbb{Q}, +, \cdot, 0, 1, <)$ is an ordered field. For those who haven't had algebra, that essentially says RCA_0 can prove arithmetic behaves as we expect in each of those three sets.

RCA_0 can prove the intermediate value theorem (once continuous functions have been appropriately coded): if f is continuous and $f(0) < 0 < f(1)$, then for some $0 < x < 1$, $f(x) = 0$. It can prove paracompactness: given an open cover $\{U_n : n \in \mathbb{N}\}$ of a set X, there is an open cover $\{V_n : n \in \mathbb{N}\}$ of X such that for every $x \in X$ there is an open set W containing x such that $W \cap V_n = \emptyset$ for all but finitely many n. As a final example, RCA_0 can prove that every countable field has an algebraic closure – but not that it has a *unique* algebraic closure. More on that momentarily.

The canonical model of RCA_0 is called REC. Its universe is $\mathbb{N}$, and its collection of sets is exactly the computable sets, meaning a formula that begins $\forall X$ is read "for all computable sets X." In fact, REC is the smallest model of RCA_0 possible with universe $\mathbb{N}$ (we call it the *minimal ω-model*).

An aside on models and parameters here: All systems of reverse math are *relative*, in the sense that the induction and comprehension

[9]Formally it is $\forall n(\varphi(n) \leftrightarrow \psi(n)) \rightarrow \exists X \forall n(n \in X \leftrightarrow \varphi(n))$ for Σ^0_1 φ and Π^0_1 ψ, which could be read as "*demonstrably* computable (i.e., Δ^0_1) sets exist."

[10]Though that system has been studied, under the name RCA^*_0.

formulas are allowed to use parameters from the model. It is tempting to think of every model of RCA_0 as simply REC, but that would be a harmful assumption. We can have noncomputable sets in a model of RCA_0; if we have such a set A the axioms provide comprehension for sets that are Δ^0_1 *in* A (that is, $\Delta^{0,A}_1$) and induction for $\Sigma^{0,A}_1$ formulas. Moreover, the universe of the model need not be $\mathbb{N}$. Every subsystem of second-order arithmetic has infinitely many nonstandard models, and as we have seen, these can act in ways counter to the intuition we have developed from $\mathbb{N}$.

Theorems pop us out of RCA_0 when they have cases that require a noncomputable set or function. For example, in reality, every field's algebraic closure is unique (up to isomorphism), so the fact that RCA_0 can't prove the uniqueness tells us that the isomorphism between two computable algebraic closures might necessarily be noncomputable.

Another example of something RCA_0 cannot prove – which comes directly from comprehension and REC – is *weak König's lemma.* This says that if you have a subtree of $2^{<\mathbb{N}}$ and there are infinitely many nodes in the tree, then there must be an infinite path through the tree (full König's lemma allows arbitrary finite branching rather than restricting to the children 0 and 1).[11] The proof is quite easy if you allow yourself noncomputable techniques: start at the root. Since there are infinitely many nodes above the root, there must be infinitely many nodes above at least one of the children of the root. Choose the left child if it has infinitely many nodes above it, and otherwise choose the right. Repeat, walking upward until you hit a branching node if you are at a node with only one child. Since each time you have infinitely many nodes above you, you never have to stop, so you trace out an infinite path.

Such a tree is *computable* if the set of its nodes is computable. There exist computable infinite trees with no computable infinite paths, so these trees are in REC but none of their infinite paths are. Hence RCA_0 cannot prove they have infinite paths at all. I'll note in passing that although the failure of weak König's lemma in the specific model REC is sufficient to show it does not follow from RCA_0, the result that not every computable tree has a computable

[11] Jeff Hirst states this lemma as "big skinny trees are tall."

path relativizes to say that for any set A, there is an A-computable tree with no A-computable path.

9.5.3. Weak König's Lemma. WKL_0, or *weak König's lemma*, does not fit the same mold as RCA_0, ACA_0, or Π^1_1-CA_0 (described below). In this system comprehension has been restricted, but not via the syntactic complexity of φ in the basic comprehension axiom scheme. It is more easily stated in the form of the previous section, that every infinite subtree of $2^{<\mathbb{N}}$ has an infinite path. WKL_0 is RCA_0 together with that comprehension axiom.[12]

I want to note here that it is important that the tree be a subset of $2^{<\mathbb{N}}$ rather than any old tree where every node has at most two children. If we let the *labels* of the nodes be unbounded, we get something equivalent to full König's lemma, which says that any infinite, finitely-branching tree (subtree of $\mathbb{N}^{\mathbb{N}}$) has a path and is equivalent to ACA_0. In fact, some reverse mathematicians refer to 0-1 trees rather than binary-branching trees to highlight the distinction. The difference is one of computability versus enumerability of the children of a node in a computable tree. If the labels have a bound, we have only to ask a finite number of membership questions to determine how many children a node actually has. If not, even knowing there are at most 2 children, if we have not yet found 2, we must continue to ask about children with higher and higher labels, a process that only halts if the node has the full complement of children.

Over RCA_0, WKL_0 is equivalent to the existence of a unique algebraic closure for any countable field. It is also equivalent to the statement that every continuous function on $[0, 1]$ attains a maximum, every countable commutative ring has a prime ideal, and the Heine-Borel theorem. Heine-Borel says that every open cover of $[0, 1]$ has a finite subcover.

