Math 29: Computability Theory

Rebecca Weber

Spring 2011

Contents

1 Introduction **3**
1.1 Mindset . . . 3
1.2 Some History . . . 4
1.3 Some References . . . 6
1.4 A Little Request . . . 6

2 Background **7**
2.1 First-Order Logic . . . 7
2.2 Sets . . . 12
2.3 Relations . . . 17
2.4 Recursion and Induction . . . 23
2.5 Some Notes on Proofs and Abstraction . . . 29

3 Defining Computability **33**
3.1 Functions, Sets, and Sequences . . . 33
3.2 Turing Machines . . . 35
3.3 Partial Recursive Functions . . . 39
3.4 Coding and Countability . . . 42
3.5 The Church-Turing Thesis . . . 46
3.6 Other Definitions of Computability . . . 47

4 Working with Computable Functions **57**
4.1 A Universal Turing Machine . . . 57
4.2 The Halting Problem . . . 58
4.3 Parametrization . . . 59
4.4 The Recursion Theorem . . . 60
4.5 Unsolvability . . . 63

5 Computable and Computably Enumerable Sets **71**
5.1 Dovetailing . . . 71
5.2 Computing and Enumerating . . . 72
5.3 Noncomputable Sets Part I . . . 76

5.4 Noncomputable Sets Part II: Simple Sets 77

6 Turing Reduction and Post's Problem 79
6.1 Reducibility of Sets 79
6.2 Finite Injury Priority Arguments 82

7 Turing Degrees 91
7.1 Turing Degrees 91
7.2 Relativization and the Turing Jump 92

8 More Advanced Results 97
8.1 The Limit Lemma 97
8.2 The Arslanov Completeness Criterion 99
8.3 $\mathcal{E}$ Modulo Finite Difference 101

9 Areas of Research 105
9.1 Lattice-Theoretic Properties 105
9.2 Randomness 111
9.3 Some Model Theory 122
9.4 Computable Model Theory 124
9.5 Reverse Mathematics 127

A Mathematical Asides 137
A.1 The Greek Alphabet 137
A.2 Summations 137
A.3 Cantor's Cardinality Proofs 138

Bibliography 141

Chapter 1

Introduction

This chapter is one I expect you will initially skim. The first section I hope you will come back to halfway through the course, to get a high-level view of the subject; it may not make total sense before the course begins.

1.1 Mindset

What does it mean for a function or set to be computable?

Computability is a dynamic field. I mean that in two ways. One, of course, is that research into computability is ongoing and varied. However, I also mean that the mindset when working in computability is dynamic rather than static. The objects in computability are rarely accessible "all at once" or in their exact form. Rather, they are approximated or enumerated. Sets will be presented element by element; a function's output must be waited for, and indeed may never come.

The aspect of computability theory that tends to bother people the most is that it is highly *nonconstructive*. By that I mean many proofs are existence proofs rather than constructive proofs: when we say a set is computable, we mean an algorithm *exists* to compute it, not that we necessarily have such an algorithm explicitly. One common application of this is being unconcerned when a program requires some magic number to operate correctly: for example, a value n such that on inputs at least n two functions are equal, though they might differ on inputs below n. We think of a fleet of programs, each "guessing" a different value; if we can show such a value exists we know one of those programs (indeed, infinitely many) will operate correctly, and that is all we care about. This is called *nonuniformity* and can inhibit some further uses of the algorithm, so we always pay attention to whether or not processes are uniform.

The other initially troublesome aspect, which perhaps only bothers computability theorists because no one else sees it, is that computability uses self-reference in very strong and perhaps illegal-looking ways. For now we will leave this to §4.4.

The primary tool in computability is called a *priority argument* (see §6.2). This is a ramped-up version of a *diagonal argument*, such as Cantor's proof the reals are uncountable (Appendix A.3). Essentially, we break up what we want to accomplish in a construction into an infinite collection of requirements to meet. The requirements have a priority ordering, where lower-priority requirements are restricted from doing anything that harms higher-priority requirements, but not vice-versa. As long as each requirement will only cause harm finitely many times, and each can recover from a finite number of injuries, the construction will succeed. This allows us to work with information that is being approximated and will throughout the construction be incomplete and possibly incorrect: the requirements must act based on the approximation and thus might act wrongly or have their work later undone by a higher-priority requirement. Proofs that priority constructions accomplish their goals are done by *induction*, which also works step by step.

As a simple example of the sorts of conflicts that arise, consider building a set A by putting numbers into it stage by stage during a construction. We may have some requirements that want to put elements into A, say, to cause it to have nonempty intersection with another set. Then we may have requirements that are trying to maintain computations that are based on A – that say things like "if 5 is in A, output 1, and if not, output 0". If one requirement wants to put 10 into A to cause an intersection and another wants to keep 10 out to preserve a computation, priority allows us to break the tie. Regardless of which one wins the other has to recover: the one that wanted 10 out would need to get its computation back or be able to use a different one, and the one that wanted 10 in would need to be able to use a different element to create the intersection.

The point of a priority argument is that we can make assertions about the computability of the object we are building. Computability theorists are concerned not only with *whether* an object exists but with *how complicated* it must be.

1.2 Some History

We begin with the mathematical philosophy of *formalism*, which holds that all mathematics is just symbol-pushing without deeper meaning (to be contrasted with *Platonism*, which holds that mathematics represents something real, and in particular that even statements we can't prove or disprove with our mathematical axioms *are* true or false in reality). In 1910, Whitehead and Russell published the *Principia Mathematica*, which was in part an effort to put all of mathematics into symbolic form.

Would this take the creativity out of mathematics? If you can formalize everything, you can generate all possible theorems by applying your set of logical deduction rules to sets of axioms. Repeat, adding your conclusions at every step to the axioms. It would take forever, but would be totally deterministic and complete.

In 1900, David Hilbert gave a famous talk in which he listed problems he thought should direct mathematical effort. His tenth problem, paraphrased, was to find an algorithm to determine whether any given polynomial equation with integer coefficients (a *Diophantine equation*) has an integer solution. This was, again, part of the program to make all mathematics computational. People were once famous for being able to solve lots of quadratic equations, but the discovery of the quadratic formula put an end to that. Could we find such a formula for any degree equation?

Gödel showed Whitehead and Russell's quest to formalize mathematics was doomed to fail. His First Incompleteness Theorem shows any sufficiently strong axiomatic system has true but unprovable theorems: theorems that would never appear in the list being generated by automated deduction. Furthermore, what is meant by "sufficiently strong" is well within the bounds of what mathematicians would consider reasonable axioms for mathematics.

Church showed even Hilbert's more modest goal of mechanizing finding roots of polynomials was impossible [8]. However, Hilbert's tenth problem didn't acknowledge the possibility of such an algorithm simply not existing; at the time, there was no mathematical basis to approach such questions.

Hilbert's phrasing was as follows:

> Given a Diophantine equation with any number of unknown quantities and with rational integral[1] numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.

Church's response:

> 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 non-essential.]
>
> ... 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.

That is, it may not be *possible* to devise Hilbert's desired process, and in fact it is not, though that was shown much later. Church's major contribution is the

[1] *rational integer* = integer.

point that we need some *formal* notion of finite process to answer Hilbert – this is his *effective calculability*.

Church proposes two options in this paper: the lambda calculus, and what would later be called primitive recursive functions. Shortly thereafter Kleene proposed what we now call the *partial recursive functions* [32]. It was not widely accepted at the time that any was a good characterization of "effectively computable", however. It was not until Turing developed his Turing machine [59], 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 needed in such proofs.

This area of study took on a life of its own beyond simply answering Hilbert's challenge (often the way new fields of mathematics are introduced), becoming known as computability theory or recursion theory. Chapters 5–8 explore some of the additional topics and fundamental results of the area, and Chapter 9 contains a survey of the sorts of questions of current interest to computability theorists.

1.3 Some 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 [49], *Theory of Recursive Functions and Effective Computability* by H. Rogers [51], and *Recursively Enumerable Sets and Degrees* by R.I. Soare [56]. The material in here owes a great deal to those three texts. More recently, I have enjoyed A. Nies' book *Computability and Randomness* [48].

In how to present such material to undergraduates, I was influenced by such books as *Computability and Logic* by Boolos, Burgess, and Jeffrey [4], *Computability* by Cutland [10], *A Mathematical Introduction to Logic* by Enderton [19], *An Introduction to Formal Languages and Automata* by Linz [41], and *A Transition to Advanced Mathematics* by Smith, Eggen, and St. Andre [55].

1.4 A Little Request

I am looking to turn these course notes into a textbook in the near future, so any comments as to places you feel it could be improved are welcome (but do not feel obligated). Typos, yes, but more importantly spots where I was too terse, too verbose, or disjointed, sections or chapters you feel would benefit from reorganization, or places where you were left wanting more and would like to at least see a reference to outside sources.

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 as a reference, but we will cover them 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 grows 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 and ψ, sometimes ρ 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 "$\varphi \;\&\; \psi$". It is true when both φ and ψ are true, and false otherwise. Logically "and" and "but" are equivalent, and so are $\varphi \;\&\; \psi$ and $\psi \;\&\; \varphi$, 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*. Natural language tends to use *exclusive or*, where only one of the clauses will be 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? $\varphi \;\&\; \psi$ 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 natural language's "neither...nor" construction. These two negation rules are called *DeMorgan's Laws.*

Exercise 2.1.1. Simplify the following formulas.

(i) $\varphi \;\&\; ((\neg\varphi) \vee \psi)$.

(ii) $(\varphi \;\&\; (\neg\psi) \;\&\; \theta) \vee (\varphi \;\&\; (\neg\psi) \;\&\; (\neg\theta))$.

(iii) $\neg((\varphi \;\&\; \neg\psi) \;\&\; \varphi)$.

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 $\varphi \;\&\; (\neg\varphi)$. You will sometimes see the former expression denoted T (or $\top$) and the latter $\perp$.

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* and ψ the *consequent* 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 $\varphi \rightarrow \psi$ 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). $\varphi \rightarrow \psi$ 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 $\varphi \rightarrow \psi$ is only false when proven false, and that can only happen when you see a true antecedent and a false consequent. From this it is clear that $\neg(\varphi \rightarrow \psi)$ is $\varphi \;\&\; (\neg\psi)$.

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 $\varphi \rightarrow \psi$ 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 $(\varphi \to \psi)$ & $(\psi \to \varphi)$, 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 $(\varphi \,\&\, \psi) \vee ((\neg\varphi) \,\&\, (\neg\psi))$. Its negation is *exclusive or*, $(\varphi \vee \psi) \,\&\, \neg(\varphi \,\&\, \psi)$.

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 build 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	$\neg\varphi$	$\neg\psi$	$\varphi \,\&\, \psi$	$\varphi \vee \psi$	$\varphi \to \psi$	$\varphi \leftrightarrow \psi$
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 $(x < 0)$ & $(x + 100 > 0)$, 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.

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 or the rational numbers, this statement is true, because the reciprocal of x is available to play the role of y. However, in the integers or natural numbers 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 the natural numbers, integers, rational numbers, and real numbers. 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) $(\exists x)(x^3 + 8 = 0)$.

(vi) $(\exists x)(x^2 - 2 = 0)$.

When working with multiple quantifiers the order of quantification can matter a great deal. For example, take the two 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. However, ψ says "there is a number which is all other numbers' square" and is true only if you are working over the domain containing 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 an 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 notated $A \subseteq B$. 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. That is, $\{1,2,3\}$, $\{2,3,1\}$, and $\{1,1,2,2,3,3\}$ are all representations of the same set. We may also write it in 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. 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 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 all of our sets are subsets of $\mathbb{N}$. That is, we assume $\mathbb{N}$ is our universe. 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. If we have sets A_i for $i \in \mathbb{N}$, these are

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

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

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 $\bigcap_i B_i$?

When sets are constructed in computability theory, elements are typically put in a few at a time, stagewise. For set A, we denote the (finite) set of elements added to A at stage s or earlier as A_s, and when the writing is formal enough the construction will say A is defined as $\bigcup_s A_s$. When the writing is informal that is left unsaid, but is still true.

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 US 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 Japanese 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*, things like de Morgan's Laws,

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

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

and 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.

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. To prove that a set is countable, 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 about 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 formula 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 formulas 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 (2)? We may describe it in two ways: as "the negation of the relation in (2)", and how?

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

(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 need to be added to make it reflexive?

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

(iv) If the relation is not transitive, what is the smallest collection of pairs that need 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: for any pair of vertices, either there are edges in both directions or there are no edges between them.
- Antisymmetry: for two distinct vertices there is at most one edge 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 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. Within each box all possible pairings 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*, 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

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

That is, it doesn't matter which representative of the equivalence class you pick to do your addition.

The second important class of relations we will look at is partial orders.

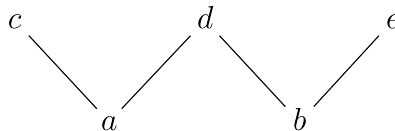
Definition 2.3.20. A *partial order* $\leq$ on a set A is a binary relation that is reflexive, antisymmetric, and transitive. A with $\leq$ is called a *partially ordered set*, or *poset*.

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:



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

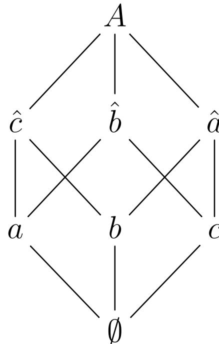
It is easy to check the relation $\subseteq$ is reflexive, transitive, and antisymmetric. Not every pair of elements is comparable: for example, neither $\{1, 2, 3\}$ nor $\{4, 5, 6\}$ is a subset of the other. This poset actually has some very nice properties that not every poset has: it has a top element ($\mathbb{N}$) 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 then as follows:



You could think of this as a cube standing on one corner.

Exercise 2.3.24. How many partial orders are possible on a set of two elements? Three elements?

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 $[(x, y) \in R \ \& \ (x, z) \in R] \to y = z$ (the vertical line rule for functions).[1] We can ramp this up even further to multivariable 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 Recursion and Induction

Recursive definitions and proofs by induction are essentially 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 holds of all the natural numbers. To do so, you prove that P holds of

[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.

0, and then prove that *if* P holds of some $n \geq 0$, *then* P also holds 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.

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.

Principle of Mathematical Induction, basic form:
If S is a subset of the positive integers such that $1 \in S$ and $n \in S$ 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 actually 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 2.4.1. Prove that for every positive integer n, the equation

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

holds.

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. □

For the next example we need to know a *convex* polygon is one where all the corners point out. The outline of a big block-printed V is a polygon, but not a convex one. The importance of this will be that if you connect two corners of a convex polygon with a straight line segment, the segment will lie entirely within the polygon.

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

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

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 our 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 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 $(n-2) \cdot 180°$. The $n+1$-gon therefore has interior angle sum $180° + (n-2)180° = (n+1-2) \cdot 180°$, 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 2.4.3. 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.

(vi) 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 which preserves order will give the same result.

Exercise 2.4.4. 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

(a) induction on the number of vertices;

(b) induction on the number of edges.

Exercise 2.4.5. 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.

Recursion

To define a class *recursively* means to define it via a set of basic objects and a set of rules allowing you to extend the set of basic objects. We may give some simple examples.

Example 2.4.6. 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 2.4.7. 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) $(\varphi \,\&\, \psi)$;
  - (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 building-up rules repeatedly to get more and more complicated objects.

For example, $((A\&B)\vee((P\&Q)\to(\neg A)))$ 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), $((P\&Q)\to(\neg A))$ from (iii), and finally our original formula by (ii).

Exercise 2.4.8. (i) Prove that $((A\vee(B\&C))\leftrightarrow C)$ is a wff.

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

Exercise 2.4.9. (i) Add a building-up rule to the recursive definition of $\mathbb{N}$ to get a recursive definition of $\mathbb{Z}$.

(ii) Add a building-up rule to the recursive definition of $\mathbb{Z}$ to get a recursive definition of $\mathbb{Q}$.

Exercise 2.4.10. 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. For this exercise, don't worry about the problem of division by zero.

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:

$$a(0,0) = 0;$$

$$a(m+1,n) = a(m,n)+1;$$

$$a(m,n+1) = a(m,n)+1.$$

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

Induction Again

Beyond simply resembling each other, induction and recursion have a strong tie in proofs. To prove something about a recursively-defined class requires induction. This use of induction is less codified than the induction on $\mathbb{N}$ we saw above. In fact, the limited version of induction we saw above is simply the induction that goes with the recursively-defined set of natural numbers, as in Example 2.4.6. Let's explore how this works in general.

The base case of the inductive argument will match the basic objects of the recursive class. The inductive step will come from the operations that build up the rest of the class. If they match exactly, 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 be the entire class.

Example 2.4.12. Consider the class of wffs, defined in Example 2.4.7. We may prove by induction that for any wff φ, the number of positions where binary connective symbols occur in φ (that it, $\&, \vee, \rightarrow$, 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, one less than 1.

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

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

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

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

Exercise 2.4.13. Suppose φ is a wff which does not contain negation (that is, it comes from the class defined as in Example 2.4.7 but without closure operation (v)). Prove by induction that the length of φ is of the form $4k + 1$ for some $k \geq 0$, and that the number of positions at which propositional symbols occur is $k + 1$ (for the same k).

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

Exercise 2.4.14. 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 2.4.10, prove that every rational function is continuous on its entire domain.

Exercise 2.4.15. 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, $a(m, n) = a(n, m)$).

2.5 Some Notes on Proofs and Abstraction

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 speakers of present-day English I doubt the original planting 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 which fulfill the terms of the definition. To say something is "by definition impossible" has a rigorous meaning in mathematics: if it contradicts one 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 being 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.

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

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. That is, avoid vagueness and leaps of deduction, and strip out irrelevant statements.

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, ...").

Our proofs will be very verbal – they will bear 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 be clumsy and expand proofs out of readability to avoid symbols altogether. It is also 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.

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.

Cautionary notes:

* 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.

* If you have a definition before you of a particular concept and are asked to prove something about the concept, you must stick to the definition.

* Be wary of mentally adding words like *only*, *for all*, *for every*, or *for some* which are not actually there; likewise 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.

* If you are asked to prove something holds of all objects of some type, you cannot pick a specific example and show the property holds of *that* object – it is not a proof that it works for all. Instead give a symbolic name to an arbitrary example and prove the property holds using only facts that are true for all objects of the given type.