There is no canonical model of WKL_0. In fact, any model of WKL_0 with universe $\mathbb{N}$ contains a proper submodel that is also a model of WKL_0. The intersection of all such models is REC, which as we saw is not a model of WKL_0. There is a deep connection between

[12]This makes WKL_0 another system with more induction than is simply given by comprehension plus set-based induction; the three stronger systems do not have this trait. Without the extra induction we call the system WKL_0^*.

models of WKL_0 and Peano Arithmetic (PA; see §9.3.1). Formally, a degree $\boldsymbol{d}$ is the degree of a nonstandard model of PA if and only if there is a model of WKL_0 with universe $\mathbb{N}$ consisting entirely of sets computable from $\boldsymbol{d}$. Informally, “PA-degree” is to WKL_0 what “computable” is to RCA_0 and “arithmetical” to ACA_0.

The study of computable trees gives us a result called the *low basis theorem* [43], which says any computable tree has a path of low degree (where A is *low* if $A' \equiv \emptyset'$; noncomputable low sets exist). This and a little extra computability theory shows that WKL_0 has a model with only low sets.

9.5.4. Arithmetical Comprehension. ACA_0 stands for *arithmetical comprehension axiom.* As mentioned, we obtain it from Z_2 by restricting the formulas φ in the comprehension scheme to those that may be written using number quantifiers only, no set quantifiers. There is no middle ground between RCA_0 and ACA_0 in terms of capping the complexity of φ via the arithmetic hierarchy: if we allow φ to be even Σ^0_1, we get the full power of ACA_0. The proof is by relativization: given the existence of a set X, we get the existence of every set that is $\Sigma^{0,X}_1$, which includes X'. From there, we get the existence of all sets that are $\Sigma^{0,X'}_1$, which by Post's Theorem 7.2.9 are the sets that are $\Sigma^{0,X}_2$. Continuing this process, we bootstrap our way all the way up the arithmetical hierarchy.

RCA_0 can prove that the statement “for all X, the Turing jump X' exists” (suitably coded) is equivalent to ACA_0. Other equivalent statements include: every sequence of points in a compact metric space has a convergent subsequence; every countable vector space over a countable scalar field has a basis; and every countable commutative ring has a maximal ideal.

ACA_0, like RCA_0, has a minimal model with universe $\mathbb{N}$. It is ARITH, the collection of all arithmetic sets. These sets are exactly those definable by formulas with no set quantifiers but arbitrarily many number quantifiers, or equivalently, sets that are Turing reducible to $\emptyset^{(n)}$ for some n.

9.5.5. Arithmetic Transfinite Recursion and Π^1_1 Comprehension. ATR_0, like WKL_0, is obtained via a restriction to comprehension that feels less natural than the other systems. Arithmetic transfinite recursion roughly says that starting at any set that exists, we may iterate the Turing jump on it as many times as we like and those sets will all exist. This is a very imprecise version, clearly, since it is not at all apparent this gives more than ACA_0; the real thing is quite technical ("as many times as we like" is a lot).

Π^1_1-CA_0 stands for Π^1_1 *comprehension axiom*. It is the last of the "Big Five," and in the same family as RCA_0 and ACA_0, where the comprehension scheme has been restricted by capping the complexity of the formula φ. In this case, φ is allowed to have one universal set quantifier and an unlimited (finite) list of number quantifiers.

One theorem equivalent to ATR_0 is the *perfect set theorem*. A set X is (topologically) perfect if it has no isolated points; every point $x \in X$ is the limit of some sequence of points $\{y_i : y_i \in X, i \in \mathbb{N}, y_i \neq x\}$. A tree is perfect if every node of the tree has more than one infinite path extending it, which is exactly the previous statement but specific to the tree topology. The perfect set theorem states that every uncountable closed set has a nonempty perfect subset, and the version for trees says every tree with uncountably many paths has a nonempty perfect subtree. Both are equivalent to ATR_0 over RCA_0; note that both are comprehension theorems. Π^1_1-CA_0 is equivalent to some related but stronger theorems: (a) every tree is the union of a perfect subtree and a countable set of paths, and (b) for any uncountable tree the *perfect kernel* of the tree exists (that is, the union of all its perfect subtrees).

Another comprehension theorem equivalent to ATR_0 is that, for any sequence of trees $\{T_i : i \in \mathbb{N}\}$ such that each T_i has at most one path, the set $\{i : T_i \text{ has a path}\}$ exists. If you remove the restriction on number of paths, you have a statement equivalent to Π^1_1-CA_0.

From ATR_0 up, there are no minimal models. ATR_0 is similar to WKL_0 in that it has no minimal model but the intersection of all of its models with universe $\mathbb{N}$ is a natural class of sets. In this case it is HYP, the *hyperarithmetic* sets, which we will not define.

9.5.6. A Spiderweb. It is remarkable that so many theorems of ordinary mathematics fall into five major equivalence classes under relative provability. However, it would be misleading to close this section without mentioning that not every theorem has such a clean relationship to the Big Five. A lot of research has been done that establishes a cobweb of implications for results that lie between RCA_0 and ACA_0, and for many researchers this is the most interesting part of reverse mathematics. If I were willing to drown you in acronyms I could draw a very large picture with one-way arrows, two-way arrows, unknown implications, and non-implications, but we'll look at just a few examples.

One principle strictly between RCA_0 and WKL_0 is DNR, which stands for *diagonally non-recursive.* DNR says there exists a function f such that $(\forall e)(f(e) \neq \varphi_e(e))$. It is clear that DNR fails in REC, since such a function is designed exactly to be unequal to every computable function. It is true in WKL_0 because we may construct a computable tree such that every path through the tree represents a diagonally non-recursive function: let the levels of the tree represent values of e, leftward branching at e represent $f(e) = 0$, and rightward branching $f(e) = 1$. Dovetail the computations of $\varphi_e(e)$, and when you see one halt, cease extending the descendants of the nodes representing $f(e) = \varphi_e(e)$ for that e. Unless instructed by such a computation, extend all nodes by both 0 and 1 at each level.

WWKL_0, or *weak weak König's lemma*, is a system strictly between DNR and WKL_0. The lemma says that if $T \subseteq 2^{<\mathbb{N}}$ is a tree with no infinite path, then

$$\lim_{n\to\infty} \frac{|\{\sigma \in T : |\sigma| = n\}|}{2^n} = 0.$$

This is clearly implied by weak König's lemma, which says in contrapositive that if T has no infinite path it must be finite (so this fraction does not just approach 0, it is identically 0 from some n on). A decent amount of measure theory can be carried out in WWKL_0, but I wanted to mention it in particular because it has connections to randomness as laid out in §9.2. A model $\mathcal{M}$ of RCA_0 with standard first-order part is also a model of WWKL_0 if and only if, for every X in $\mathcal{M}$, there is some Y in $\mathcal{M}$ that is 1-random relative to X [3].

Finally, we will see a proof of reverse mathematical equivalence. Some very weak reverse mathematical systems are referred to as *fragments of (first-order) arithmetic*. The book by Hájek and Pudlák [35] is a good reference for this material. The two we will consider are *bounding* (or *collection*) systems. Bounding for a specific logical formula φ is the following.

$$B_\varphi : (\forall u)\left[((\forall x \leq u)(\exists y)\varphi(x,y)) \rightarrow ((\exists v)((\forall x \leq u)(\exists y \leq v)\varphi(x,y)))\right]$$

We can make axiom systems by adding bounding for certain kinds of formulas to RCA_0. In particular, we may consider $B\Sigma^0_n$, which includes bounding for Σ^0_n, and $B\Pi^0_n$. Because each of the φ pieces of the statement of B_φ has an existential quantifier out front, with only bounded quantifiers in between it and φ, syntax shows $B\Sigma^0_{n+1}$ is equivalent to $B\Pi^0_n$.

$B\Sigma^0_2$, or $B\Pi^0_1$, seems like an incredibly weak system; how could it be any more than just RCA_0? The intuitive explanation is that while in the standard model everything is finite and works easily, in a nonstandard model we have to bound y values for infinite u, and RCA_0 can't do that.

That explanation also sheds some light on the reason RCA_0 can't prove the pigeonhole principle: If f is a function with finite range $\{0,\ldots,c\}$, then there must be an infinite set on which f is constant. That is, if we have finitely many boxes 0 to c and are putting infinitely many numbers into the boxes, some box must contain infinitely many numbers. Call this axiom, together with RCA_0, the system PHP.

Theorem 9.5.1 (Hirst [40], see [13] Thm 2.10)**.** *Over* RCA_0*, PHP is equivalent to* $B\Sigma^0_2$ *(equivalently, to* $B\Pi^0_1$*).*

Proof. Let $\mathcal{M}$ be a model of PHP with first-order part M. Let θ be a Σ^0_0 formula; it is allowed to have parameters from the second-order part of $\mathcal{M}$. Fix u and assume that $(\forall x \leq u)(\exists y)(\forall z)\theta(x,y,z)$; this is the antecedent of the bounding principle for the Π^0_1 formula $(\forall z)\theta(x,y,z)$. We define a function F that by the PHP will either be constant on an infinite set or have an unbounded range; the latter case will lead to a contradiction. Let $F(t)$ be the least $n < t$ such that $(\forall x \leq u)(\exists y < n)(\forall z < t)\theta(x,y,z)$ if such an n exists, and $F(t) = t$

otherwise. Since θ is Σ_0^0 and all quantifiers are bounded, this function is in $\mathcal{M}$.

If there is an infinite set H such that $F(t) = \ell$ for all $t \in H$, then ℓ is a suitable bound for $(\forall z)\theta(x, y, z)$: $(\forall x \leq u)(\exists y < \ell)(\forall z)\theta(x, y, z)$. If there is no such infinite set, PHP says the range of F must be unbounded, so covering all the z values requires larger y values infinitely often. In this case, Δ_0^0 comprehension (from RCA_0) demonstrates existence of a sequence $\{t_i : i \in M\}$ such that $t_i < t_{i+1}$ and $F(t_i) < F(t_{i+1})$. Let $G(i)$ be the least $x < u$ such that $(\forall y < F(t_i) - 1)(\exists z < t_i)\neg\theta(x, y, z)$. G certainly has a finite range, so by the PHP there is an infinite set S in $\mathcal{M}$ on which G is constant; suppose G's output on S is x_0. Since S is infinite, for any y_0 there is some $i \in S$ such that $F(t_i) - 1 > y_0$. This gives $(\exists z < t_i)\neg\theta(x_0, y_0, z)$ and hence $(\forall y)(\exists z)\neg\theta(x_0, y, z)$, contrary to assumption.