* 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 which is not true, it may feel more natural to use the subjunctive, but that comes across as tentative. 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.
* And finally, though math class is indeed not English class, 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.

Exercise 2.5.1. 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 $A = B$.

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 which 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 §3.5).

3.1 Functions, Sets, and Sequences

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

Limits

Our functions take only whole-number values. Therefore, for the limit $\lim_{n\to\infty} f(n)$ 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 $\lim_n f(n)$.

Partial Functions

Let's go back to calculus, or possibly even algebra. A function definition is supposed to include not only the rule that associates domain elements with range elements, but also the domain. However, in calculus, we abuse this to give functions as algebraic formulas that calculate a range element from a domain element, and don't specify their domains; instead we say their domain is all elements of $\mathbb{R}$ on which they are 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 is discontinuous.

Here we take that mentality and make it official. In computability we use *partial functions* on $\mathbb{N}$, functions which 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 $\mathbb{N}$.

The intuition here is that the function is a computational procedure which may legally be given any natural number as input, but might go into an infinite loop on certain inputs and never output a result. Because we want to allow all computational procedures, we have to work with this possibility.

Most basically, we need notation. If x is in the domain of f, we write $f(x)\downarrow$ and say the computation *halts*, or *converges*. We might specify halting when saying what the output of the function is, $f(x)\downarrow = y$, though there the $\downarrow$ is fairly superfluous. When x is not in the domain of f we say the computation *diverges* and write $f(x)\uparrow$. We also still talk about $f(x)$, and by extension the computation, being *defined* or *undefined*.

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 meant is clear, $f(x) = y$ may be written $x \mapsto y$.

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 those objects aren't different at all. 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 that is the outputs of the function in order of input size.

Example 3.1.2. The sequence 1010101010... can represent

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

(ii) The function $f(n) = n \mod 2$.

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

(i) Infinite binary sequences, $2^{\mathbb{N}}$.

(ii) Total functions from $\mathbb{N}$ to $\{0, 1\}$.

(iii) Subsets of $\mathbb{N}$.

Sometimes it is useful to limit ourselves to finite objects.

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

(i) Finite binary sequences, $2^{<\mathbb{N}}$.

(ii) Finite subsets of $\mathbb{N}$.

(iii) $\mathbb{N}$.

3.2 Turing Machines

Our first rigorous definition of computation is due to Turing [59].

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's happening on the tape. Everything here is discrete: we think of the tape as being divided into squares, each of which can hold one symbol, and the read/write head as resting on an individual square and moving from square to square. 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". The symbol in position c may also be a blank, indicating the machine should erase

whatever symbol it is reading. For any fixed a, b, there is *at most* one quadruple. It is not necessary that there be any instruction at all; the computation may halt by hitting a dead end.

Since Turing machines represent idealized computers, we allow them unlimited time and memory to perform their computations. Not *infinite* time or memory, but we can't bound them from the beginning; what if our bound was just one step short of completion or one square of tape too small? So the TM's tape is infinite, though any given computation uses only a finite length of it.

The symbols and states come from a finite list and hence the collection of instructions must 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. Let's begin by writing a Turing machine that outputs $x + 1$ in tally notation given input x in tally notation (i.e., the tape begins by holding x consecutive 1s and ends with $x + 1$ consecutive 1s). Here is a sample input tape:

	*	1	1	1	1	*	*	*	
		↑							

The arrow indicates the starting position of the read/write head; we are allowed to specify that the input must be in tally notation and the TM's head be positioned 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 $*$ from state q_0 will be the first blank at the end of x.

What about binary notation instead? For example:

	*	0	1	1	0	0	1	*	*	
		↑								

We can choose to interpret this with the leftmost digit as the smallest, specifying that this is the sort of input our TM is designed to handle. Under that interpretation this input 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? 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:

$\langle q_0, 0, 1, q_1\rangle$
$\langle q_0, 1, 0, q_2\rangle$
$\langle q_2, 0, R, q_0\rangle$
$\langle q_0, *, 1, q_1\rangle$

Exercise 3.2.2. Step through the program above with the following tapes, where you may assume the read/write head begins at the leftmost non-blank square. Write the contents of the tape, position of read/write head, and current state of the machine for each step.

(i)

	*	0	1	1	*	

(ii)

	*	1	0	1	*	

(iii)

	*	1	1	1	*	

Exercise 3.2.3. Write a Turing machine to compute the function $f(x) = 4x$. Use tally or binary notation as desired.

Exercise 3.2.4. This exercise will walk you through writing an inverter, a Turing machine that, given a string of 1s and 0s, outputs the reversal of that string.

Here is what ought to happen:

Instruction Block A

	*	0	1	1	*	*	*	*	
		↑							

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

Instruction Block B

	*	0	1	*	1	*	*	*	
					↑				

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

Likewise:

	*	0	*	*	1	1	*	*	
						↑			

Instruction Block C

	*	*	*	*	1	1	0	*	
							↑		

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 (switch state at blanks). 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 (i.e., to see if the next spot is blank or not) and use state to account for a yes or no answer.

Exercise 3.2.5. Write a Turing machine to compute the function $f(x) = x \mod 3$. Use tally or binary notation as desired.

3.3 Partial Recursive Functions

Turing's machine definition of computability was far from the only competitor on the field. We will only explore one other in depth, but survey a few more in the next section. The partial recursive functions, where "recursive" is used as in §2.4, were Kleene's contribution.

Primitive Recursive Functions

We begin with a more restricted set of functions, the *primitive recursive* functions. This definition can be a little opaque at first, so 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.

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

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

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

(iv) (Composition.) 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.

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 that h to spit 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.

Exercise 3.3.3. Prove that the maximum function, $m(x, y) = \max\{x, y\}$, is primitive recursive.

Exercise 3.3.4. Prove that the multiplication function, $g(x, y) = x \cdot y$, is primitive recursive. You may use the addition function $f(x, y) = x + y$ in your derivation.

Exercise 3.3.5. 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, all the usual arithmetic functions on $\mathbb{N}$ are primitive recursive, such as exponentiation, factorial, and the modified subtraction

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

It is a very large class, including nearly all functions encountered in usual mathematical work, and perhaps has claim on the label "computable" by itself. We will argue in the following sections that it is insufficient.

The Ackermann Function

The Ackermann function is the most common example of a (total) computable function that is not primitive recursive; in other words, 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 be total. In fact, it was custom-built to meet that criterion, since the primitive recursive functions cover so much ground it seemed they might actually constitute all computable functions. The Ackermann functions is defined recursively for non-negative integers m and n as follows:

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

The version above is a simplification of Wilhelm Ackermann's original function due to Rózsa Péter and Raphael Robinson. It is not necessarily immediately clear that this function is computable – that the recursive definition always hits bottom. The proof that it is came later than the definition of the function itself.

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:

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

where 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.

The key is the stack of 2s. 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.

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 3.3.6. The class of *partial recursive functions* is the smallest class of functions such that the five conditions from Definition 3.3.1 of the primitive recursive functions hold, and additionally

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

One of the most important features of this closure scheme is that it introduces partiality; the primitive recursive functions are all total. A function using unbounded search *can* be total, of course, and in fact the Ackermann function requires unbounded search, despite being total. That is a sign that we perhaps need more than just the primitive recursive functions to capture all of computability.

Why should partiality be allowed? The first reason is that allowing all operations that seem like they ought to be allowed results in the possibility of partial functions. That is, from the modern perspective, real computers sometimes get caught in infinite loops. A more practical 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 3.4.6, below.

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

Example 3.3.7. Using unbounded search we can easily write a square root 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 $\dot{-}$ (where $x \dot{-} y = \max\{0, x - y\}$) 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 computable function

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

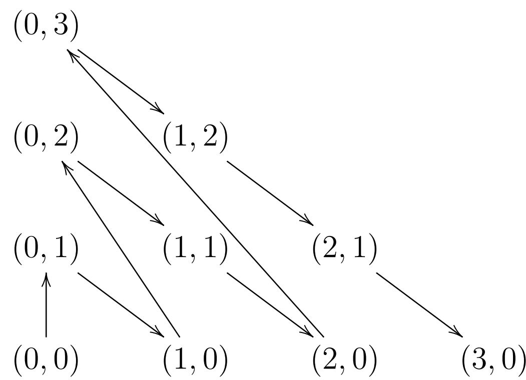
To move into $\mathbb{N}^2$, the set of ordered pairs of natural numbers, there is a standard *pairing function* indicated by angle brackets:

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

For longer tuples we iterate, so for example $\langle x, y, z \rangle := \langle \langle x, y \rangle, z \rangle$. Note this gives us a way to encode the rational numbers, $\mathbb{Q}$. It also lets 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.



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 one on its own diagonal is its first entry, so if we want to number these from 0, we let (x, y) map to

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

where all terms except the last correspond to the diagonals below (x, y)'s diagonal. This sums to

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

If you are unfamiliar with the formula for the summation of the integers 1 through n, you can find it in Appendix A.2.

The key elements of any coding function are that it be *bijective* and *computable*. There are two ways to think about how computation is performed under coding.

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 which takes $f(k)$ as input and outputs $f(2k)$ using approach 2 above. By the Church-Turing thesis you need not write a Turing machine or a formal partial recursive function; an algebraic expression will suffice.

There are limitations on what kinds of objects can be encoded: they must come from a set that is *effectively countable*. Here, in *countable* we include finite; all finite sets are effectively countable. An infinite countable set is effectively countable if there exists a *computable* bijection between the set and $\mathbb{N}$ which also has computable inverse. If we have such a bijection, we can use the image of an object, which will be a natural number, as the code of the object. This is equivalent to the objects

being representable by finite sequences of symbols that come from a finite alphabet, so that the symbols can be represented by numbers and the sequences of numbers collapsed down via pairing or a similar function. In fact, pairing is a bijection with $\mathbb{N}$ that shows that $\mathbb{N}^2$ and in fact $\mathbb{N}^k$ for all k are effectively countable.

In fact, $\bigcup_k \mathbb{N}^k$ is effectively countable, where here I intend k to start at 0 ($\mathbb{N}^0 = \{\emptyset\}$). The function

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

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

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

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 in the exponents?

(iii) What is the purpose of adding $1, 2, \ldots, k-1$ in the exponents?

(iv) Prove that τ is a bijection.

Exercise 3.4.3. (i) Given disjoint effectively countable sets A and B, prove that $A \cup B$ is effectively countable.

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

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 building-up rules (see §2.4), 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, so is A. §5.1 may be helpful.

Coding is generally 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 a need for separate definitions of *algorithm* for different kinds of objects.

There is another particular coding we need to discuss. 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 interpret an element of $\mathbb{N}$ as a finite subset of $\mathbb{N}$, as

in Exercise 3.1.4, and then interpret the elements of that finite subset as quadruples. A natural number n will be read as $k + \ell$, where $0 \leq \ell \leq 7$ and k is a multiple of 8. We decode $k/8$ into $\langle i, j \rangle$ and interpret it to say the starting and ending states of the quadruple are q_i and q_j. Then ℓ will give the symbol read and action taken; $0 \mapsto **$, $1 \mapsto *1$, $2 \mapsto *L$, $3 \mapsto *R$, and likewise for the four pairs beginning with 1. Note in this example we are using just the symbols $*$ and 1, but it is clear how this generalizes to any finite symbol set.

It is important to note that any method of coding 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. In fact, we can prove the Padding Theorem, Exercise 3.4.5, after a bit of vocabulary.

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 eth function in the enumeration, the function with index e, and the function that encodes to e. 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.

Exercise 3.4.5. 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. This is called the Padding Theorem.

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

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.

Theorem 3.4.6. *The total computable functions are not effectively countable. That is, there is no computable indexing of exactly the total computable functions.*

Proof. Suppose the contrary and let f_e denote the eth function in an enumeration of all total computable functions. We define a new total computable function as follows:

$$g(e) = f_e(e) + 1.$$

Since all f_e are total computable, it is clear that g is total computable.[1] Hence g must have an index; that is, there must be some e' such that $g = f_{e'}$. However, $g(e') \neq f_{e'}(e')$, which is a contradiction. Therefore no such indexing can exist. □

[1] Or perhaps it is not. See §4.1.

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 eth 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 which rely completely on the existence of an indexing in order to work; this combined with the justifications in §3.3 weigh heavily on the side of allowing partial functions to be called computable.

The take-home messages of this section, which are vital for our later work, are the following:

1. Via coding, we can treat any effectively countable set as though it were $\mathbb{N}$.
2. 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 The Church-Turing Thesis

It is not at all 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 which give the same class of functions as TMs and partial recursive functions (see §3.6 for a sample). It is even possible to introduce nondeterminism into our Turing machines without increasing their power!

Turing showed it is possible to use a Turing machine to evaluate a partial recursive function [59]. One can explicitly construct machines that compute successor, constants, and projections, and then show the Turing machines are closed under composition, primitive recursion, and unbounded search. Conversely, given a Turing machine we 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 [4] that we sketch.

The tape contents are viewed in two pieces, the spaces to the left of the read/write head as a 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: tape to left, current state, tape to right. 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 [59], Turing does a thought experiment 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 (or at least I find it compelling). This idea that we really have captured the correct/full notion is called the **Church-Turing Thesis**: the computable functions are exactly the Turing-computable functions.

3.6 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 programmable calculator's lobotomized 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.

In this chapter we have explored a machine definition of computability and a function definition. The definitions we add here fall into those same two categories, though there is a third category of symbol manipulation systems (this is, however, arguably a function definition for a different sort of function). We begin with modifications to Turing machines.

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 3.6.1. Prove that a two-symbol Turing machine can code (or *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 3.6.2. 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 3.6.3. 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 a read/write head 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 3.6.4. A *nondeterministic Turing machine* has an infinite tape, single read/write head that works from a finite list of symbols, and 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 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*.[2]

Claim 3.6.5. *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' 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 3.6.6. Turn the idea above into a proof of Claim 3.6.5.

The Lambda Calculus

This is an important function definition of computability. 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. This was Church's main contender for the definition of *computable*. It is one of the many models of computation which is equivalent to Turing machines and partial recursive functions; since it is important we'll explore it in some depth, though still only getting a taste of it. I learned about the lambda calculus in a programming languages class I took from Peter Kogge at Notre Dame, and this section is drawn from his lecture notes and book [28].

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

The lambda calculus is based entirely on *substitution*; typical expressions look like

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

which then is written $[A/x]M$, and means "replace every instance of x in M by A."

Expressions are built recursively. We have a symbol set which consists of parentheses, $|$, λ, and an infinite collection of *identifiers*, generally represented by lower-case 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,

$$(\lambda xy|M)AB = ((\lambda x|(\lambda y|M)))A)B = [B/y]([A/x]M).$$

Identifiers are essentially variables, but are called identifiers instead because their values don't change over time. We solve problems with lambda calculus by manipulating the form the variables appear in, 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 $(\lambda x|M)$, only free occurrences of x are candidates for substitution, and no substitution is allowed which 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 $(\lambda x|E)A \to [A/x]E \to E'$. They are defined recursively, in cases matching those of the recursive definition of expression.

1. If $E = y$, an identifier, then if $y = x$, $E' = A$. Otherwise $E' = E$.
2. If $E = BC$ for some expressions B, C, then $E' = (([A/x]B)([A/x]C))$.
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, then $E' = (\lambda z|[A/x]([z/y]C))$, where z is a symbol that does not occur free in A. This is the renaming rule.

Example 3.6.7. 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 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 $(\lambda x_1 \ldots x_n|E)A_1 \ldots A_m$, where $m \leq n$, the sequential evaluation

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

is equivalent to the *simultaneous* evaluation

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

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

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

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

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

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

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

I don't know that there is any way to understand these without stepping through an example.

Example 3.6.8. $2+3$.

To avoid variable clashes, we'll use s and a for s and z in 2 and r and b in 3.

$$2+3 = (\lambda wzyx|wy(zyx))(\lambda sa|s(s(a)))(\lambda rb|r(r(r(b))))$$

$$= (\lambda yx|(\lambda sa|s(s(a)))y((\lambda rb|r(r(r(b))))yx))$$

$$= (\lambda yx|(\lambda a|y(y(a)))(y(y(y(x)))))$$

$$= (\lambda yx|y(y(y(y(y(x)))))) = 5.$$

Exercise 3.6.9. Evaluate $S(3)$.

Exercise 3.6.10. Evaluate $2 \cdot 3$.

Similarly we can define lambda expressions that execute "if...then...else" operations. That is, we want expressions P such that PQR returns Q if P is true, and R if P is false. Then, additional Boolean operations are useful. We won't step through these, but I'll give you the definitions and you can work some examples out yourself.

$$\text{true} : T = (\lambda xy|x) \qquad \text{false} : F = (\lambda xy|y)$$
$$\text{and} : (\lambda zw|zwF) \qquad \text{or} : (\lambda zw|zTw)$$
$$\text{not} : (\lambda z|zFT)$$

Exercise 3.6.11. Work out the following operations:

not T, not F
and TT, and TF, and FT, and FF
or TT, or TF, or FT, or FF
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 something like RB, where B is somehow 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 the magic function

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

Try doing the substitution called for. Next, given some expression R wherein x does not occur free, try evaluating

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

This is not so 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'll omit any examples.

Unlimited Register Machines

Unlimited register machines, or URMs, are (as you would guess) a machine definition of computability. Nigel Cutland uses them as the main model of computation in his book *Computability* [10]; they 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* which is 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 ith 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 ith instruction, and nothing between the ith 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 3.6.12. Using three registers we can compute sums. The initial contents of the registers will be x, y, 0, 0, 0, ..., 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 our jump instruction jumps when the two values checked are different rather than the same, we have to be clever about how we use it. Here is a program that will add x and y:

Instructions:	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 3.6.13. 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 3.6.14. 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 A Universal Turing Machine

From the enumeration of all Turing machines we can denote a *universal* Turing machine; that is, a machine which will emulate *every* other machine. Using σ to denote an arbitrary input and 1^e to denote a string of e 1s, we can define the universal machine U by

$$U(1^e 0\sigma) = \varphi_e(\sigma).$$

U counts the 1s at the beginning of the input string, decodes that value into the appropriate set of quadruples, throws out the 0 it sees next, and uses the rest of the string as input, acting according to the quadruples it decoded. This procedure is computable because the coding of Turing machines as indices is computable.