For the converse, now let $\mathcal{M}$ be a model of $B\Pi_1^0$. Let $c \in M$ and F be a function from M to $\{0, \ldots, c-1\}$. Suppose no $x < c$ is the image under F of infinitely many elements of M, or, in symbols, that $(\forall x < c)(\exists y)(\forall z)(z > y \rightarrow F(z) \neq x)$. By $B\Pi_1^0$, $(\exists v)(\forall x < c)(\exists y < v)(\forall z)(z > y \rightarrow F(z) \neq x)$. In fact, $(\exists v)(\forall x < c)(\forall z)(z > v \rightarrow F(z) \neq x)$. If v_0 is such a v, we have $(\forall x < c)F(v_0 + 1) \neq x$, contradicting the choice of F. Therefore it must be the case that $(\exists x < c)(\forall y)(\exists z)(z > y \;\&\; F(z) = x)$. Letting x_0 be such an x, $\{t : F(t) = x_0\}$ is the desired infinite set. $\square$

For more easy to state combinatorial principles that fall below or to the side of WKL_0, see the papers by Cholak, Jockusch, and Slaman [13] and Hirschfeldt and Shore [39].

Appendix A

Mathematical Asides

In this appendix, I've included a few proofs and other tidbits that aren't really part of computability theory but have been referenced in the text.

A.1. The Greek Alphabet

As you progress through mathematics you'll learn much of the Greek alphabet by osmosis, but here is a list for reference.

alpha	A	α	nu	N	ν
beta	B	β	xi	Ξ	ξ
gamma	Γ	γ	omicron	O	o
delta	Δ	δ	pi	Π	π
epsilon	E	ϵ or ε	rho	P	ρ
zeta	Z	ζ	sigma	Σ	σ
eta	H	η	tau	T	τ
theta	Θ	θ	upsilon	Υ	υ
iota	I	ι	phi	Φ	φ or ϕ
kappa	K	κ	chi	X	χ
lambda	Λ	λ	psi	Ψ	ψ
mu	M	μ	omega	Ω	ω

A.2. Summations

When defining the pairing function we needed to sum from 1 to $x+y$. There is a clever way to find a closed form for the sum $1+2+\ldots+n$. Write out the terms twice, in two directions:

$$\begin{array}{ccccccccc} 1 & + & 2 & + & \ldots & + & (n-1) & + & n \\ n & + & (n-1) & + & \ldots & + & 2 & + & 1 \end{array}$$

Adding downward, we see n copies of $n+1$ added together. As this is twice the desired sum, we find

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

Related, though not relevant to this material, is the way one proves the sum of the geometric series with terms ar^i, $i \geq 0$, is $\frac{a}{1-r}$ whenever $|r| < 1$. We take the partial sum, stopping at some $i = n$, and we subtract from it its product with r.

$$\begin{array}{ccccccccccc} a & + & ar & + & ar^2 & + & \ldots & + & ar^n & & \\ & - & (ar & + & ar^2 & + & \ldots & + & ar^n & + & ar^{n+1}) \end{array}$$

Letting s_n be the n^{th} partial sum of the series, we get $s_n - rs_n = a - ar^{n+1}$, or $s_n = (a - ar^{n+1})/(1-r)$. The sum of any series is the limit of its partial sums, so we see that

$$\sum_{i=0}^{\infty} ar^i = \lim_{n\to\infty} \frac{a - ar^{n+1}}{1-r} = \frac{a}{1-r} \lim_{n\to\infty} (1 - r^{n+1}),$$

and the rightmost limit is 1 whenever $|r| < 1$.

A.3. Cantor's Cardinality Proofs

Cantor had two beautifully simple diagonal proofs to show that the rational numbers are no more numerous than the natural numbers, but the real numbers are strictly more numerous. The ideas of these proofs are used for some of the most fundamental results in computability theory, such as the proof that the halting problem is noncomputable. This section recaps some material from earlier in the book in order to be self-contained.

First we show that $\mathbb{Q}$ has the same cardinality as $\mathbb{N}$. Take the grid of all pairs of natural numbers, i.e., all integer-coordinate points in the first quadrant of the Cartesian plane. The pair (n, m) represents the rational number n/m; all positive rational numbers are representable as fractions of natural numbers. We may count these with the natural numbers if we go along diagonals of slope -1. Note that it does not work to try to go row by row or column by column, as you will never finish the first one; you must dovetail the rows and columns, doing a bit from the first, then a bit from the second and some more from the first, then a bit from the third, more from the second, yet more from the first, and so on. To count exactly the rationals, start by labeling 0 with 0, then proceed along the diagonals, skipping (n, m) if n/m reduces to a rational we've already counted, and otherwise counting it twice to account for the negation of n/m.

Cantor's proof that $\mathbb{R}$ is strictly bigger than $\mathbb{N}$ is necessarily more subtle, as constructing a bijection with $\mathbb{N}$ (which is exactly what counting with the natural numbers accomplishes) is generally more straightforward than demonstrating that no such bijection exists.

In fact, we will show even just the interval from 0 to 1 is larger than $\mathbb{N}$. Suppose for a contradiction that we have a bijection between $[0, 1]$ and $\mathbb{N}$. List the elements of $[0, 1]$ out as infinite repeating decimals (using all-0 tails if needed) in the order given by the bijection:

$$\begin{array}{l}
.65479362895\ldots\\
.00032797584\ldots\\
.35271900000\ldots\\
.00000000063\ldots\\
.98989898989\ldots\\
\vdots
\end{array}$$

Now construct a new number $d \in [0, 1]$ decimal-by-decimal using the numbers on the list. If the n^{th} decimal place of the n^{th} number on the list is k, then the n^{th} decimal place of d will be $k + 1$, or 0 if $k = 9$. In our example above, d would begin .71310. While d is clearly a number between 0 and 1, it does not appear on the list, because it differs from every number x on the list in at least one decimal place: the one corresponding to x's position on the list.

For an interesting twist on this, look up Richard's Paradox, due to French mathematician Jules Richard. Its resolution is more akin to Theorem 3.5.1 than Cantor's proof above is.

Cantor also proved that for any set A, $|A| < |\mathcal{P}(A)|$. For finite sets, this is clear, because if $|A| = n$, then $|\mathcal{P}(A)| = 2^n$. For infinite sets, however, it is not immediate. Sets can "seem bigger" and not be, such as the rationals compared to the naturals. The proof is diagonal in nature and not too different from the proof that the reals are more numerous than the rationals.

To begin, let A be $\mathbb{N}$. It is clear that $\mathcal{P}(\mathbb{N})$ is at least as large as $\mathbb{N}$, since it contains $\{n\}$ for every $n \in \mathbb{N}$, giving a natural injection. If $\mathbb{N}$ and $\mathcal{P}(\mathbb{N})$ are bijective, then the elements of $\mathcal{P}(\mathbb{N})$ may be listed out in order, according to the natural number with which the bijection associates them. We construct a subset X of $\mathbb{N}$ that cannot be on the list, using the membership of n in the n^{th} set on the list to determine whether n is in X. To wit: put n in X if and only if n is *not* in the n^{th} set on the list.

Since the only elements we put in X are elements of $\mathbb{N}$, $X \in \mathcal{P}(\mathbb{N})$ and so X must be on the list; suppose it is entry m. However, then $m \in X \Leftrightarrow m \notin X$, a contradiction. As before, the conclusion is that such a list is impossible: $\mathcal{P}(\mathbb{N})$ is strictly larger than $\mathbb{N}$.

To extend this proof to sets other than $\mathbb{N}$, note that the fact that a bijection between $\mathbb{N}$ and $\mathcal{P}(\mathbb{N})$ gives a *list* is irrelevant. For any set A and an injection $f : A \to \mathcal{P}(A)$, we can define $X \subseteq A$ by letting $a \in X \Leftrightarrow a \notin f(a)$. If X is the image of some $\hat{a}$, then that $\hat{a} \in X \Leftrightarrow \hat{a} \notin X$, as before.

Bibliography

[1] Wilhelm Ackermann, *Zum Hilbertschen Aufbau der reellen Zahlen*, Math. Ann. **99** (1928), no. 1, 118–133 (German).

[2] Klaus Ambos-Spies, Carl G. Jockusch Jr., Richard A. Shore, and Robert I. Soare, *An algebraic decomposition of the recursively enumerable degrees and the coincidence of several degree classes with the promptly simple degrees*, Trans. Amer. Math. Soc. **281** (1984), no. 1, 109–128.

[3] Klaus Ambos-Spies, Bjørn Kjos-Hanssen, Steffen Lempp, and Theodore A. Slaman, *Comparing DNR and WWKL*, J. Symbolic Logic **69** (2004), no. 4, 1089–1104.

[4] Klaus Ambos-Spies and Antonín Kučera, *Randomness in computability theory*, Computability Theory and its Applications (Boulder, CO, 1999), 2000, pp. 1–14.

[5] M. M. Arslanov, *Some generalizations of a fixed-point theorem*, Izv. Vyssh. Uchebn. Zaved. Mat. **5** (1981), 9–16 (Russian).

[6] M. M. Arslanov, R. F. Nadyrov, and V. D. Solov′ev, *A criterion for the completeness of recursively enumerable sets, and some generalizations of a fixed point theorem*, Izv. Vysš. Učebn. Zaved. Matematika **4 (179)** (1977), 3–7 (Russian).

[7] C. J. Ash and J. Knight, *Computable Structures and the Hyperarithmetical Hierarchy*, Studies in Logic and the Foundations of Mathematics, vol. 144, North-Holland Publishing Co., Amsterdam, 2000.

[8] Jon Barwise (ed.), *Handbook of mathematical logic*, with the cooperation of H. J. Keisler, K. Kunen, Y. N. Moschovakis and A. S. Troelstra, Studies in Logic and the Foundations of Mathematics, Vol. 90, North-Holland Publishing Co., Amsterdam, 1977.

[9] Charles H. Bennett and Martin Gardner, *The random number Omega bids fair to hold the mysteries of the universe*, Scientific American **241** (1979), no. 5, 20–34.