This is why it is "clear" in Theorem 3.4.6 that g is total computable: were the total computable functions effectively enumerable, we wouldn't have a fleet of disparate $f_e(e)$ to evaluate for each e; we would have one $U(1^e 0e)$ for all e.

Note that of course there are infinitely-many universal Turing machines, as there are for any program via padding.

We can use the universal machine to construct a total recursive function that is not primitive recursive, in a different way from the Ackermann function. We'll still be pretty sketchy about it, though. Here's the outline:

1. Code all the primitive recursive functions as θ_n.

2. Show there exists a computable $p(n)$ such that for all n $\varphi_{p(n)} = \theta_n$, where $\varphi_{p(n)}$ lives in the standard enumeration of partial recursive functions.

3. Use the universal machine to define a new function which is total recursive but not any θ_n.

Some more detail:

1. Conceptually straightforward though technically annoying. We can code the *derivation* of our function via composition and primitive recursion, from constants, successor, and projection.

2. Start by arguing we have indices in the standard enumeration φ_n for the basic primitive recursive functions, which is true essentially because we can code them up on the fly in a uniform way (e.g., for constants, with a single function that takes any pair c, n to an index for the n-ary constant function with output c). Then we argue that we have explicit indices for composition and recursion as functions of the constituent functions' indices (again exploiting the fact that the index *is* the function), which is again true because we can explicitly code them.

   Then, given a θ-index n, we can uniformly find $p(n)$, a code for θ_n in the standard enumeration of partial recursive functions. We simply decode the θ-index and recode into a φ-index using the functions whose indices we just argued we have.

3. Using n to denote not only the integer but also its representation in binary, define the function
$$f(n) = U(1^{p(n)}0n) + 1.$$
   That is, $f(n) = \varphi_{p(n)}(n) + 1 = \theta_n(n) + 1$. Since θ_n is primitive recursive, it is total, which means f is total. However, it is not equal to any θ, as it differs from each on at least one input.

4.2 The Halting Problem

Is it possible to define a specific function which is not computable? Yes and no. We can't write down a procedure, because by the Church-Turing thesis that leads to a computable function. However, via the indexing of all partial computable functions we can define a noncomputable function.

First, a little notation recalled from §3.1. We use arrows to denote halting behavior: for a function φ_e, the notation $\varphi_e(n)\downarrow$ means n is in the domain of φ_e, and $\varphi_e(n)\uparrow$ means n is not in the domain of φ_e, so φ_e fails to halt on input n.

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:

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

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, so is g. However, it is straightforward to show 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 4.2.1. Prove that g defined above is not computable. You may find the contemplation of Theorem 3.4.6 helpful.

4.3 Parametrization

Parametrization means something different in computability theory than it does in calculus. What we mean here is the ability to push input parameters into the index of a function. Here is the first place that it is important that the indexing of Turing machines be fixed, and where we take major advantage of the fact that the index – a natural number – 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 e, x, and y, $\varphi_e(x,y) = \varphi_{S^1_1(e,x)}(y)$.*

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

This is a good moment to pause and think about uniformity, a key idea in computability. A process is uniform in its inputs if it is like a choose-your-own-adventure book: all possible paths from start to finish are already there in the book, 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 $\varphi_e(x,y)$.

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.

Theorem 4.3.3 (Kleene 1938)**.** *Given m, n, there is a primitive recursive one-to-one function S_n^m such that for all e, all n-tuples $\bar{x}$, and all m-tuples $\bar{y}$,*

$$\varphi_{S_n^m(e,\bar{x})}(\bar{y}) = \varphi_e(\bar{x}, \bar{y}).$$

That you can get this to be primitive recursive is interesting but not too important. The fact that you can force it to be one-to-one follows from the Padding Theorem (Exercise 3.4.5).

I'll note that while it looks at first like 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.

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 $U(1^x0z) + U(1^y0z)$, 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 $\varphi_{s(x,y)}(z) = f(x, y, z)$, so that is the index for $\varphi_x + \varphi_y$ as a (total computable) function of x and y.

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*.

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, at least in the way it is usually presented. It is extremely useful – vital, in fact – for a large number of proofs in the field. We will discuss this a bit 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, Kleene)**.** *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. 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, putting $x = m$ and letting $n = \varphi_m(m)$ (which is defined because s is total), we have

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

as required. □

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 for f from a different function g defined to be 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) There is a number n such that $\varphi_n(x) = x^n$.

(ii) There is a number n such that the domain of φ_n is $\{n\}$.

We may prove index set results easily from the Recursion Theorem, where $A \subseteq \mathbb{N}$ is an *index set* if it has the property that if $x \in A$ and $\varphi_x = \varphi_y$, then $y \in A$.

Theorem 4.4.7 (Rice's Theorem)**.** *Suppose that A is an index set not equal to $\emptyset$ or $\mathbb{N}$. Then A is not computable.*

Proof. Begins as follows: work by contradiction, supposing A is computable. Set some $a \in A$ and $b \notin A$ and consider the function

$$f(x) = \begin{cases} a & x \notin A \\ b & x \in A \end{cases}$$

Apply the Recursion Theorem. □

Exercise 4.4.8. Complete the proof of Rice's Theorem.

We may also use the Recursion Theorem to prove results about enumeration of Turing machines. In particular, there is no effective enumeration which takes the first instance of each function and omits the rest.

Theorem 4.4.9. *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. □

Now let's go back and discuss the theorem and its use in the wider world. The Recursion Theorem is often described as "a diagonalization argument that fails"; partiality is, in some sense, a built-in defense against diagonalization. In particular, if we wanted to define a function that differed from φ_e on input e, we would have to know whether $\varphi_e(e)\downarrow$, which bounces us out of the realm of the computable. The Recursion Theorem is a strong statement of the failure of that attempt.

In more detail, define the diagonal function δ by $\delta(e) = \varphi_e(e)$. This is a partial computable function; its domain is K, the Halting Set. For any total f we can define $f \circ \delta(x)$ as the result of the usual composition if $\delta(x)$ halts and undefined otherwise (confirm to yourself that composition defined in that way gives a partial computable function). Hence $f \circ \delta$ is φ_e for some e, and *if* $f \circ \delta(e)$ *is defined* it equals $\delta(e)$. In that case $\delta(e)$ is a fixed point for f, in the *literal* sense rather than the machine index sense. Now, we can see $f \circ \delta(e)$ can't always be defined, because $f(e) = e + 1$ is partial computable, but has no literal fixed point.

What we get instead is Corollary 4.4.2, a fixed point at the machine index level. The s-m-n theorem gives a *total* computable function d such that $\varphi_{d(i)} = \varphi_{\delta(i)}$ for all i such that $\delta(i)\downarrow$, and then the function s such that $\varphi_{s(i)} = \varphi_{f\circ d(i)}$. The argument from the previous paragraph gives us the rest, with adjusted functions: $f \circ d$ will be φ_e for some e, so $f(d(e)) = \delta(e)$ (now we are able to assert this is defined). By definition of d, $d(e)$ and $\delta(e)$ index the same function, so $\varphi_{d(e)} = \varphi_{f(d(e))}$ and $d(e)$ is the sought-after fixed point.

This is extraordinarily useful in constructions. Many of the uses can be summed up as building a Turing machine using the index of the finished machine. The construction will have early on a line something 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 then the construction's if/then statements would give different results depending on which particular f was in play, but such variations

will be *uniform*, as described in §4.3. That is, the construction is like a choose-your-own-adventure book, or a complicated flowchart. The particular function f selects the option, but what happens for all possible sequences of options is already laid out. Likewise, 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 uniformity has such a well-defined index.

4.5 Unsolvability

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

The celebrity example is Diophantine equations. In 1900 Hilbert posed a list of unsolved problems in mathematics as a challenge to drive mathematical development [25]. The tenth problem on the list asked for an algorithm to determine whether an arbitrary 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, Matijacevič proved the problem is unsolvable [44]. The full proof and story are laid out in a paper by Davis [13].

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. We omit all details but note that this is the main method to show a problem is undecidable: show you can encode the halting problem into it.

This section owes a lot to Martin Davis's book *Computability and Unsolvability* [11], where you can find more details about most of the topics below as well as additional topics, though you must translate into current notation. It is also drawn from course notes by Rod Downey at Victoria University of Wellington.

Index Sets

Rice's Theorem 4.4.7 can be viewed as a summary of a large number of undecidability results. It essentially says that any nontrivial property of the partial computable functions is unsolvable. Noting the domain of φ_e is typically denoted W_e, among

the index sets we might consider are the following:

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

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

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

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

All of the sets above are not only noncomputable, they are at a higher level of the noncomputability hierarchy than the Halting Set.

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. For production systems these procedures involve replacing certain subsequences of a string with other subsequences. We usually call the sequences *words*, and abbreviate strings of the same symbol using superscripts.

Example 4.5.1. The most general of productions allows us to replace strings at the beginning, middle, and end of a given word. 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 denote the empty string by λ.

We may apply this production to any word which 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 which are playing the roles of P and Q. Most simply, we could convert the word a^2b^3a to b^3a^3, which is the application of this production to the word $a\lambda ab^2\lambda ba$.

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

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

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

Definition 4.5.3. 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 4.5.4. Let the semi-Thue system Γ be the axiom ab^2ab together with the 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 or not that word is a theorem of Γ or another semi-Thue system?

Exercise 4.5.5. Construct a semi-Thue system with infinitely many theorems.

Exercise 4.5.6. 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 4.5.7. (i) Write an algorithm to determine whether or not a given word W is a theorem of the semi-Thue system S. Exercise 4.5.6 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 Γ_M. We extract and modify the contents of M's tape, treating it as a particular word. We insert an additional symbol into the word beyond the tape contents to indicate M's current state and location of the read/write head (put it just left of the current tape square), and the productions of Γ_M follow naturally:

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

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

The axiom of Γ_M is the initial state followed by the initial contents of the tape; i.e., the input $\bar{m}$.

The mimicry of M we're aiming for is to have a particular word be a theorem of Γ_M if and only if M halts on the input $\bar{m}$. To clean things up and take care of special cases, we add a special unused symbol (h) to the beginning and end of each word, and add productions that deal with that. 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 $q_i S_j \to q S_j$. Once we're in q we delete symbols to the right: for every symbol S_i, Γ_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: $S_i q' \to q'$. Ultimately, if M halts on $\bar{m}$, our production system with axiom $hq_0\bar{m}h$ will produce the theorem $hq'h$.

We have "proved" the following theorem:

Theorem 4.5.8. *It is not possible in general to decide whether or not a word is a theorem of a semi-Thue system.*

Exercise 4.5.9. How did the mimicry of Turing machines by semi-Thue systems give us Theorem 4.5.8?

Exercise 4.5.10. Write a proof of Theorem 4.5.8. In particular, fill in the details of the symbol h, formally verify that the construction works, and include the explanation of Exercise 4.5.9.

Post Correspondence

I'm including this one mostly because it's cute. We use the term *alphabet* for the set of all symbols used. If A is an alphabet and w a word all of whose symbols are in A we call w a word *on* A.

Definition 4.5.11 (Post, 1946 [50])**.** 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 $i \leq i_1, i_2, \ldots, i_n \leq m$ (the i_j need not be distinct) 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 4.5.12. 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 process Γ and a word v, we can construct a Post correspondence system that has a solution if and only if v is a theorem of Γ. Then we can conclude the following.

Theorem 4.5.13. *There is no algorithm for determining whether or not a given arbitrary Post correspondence system has a solution.*

Proof. Let Γ be a semi-Thue process 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 Γ. The alphabet of P is

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

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 Γ are $g_i \to \bar{g}_i$, $1 \le i \le k$, 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 Γ. However, we may now assert that v is a theorem of Γ 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:

$$\begin{array}{l} \langle [u\star, [\rangle, \langle \star, \star'\rangle, \langle \star', \star\rangle, \langle], \star' v]\rangle, \\ \left.\begin{array}{l} \langle \bar{g}_j, g_j'\rangle, \\ \langle \bar{g}_j', g_j\rangle \end{array}\right\} \text{ for } 1 \le j \le k \end{array}$$

Let $u = u_1 \to u_2 \to \cdots \to u_m = v$, where m is odd. Then the word

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

is a solution of P, with the decompositions

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

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 $1 \leq j \leq k$. Then $u_2' = r'\bar{g}_j's'$ and the correspondence is given by the productions $r \to r$, $s \to s$, and $g_j \to \bar{g}_j$.

For the converse, we show that if $\bar{w}$ is a solution, it is a derivation or concatenation of derivations of v from u. If $\bar{w}$ begins with [and later has a], let w be the portion of $\bar{w}$ up to the first]. Then

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

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

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

We must have u corresponding to some r' and v to some s', where $u \to r$ and $s \to v$. Then the $\star$ and $\star'$ must correspond to a $\star'$ and $\star$, respectively. If they do not correspond to each other we have

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

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

Furthermore, any solution $\bar{w}$ must begin with [and end with] (possibly with additional brackets in the middle). We have forced this by adding $'$ to the symbols in half of every pair. For $\bar{w}$ to be a solution, the symbol at the beginning of $\bar{w}$ must also begin both elements of a pair of P, and the only symbol that does so is [; likewise the only symbol that ends both elements of a pair of P is].

Hence, P has a solution if and only if v is a theorem of Γ; if we can always decide whether a Post correspondence problem has a solution we have contradicted Theorem 4.5.8. □

As a final note, we point out that this undecidability result is for *arbitrary* Post correspondence 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 [18], and at 7 pairs it is undecidable [45]. Between three and six pairs inclusive the question is still open.

Mathematical Logic

In §2.1 we met predicate logic as a way of writing formulas; it includes negation ($\neg$), the connectives &, $\vee$, $\to$ and $\leftrightarrow$, and the quantifiers $\forall$ and $\exists$. Here we must

broaden our perspective a bit to define a *logic* which is a whole unto itself rather than just a notational system.

A particular (predicate) logic consists of the symbols above as well as variables, constants, functions, and relations; all together they are called the *language*. We used this idea without comment in §2.1 when writing formulas about $\mathbb{N}$ or other number systems, using constants for elements of $\mathbb{N}$, arithmetic functions, and the relations $=$ and $<$. We define the notion of *formula* recursively (see §2.4), beginning with simpler notions.

A *term* is a constant, a variable, or a function of terms. For example, if $\cdot$, $+$, and 2 are in our language, $2 \cdot (x + y)$ is a term because 2, x, and y are individually terms, $x + y$ is a term because it is a function of x and y, and $2 \cdot (x + y)$ is a term because it is a function of the terms 2 and $x + y$.

An *atomic formula* is a relation of terms, such as $2 \cdot (x + y) > 5$. All atomic formulas are *formulas*, and if φ, ψ are formulas, so are $(\varphi \& \psi)$, $(\varphi \vee \psi)$, $\neg(\varphi)$, $\forall x(\varphi)$, and $\exists x(\varphi)$.

To define a particular logic we explicitly state the collections of constants, functions, and relations included. As an aside, this allows distinctions of *in which systems* a particular property is definable by a predicate formula. Any given logic may have multiple *interpretations*, structures $\mathcal{M}$ in which every constant, predicate, and function has a specified meaning. For example, $\mathbb{N}$ with the usual arithmetic and $\{0, 1, \ldots, 11\}$ with arithmetic modulo 12 are distinct interpretations of the logic with constants 0 and 1, functions $+$ and $\cdot$, and relation $=$. The structure $\mathcal{M}$ *models* a formula ψ, written $\mathcal{M} \models \psi$, if ψ is true under the interpretation. ψ is *valid* (tautological) if it is true in all interpretations, denoted $\vdash \psi$.

The problem is this: given a logic Σ, determine whether the Σ-formula ψ is valid. To prove that this is unsolvable, for each Turing machine M we construct a logic Σ_M and a formula ψ_M of Σ_M such that ψ_M is valid if and only if M halts. We give only a sketch of the proof, which is very similar to the construction for semi-Thue systems, Theorem 4.5.8.

Given a Turing machine M on the alphabet $\{0, 1\}$ with internal states $\{q_0, \ldots, q_n\}$, let the constants of the language Σ_M be $\{0, 1, q_0, \ldots, q_n, q, q', h\}$. Σ_M has one function f and one relation Q. The function $f(x, y)$, also written (xy), concatenates constants: the terms of Σ_M are words on the alphabet of constants and variables. The binary relation $Q(t_1, t_2)$ (for *quadruple*), which we will also write $t_1 \mapsto t_2$, holds exactly when t_2 is obtained from t_1 by one of the semi-Thue productions in the proof of Theorem 4.5.8. That is, we have the formula $(\forall x, y)[xAy \mapsto xBy]$ whenever $A \to B$ is one of the productions of Γ_M.

We now have a collection of finitely-many formulas we'll call *axioms*, one for each semi-Thue production. We need two more. The first says f is associative:

$$(\forall x, y, z)[((xy)z) = (x(yz))].$$

The second says Q is transitive:

$$(\forall x, y, z)[(x \mapsto y \;\&\; y \mapsto z) \to (x \mapsto z)].$$

Let the conjunction of all the above axioms be called φ. Then the Turing machine M halts on input x with initial configuration q_0 if and only if the formula $\psi := \varphi \to (h\, q_0\, x\, h \mapsto h\, q'\, h)$ holds. That is, M halts if and only if $\vdash \psi$.

Exercise 4.5.14. Formally prove that mathematical logic is undecidable by filling in the gaps in the proof sketch above.

Chapter 5

Computable and Computably Enumerable Sets

We've talked about computability for functions; now let's discuss 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 obtain their outputs, when they 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 that remain are trapezoidal, reminiscent of the tail of a bird seen from above, giving the source of the name in carpentry). In computability, we interleave portions of the computations.

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 for $f(0)$. If it halts, then we know $f(0)$. Either way, we run two steps of the computation for $f(1)$, and if necessary, two steps of the computation of $f(0)$. Step (or *stage*) n of this procedure is to run the computations for $f(0)$ 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

[1]Likewise, we could record our stopping point and just run, e.g., $f(0)$ one additional step 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

in finitely-many stages, each element of f's domain will eventually give its output.

To denote the computation after s steps of computation we give the function a subscript s: $f_s(n)$. We could 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.