[10] George S. Boolos, John P. Burgess, and Richard C. Jeffrey, *Computability and Logic*, 4th ed., Cambridge University Press, Cambridge, 2002.

[11] Gregory J. Chaitin, *Information-theoretic characterizations of recursive infinite strings*, Theoret. Comput. Sci. **2** (1976), no. 1, 45–48.

[12] G. J. Chaitin, *Incompleteness theorems for random reals*, Adv. in Appl. Math. **8** (1987), no. 2, 119–146.

[13] Peter A. Cholak, Carl G. Jockusch, and Theodore A. Slaman, *On the strength of Ramsey's theorem for pairs*, J. Symbolic Logic **66** (2001), no. 1, 1–55.

[14] Alonzo Church, *An Unsolvable Problem of Elementary Number Theory*, Amer. J. Math. **58** (1936), no. 2, 345–363. Reprinted in [18].

[15] ———, *A note on the Entscheidungsproblem*, J. Symbolic Logic **1** (1936), no. 1, 40–41. Correction in J. Symbolic Logic **1** No. 3 (1936), 101–102. Reprinted in [18].

[16] Robert L. Constable, Harry B. Hunt, and Sartaj Sahni, *On the computational complexity of scheme equivalence*, Technical Report 74–201, Department of Computer Science, Cornell University, Ithaca, NY, 1974.

[17] Nigel Cutland, *Computability*: *An introduction to recursive function theory*, Cambridge University Press, Cambridge, 1980.

[18] Martin Davis (ed.), *The Undecidable*: *Basic papers on undecidable propositions, unsolvable problems and computable functions*, Raven Press, Hewlett, N.Y., 1965.

[19] Martin Davis, *Hilbert's tenth problem is unsolvable*, Amer. Math. Monthly **80** (1973), 233–269.

[20] ———, *Why Gödel didn't have Church's thesis*, Inform. and Control **54** (1982), no. 1-2, 3–24.

[21] ———, *American logic in the 1920s*, Bull. Symbolic Logic **1** (1995), no. 3, 273–278.

[22] Rodney G. Downey and Denis R. Hirschfeldt, *Algorithmic Randomness and Complexity*, Theory and Applications of Computability, Springer, New York, 2010.

[23] Bruno Durand and Alexander Zvonkin, *Kolmogorov complexity*, Kolmogorov's Heritage in Mathematics, 2007, pp. 281–299.

[24] A. Ehrenfeucht, J. Karhumäki, and G. Rozenberg, *The (generalized) Post correspondence problem with lists consisting of two words is decidable*, Theoret. Comput. Sci. **21** (1982), no. 2, 119–144.

[25] Herbert B. Enderton, *A Mathematical Introduction to Logic*, 2nd ed., Harcourt/Academic Press, Burlington, MA, 2001.

[26] Yu. L. Ershov, S. S. Goncharov, A. Nerode, J. B. Remmel, and V. W. Marek (eds.), *Handbook of Recursive Mathematics. Vol. 1*: *Recursive Model Theory*, Studies in Logic and the Foundations of Mathematics, vol. 138, North-Holland, Amsterdam, 1998.

[27] Richard M. Friedberg, *Two recursively enumerable sets of incomparable degrees of unsolvability (solution of Post's problem, 1944)*, Proc. Nat. Acad. Sci. U.S.A. **43** (1957), 236–238.

[28] ———, *Three theorems on recursive enumeration. I. Decomposition. II. Maximal set. III. Enumeration without duplication*, J. Symb. Logic **23** (1958), 309–316.

[29] Péter Gács, *Every sequence is reducible to a random one*, Inform. and Control **70** (1986), no. 2-3, 186–192.

[30] Kurt Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatsh. Math. Phys. **38** (1931), no. 1, 173–198 (German). An English translation appears in [18].

[31] S. S. Gončarov, *The number of nonautoequivalent constructivizations*, Algebra Logika **16** (1977), no. 3, 257–282, 377 (Russian); English transl., Algebra Logic **16** (1977), no. 3, 169–185.

[32] ———, *The problem of the number of nonautoequivalent constructivizations*, Algebra Logika **19** (1980), no. 6, 621–639, 745 (Russian); English transl., Algebra Logic **19** (1980), no. 6, 404–414.

[33] S. S. Gončarov and V. D. Dzgoev, *Autostability of models*, Algebra Logika **19** (1980), no. 1, 45–58, 132 (Russian); English transl., Algebra Logic **19** (1980), no. 1, 28–37.

[34] Eitan Gurari, *An Introduction to the Theory of Computation*, Computer Science Press, 1989. Available at `http://www.cse.ohio-state.edu/~gurari/theory-bk/theory-bk.html`.

[35] Petr Hájek and Pavel Pudlák, *Metamathematics of First-Order Arithmetic*, Perspectives in Mathematical Logic, Springer-Verlag, Berlin, 1993.

[36] Valentina S. Harizanov, *Computability-theoretic complexity of countable structures*, Bull. Symbolic Logic **8** (2002), no. 4, 457–477.

[37] David Hilbert, *Mathematical problems*, Bull. Amer. Math. Soc. **8** (1902), no. 10, 437–479.

[38] Denis R. Hirschfeldt, André Nies, and Frank Stephan, *Using random sets as oracles*, J. Lond. Math. Soc. (2) **75** (2007), no. 3, 610–622.

[39] Denis R. Hirschfeldt and Richard A. Shore, *Combinatorial principles weaker than Ramsey's theorem for pairs*, J. Symbolic Logic **72** (2007), no. 1, 171–206.

[40] Jeffry Lynn Hirst, *Combinatorics in Subsystems of Second Order Arithmetic*, Ph.D. Thesis, The Pennsylvania State University, 1987.

[41] John E. Hopcroft, *Turing Machines*, Scientific American **250** (1984), no. 5, 86–98.

[42] C. G. Jockusch Jr., M. Lerman, R. I. Soare, and R. M. Solovay, *Recursively enumerable sets modulo iterated jumps and extensions of Arslanov's completeness criterion*, J. Symbolic Logic **54** (1989), no. 4, 1288–1323.

[43] Carl G. Jockusch Jr. and Robert I. Soare, *Π^0_1 classes and degrees of theories*, Trans. Amer. Math. Soc. **173** (1972), 33–56.

[44] S. C. Kleene, *General recursive functions of natural numbers*, Math. Ann. **112** (1936), no. 1, 727–742. Reprinted in [18].

[45] Peter M. Kogge, *The Architecture of Symbolic Computers*, McGraw-Hill Series in Supercomputing and Parallel Processing, McGraw-Hill, 1990.

[46] A. N. Kolmogorov, *On tables of random numbers*, Sankhyā Ser. A **25** (1963), 369–376.

[47] ———, *Three approaches to the definition of the concept "quantity of information"*, Problemy Peredači Informacii **1** (1965), no. vyp. 1, 3–11 (Russian); English transl., Probl. Inf. Transm. **1** (1965), 1–7.

[48] Julia F. Knight, *Degrees coded in jumps of orderings*, J. Symbolic Logic **51** (1986), no. 4, 1034–1042.

[49] Leon G. Kraft, *A Device for Quantizing, Grouping, and Coding Amplitude Modulated Pulses*, Electrical Engineering Master's Thesis, MIT, Cambridge, MA, 1949.

[50] Shira Kritchman and Ran Raz, *The surprise examination paradox and the second incompleteness theorem*, Notices Amer. Math. Soc. **57** (2010), no. 11, 1454–1458.

[51] Antonín Kučera, *Measure, Π^0_1-classes and complete extensions of* PA, Recursion Theory Week (Oberwolfach, 1984), 1985, pp. 245–259.

[52] Antonín Kučera and Sebastiaan A. Terwijn, *Lowness for the class of random sets*, J. Symbolic Logic **64** (1999), no. 4, 1396–1402.

[53] A. H. Lachlan, *Lower bounds for pairs of recursively enumerable degrees*, Proc. London Math. Soc. (3) **16** (1966), 537–569.

[54] Steffen Lempp, Manuel Lerman, and Reed Solomon, *Embedding finite lattices into the computably enumerable degrees—a status survey*, Logic Colloquium '02, 2006, pp. 206–229.

[55] L. A. Levin, *Laws on the conservation (zero increase) of information, and questions on the foundations of probability theory*, Problemy Peredači Informacii **10** (1974), no. 3, 30–35 (Russian); English transl., Probl. Inf. Transm. **10** (1974), 206–210.

[56] Ming Li and Paul Vitányi, *An Introduction to Kolmogorov Complexity and its Applications*, 2nd ed., Graduate Texts in Computer Science, Springer-Verlag, New York, 1997.

[57] Peter Linz, *An Introduction to Formal Languages and Automata*, 2nd ed., Jones and Bartlett Publishers, 1997.

[58] Wolfgang Maass, *Recursively enumerable generic sets*, J. Symbolic Logic **47** (1982), no. 4, 809–823 (1983).

[59] A. Markoff, *On the impossibility of certain algorithms in the theory of associative systems*, C. R. (Doklady) Acad. Sci. URSS (N.S.) **55** (1947), 583–586.

[60] Donald A. Martin, *Classes of recursively enumerable sets and degrees of unsolvability*, Z. Math. Logik Grundlagen Math. **12** (1966), 295–310.

[61] Yu. V. Matijasevič, *On recursive unsolvability of Hilbert's tenth problem*, Logic, Methodology and Philosophy of Science, IV (Proc. Fourth Internat. Congr., Bucharest, 1971), 1973, pp. 89–110.

[62] Yuri Matiyasevich and Géraud Sénizergues, *Decision problems for semi-Thue systems with a few rules*, 11th Annual IEEE Symposium on Logic in Computer Science (New Brunswick, NJ, 1996), 1996, pp. 523–531.

[63] Russell Miller, *The Δ^0_2-spectrum of a linear order*, J. Symbolic Logic **66** (2001), no. 2, 470–486.

[64] A. A. Mučnik, *On the unsolvability of the problem of reducibility in the theory of algorithms*, Dokl. Akad. Nauk SSSR (N.S.) **108** (1956), 194–197 (Russian).

[65] Roman Murawski, *Recursive Functions and Metamathematics: Problems of Completeness and Decidability, Gödel's Theorems*, Synthese Library, vol. 286, Kluwer Academic Publishers Group, Dordrecht, 1999.