If we are drawing from an indexed list of functions, we put both the index and the stage into the subscript: $\varphi_{e,s}(n)$. 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 we have more than just the function being approximated, as in §6.1. In this case the up or down arrow goes after everything: $\varphi_e(n)[s]\downarrow$.

Any collection of computations we can index can be dovetailed, gradually running more of them for more steps.

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.

The word *effective* is often used as a synonym for computable/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 they are not always exactly synonymous! Exercise 5.2.17 introduces the notion of *computably inseparable* sets. While we could say *recursively inseparable* for that, *effectively inseparable* sets are a different collection.

A good way to think about computable sets is that they have an associated 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". This is one use of a fact that comes up again and again,

cause $f(i)$ to have its $n-1$ step run at stage n, making the dovetailing truly diagonal.

which is that any finite amount of information may be assumed and not violate the computability of a procedure.

□

What about sets whose characteristic functions are noncomputable?

Definition 5.2.3. A set is *computably enumerable* (or *recursively enumerable*, abbreviated *c.e.*, *r.e.*) if there is a computable procedure to list its elements (not necessarily in order).

That definition is maybe 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.*

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

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

We must show this is 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)⇒(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

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

$\hat{\varphi}$ is computable because φ is. □

Exercise 5.2.5. 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.

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 not computable. What's the difference? 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.

As in §5.1, we use subscripts to denote partially-enumerated sets. In any construction we will have seen only finitely-many elements of A enumerated at a given stage s, and we denote that finite subset of A by A_s. When c.e. sets are constructed, formally it is the finite sets A_0, A_1, ..., that are built, and then A is defined as $\bigcup_s A_s$.

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 only infinitely many but uncountably many sets that are not computably enumerable.

Exercise 5.2.6. Prove that if A is c.e., A is computable if and only if $\overline{A}$ is c.e.

Exercise 5.2.7. Use Exercise 5.2.6 and the enumeration of c.e. sets, $\{W_e\}_{e\in\mathbb{N}}$, to give an alternate proof of the noncomputability of K.

Exercise 5.2.8. 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 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 B is computable even if $A - B$ is *computable*.

Exercise 5.2.10. 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

$$\hat{A} \cap \hat{B} = \emptyset \text{ and } \hat{A} \cup \hat{B} = A \cup B.$$

Exercise 5.2.11. Prove that the c.e. sets are *uniformly enumerable*: there is a single computable procedure that enumerates the pair $\langle e, x$ if and only if $x \in W_e$.

Exercise 5.2.12. Prove that the collection $\{(A_n, B_n)\}_{n\in\mathbb{N}}$ of all pairs of disjoint c.e. sets is uniformly enumerable, with the definition in Exercise 5.2.11 modified to involve triples $\langle n, i, x\rangle$, where $i \in \{0, 1\}$ indicates A or B. Note that as with the c.e. sets, the enumeration will contain repeats.

Exercise 5.2.13. Show that any infinite c.e. set contains an infinite computable subset.

Exercise 5.2.14. Show that any infinite set contains a noncomputable subset.

Exercise 5.2.15. Prove that if A and B are both computable (respectively, c.e.), then the following sets are also computable (respectively, 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 5.2.16. Show that if $A \oplus B$, as defined above, is computable (respectively, c.e.), then A and B are both computable (c.e.).

Exercise 5.2.17. 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 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 5.2.12. Let $x \in A$ iff $x \in A_x$ and $x \in B$ iff $x \in B_x$, and show A and B are computably inseparable. Hint: What if C were one of the B_n?

Exercise 5.2.18. Show that if A is computably enumerable, the union $B = \bigcup_{e\in A} W_e$ is computably enumerable. If A is computable, is B computable? Can you make any claims about $C = \bigcap_{e\in A} W_e$ given the computability or enumerability of A?

Exercise 5.2.19. (i) Call a relation computable if when coded into a subset of $\mathbb{N}$, that set is computable. Given a computable binary relation R, prove the set $A = \{x : (\exists y)((x, y) \in R)\}$ is c.e.

(ii) With R and A as above, prove that if A is noncomputable, for every total computable function f, there is some $x \in A$ such that every y such that $(x, y) \in R$ satisfies $y > f(x)$.

5.3 Noncomputable Sets Part I

So how do we create a noncomputable set? One way is by making its characteristic function nonequal to every total computable function. We can do this diagonally, by making A such that $\chi_A(e) \neq \varphi_e(e)$.

We want to say "Let $\chi_A(e) = 1$ if $\varphi_e(e) = 0$, and otherwise let it be 0." It's not so hard to prove A defined that way is c.e., but to generalize we have to be a little more careful, enumerating A gradually as we learn about the results of the various $\varphi_e(e)$ computations.

Let's consider this diagonal example further. 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 gives a value different from $\chi_A(e)$. We must also make sure A is c.e., which is a single requirement that permeates the construction.

To make sure A is c.e., we put elements into it but never take them out, and we make sure every step of the construction is computable. The construction itself, then, 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 we keep e out; that's the whole construction.

Why does this give a noncomputable set? In other words, why does it satisfy the requirements? Because if $\varphi_e(e)\uparrow$, we win that requirement. Otherwise the computation converges at some finite stage. If it converges to 0, at that stage we put e into A, and $\chi_A(e) = 1 \neq \varphi_e(e)$. Otherwise we keep e out of A, and $\chi_A(e) = 0 \neq \varphi_e(e)$.

5.4 Noncomputable Sets Part II: Simple Sets

Definition 5.4.1. A c.e. set A is *simple* if its complement is infinite but contains no infinite c.e. subsets.

That is, 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 having an infinite or finite complement is often called being *coinfinite* or *cofinite*, respectively.

Exercise 5.4.2. (i) Prove that if A is simple, it is not computable.

(ii) Prove that a coinfinite c.e. set is simple if and only if it is not contained in any coinfinite computable set.

(iii) Prove that if A and B are simple, $A \cap B$ is simple and $A \cup B$ is either simple or cofinite.

(iv) Prove that if A is simple and W_e is infinite, $A \cap W_e$ must be infinite (not just nonempty).

We now discuss the construction of a simple set. This perhaps seems technical, but 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 turns out to be easier to work with than the module we met in §5.3).

Just as before, we have an infinite collection of requirements to meet:

$$R_e : (|W_e| = \infty) \Rightarrow (A \cap W_e \neq \emptyset).$$

Additionally we have two overarching requirements,

$$A \text{ is c.e.}$$

and

$$|\overline{A}| = \infty.$$

As before, to make sure A is c.e., we will enumerate it during the construction and make sure 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 k-many requirements R_e are allowed to put numbers below $2k$ into A for any k, 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 during enumeration whether any given W_e will be finite or infinite. There could be a long lag time between enumerations, and we can't tell whether we need to wait longer to get more elements or whether we're done. 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 extra elements in A could hurt the construction.

Chapter 6

Turing Reduction and Post's Problem

6.1 Reducibility of Sets

We'd like a definition of relative computability that allows us to say (roughly) that set A is more or less computable than set B. Certainly it should be the case that the computable sets are "more computable" than all noncomputable sets; we can get a finer-grade division than those two layers, however. Intuitively, A is *reducible* to B if knowing B *makes* A computable.

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 single computation.

The restriction to only finitely-many questions is so the computation remains finite. We 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 the machine having an additional internal tape that is read-only and pre-printed with the characteristic function of A. That perhaps clarifies how we might code oracle Turing machines, 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; i.e., with the oracle. However, once again we must have uniformity; you can think of a pre-existing flowchart or tree diagram of the procedure. For example, a simple, pointless function might have a process as follows:

- Given input n, find if $n \in A$.

- If so, test each $k \in \mathbb{N}$. If any $k^2 = n$, halt and output k.
- If not, ask if $n^2 \in A$.
  * If so, halt and output n.
  * If not, go into an infinite loop.

We notate 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: $\varphi_{e,s}^{A_s}(n)$ abbreviates to $\varphi_e^A(n)[s]$.

Definition 6.1.2. A set A is *Turing reducible* to a set B, written $A \leq_T B$, if for some e, $\varphi_e^B = \chi_A$. A and B are *Turing equivalent*, $A \equiv_T B$, if $A \leq_T B$ and $B \leq_T A$.

This definition may also be made with functions. To match it to the above, we conflate a function f with its (coded) graph $\{\langle x, y\rangle : f(x) = y\}$.

Exercise 6.1.3. Prove $A \equiv_T \overline{A}$.

Exercise 6.1.4. (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, which is easiest from the function point of view: prove there is a function k such that for all i, e, f, g, h, if $h = \varphi_e^g$ and $g = \varphi_i^f$, then $h = \varphi_{k(e,i)}^f$.

(iii) Prove that $\equiv_T$ is an equivalence relation on $\mathcal{P}(\mathbb{N})$.

Exercise 6.1.5. (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.

One could think of Turing-equivalent sets as being closely related, like A and $\overline{A}$ are. The following is about as close as a relationship can get.

Definition 6.1.6. 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 6.1.7. Prove $=^*$ is an equivalence relation on $\mathcal{P}(\mathbb{N})$.

Exercise 6.1.8. (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 *complete* sets (or *Turing complete*): 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 6.1.9 (Post, 1944; see Davis [12]). *K is c.e., and if A is computably enumerable, then $A \leq_T K$.*

Proof. 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 the *s-m-n* 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, $f(x)$. Now,

$$x \in A \;\Rightarrow\; \varphi_e(x)\downarrow \Rightarrow\; \forall y(\varphi_{f(x)}(y) = 0) \;\Rightarrow\; \varphi_{f(x)}(f(x))\downarrow \Rightarrow\; f(x) \in K;$$

$$x \notin A \;\Rightarrow\; \varphi_e(x)\uparrow \Rightarrow\; \forall y(\varphi_{f(x)}(y)\uparrow) \;\Rightarrow\; \varphi_{f(x)}(f(x))\uparrow \Rightarrow\; f(x) \notin K.$$

□

Exercise 6.1.10. Another way to show K is complete is via an augmented halting set. Recall that $\langle x, y\rangle$ is the code of the pair x, y under the standard pairing function. Define $K_0 = \{\langle x, y\rangle : \varphi_x(y)\downarrow\}$. You show K_0 is c.e. via dovetailing, as with K, and it is clear that every c.e. set is computable from K_0. To complete Theorem 6.1.9, we need only show $K_0 \leq_T K$. Prove this directly, using the *s-m-n* Theorem 4.3.1.

When working with oracle computations we need to know how changes in the oracle affect the computation, or really, when we can be sure 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. There are various notations for use; I'll stick to $u(A, e, x)$, the maximum $n \in \mathbb{N}$ that φ_e asks A about during its computation on input x, plus one.

Another piece of notation: $A \upharpoonright n$ (A restricted to the first n elements) means $A \cap \{0, 1, \ldots, n-1\}$ (the reason for the "plus one" above). Note that if $A \upharpoonright (u(A, e, x)) = B \upharpoonright (u(A, e, x))$, then $u(B, e, x) = u(A, e, x)$ 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 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.

What this means for constructions is that if you want to preserve a computation $\varphi_e^A(x)$ while still allowing enumeration into A, you need only prevent enumeration of numbers $\leq u(A, e, x)$. Any others will leave the computation unharmed.

In §4.5 we saw a collection of problems that correspond to c.e. sets: the set of theorems of a semi-Thue system or logic is enumerable; the set of concatenations that might be solutions to a Post correspondence system is enumerable. In full generality every one of those is equivalent to the halting problem (we noted that embedding the halting problem into the decision problem was a standard method to prove undecidability), and it is not clear how one would reduce the generality in such a way as to become weaker than the halting problem without becoming computable.

This prompted Emil Post to ask the following question.

Question 6.1.11 (Post's Problem)**.** *Is there a set A such that A is noncomputable and incomplete?*

The answer is yes, though it took a while to get there. A refinement of the question asks whether there is a c.e. set that is noncomputable and incomplete. Why is that a refinement? Because any complete set will be Turing-above some non-c.e. sets as well as all the c.e. ones. So if we build an intermediate set without worrying about its enumerability, we might well end up with a non-c.e. one. However, we can also answer yes to the problem of whether there is a c.e. set between computable and complete, as proved in the next section.

Theorem 6.2.3. *There is a c.e. set A such that A is noncomputable and incomplete.*

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 §5.3 and §5.4. However, suppose further that these requirements may interact with each other, and to each other's detriment. As 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 in a vicious circle 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 was no waiting until we determine whether a set is finite or not. 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). This "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 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 only allow each to injure requirements further down the list. Then 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 the definition.

Definition 6.2.1. A c.e. set A is *simple* if $\overline{A}$ is infinite but contains no infinite c.e. subset.

Theorem 6.2.2. *There is a simple set.*

Proof. We will construct A to be simple via meeting the following two sets of requirements:

$$R_e : (|W_e| = \infty) \Rightarrow (A \cap W_e \neq \emptyset).$$

$$N_e : (\exists n > e)(n \in \overline{A}).$$

The construction will guarantee that A is computably enumerable. It is clear, as discussed in §5.4, that meeting all R_e will guarantee $\overline{A}$ contains no infinite c.e. subsets.

To see that meeting all N_e guarantees $\overline{A}$ is infinite, consider a specific N_e. If it is met, there is some $n > e$ in $\overline{A}$. Now consider N_n. If it is met, there is some $n' > n$

in $\overline{A}$; we may continue in this way. Thus satisfying all N_e requires infinitely many elements in $\overline{A}$.

We first discuss meeting each requirement in isolation, starting with R_e. If $W_{e,s} \cap A_s = \emptyset$, but an element enters W_e at stage $s+1$, R_e puts that element into A. It is then permanently satisfied, as we do not take elements out of A. Each N_e chooses a *marker* $n_e > e$ and prevents its enumeration into A.

The negative (N) requirements will prohibit some positive (R) requirements from enumerating elements into A. Some R requirements will enumerate elements into A that N requirements want to keep out of A. The priority ordering on these requirements is as follows:

$$R_0, N_0, R_1, N_1, R_2, N_2, \ldots$$

Recall that requirements earlier in the list have higher priority.

Now each R_e requirement may enumerate anything from W_e into A except for the elements prohibited by $N_0, N_1, \ldots, N_{e-1}$. Thus R_e might injure $N_{e'}$ for some $e' > e$, by enumerating its chosen value into A. This will cause the negative requirements from that point on to move their chosen n_e values past 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 $k < e$ (there is a suitable *witness* that we are able to use to satisfy R_e at stage $s+1$).

Construction:

Stage 0: $A_0 = \emptyset$. Each N_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 $s+1$.

End construction.

Now we must verify the construction has succeeded in meeting the requirements.

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, $n_e \notin A$.

Proof. The value of a marker changes only when 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. The value n_e is in $\overline{A}$ because each time a marker's value is enumerated into A, that marker is moved to a different value. ⊣

Lemma 3. If W_e is infinite, then $W_e \cap A$ is nonempty.

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 enumeration values is fixed. As W_e is infinite, it must contain values 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 a value not on the list, R_e will have the opportunity to act and will be satisfied. ⊣

Lemma 2 shows each N_e is satisfied, and Lemma 3 that each R_e is satisfied. Combined with the discussion preceding the construction, we see the proof of the theorem is complete. □

We can now construct a set that gives a positive answer to Post's Problem, Question 6.1.11.

Theorem 6.2.3. *There is a c.e. set A such that A is noncomputable and incomplete.*

The proof I will give combines the construction of A simple with creation of a specific c.e. set B that is not computed by A, to show 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 then enumerating elements into A and B but not taking them back out. The construction that makes an intermediate but not necessarily c.e. set takes elements back out again.

Proof of Theorem 6.2.3. We build A simple and B c.e. such that $B \not\leq_T A$. The requirements are the following.

$$R_e : (|W_e| = \infty) \Rightarrow (A \cap W_e \cap \{x \in \mathbb{N} : x > 2e\} \neq \emptyset).$$

$$N_e : \varphi_e^A \neq \chi_B.$$

A and B will be c.e. by construction.

The method for solving an R_e requirement in isolation is the same as in our original work on the simple set construction in §5.4.

We win an N_e requirement if φ_e^A is not total or if 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 §5.3. Each N_e will again pick a

witness n_e, as in Theorem 6.2.2, 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).[1] In that case N_e puts the witness into B. The difference between this and §5.3 is that with an oracle, the computation might not "stay halted." That is, as A changes, the behavior of φ_e^A may also change. Therefore N_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 the *use* function from the end of §6.1).

The priority ordering is as in Theorem 6.2.2:

$$R_0, N_0, R_1, N_1, R_2, N_2, \ldots$$

Each R_e must obey restraints set by N_k for $k < e$, but may injure later N requirements by enumerating into A below the restraint of that N, after N has enumerated its witness into B. In that case, N 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 N_k for $k < e$. N_e *requires attention* at stage s if it has a defined witness $n_{e,s}$, and $\varphi_e^A(n_e)[s]\downarrow = 0$ but $n_{e,s} \notin B_s$.

Construction:

$A_0 = B_0 = \emptyset$.

Stage s: Set $n_{s,s}$ to be the least number not yet used in the construction (as there have been only finitely-many stages and only finitely much happens in any given stage, there will always be an infinite tail of $\mathbb{N}$ to work with).

Ask if any R_e or N_e with $e < s$ requires attention. If so, take the highest-priority such and act to satisfy it:

- If N_e, put $n_{e,s}$ into B_{s+1} and restrain A up to $u(A_s, e, n_{e,s})$.
- If R_e, put the least applicable x into A_{s+1}. Cancel the witnesses (and restraints, if any were set) of requirements N_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, any $n_{e,s+1}$ that was not specifically defined is equal to $n_{e,s}$; restraints hold until they are cancelled by injury.

End construction.

Now, the verification.

Lemma 1. Each R_e requirement acts at most once.

Proof. Clear. ⊣

Lemma 2. For every e, $n_e = \lim_s n_{e,s}$ is defined.

[1] Why pick a specific n_e instead of just looking for some difference somewhere? Because it streamlines the construction and doesn't make it any more difficult to verify.

Proof. As before, this lemma follows entirely from Lemma 1: since every R_e requirement acts at most once, the finite collection of them preceding N_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. ⊣

Lemma 3. Every N_e requirement is met. Moreover, each N_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 N_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, N_e will never be injured again and will have a single witness n_e to worry about. By induction, assume all higher-priority N_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 N_e does by default. N_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 N_e never sets a restraint for the permanent value of n_e, and any restraints it set on behalf of earlier witnesses are cancelled. N_e will never act again.