[66] André Nies, *Lowness properties and randomness*, Adv. Math. **197** (2005), no. 1, 274–305.

[67] ———, *Computability and Randomness*, Oxford Logic Guides, vol. 51, Oxford University Press, Oxford, 2009.

[68] Piergiorgio Odifreddi, *Classical Recursion Theory: The theory of functions and sets of natural numbers*, with a foreword by G. E. Sacks, Studies in Logic and the Foundations of Mathematics, vol. 125, North-Holland Publishing Co., Amsterdam, 1989.

[69] Rózsa Péter, *Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktion*, Math. Ann. **110** (1935), no. 1, 612–632 (German).

[70] ———, *Konstruktion nichtrekursiver Funktionen*, Math. Ann. **111** (1935), no. 1, 42–60 (German).

[71] Emil L. Post, *Finite combinatory processes. Formulation I*, J. Symbolic Logic **1** (1936), no. 3, 103–105. Reprinted in [18].

[72] ———, *Recursively enumerable sets of positive integers and their decision problems*, Bull. Amer. Math. Soc. **50** (1944), 284–316. Reprinted in [18].

[73] ———, *A variant of a recursively unsolvable problem*, Bull. Amer. Math. Soc. **52** (1946), 264–268.

[74] ———, *Recursive unsolvability of a problem of Thue*, J. Symbolic Logic **12** (1947), 1–11.

[75] Hartley Rogers Jr., *Theory of Recursive Functions and Effective Computability*, 2nd ed., MIT Press, Cambridge, MA, 1987.

[76] C.-P. Schnorr, *A unified approach to the definition of random sequences*, Math. Systems Theory **5** (1971), 246–258.

[77] J. C. Shepherdson and H. E. Sturgis, *Computability of recursive functions*, J. Assoc. Comput. Mach. **10** (1963), 217–255.

[78] J. R. Shoenfield, *On degrees of unsolvability*, Ann. of Math. (2) **69** (1959), 644–653.

[79] Richard A. Shore, *Nowhere simple sets and the lattice of recursively enumerable sets*, J. Symbolic Logic **43** (1978), no. 2, 322–330.

[80] Stephen G. Simpson, *Subsystems of Second Order Arithmetic*, Perspectives in Mathematical Logic, Springer-Verlag, Berlin, 1999.

[81] Douglas Smith, Maurice Eggen, and Richard St. Andre, *A Transition to Advanced Mathematics*, 3rd ed., Brooks/Cole Publishing Company, Wadsworth, Inc., 1990.

[82] Robert I. Soare, *Recursively Enumerable Sets and Degrees*: *A Study of Computable Functions and Computably Generated Sets*, Perspectives in Mathematical Logic, Springer-Verlag, Berlin, 1987.

[83] Robert Solovay, *Draft of paper (or series of papers) on Chaitin's work*, May, 2004. Unpublished notes.

[84] S. Tennenbaum, *Non-Archimedean models for arithmetic*, Notices Amer. Math. Soc. **6** (1959), 270.

[85] A. M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proc. Lond. Math. Soc. (2) **42** (1937), 230–265. A correction appears in Proc. Lond. Math. Soc. (2), **43** (1937), 544–546. Both reprinted in [18].

[86] Jean van Heijenoort, *From Frege to Gödel*: *A Source Book in Mathematical Logic, 1879–1931*, Harvard University Press, Cambridge, Mass., 1967.

[87] Sérgio B. Volchan, *What is a random sequence?*, Amer. Math. Monthly **109** (2002), no. 1, 46–63.

[88] Judson Chambers Webb, *Mechanism, Mentalism, and Metamathematics*: *An essay on finitism*, Synthese Library, vol. 137, D. Reidel Publishing Co., Dordrecht, 1980.

[89] Alfred North Whitehead and Bertrand Russell, *Principia Mathematica*, Vol. 1–3, Cambridge University Press, 1910–1913. Second edition, strongly edited, 1927.

[90] D. Zambella, *On sequences with simple initial segments*, University of Amsterdam ILLC technical report ML-1990-05, 1990.

[91] A. K. Zvonkin and L. A. Levin, *The complexity of finite objects and the basing of the concepts of information and randomness on the theory of algorithms*, Uspehi Mat. Nauk **25** (1970), no. 6(156), 85–127 (Russian).

Index

Selected Titles in This Series

What can we compute—even with unlimited resources? Is everything within reach? Or are computations necessarily drastically limited, not just in practice, but theoretically? These questions are at the heart of computability theory.

The goal of this book is to give the reader a firm grounding in the fundamentals of computability theory and an overview of currently active areas of research, such as reverse mathematics and algorithmic randomness. Turing machines and partial recursive functions are explored in detail, and vital tools and concepts including coding, uniformity, and diagonalization are described explicitly. From there the material continues with universal machines, the halting problem, parametrization and the recursion theorem, and thence to computability for sets, enumerability, and Turing reduction and degrees. A few more advanced topics round out the book before the chapter on areas of research. The text is designed to be self-contained, with an entire chapter of preliminary material including relations, recursion, induction, and logical and set notation and operators. That background, along with ample explanation, examples, exercises, and suggestions for further reading, make this book ideal for independent study or courses with few prerequisites.

ISBN 978-0-8218-7392-2

9 780821 873922

STML/62