Case 2: $\varphi_e^A(n_e)\downarrow= 0$. Suppose the final value of n_e is assigned to N_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 N requirements have stopped acting. Then at the first stage $s'' \geq s'$ such that $\varphi_e^A(n_e)[s'']\downarrow= 0$, N_e will be the highest-priority requirement needing attention and will set restraint on A up to $u(A_{s''}, e, n_e)$ 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 at its final level, and N_e will never act again. ⊣

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 N_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 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. ⊣

This completes the proof of the theorem. □

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 $0'$ is simply to be computable from $0'$, or Δ^0_2 (see §7.2.5). For A to be Δ^0_2 means there is a computable function $g(x,s)$ such that for each x, $\lim_{s\to\infty} g(x,s) = \chi_A(x)$, and the number of times g "changes its mind" on each x is finite:

$$(\forall x)\,(|\{s : g(x,s) \neq g(x,s+1)\}| < \infty)\,.$$

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 additional elements we hadn't counted on as well as the removal of elements we thought were in the set. Restraint on the set being constructed serves to control 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.

In between 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.*

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 less 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.

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. For the remainder, satisfy the following requirements:

$$P_e : \varphi_e \text{ total } \Rightarrow (\exists x,y)[\varphi_e(x,y) = 1 \text{ but } (\exists z)(x \leq_L z \leq_L y)]$$

$$N_x : \text{there are only finitely-many elements } L\text{-below } x$$

Exercise 6.2.5. Using a finite injury priority argument, build a bi-immune Δ^0_2 set A. That is, A such that A and $\overline{A}$ both intersect every infinite c.e. set. Meet the following requirements:

$$P_e : |W_e| = \infty \Rightarrow (\exists n)(n \in W_e \cap A)$$

$$R_e : |W_e| = \infty \Rightarrow (\exists n)(n \in W_e - A)$$

P_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.

Chapter 7

Turing Degrees

7.1 Turing Degrees

Recall from Exercise 6.1.4 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 $\deg(A) = [A]$, the equivalence class of A under Turing equivalence. This is often notated $\boldsymbol{a}$, 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 $\deg(A \oplus B)$ ($A \oplus B$ is defined in Exercise 5.2.15).

(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*.

Part (iii) of Exercise 7.1.2 explains the wording of the following definition.

Definition 7.1.3. 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 6.1.9.

Exercise 7.1.4. Prove that each Turing degree contains only countably many sets.

Corollary 7.1.5. *There are uncountably many Turing degrees.*

7.2 Relativization and the Turing Jump

The notion of *relativization* is one of fixing some set A and always working with A as your oracle: working *relative to* A. Then computability becomes computability in A (being equal to φ_e^A for some e, also called A-computability) and enumerability become enumerability in A (being equal to $W_e^A := \mathrm{dom}(\varphi_e^A)$ for some e). Some examples follow.

Theorem 7.2.1 (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 $e, y_1, \ldots, y_m \in \mathbb{N}$,*

$$\varphi^A_{S_n^m(e,y_1,\ldots,y_m)}(z_1,\ldots,z_n) = \varphi_e^A(y_1,\ldots,y_m,z_1,\ldots,z_n).$$

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 computable in A, it is computable. 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:

Exercise 7.2.2. Prove that $B \leq_T A$ if and only if B and $\overline{B}$ are both c.e. in A.

Here's a very important example of relativization.

Theorem 7.2.3. *The set $A' = \{e : \varphi_e^A(e)\downarrow\}$ is c.e. in A but not A-computable.*

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). If we want to iterate the jump that 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''''.

The jump of a set is always strictly Turing-above the original set, and computes it. Jump never inverts the order of Turing reducibility, though it may collapse it. That is, the jump operator is not one-to-one.

Proposition 7.2.4. (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 $A' \equiv_T \emptyset'$.

The degrees $\boldsymbol{\emptyset}, \boldsymbol{\emptyset'}, \dots, \boldsymbol{\emptyset^{(n)}}, \dots$ 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 *arithmetic hierarchy* is a way of categorizing relations according to how complicated the logical predicate representing them has to be. Let's have some definitions.

Definition 7.2.5. (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 $R(x, y_1, \dots, y_n)$ such that $x \in B \iff (\exists y_1)(\forall y_2)(\exists y_3) \dots (Q y_n)(R(x, y_1, \dots, y_n))$, where the quantifiers alternate and hence $Q = \exists$ if n is odd, and $Q = \forall$ if n is even.

(v) Likewise, B is in Π_n if there is a computable relation $R(x, y_1, \dots, y_n)$ such that $x \in B \iff (\forall y_1)(\exists y_2)(\forall y_3) \dots (Q y_n)(R(x, y_1, \dots, y_n))$, 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 in addition to numbers, and would obtain the *analytic hierarchy.* That's outside the scope of this course.

Exercise 7.2.6. Prove the following basic results.

(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 computable if and only if it is in Δ_1 (i.e., $\Delta_0 = \Delta_1$).

(iv) B is c.e. if and only it is in Σ_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. Note that the Σ_n formulas with one free variable (the formulas that define subsets of $\mathbb{N}$) are effectively countable, as in Exercise 3.4.4. 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 eth 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 7.2.6, $\overline{P}$ would be Σ_n. However, 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 $\hat{e} \in \overline{P} \Leftrightarrow \hat{e} \notin P \Leftrightarrow \langle \hat{e}, \hat{e}\rangle \notin S$, for a contradiction.

The complement $\overline{P}$, then, is Π_n but not Σ_n.

Exercise 7.2.7. Prove that there is a Δ_{n+1} set that is neither Π_n nor Σ_n. Hint: use P and $\overline{P}$ as above, merge them, and use parts (i) and (v) of Exercise 7.2.6.

The strong connection to Turing degree continues as we move up the scale of complexity.

Definition 7.2.8. 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.

Theorem 7.2.9. $\emptyset^{(n)}$ *is* Σ_n*-complete and* $\overline{\emptyset^{(n)}}$ *is* Π_n*-complete for all* $n > 0$.

The index sets we saw in §4.5 are all complete at some level of the hierarchy. Fin is Σ_2-complete, Inf and Tot are Π_2-complete, and Rec is Σ_3-complete.

In fact, the following also hold. Theorem 7.2.9 combined with the proposition below is known as Post's Theorem.

Proposition 7.2.10. (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, it holds for *some* value of the parameter.

Theorem 7.2.9 and Proposition 7.2.10 both relativize. For Theorem 7.2.9 the relativized version starts with "for every $n > 0$ and every set A, $A^{(n)}$ is Σ_n^A-complete" (recall 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

$$x \in A \iff \lim_s g(x, s) = 1.$$

(ii) Prove A is Π_2 if and only if there is a total computable function $g(x, s)$ with codomain $\{0, 1\}$ such that

$$x \in A \iff \lim_s g(x, s) \neq 0.$$

Some general rules for working in the arithmetic hierarchy:

- Like quantifiers can be collapsed to a single quantifier. For example, $(\exists x_1)(\exists x_2)(\exists x_3)(R(y, x_1, x_2, x_3))$ is still Σ_1. This follows from codability of tuples.

- 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.

- Bound quantifiers, ones which only check parameter values on some initial segment of $\mathbb{N}$, do not increase the complexity of a formula. For example, $(\exists x < 30)(\forall y)(R(x, y))$ is just Π_1.

- It is possible to turn a Π_n statement into an infinite collection of Σ_{n-1} statements by bounding the leading universal quantifier with larger and larger bounds. That is, turn $(\forall x)[\Sigma_{n-1}]$ into the set $\{(\forall x < m)[\Sigma_{n-1}] : m \in \mathbb{N}\}$. The original Π_n formula is true if and only if all Σ_{n-1} formulas in the set are true. This is useful because in general we have a much better idea of how to cope with Σ formulas (since they are c.e. over some iteration of the halting problem) than Π formulas (which are co-c.e.; there is not as much machinery developed for them).

Chapter 8

More Advanced Results

This chapter contains some miscellaneous results and ideas in computability theory that come up with some frequency.

8.1 The Limit Lemma

Recall that a set 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. We argue that if we can accomplish such a feat for all e, A is low. The following general theorem proves rigorously why that works and is useful for more than just constructing low sets.

Theorem 8.1.1 (Limit Lemma)**.** *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 which limits to f. Since we are working with output values in $\mathbb{N}$, note that to limit to a value means to take that value on all but finitely-many inputs.

To make A low in the discussion above, we are taking a computable function which limits to the characteristic function of A'. That computable function is defined as follows.

$$g(e, s) = \begin{cases} 0 & \text{if } \varphi_e^A(e)[s]\uparrow \\ 1 & \text{if } \varphi_e^A(e)[s]\downarrow \end{cases}$$

It is computable provided the construction is computable (so that A_s may be obtained from s), and by making sure the computation eventually either converges or diverges, we are making sure for each e $\lim_s g(e, s)$ exists. It is clear from the definition that if the limit exists it has to match the characteristic function of A'.

Since not all sets which are reducible to $\emptyset'$ (i.e., Δ^0_2 sets) are c.e., this is often the most useful way to work with them. There is a way to use the approximating function to distinguish between c.e. and general Δ^0_2 sets, and in fact we need part of it in order to prove the Limit Lemma, so let's consider it now.

Recall that the *mu-operator* returns the minimal example satisfying its formula: $(\mu x)[x > 1 \,\&\, (\exists y)(y^2 = x)]$ would be 4.

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. All notation is as in Definition 8.1.2.

(i) Prove that the least modulus is computable in any modulus.

(ii) Prove that $B \geq_T g(x, s)$.

(iii) Prove that f is computable from g and any modulus m for g.

In general we cannot turn the reducibility of (iii) 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 which is computable from B.*

Proof. Let B be c.e. and let $f = \varphi_e^B$. Define the 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 quantifier on z is bounded and hence does not increase complexity, the first clause is computable, and the second clause is clearly B-computable (it 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. ($\Longrightarrow$) 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 relativized to B.

($\Longleftarrow$) 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 notated $\oplus_x B_x$), 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 $f \leq_T m \oplus B \leq_T C \oplus B \leq_T B'$. □

Properties of the modulus of the limiting function are what give us a characterization of the c.e. degrees.

Corollary 8.1.5. *A function f has c.e. degree iff f is the limit of a computable function $g(x,s)$ which has a modulus $m \leq_T f$.*

Exercise 8.1.6. Prove Corollary 8.1.5; for ($\Longrightarrow$) apply the Modulus Lemma; for ($\Longleftarrow$) use C from the proof of the Limit Lemma.

8.2 The Arslanov Completeness Criterion

This is a result that can be viewed as the flip side of the Recursion Theorem, and is presented mostly as a companion to the Recursion Theorem. Recall that a *complete* c.e. set is one that has the same degree as the halting problem. 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 $n(y)$ such that $\varphi_{n(y)} = \varphi_{f(n(y),y)}$ for all y.*

Proof. Define a computable function d by

$$\varphi_{d(x,y)}(z) = \begin{cases} \varphi_{\varphi_x(x,y)}(z) & \text{if } \varphi_x(x,y)\downarrow; \\ \uparrow & \text{otherwise.} \end{cases}$$

Choose v such that $\varphi_v(x,y) = f(d(x,y),y)$. Then $n(y) = d(v,y)$ is a fixed point, since unpacking the definitions of n, d and v (and then repacking n) we see

$$\varphi_{n(y)} = \varphi_{d(v,y)} = \varphi_{\varphi_v(v,y)} = \varphi_{f(d(v,y),y)} = \varphi_{f(n(y),y)}.$$

□

In fact we may replace the total function $f(x,y)$ with a partial function $\psi(x,y)$ and have total computable n such that whenever $\psi(n(y),y)$ is defined, $n(y)$ is a fixed point. The proof is identical to the proof of the Recursion Theorem with Parameters. Note that the parametrized version implies the original version by considering functions which ignore their second input.

Theorem 8.2.2 (Arslanov Completeness Criterion, Arslanov 1977/1981)**.** *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. ($\Longrightarrow$) We note without proof $\{x : W_x = \emptyset\}$ is of complete c.e. degree and hence Turing equivalent to whatever A we were given. Define f by

$$W_{f(x)} = \begin{cases} \emptyset & \text{if } W_x \neq \emptyset \\ \{0\} & \text{otherwise.} \end{cases}$$

By the observation $f \leq_T A$, and it clearly satisfies the right-hand side of the theorem.

($\Longleftarrow$) Let A be c.e., and assume $(\forall x)[W_{f(x)} \neq W_x]$ where $f \leq_T A$. By the Modulus Lemma 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 $m \leq_T A$). Let K denote the halting set, and let $\theta(x) = (\mu s)[x \in K_s]$ if $x \in K$; $\theta(x)\uparrow$ otherwise. By the Recursion Theorem with Parameters define the computable function h by

$$W_{h(x)} = \begin{cases} W_{g(h(x),\theta(x))} & \text{if } x \in K; \\ \emptyset & \text{otherwise.} \end{cases}$$

Now if $x \in K$ 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. Hence[1] for all x

$$x \in K \iff x \in K_{m(h(x))}$$

so $K \leq_T A$. □

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 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 §8.3). These are also called *almost fixed points*. Weaker still are *Turing fixed points*, n such that $W_n \equiv_T W_{f(n)}$.

Just as a catalogue:

- Any 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.
- If f is total and computable in $\emptyset''$ then f 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 $A = B$,

[1] This "hence" hides a long string of consequences of $\theta(x) < m(h(x))$.

(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). *Fix $\alpha \in \mathbb{N}$. Suppose $\emptyset^{(\alpha)} \leq_T A$ and A is c.e. in $\emptyset^{(\alpha)}$. Then*

$$A \equiv_T \emptyset^{(\alpha+1)} \iff (\exists f \leq_T A)(\forall x)[W_{f(x)} \not\sim_\alpha W_x].$$

8.3 $\mathcal{E}$ Modulo Finite Difference

Recall from §6.1 that $A =^* B$ means 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 which is not equal to what you want, but is $*$-equal. Momentarily I'll show you the main use (which is essentially the above but on a grander scale).

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 (well, at least more than with degrees).

Definition 8.3.1. $\mathcal{E}$, $\mathcal{R}$, and $\mathcal{N}$ are all *lattices*; that is, they are 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*. All three lattices have least and greatest element, moreover (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 8.3.2. (i) Show $\mathcal{N}$ and $\mathcal{R}$ are Boolean algebras but $\mathcal{E}$ is not.

(ii) Characterize the complemented elements of $\mathcal{E}$.

(iii) How small may a Boolean algebra be?

Definition 8.3.3. A property P is *definable* in a language (i.e., a set of relations, functions, and constants) 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 sequences 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 would be $\subseteq$) by the formula

$$(\forall x)[y \leq x].$$

The formula is true of y 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 8.3.4. Let the language $L = \{\leq\}$ be fixed.

(i) Show greatest element is definable in L.

(ii) Show meet and join are definable (via formulas with three free variables) in L.

Definition 8.3.5. An *automorphism* of a lattice $\mathcal{L}$ is a bijective function from $\mathcal{L}$ to $\mathcal{L}$ which preserves the partial order.

Exercise 8.3.6. (i) Show that automorphisms preserve meets and joins.

(ii) Show that a permutation of $\mathbb{N}$ induces an automorphism of $\mathcal{N}$.

(iii) What restrictions could we set on permutations of $\mathbb{N}$ to ensure they induce automorphisms of $\mathcal{R}$? Of $\mathcal{E}$?

Definition 8.3.7. 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*: that is, for any $x, y \in X$ there is an automorphism f of $\mathcal{L}$ such that $f(x) = y$.

Exercise 8.3.8. What sort of structure (relative to automorphisms) must an invariant class that is not an orbit have?

Definition 8.3.9. 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 $\mathcal{L} = \{\leq, \vee, \wedge, 0, 1\}$ which defines the class of sets with property P in $\mathcal{E}$, where $\leq, 0, 1$ are interpreted as $\subseteq, \emptyset, \mathbb{N}$, respectively.

Exercise 8.3.10. 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. We almost always are worried about properties which are preserved if only finitely many elements of the set are changed; that is, properties which are *closed* under finite difference. One can show that the collection of all finite sets and the relation $=^*$ are both definable in $\mathcal{E}$, and from there it is straightforward to 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 $\mathcal{E}$, and by the agreement of definability between those two structures we can get results about $\mathcal{E}$ from $\mathcal{E}^*$.

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, with some of the questions currently under investigation. Actually, only §9.1 discusses "pure" computability theory; the others are independent areas that intersect significantly with computability.

9.1 Lattice-Theoretic Properties

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 the structure of 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 $\mathcal{R} \subset \mathcal{D}$).

An *automorphism* of a poset is a bijection f from the poset to itself that preserves the order relation; that is, if $x \leq y$, $f(x) \leq f(y)$. It is nontrivial if it is not the identity. The big open question here is:

Question 9.1.2. *Is there a nontrivial automorphism of* $\mathcal{D}$*? Of* $\mathcal{R}$*?*

I should note that it is still open whether there is a "natural" intermediate degree. That is, the decidability problems we stated all gave rise to sets which were 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? Researchers in the area have varying opinions on how important that question is, given the many ways we have to construct intermediate degrees.

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)".

An example of a lattice-theoretic definition would be least upper bound and greatest lower bound. Least upper bound (join) may be defined as follows.

$$z = x \vee y \iff (z \geq x \;\&\; z \geq y \;\&\; (\forall w)[(w \geq x \;\&\; w \geq y) \rightarrow w \geq z]).$$

Greatest lower bound is defined similarly; the top and bottom element can also be defined.

Here's a more complicated but still purely lattice-theoretic definition.

Definition 9.1.3. A degree $\boldsymbol{a} \in \mathcal{R}$ is *cuppable* if there is some c.e. degree $\boldsymbol{b} < \boldsymbol{\emptyset'}$ such that $\boldsymbol{a} \vee \boldsymbol{b} = \boldsymbol{\emptyset'}$. The degree $\boldsymbol{a}$ is *cappable* if there is some c.e. degree $\boldsymbol{c} > \boldsymbol{\emptyset}$ such that $\boldsymbol{a} \wedge \boldsymbol{c} = \boldsymbol{\emptyset}$.

Another aspect of lattices that help 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.

For example, in Example 2.3.23 we had a lattice with eight elements. We can embed the four-element diamond lattice into that lattice in many different ways, where in some embeddings the least element of the diamond maps to the least element of the eight-node lattice, in some the greatest element of the diamond maps to the greatest element of the eight-node lattice, 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 preserve lattice properties like 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 purely computability-theoretic property. First a definition.

Definition 9.1.5. 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 is some s such that for some x that enters W_e at stage s, x enters A no later than stage $p(s)$.

As usual, a degree is promptly simple if it contains a promptly simple set.

Theorem 9.1.6. *The promptly simple degrees are exactly the non-cappable degrees.*

Remember 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. *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.

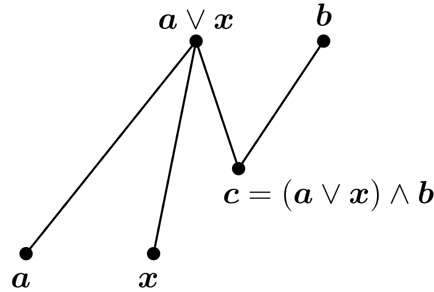
An example of a lattice-theoretic definition of a degree which may or may not have any examples is the following.

Definition 9.1.8 (Li)**.** A c.e. degree $\boldsymbol{a}$ is a *center* of $\mathcal{R}$ if for every c.e. degree $\boldsymbol{x} \neq \boldsymbol{\emptyset'}$, there are c.e. degrees $\boldsymbol{c} < \boldsymbol{b}$ such that

- $\boldsymbol{c} < \boldsymbol{a} \vee \boldsymbol{x}$, and
- $(\boldsymbol{a} \vee \boldsymbol{x}) \wedge \boldsymbol{b} = \boldsymbol{c}$,

where $\vee$ denotes least upper bound and $\wedge$ denotes greatest lower bound.

Recall greatest lower bound need not even exist for a pair of degrees. When $\boldsymbol{x}$ is above $\boldsymbol{a}$, so their join is simply $\boldsymbol{x}$ again, we just need some $\boldsymbol{b}$ incomparable to $\boldsymbol{x}$ such that the meet of $\boldsymbol{b}$ and $\boldsymbol{x}$ exists (that will be $\boldsymbol{c}$); thus everything above a center has a meet with *something*. If $\boldsymbol{x}$ is below $\boldsymbol{a}$, so their join is $\boldsymbol{a}$, then we need some $\boldsymbol{b}$ incomparable to $\boldsymbol{a}$ which has a defined meet with $\boldsymbol{a}$. If $\boldsymbol{a}$ and $\boldsymbol{x}$ are incomparable we get the picture below. For any $\boldsymbol{x}$, there are $\boldsymbol{b}$ and $\boldsymbol{c}$ such that:



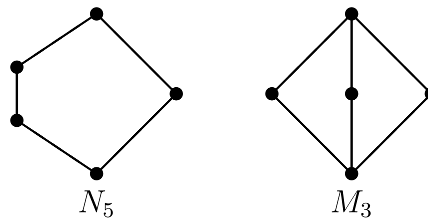
Note that neither $\boldsymbol{a}$ nor $\boldsymbol{x}$ need be above $\boldsymbol{c}$ individually. The following question, as far as I can tell, is still open.

Question 9.1.9. *Is there a center in* $\mathcal{R}$*?*

Definition 9.1.8 is an example of a definition made purely in the language of lattice theory: we do not have to know where the poset $\mathcal{R}$ comes from to understand it, it simply uses the partial order relation (meet and join are both definable in terms of the partial order relation: exercise). However, solving it will likely take the form of a priority argument, constructing a set A and at the same time constructing an infinite sequence of sets B_e, C_e so that for the c.e. set W_e (using the standard enumeration of c.e. sets to represent all possible c.e. degrees $\boldsymbol{x}$) $\deg(B_e)$ and $\deg(C_e)$ allow $\deg(A)$ to satisfy the definition of center with $\deg(W_e)$.

Onward to embeddability. First let me introduce some simple lattices. All of these have a least and greatest element.

The *diamond* lattice is a four-element lattice with two incomparable intermediate elements. The *pentagon*, or N_5, has five elements; two of the intermediate elements are comparable and the other is not. The *1-3-1*, or M_3, is a five-element lattice with three incomparable intermediate elements. Finally, S_8 is a diamond on top of a 1-3-1, for eight total elements.



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, is distributive. In fact, the non-distributive lattices are exactly those that contain the pentagon and/or the 1-3-1 as a sublattice.

Let's consider embeddings that preserve the least and/or greatest elements separately.

- All finite distributive lattices embed into $\mathcal{R}$ preserving least element, as do the pentagon and 1-3-1, but not S_8.
  Open: where does the embeddable/nonembeddable cutoff lie?
- The same results as above hold preserving greatest element.
  Conjecture: a lattice embeds into $\mathcal{R}$ preserving greatest element if and only if it embeds preserving least element.
- We lose out when we try to embed preserving both least and greatest element. The *Lachlan non-diamond theorem* says even the diamond does not embed into $\mathcal{R}$ preserving both least and greatest element. 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 $\boldsymbol{\emptyset'}$.

  We meet with success if the lattice L to be embedded can be decomposed into two sublattices L_1, L_2 such that all elements of L_1 are above all elements of L_2, L_1 can be embedded preserving greatest element, and L_2 can be embedded preserving least element. In that case we can stitch together the embeddings of the sublattices to get an embedding of L that preserves both least and greatest element.

We can also consider embedding questions for *intervals* of $\mathcal{R}$, where the interval $[\boldsymbol{a}, \boldsymbol{b}]$ is $\{\boldsymbol{c} : \boldsymbol{a} \leq \boldsymbol{c} \leq \boldsymbol{b}\}$.

Now an open question about the Turing degrees as a whole. For a poset to be *locally countable* means the set of predecessors of any one element x (that is, the set of elements of the poset less than or equal to x) is countable.

Question 9.1.10 (Sacks)**.** *Suppose P is a locally countable partially ordered set of cardinality less than or equal to that of $\mathcal{P}(\mathbb{N})$. Is P embeddable into $\mathcal{D}$?*

This question is essentially asking "if a partially ordered set has no obvious blockades to embeddability, does it embed?" The Turing degrees are locally countable and of the same size as $\mathcal{P}(\mathbb{N})$, so anything that is not locally countable or is any bigger clearly cannot be embedded.

We can also discuss the lattice of c.e. *sets*, ordered by $\subseteq$. We call this $\mathcal{E}$. The top element is $\mathbb{N}$ and the bottom is $\emptyset$. Every pair of sets has a defined meet and join, given by intersection and union. The complemented sets are those c.e. sets whose

set-theoretic complement is also c.e.; in other words, the computable sets. Recall that the set-theoretic *difference* of two sets A and B is $A-B = \{x : x \in A \ \& \ x \notin B\}$.

Definition 9.1.11. Two sets A and B are *equivalent modulo finite difference*, denoted $A =^* B$, if the *symmetric difference* $(A - B) \cup (B - A)$ is finite.

Definition 9.1.12. A coinfinite c.e. set $W \neq \mathbb{N}$ is *maximal* in $\mathcal{E}$ if for any c.e. set Z such that $W \subset Z \subset \mathbb{N}$, either $W =^* Z$ or $Z =^* \mathbb{N}$.

That is, anything between a maximal set and $\mathbb{N}$ is either essentially the maximal set or essentially all of $\mathbb{N}$. The complement of a maximal set is called *cohesive.* Maximal sets do exist. They put an end to Post's program to find a 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 incomplete. A maximal set has the smallest possible complement from a c.e. set point of view, but not all maximal sets are incomplete.

One family of theorems particularly suited to $\mathcal{E}$ are *splitting theorems.* A *splitting* of a c.e. set B is a pair of disjoint c.e. sets that union to B. Two useful splitting theorems follow.

Theorem 9.1.13 (Friedberg Splitting)**.** *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 meets the hypothesis of the implication, it must have not only an infinite intersection 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 computable enumerability. 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 surprising and useful; the splitting is somehow very much down the "middle" of B.

A generalization of the Friedberg Splitting Theorem is the Owings Splitting Theorem. It is said of a very good computability theorist that he has made his career on clever uses of the Owings Splitting Theorem. It's not entirely true, of course, but he has several very nice theorems that rely heavily on innovative applications of splitting.

Theorem 9.1.14 (Owings Splitting)**.** *Let $C \subseteq B$ be c.e. such that $B - C$ is not co-c.e. (so if it is c.e., it is not computable). Then there is a splitting of B into c.e. sets A_0, A_1, such that*

(i) *$A_i - C$ is not co-c.e. for $i = 0, 1$, and*

(ii) *for all c.e.* W*, if* $C \cup (W - B)$ *is not c.e., then* $C \cup (W - A_i)$ *is not c.e. for* $i = 0, 1$.

As in Friedberg, setting $W = \mathbb{N}$ in (ii) gives (i). Setting $C = \emptyset$ gives the Friedberg Splitting Theorem.

Splitting questions in general ask "if B has a certain computability-theoretic property, can we split it into two sets which 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 9.1.15. 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 9.1.16 (Shore, 1978)**.** *Every c.e. set can be split into two nowhere simple sets.*

9.2 Randomness

Birds-Eye View

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 quantify that intuitive idea of randomness and briefly explore some of the consequences and applications.

We'll be discussing randomness for infinite binary sequences. There are three main approaches to randomness.

- Compression: is there a short description of the sequence?
- Betting: can you get unboundedly rich betting on the bits of the sequence?
- 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 order as above.

- 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, the sequence is random. (prefix-free TMs will be defined shortly.)

- 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; double-or-nothing means that for every string σ, the capital held at $\sigma 0$ and $\sigma 1$ must average to the capital held at σ). 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, the sequence is random.

- 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, it is random. (Measure will also be defined shortly.)

The nice thing about the implementations above is that they coincide [52]: 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 be prefix-free, but 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.

What does one do with this concept?

- Prove random sequences exist.

- Look at computability-theoretic properties of random sequences, considering them as characteristic functions.

- Compare different definitions of randomness.

- Consider *relative randomness*: if I know this sequence, does it help me bet on/compress this other sequence? (just like oracles for Turing machines)

- Look for sequences to which every random sequence is relatively random. Prove noncomputable examples exist.

- Extend the definition to other realms, like sets or functions instead of just sequences.

Significant references for randomness are the books by Downey and Hirschfeldt [14] (in preparation and available online), Nies [48], and Li and Vitányi [40]. For historical reading I suggest Ambos-Spies and Kučera [2], section 1.9 of Li and Vitányi [40], and Volchan [63].

Some Notation and Basics

Some of this is reminders and some is new.

$2^{\mathbb{N}}$ is the collection of all infinite binary strings (often referred to as *reals*) and $2^{<\mathbb{N}}$ the collection of all finite binary strings. If you read computer science papers, you may see $\{0,1\}^*$ for $2^{<\mathbb{N}}$. I will use λ for the empty string; it is also often called $\langle\rangle$. 1^n is the string of n 1s and likewise for 0, and if σ and τ are strings, $\sigma\tau$ (or $\sigma^\frown\tau$, when it seems clearer) is 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 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, notated $|n|$, is $\log n$ (all our logarithms have base 2). If we are working with a string σ, then $|\sigma|$ is simply the number of bits in σ.

A bit of topology: the *basic open sets* in $2^{\mathbb{N}}$ are *intervals* $[\sigma] = \{X : \sigma \subset X\}$, for any finite binary string σ. The *open sets* are countable unions of intervals. *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 all intervals generated by strings of a fixed length is 1, and the measure of the whole space is 1. The measure of the union of a pairwise-disjoint collection of intervals is the sum of the measure of the intervals; the notion of measure extends to all subsets of $2^{\mathbb{N}}$ in a straightforward manner.

Big O notation is shorthand to describe the asymptotic behavior of functions, or really their asymptotic upper bound. To say the function $f(n)$ is $\mathcal{O}(g(n))$ means there is some number n_0 and constant M such that $|f(n)| \leq M|g(n)|$ for all $n \geq n_0$. Big O notation can also be used as functions approach a finite value, but we will only use it as an asymptotic. The order of a function will correspond to its fastest-growing term; coefficients are disregarded as well as lower-order terms. For more see any computer algorithms textbook. We will primarily be concerned with $\mathcal{O}(1)$. If we write $f(x, y) \leq$ (something)+$\mathcal{O}(1)$, the constant is always independent of both x and y (all input variables).

Kolmogorov Complexity

Prefix-free machines and 1-randomness

Prefix-free Turing machines were suggested by Levin [38, 67] and later Chaitin [5] as the best way to approach the compressibility of strings.

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 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 eth prefix-free machine on input σ."[1] Call such a machine U. It is prefix-free because in order to have $\sigma \subset \tau$ we must have $\sigma = 1^e0\sigma'$ and $\tau = 1^e0\tau'$ 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 x is

$$K(x) = \min\{|p| : U(p) = x\}.$$

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 is going to 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 x is *random* if $K(x) \geq |x|$.

(ii) An infinite binary sequence X is 1-*random* if all of its initial segments are random, up to a constant. That is, for all n, $K(X \upharpoonright n) \geq n - \mathcal{O}(1)$.

[1] Of course this begs the question of whether the prefix-free machines can be enumerated. They can, by taking the enumeration of all Turing machines and modifying the machines which turn out to be non-prefix-free. We work stagewise, M the given machine and P 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, at some finite stage s^* M will halt on a string which is 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^* 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 (in terms of halting behavior, input, and output).

Here is an interesting theorem I stumbled across while researching finite injury priority arguments for a seminar talk. I include it because it's sort of magic and cool. 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 9.2.4 (Kolmogorov [30, 31]). *The set of nonrandom numbers is simple.*

Proof. The set we need to prove simplicity for is $A = \{x : K(x) < |x|\}$. It is c.e. because $x \in A$ if and only if $(\exists e < x)(U(e) = x)^2$, or to make it clearer $(\exists s)(\exists e < x)(U_s(e) \downarrow= x)$. This is a computable predicate preceded by a single existential quantifier, so it is Σ^0_1 and hence corresponds to a c.e. set. 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. There is a uniform description of the elements of W_e: $x_{e,n}$ is the nth element in the enumeration of W_e. Therefore by an application of the S-m-n Theorem, there is a one-to-one computable function h such that $U(h(e,n)) = x_{e,n}$. We use this to show every infinite c.e. set has an infinite subset such that h is a description of x_n, shorter than x_n, for some n. That element x_n will be nonrandom and in the original set, so the original set has a nonrandom element.

To that end, set $t(n) = \max_{e \leq n} h(e,n)$. The important point is that for any index e, $t(n)$ will take $h(e,n)$ into account on all but finitely-many values of n.

Given a c.e. set X, enumerate a subset Y so that the nth element of Y, y_n, is greater than $t(n)$. This is possible, and results in an infinite set Y, when X is infinite because $t(n)$ is some fixed value for each n, so X will contain infinitely many elements greater than it. Y will be c.e. because of that, and because t is computable.

Since Y is c.e., it is W_e for some e. However, by the choice of $y_n > t(n)$ and the fact that for almost all n, $t(n) \geq h(e,n)$, we know there is some n such that $x_{e,n} = y_n > t(n) \geq h(e,n)$. For that n, $h(e,n)$ gives a short description of $x_{e,n}$, so $x_{e,n}$ is a nonrandom element of $W_e = Y$ and hence of X. □

This uses *Berry's paradox* (Russell 1906): Given n, consider the least number indescribable by $< n$ characters. This gives a description of length $c + |n|$ for fixed c, which is paradoxical whenever $c + |n| \leq n$ (i.e., for almost every n).

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 length of

[2] We're fudging a little here since the complexity will be determined by the length of such e and the length of two nonequal numbers may be equal, but it's not too important and streamlines the argument substantially.

x? The following is part of a larger, more technical theorem, which I have trimmed in half.

Theorem 9.2.5 (Chaitin, [5]). *For every x of length n,*

$$K(x) \leq n + K(n) + \mathcal{O}(1). \tag{9.2.1}$$

Proof. To obtain (9.2.1), consider a prefix-free Turing machine T which computes $T(qx) = x$ for any q such that the universal prefix-free machine U gives $U(q) = |x|$. Since T is prefix-free it has an index m in the enumeration of all prefix-free machines, and hence $U(1^{|m|}0mqx) = x$. That description has length $2|m| + |q| + |x|$, or (if q is as short as possible) $|x| + K(|x|) + 2|m|$, where m does not depend on x, and its length certainly bounds the size of $K(x)$. □

This upper bound leads to recursive further bounds like

$$K(x) \leq n + |n| + ||n|| + |||n||| + \dots$$

Why do we not use this upper bound in our definition of randomness? Because there is no infinite string X such that for all n, $K(X \upharpoonright n) \geq n + K(n) - \mathcal{O}(1)$. We could kludge by saying "for infinitely-many n" instead of for all, but that's unsatisfying. 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 comes from the Kraft Inequality, a very useful tool in randomness. In a prefix-free set there are a lot of binary strings missing. Thus we would expect the length of these strings to grow rapidly. They do, as the theorem below shows. The proof is included because it's not so difficult.

Theorem 9.2.6 (Kraft Inequality, [34]). *Let $\ell_1, \ell_2, \dots$ be a finite or infinite sequence of natural numbers. There is a prefix-free set of binary strings with this sequence as its elements' lengths if and only if*

$$\sum_n 2^{-\ell_n} \leq 1.$$

Proof. First suppose there is a prefix-free set of finite binary strings x_1 with lengths ℓ_i. Consider

$$\mu\left(\bigcup_i [x_i]\right)$$

as a subset of $2^{\mathbb{N}}$. 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 there is a set of values $\ell_1, \ell_2, \dots$ such that the inequality holds. Because we are not working effectively, we may assume the set is nondecreasing. To

find a prefix-free set of binary strings which have those values as their lengths, start carving up the complete binary tree, taking the leftmost string of length ℓ_i which 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 $K(x)$ has to grow significantly faster than length (overall). The set of programs giving the strings x is the prefix-free set here, and if those programs have length approximately $|x|$, the sum $\sum_n 2^{-\ell_n}$ is essentially $2^{-1} + 2^{-1} + 2^{-2} + 2^{-2} + 2^{-2} + 2^{-2} + \ldots$, which diverges.

Notice that in the proof, we assumed the lengths we were given were in increasing order. Next we will see an effectivized version of Theorem 9.2.6 (Theorem 9.2.7), where we can be given the required lengths of strings in any order and still create a prefix-free set with those lengths. As written, 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 we get to length ℓ_n, and although there is at least $2^{-\ell_n}$ measure unused in the tree, it is not all in one piece. Having the ability to cope with strings given in no particular order is very useful.

Theorem 9.2.7 (Kraft-Chaitin [5, 6], also Levin [38])**.** *Let $\ell_1, \ell_2, \ldots$ be a collection of values (in no particular order, possibly with repeats, possibly finite) satisfying the Kraft Inequality; in other words, 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. The organization of this proof is as written in Downey and Hirschfeldt [14], where they say it was suggested by Joe Miller.

Assume that we have selected strings σ_i, $i \leq n$, such that $|\sigma_i| = \ell_i$. Suppose also that we have a string $x[n] = .x_1x_2\ldots x_m = 1 - \sum_{j \leq n} 2^{-\ell_j}$, 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 $\sigma_j, j \leq n$ and all $\tau_j, j < k$ and $x_j = 1$.

Notice 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 measure 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 $x_j = 1$ and the leftmost string τ of length ℓ_{n+1} extending τ_j, and let $\sigma_{n+1} = \tau$. Let $x[n+1] = x[n] - .0^{\ell_{n+1}-1}1$. 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 9.2.8 (Kraft-Chaitin, 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 recursively build a prefix-free machine M and a collection of strings (descriptions) τ_k such that $|\tau_k| = n_k$ and $M(\tau_k) = \sigma_k$.*

Kraft-Chaitin allows us to implicitly build machines by computably enumerating "axioms" $\langle n_k, \sigma_k \rangle$ and arguing that the set $\{n_k\}_{k \in \mathbb{N}}$ satisfies the Kraft Inequality. The machine houses the construction, from which the axioms are enumerated, and on input τ enumerates them while performing Kraft-Chaitin 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.

So why prefix-free?

We could define the complexity of x as the minimum length input that produces x when given to the standard universal Turing machine, rather than the universal prefix-free Turing machine. That is studied; we call it the *plain Kolmogorov complexity* of x and denote it $C(x)$. However, as a standard for complexity it has some problems.

To say an infinite string is random if and only if all its initial segments are random sounds right, but without the restriction to prefix-free machines it is an empty definition: no such string exists. In fact, this is the reason there is no infinite string X such that for all n, $K(X \upharpoonright n) \geq n + K(n) - \mathcal{O}(1)$. The proof is from Martin-Löf [43] and may be found in Li and Vitányi [40], §2.5.

Even at the level of finite strings plain Kolmogorov complexity has some undesirable properties. The first is non-subadditivity. That is, for any constant c you like there are x and y such that the plain complexity of the coded pair $\langle x, y \rangle$ is more than $C(x)+C(y)+c$. K, on the other hand, *is* subadditive, because with K we can concatenate the descriptions p and q of x and y, respectively, and the machine will be able to tell them apart: the machine can read until it halts, assume that is the end of p, and then read again until it halts to obtain q. 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 undesired property of C 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

$C(1^n) \leq \log\log n + \mathcal{O}(1)$ (i.e., a description of k, which is no more than $\log k$ in size, plus some machinery to print 1s). However, once k is big enough, there will be numbers smaller than n which have much higher complexity because they have no nice concise description in terms of, say, powers of smaller numbers. For such a number m, the plain complexity of 1^m would be higher than that of 1^n even though 1^m is a proper initial segment of 1^n.

What is the underlying problem? The $C(x)$ measure contains information about the length of x (that is, n) as well as the pattern of bits. For most n, about $\log n$ of the bits of the shortest description of x will be used to determine n. What that means is that for simple strings of the same length n, where by "simple" I mean each having plain Kolmogorov complexity less than $\log n$, any distinction between the information content 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 problems as well as philosophical ones.

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.

Relative Randomness

Next we would like to be able to compare sets to each other, in a finer-grained way than saying both, one, or neither is n-random for some n. 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.

The generalization is very simple: add an oracle to the prefix-free Turing machines.

Definition 9.2.9. (i) The *prefix-free Kolmogorov complexity of x relative to A* is $K^A(x) = \min\{|p| : U^A(p) = x\}$.

(ii) A set or sequence B is *A-random* (or 1-A-random) if

$$(\forall n)[K^A(B \upharpoonright n) \geq n - \mathcal{O}(1)].$$

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.

Two extremely useful theorems about relative randomness rely on the join of sequences. We've seen this definition before, but to remind you:

Definition 9.2.10. The *join* of two sets (or sequences) A and B is their disjoint union

$$A \oplus B = \{2n : n \in A\} \cup \{2n+1 : n \in B\}.$$

Its Turing degree is the least upper bound of the degrees of A and B, so $A \oplus B$ corresponds to join in the lattice of Turing degrees.

Theorem 9.2.11 (van Lambalgen [60])**.** *If $A \oplus B$ is 1-random, then B is 1-A-random (and hence 1-random).*

Note that likewise A will be 1-B-random in the theorem above. There is a converse to this, though it is slightly stronger: instead of needing A to be 1-B-random *and* B to be 1-A-random, we only need one of those conditions plus 1-randomness for the other set.

Theorem 9.2.12 (van Lambalgen [61])**.** *If A is 1-random and B is 1-A-random, then $A \oplus B$ is 1-random.*

The two theorems together show that if A and B are 1-random and one is 1-random relative to the other, the other is 1-random relative to the first. In fact, they are only special cases of a much more general pair of theorems that we don't have the vocabulary to state.

Lowness and K-Triviality

The idea of lowness for randomness comes from several perspectives. First, there is the observation that taking 1-randomness relative to A may only decrease the set of random reals. 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, $\mathrm{RAND}^A \subseteq \mathrm{RAND}$. 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 an empty one.

Definition 9.2.13. A set A is *low for random* if it is noncomputable and $\mathrm{RAND}^A = \mathrm{RAND}$; that is, any real which is 1-random is still random relative to A.

The term "low" is by analogy with ordinary computability theory, where A is low if the halting problem relativized to A has the same Turing degree as the non-relativized halting problem.

We think of a low set as being "nearly computable". It clearly cannot itself be random, else in derandomizing itself and its infinite subsequences it would change the set of randoms. Therefore, the existence of low for random sets gives a middle ground between computable and random.

Theorem 9.2.14 (Kučera and Terwijn [36]). *There exists a noncomputable A such that* $\mathrm{RAND}^A = \mathrm{RAND}$.

There is a different aspect of lowness we could consider; this approaches the "middle ground" between computability and randomness in a different way, directly tackling the question of initial segment complexity.

Definition 9.2.15. A real α is *K-trivial* if the prefix-free complexity of its length-n initial segments is bounded by the complexity of n; that is, for all n, $K(\alpha \upharpoonright n) \leq K(n) + \mathcal{O}(1)$.

The question is, again, whether there are any noncomputable K-trivial reals. Certainly all computable reals α are such that $K(\alpha \upharpoonright n) \leq K(n) + \mathcal{O}(1)$; the $\mathcal{O}(1)$ term holds the function which generates the initial segments of α, and then getting an initial segment is as simple as specifying the length you want.

Theorem 9.2.16 (Zambella [66], after Solovay [57]). *There is a noncomputable c.e. set A such that $K(A \upharpoonright n) \leq K(n) + \mathcal{O}(1)$.*

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 really, really hard, involving work by Gács [21], Hirschfeldt, Nies, and Stephan [26], Kučera [35], and Nies [47].

There are some theorems we won't prove here about the degree properties of K-trivials; for proofs see Downey and Hirschfeldt [14] or the paper cited. Recall that a set $A \leq_T \emptyset'$ is *high* if $A' \equiv_T \emptyset''$.

Theorem 9.2.17 (Chaitin [5]). *If A is K-trivial then it is Δ^0_2; that is, $A \leq_T \emptyset'$.*

Theorem 9.2.18 (Downey, Hirschfeldt, Nies, Stephan [16]). *If A is K-trivial, then A is Turing incomplete, and in fact not even high.*

Theorem 9.2.19 ([16]). *If reals α and β are K-trivial, then so is their sum $\alpha + \beta$.*

Note that in the above we mean simply the arithmetic sum, not the join. The next theorem shows that the K-trivials hang together in a strong sense. An *ideal* of the Turing degrees is a collection of degrees which is closed downward under Turing reducibility and upward under join. Therefore the proof must show both that if $\alpha \leq_T \beta$ and β is K-trivial, α is K-trivial, and that if α and β are K-trivial, $\alpha \oplus \beta$ is K-trivial.

Theorem 9.2.20 (Nies). *The K-trivial reals form a Σ^0_3-definable ideal in the Turing degrees.*

This is touted as the only natural example of a nontrivial ideal in the Turing degrees.

9.3 Some Model Theory

Both computable model theory (§9.4) and reverse mathematics (§9.5) use some model theory, which is an entire other area of mathematical logic. Propositional and predicate logic, which undergraduates are often exposed to, tend to include elements of what would be categorized as model theory.

Suppose we have a collection of relation symbols, such as $(=,<)$; call it a *language* $\mathcal{L}$. In this example, the relations are both binary. 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. Generally we call the collection of logical sentences (recall sentences are formulas with no free variables, so that they have a truth value) we're treating as axioms a *theory*. These sentences may use all the standard logical symbols (and, not, quantifiers, etc) as well as variables and any symbol from the language. A structure for the language is a model of the theory if, interpreting the language as the structure specifies and letting the domain of quantification be 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*).

Properly speaking, languages can have not only relation symbols, but also function symbols and symbols for distinguished constant values; the arity of the relations and functions must be specified. 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 tuples $\overline{a}$ of the appropriate arity, if $F(a_i) = b_i$ and $F(k) = \ell$, then $f^{\mathcal{A}}(\overline{a}) = k \leftrightarrow f^{\mathcal{B}}(\overline{b}) = \ell$ and $R^{\mathcal{A}}(\overline{a}) \leftrightarrow R^{\mathcal{B}}(\overline{b})$. For languages with more than one constant, function, or relation, this definition extends as expected. When the 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 which have countable universes.

(i) $\mathbb{N}$, with the usual meanings for all these symbols;

(ii) $\mathbb{Q}$, with the usual meanings for all 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 we do not have to abide by the conventional uses of the symbols.[3] In fact we could have gone further afield and decided that, say, $=$ would be the relation that holds of (n, m) exactly when n is even, and $<$ the relation that holds when $n + m$ is a multiple of 42.

Let us consider some axioms on $\mathcal{L}$.

(I) $\neg(0 = 1)$

(II) $\forall x, y(x < y + 1 \rightarrow (x < y \ \vee \ x = y))$

(III) $\forall x(\neg(x < 0))$

(IV) $\forall x, y \exists z((\neg(y = 0) \ \& \ \neg(x = 0)) \rightarrow x \cdot z = y)$

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 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 theory containing axioms I, II, and III. If we call the model $\mathcal{M}$ and the theory $\mathcal{T}$, we notate this as $\mathcal{M} \models \mathcal{T}$. However, structure (i) is not a model for the last axiom; call it φ: $\mathcal{M} \not\models \varphi$. Note that for all sentences φ and models $\mathcal{M}$ over the same language, either $\mathcal{M} \models \varphi$ or $\mathcal{M} \models \neg\varphi$. However, φ may be *independent* of the theory $\mathcal{T}$, where $\mathcal{T} \not\vdash \varphi$ and $\mathcal{T} \not\vdash \neg\varphi$.

This discussion has implied we can go from structures to theories, and indeed we can. Given a theory $\mathcal{T}$ we can talk about models of that theory, $\mathcal{M} \models \mathcal{T}$, but given a structure $\mathcal{M}$, we can speak of the *theory of that structure*, $\mathrm{Th}(\mathcal{M}) := \{\varphi : \mathcal{M} \models \varphi\}$.

Since a theory is just a collection of sentences, it has consequences under logical deduction.[4] If φ is a logical consequence of the sentences in $\mathcal{T}$, we denote

[3]However, it is common, perhaps even standard, to make $=$ a special symbol that may only be interpreted as genuine equality.

[4]Some authors require theories to be closed under logical deduction.

that by $\mathcal{T} \vdash \varphi$. If for every $\mathcal{L}$-sentence φ, either $\mathcal{T} \vdash \varphi$ or $\mathcal{T} \vdash \neg\varphi$, $\mathcal{T}$ is called *complete*. For any structure $\mathcal{M}$, Th($\mathcal{M}$) is complete. I should note here that we assume our theories are *consistent* (i.e., they do not prove any contradictions); otherwise the deductive closure of the theory is literally all sentences in the language. *Gödel's completeness theorem* says that $\mathcal{T} \vdash \varphi$ if and only if for all structures $\mathcal{M}$, $\mathcal{M} \models \mathcal{T} \rightarrow \mathcal{M} \models \varphi$. What this means is that if you want to show φ follows from a $\mathcal{T}$, you must give a logical deduction, but if you want to show $\mathcal{T} \nvdash \varphi$ (which is *not* the same as $\mathcal{T} \vdash \neg\varphi$), you need only construct a model of $\mathcal{T}$ in which φ is false.

Thus far we have only spoken of *first-order* theories, ones where only one kind of object is quantified over. 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 universe M, a subset of $\mathcal{P}(M)$, and interpretations of all 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 in forthcoming sections.

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 Ash and Knight's *Computable Structures and the Hyperarithmetical Hierarchy* [3]; Volume 1 of the *Handbook of Recursive Mathematics* [20] and a survey article by Harizanov [24] are also good references. Note that in this section, all structures will be countable; in fact one frequently assumes 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:

- 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 important theory in logic is *Peano arithmetic* (PA), a theory in the language $(S, +, \cdot, 0)$, where S is a unary function, $+$ and $\cdot$ are binary functions, and 0 is a constant. The axioms of PA abstract the essence of grade-school arithmetic. That

addition and multiplication act as we expect follows from the axioms, which more explicitly fall into the following groups:

- 0 is zero: $(\forall x)(S(x) \neq 0)$; $(\forall x)(x + 0 = x)$; $(\forall x)(x \cdot 0 = 0)$.
- S means "plus one": $(\forall x)(\forall y)(S(x) = S(y) \ \rightarrow \ x = y)$; $(\forall x)(x+S(y) = S(x+y))$; $(\forall x)(x \cdot S(y) = x \cdot y + x)$.
- Induction works: for every formula $\varphi(\overline{u}, v)$ in the language of PA, we have the axiom

$$(\forall \overline{u})\left(\varphi(\overline{u}, 0) \ \& \ (\forall y)\left[\varphi(\overline{u}, y) \ \rightarrow \ \varphi(\overline{u}, S(y))\right] \ \rightarrow \ (\forall x)\varphi(\overline{u}, x)\right).$$

The standard model of PA is denoted $\mathcal{N}$, the natural numbers with successor and the usual addition, multiplication, and zero.

In fact a "standard model" of PA is any model where the universe is generated by closing 0 under successor, and those elements are also called *standard*. Nothing forbids having elements in the universe that are not obtained in that way. Those elements and the models that contain them are called *nonstandard*; nonstandard models also contain standard elements.

The induction axiom in PA leads directly to the following useful tool, since the antecedent need refer only to the successors of zero, but the consequent has an unrestricted $\forall x$.

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 [58])**.** *If $\mathcal{M} \models PA$ is nonstandard, it is not computable.*

Proof. Let X and Y be computably inseparable c.e. sets (see Exercise 5.2.17). There are natural formulas that mean $x \in X_s$, $y \in Y_s$, as well as $p_n | u$ (the nth prime divides u). Let $\psi(x, u)$ say

$$\forall y([(\exists s \leq x \ (y \in X_s)) \rightarrow p_y | u] \ \& \ [(\exists s \leq x \ (y \in Y_s)) \rightarrow p_y \nmid u]).$$

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 such that 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, $\{0, 1, 2, \ldots\}$ with finitely-many named elements and unary relations that are all either empty or the entire universe. Any permutation of $\{0, 1, 2, \ldots\}$ that preserves the named elements will also preserve the relations, and hence be an automorphism.

Theorem 9.4.4 (Solovay, Marker, Knight [33])**.** *Suppose $\mathcal{A}$ is a nontrivial structure. If $\mathcal{A} \leq X$, 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 $\mathcal{B}$. In particular, if $\mathcal{A}$ is a linear order, we may enumerate the universes A and B as $\{a_0, a_1, \ldots\}$ and $\{b_0, b_1, \ldots\}$. F maps from A to B so that if $a_{2n} <^{\mathcal{A}} a_{2n+1}$, then $F(a_{2n}) = b_{2n}$ and $F(a_{2n+1}) = b_{2n+1}$ if and only if $n \in X$. Otherwise F swaps the order; if $a_{2n+1} <^{\mathcal{A}} a_{2n}$ the opposite happens, so b_{2n+1} is on top iff $n \in X$. Then $\mathcal{B}$ interprets $<$ in the necessary way to make F an isomorphism from $\mathcal{A}$ to $\mathcal{B}$.

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.

Linear orders are a useful example for a number of our questions.

Theorem 9.4.6 (Miller [46])**.** *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 do exist noncomputable Turing degrees that are not above any noncomputable Δ^0_2 degree.

A *computably categorical* structure is $\mathcal{A}$ such that $\mathcal{A}$ is computable and 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 ordering that cannot be computably isomorphic. The discrete linear order with one endpoint is not computably categorical.

Dense linear orders without endpoints, 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: 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 where you select your element makes sure the map is onto in each direction; using only unused elements makes sure it is one-to-one. Density and lack of endpoints guarantee finding a match.

Exercise 9.4.7. Complete the proof sketch above and show it can be done computably whenever the DLOs $\mathcal{A}$ and $\mathcal{B}$ are computable, 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 [22, 23]). 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 [17]). 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 may have noticed by now that computability and the related areas we've discussed involve a lot of hierarchies: Turing degrees, the arithmetic hierarchy, 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. You might be familiar with the first one, which states that for any sufficiently sophisticated theory there are true but unprovable statements. As an aside, the sophistication is in being able to self-reference, to make a formula which almost literally says "I am not provable"; there are proofs which do not make use of self-reference, but the systems must be capable of it. The second one, however, is more relevant here, and roughly says if a theory is consistent, it cannot prove its 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.[5]

[5]We generally speak as though our theories are closed under deduction, though we specify them as a non-closed collection of axioms; since any formula that follows from the theory must be true in all models of the theory, this practice does not cause any harm.

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 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.

The main reference for reverse mathematics is Steve Simpson's book *Subsystems of Second-Order Arithmetic* [54]; this section is drawn from that book, conference talks by Simpson, and a course I had on reverse mathematics from Jeff Hirst at Notre Dame. 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.

Second-Order Arithmetic

We work in the world of *second-order arithmetic*, or Z_2. The language of Z_2 is different from the example in §9.3 in that it 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. A model of Z_2 will specify not only the universe M of the number variables, as explained in §9.3, but also the universe $\mathcal{S}$ for the set variables, where $\mathcal{S} \subseteq \mathcal{P}(M)$.

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: $(\exists X)(\forall n)(n \in X \leftrightarrow \varphi(n))$ for each formula $\varphi(n)$ in which X does not occur freely.

The last two axioms together say that for any second-order formula $\varphi(n)$,

$$(\varphi(0) \mathbin{\&} (\forall n)(\varphi(n) \to \varphi(n+1))) \to (\forall n)\varphi(n). \tag{9.5.1}$$

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 decide what restrictions to set? For comprehension, we might limit φ to certain levels of complexity, such as being recursive (Δ^0_1) or arithmetic (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 (9.5.1) holds.

Recursive Comprehension

The base system of reverse mathematics is called RCA_0, which stands for *recursive comprehension axiom*. RCA_0 contains all 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 all computable sets exist, though it is slightly more precise than that.[6] RCA_0 also has Σ^0_1 induction, which is as (9.5.1) 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.[7]

Essentially, 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 roughly says 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), that if f is continuous and $f(0) < 0 < f(1)$, then for

[6] Formally it is

$$\forall n(\varphi(n) \leftrightarrow \psi(n)) \to \exists X \forall n(n \in X \leftrightarrow \varphi(n))$$

for Σ^0_1 φ and Π^0_1 ψ, which could be read as "*demonstrably* computable (Δ^0_1) sets exist."

[7] Though that system has been studied, under the name RCA^*_0.

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 ever $x \in X$ 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 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 restriction for proving results. 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. We will not address them here, except to say the universes of such models start with $\mathbb{N}$ but include elements that are larger than every element of $\mathbb{N}$. Those "infinite" elements can act in ways counter to the intuition we have developed from $\mathbb{N}$.

Any theorem that might require a noncomputable set or function pops us out of RCA_0. 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 even for two computable algebraic closures the isomorphism between them 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).[8] 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 (walk 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

[8] Jeff Hirst states this lemma as "big skinny trees are tall."

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 path relativizes to say that for any set A, not every A-computable tree has an A-computable path.

There are, of course, many other theorems RCA_0 cannot prove, but we will discuss those in the subsequent sections.

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 in a way that uniformly addresses the 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.[9]

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 which 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 models of WKL_0 and Peano Arithmetic (PA; see §9.4). Formally, a degree $\boldsymbol{d}$ is the degree of a nonstandard model of PA iff there is a model

[9]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^*.

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 "arithmetic" to ACA_0.

The study of computable trees gives us a result called the *low basis theorem*, which says any computable tree has a path of low degree (where A is *low* if $A' \equiv \emptyset'$, and it is significant that noncomputable low sets exist). This and a little extra computability theory shows WKL_0 has a model with only low sets.

Arithmetic Comprehension

ACA_0 stands for *arithmetic comprehension axiom.* As mentioned, we obtain it from Z_2 by restricting the formulas φ in the comprehension scheme to those which may be written using number quantifiers only, no set quantifiers. Surprisingly, 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 easy, as well: 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 7.2.10 are the sets that are $\Sigma^{0,X}_2$. Continuing this process we bootstrap our way all the way up the arithmetic 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 (we may also restrict to the scalar field being $\mathbb{Q}$ and still get equivalence); 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 which are Turing reducible to $\emptyset^{(n)}$ for some n.

Arithmetic Transfinite Recursion

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), so we will skip it and discuss some of the equivalent theorems.

The main one 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.

Another comprehension theorem which is 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.

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 its models of universe $\mathbb{N}$ is a natural class of sets. In this case it is HYP, the *hyperarithmetic* sets, which we will not define.

Π^1_1 Comprehension

Π^1_1-CA_0 stands for Π^1_1 *comprehension axiom*. It is in the same family as RCA_0 and ACA_0, where the comprehension scheme has been restricted by capping the allowed 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.

We'll mention only a few results equivalent to Π^1_1-CA_0, most strengthenings or generalizations of theorems equivalent to ATR_0. It is equivalent to Π^1_1-CA_0 that every tree is the union of a perfect subtree and a countable set of paths.

Two comprehension theorems equivalent to Π^1_1-CA_0 are that (a) for any sequence of trees $\{T_i : i \in \mathbb{N}\}$, the set $\{i : T_i \text{ has a path}\}$ exists, and (b) for any uncountable tree the *perfect kernel* of the tree exists (that is, the union of all its perfect subtrees).

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 acronym definitions I could draw a very large picture with one-way arrows, two-way arrows, unknown implications, and non-implications, but we will keep to a manageable list. For this section the primary references are papers by Cholak, Jockusch, and Slaman [7] and Hirschfeldt and Shore [27].

One of the main focuses of this area of research is *Ramsey's theorem*. The general statement is the following, where $[\mathbb{N}]^n$ is the set of all n-element subsets of $\mathbb{N}$.

Theorem 9.5.1 (Ramsey's theorem)**.** *Given $f : [\mathbb{N}]^n \to \{0, \ldots, m-1\}$, there is an infinite set $H \subseteq \mathbb{N}$ such that the restricted map $f \upharpoonright H : [H]^n \to \{0, \ldots, m-1\}$ is constant. H is called a* homogeneous set *for f.*

Ramsey's theorem is a generalization of the pigeonhole principle, which is the case $n = 1$. The pigeonhole principle says if we put infinitely-many objects in finitely-many boxes, some individual box must contain infinitely-many objects.

We usually think of the range of f as consisting of *colors* and call f an *m-coloring of (unordered) n-tuples from* $\mathbb{N}$. For f a 2-coloring of pairs, we may picture $[\mathbb{N}]^2$ as a graph with vertices the natural numbers and an undirected edge between every pair of distinct vertices. The map f colors each edge red or blue, and Ramsey's theorem asserts the existence of a subset of vertices such that the induced subgraph on those vertices (the one that contains all edges from the original graph which still have both their endpoints in the subgraph) will have all edges the same color.

For our use it matters what the values of the parameters are, so we use RT^n_m to denote Ramsey's theorem restricted to functions $f : [\mathbb{N}]^n \to \{0, \ldots, m-1\}$ for specified n and m. It is the size of the subsets that matters; we can argue the number of colors can be restricted to 2 without loss of generality.

Exercise 9.5.2. Fix n. Show that by repeated applications of RT^n_2 one can obtain a homogeneous set for the coloring $f : [\mathbb{N}]^n \to \{0, \ldots, m-1\}$. Since it is clear RT^n_m implies RT^n_2 for each $m \geq 2$, this shows they are actually equivalent.

RT^n_m for fixed $n \geq 3$, $m \geq 2$ is equivalent to ACA_0, and the universal quantification $(\forall m)\,\mathrm{RT}^n_m$ is equivalent to ACA_0 for $n \geq 3$. If we quantify over both parameters we get the principle RT, which is strictly between ACA_0 and ATR_0. On the other side, RT^2_2 is strictly weaker than ACA_0, and is not implied by WKL_0. It is an open problem whether RT^2_2 implies WKL_0 or whether they are independent.

A principle which is strictly below both WKL_0 and RT^2_2 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.

CAC, or *chain-antichain*, says that every infinite partial order $(P, \leq_P)$ has an infinite subset that is either a chain (all elements are comparable by $\leq_P$; i.e., a subset that is a linear order) or an antichain (no elements are comparable by $\leq_P$). As an example, the partial order $(\mathcal{P}(\mathbb{N}), \subseteq)$ contains the infinite chain $\{\{0, 1, \ldots n\} : n \in \mathbb{N}\}$ and the infinite antichain $\{\{n\} : n \in \mathbb{N}\}$.

ADS, *ascending or descending sequence*, is implied by CAC; it is open whether they are equivalent or ADS is strictly weaker. ADS says that every infinite linear order $(L, \leq_L)$ has an infinite subset S that is either an ascending sequence $((\forall s, t \in S)(s < t \to s <_L t))$ or a descending sequence $((\forall s, t \in S)(s < t \to t <_L s))$. For example, if L is $(\mathbb{Z}, \leq_{\mathbb{Z}})$ coded by $n \to 2n$ for $n \geq 0$ and $n \to -2n-1$ for $n < 0$, the positive integers form an ascending sequence (their coded ordering $<$

matches their interpreted ordering $\leq_{\mathbb{Z}}$) and the negative integers form a descending sequence (their coded ordering is opposite from their interpreted ordering).

Neither CAC nor ADS implies DNR and neither is implied by WKL_0. Both are implied by RT^2_2. For some justification as to why CAC and ADS should be stronger than RCA_0, see Exercise 6.2.4. Even if the order relation must be computable (as in REC), things we define from the order relation need not be.

Finally, WWKL_0, or *weak weak König's lemma*, is a system intermediate between WKL_0 and DNR. 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 is also a model of WWKL_0 if and only if for every X in $\mathcal{M}$, there is some Y in $\mathcal{M}$ such that Y is 1-random relative to X [1].

Appendix A

Mathematical Asides

In this appendix I've stuck 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 very 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 get

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

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 nth 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

$$\sum_{i=0}^{\infty} ar^i = \lim_{n\to\infty} \frac{a - ar^{n+1}}{1-r} = \frac{a}{1-r} \lim_{n\to\infty} (1 - r^{n+1}),$$

and that limit is 1 whenever $|r| < 1$.

A.3 Cantor's Cardinality Proofs

Cantor had two beautifully simple diagonal proofs to show 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.

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, and 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 demonstrating the existence of an isomorphism to $\mathbb{N}$ (which is exactly what counting with the natural numbers accomplishes) is generally more straightforward than demonstrating no such isomorphism 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 an isomorphism between $[0, 1]$ and $\mathbb{N}$. List the elements of $[0, 1]$ out in the order given by the isomorphism, as infinite repeating decimals (using all-0 tails if needed):

$$
\begin{array}{l}
.65479362895\ldots \\
.00032797584\ldots \\
.35271900000\ldots \\
.00000000063\ldots \\
.98989898989\ldots \\
\vdots
\end{array}
$$

Now construct a new number $d \in [0, 1]$ decimal by decimal using the numbers on the list. If the nth decimal place of the nth number on the list is k, then the nth 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 on the list in at least one decimal place – the nth.

Bibliography

[1] Ambos-Spies K., B. Kjos-Hanssen, S. Lempp, and T.A. Slaman. Comparing DNR and WWKL. *Journal of Symbolic Logic* **69**:1089–1104, 2004.

[2] Ambos-Spies, K., and A. Kučera. Randomness in computability theory. In *Computability Theory and Its Applications: Current Trends and Open Problems* (ed. Cholak, Lempp, Lerman, Shore), vol. 257 of *Contemporary Mathematics*, pages 1–14. American Mathematical Society, 2000.

[3] Ash, C.J., and J.F. Knight. *Computable Structures and the Hyperarithmetical Hierarchy.* Elsevier Science B.V., 2000.

[4] Boolos, G.S., J.P. Burgess, and R.C. Jeffrey. *Computability and Logic*, fourth edition. Cambridge University Press, 2002.

[5] Chaitin, G.J. Information-theoretical characterizations of recursive infinite strings. *Theoretical Computer Science* **2**:45–48, 1976.

[6] Chaitin, G.J. Incompleteness theorems for random reals. *Advances in Applied Mathematics* **8**:119–146, 1987.

[7] Cholak, P.A., C.J. Jockusch, and T.A. Slaman. On the strength of Ramsey's theorem for pairs. *Journal of Symbolic Logic* **66**: 1–55, 2001.

[8] Church, A. An unsolvable problem of elementary number theory. *Journal of Symbolic Logic* **1**:73–74 (1936).

[9] Church, A. On the concept of a random sequence. *Bulletin of the American Mathematical Society* **46**:130–135, 1940.

[10] Cutland, N. *Computability: An introduction to recursive function theory.* Cambridge University Press, 1980.

[11] Davis, M. *Computability and Unsolvability.* McGraw-Hill Education, 1958. Reprinted by Dover Publications, 1985.

[12] Davis, M. *The Undecidable.* Raven Press, 1965.

[13] Davis, M. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly* **80**:233Ð269, 1973.

[14] Downey, R., and D. Hirschfeldt, *Algorithmic Randomness and Complexity*, in preparation.

[15] Downey, R., E. Griffiths, and G. LaForte. On Schnorr and computable randomness, martingales, and machines. *Mathematical Logic Quarterly* **50**(6):613–627, 2004.

[16] Downey, R., D. Hirschfeldt, A. Nies, and F. Stephan. Trivial reals, extended abstract. In *Computability and Complexity in Analysis* Malaga (Electronic Notes in Theoretical Computer Science, and proceedings; edited by Brattka, Schröder, Weihrauch, Fern Universität; 294-6/2002, 37-55), July 2002.

[17] Dzgoev, V.D., and S.S. Goncharov. Autostable models (English translation). *Algebra and Logic* **19**:28–37, 1980.

[18] Ehrenfeucht, A., J. Karhumaki, and G. Rozenberg. The (generalized) Post correspondence problem with lists consisting of two words is decidable. *Theoretical Computer Science* **21**(2), 1982.

[19] Enderton, H.B. *A Mathematical Introduction to Logic*, second edition. Harcourt/Academic Press, 2001.

[20] Ershov, Yu.L., 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 **138**, North-Holland, 1998.

[21] Gács, P. Every set is reducible to a random one. *Information and Control* **70**:186–192, 1986.

[22] Goncharov, S.S. The quantity of non-autoequivalent constructivizations (English translation). *Algebra and Logic* **16**:169–185, 1977.

[23] Goncharov, S.S. The problem of the number of non-autoequivalent constructivizations (English translation). *Algebra and Logic* **19**:401–414, 1980.

[24] Harizanov, V. Computably-theoretic complexity of countable structures *Bulletin of Symbolic Logic* **8**:457–477, 2002.

[25] Hilbert, D. Mathematical problems, *Bulletin of the American Mathematical Society* **8**(1901–1902):437–479.

[26] Hirschfeldt, D., A. Nies, and F. Stephan. Using random sets as oracles. Submitted.

[27] Hirschfeldt, D., and R.A. Shore. Combinatorial principles weaker than Ramsey's theorem for pairs. *Journal of Symbolic Logic* **72**:171–206, 2007.

[28] Kogge, P.M. *The Architecture of Symbolic Computers.* The McGraw-Hill Companies, Inc., 1998.

[29] Kolmogorov, A.N. *Grundbegriffe der Wahrscheinlichkeitsrechnung.* Springer, 1933.

[30] Kolmogorov, A.N. On tables of random numbers. *Sankhyā*, Series A, **25**:369–376, 1963.

[31] Kolmogorov, A.N., Three approaches to the quantitative definition of information. *Problems of Information Transmission (Problemy Peredachi Informatsii)* **1**:1–7, 1965.

[32] Kleene, S.C. General recursive functions of natural numbers. *Mathematische Annalen*, **112**:727–742, 1936.

[33] Knight, J.F. Degrees coded in jumps of orderings. *Journal of Symbolic Logic* **51**:1034–1042, 1986.

[34] Kraft, L.G. *A Device for Quantizing, Grouping, and Coding Amplitude Modulated Pulses.* Electrical engineering M.S. thesis. MIT, Cambridge, MA, 1949.

[35] Kučera, A. Measure, Π^0_1 classes, and complete extensions of PA. In *Springer Lecture Notes in Mathematics* Vol. 1141, pages 245–259. Springer-Verlag, 1985.

[36] Kučera, A., and S. Terwijn. Lowness for the class of random sets. *Journal of Symbolic Logic* **64**(4):1396–1402, 1999.

[37] Levin, L.A. On the notion of a random sequence. *Soviet Mathematics Doklady* **14**:1413–1416, 1973.

[38] Levin, L.A. Laws of information conservation (non-growth) and aspects of the foundation of probability theory. *Problems of Information Transmission* **10**:206–210, 1974.

[39] Lévy, P. *Théorie de l'Addition des Variables Aleatoires.* Gauthier-Villars, 1937 (second edition 1954).

[40] Li, M., and P. Vitányi, *An Introduction to Kolmogorov Complexity and its Applications*, second edition. Springer Graduate Texts in Computer Science, Springer Science+Business Media, New York, NY 1997.

[41] Linz, P. *An Introduction to Formal Languages and Automata*, second edition. Jones and Bartlett Publishers, 1997.

[42] Martin-Löf, P. The definition of random sequences. *Information and Control* **9**:602–619, 1966.

[43] Martin-Löf, P. Complexity oscillations in infinite binary sequences. *Z. Wahrscheinlichkeittheorie verw. Gebiete* **19**:225–230, 1971.

[44] Matijasevič, Yu.V. On recursive unsolvability of Hilbert's tenth problem. *Logic, methodology and philosophy of science, IV (Proc. Fourth Internat. Congr., Bucharest, 1971).* Studies in Logic and Foundations of Math. **74**: 89–110. North-Holland, 1973.

[45] Matijasevič, Y., and G. Senizergues. Decision problems for semi-Thue systems with few rules. *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, 1996.

[46] Miller, R. The Δ^0_2-spectrum of a linear order. *Journal of Symbolic Logic* **66**:470–486, 2001.

[47] Nies, A. Lowness properties and randomness. *Advances in Mathematics* **197**(1):274–305, 2005.

[48] Nies, A. *Computability and Randomness.* Oxford Logic Guides, 51. Oxford University Press, Oxford, 2009.

[49] Odifreddi, P.G. *Classical Recursion Theory. Studies in Logic and the Foundations of Mathematics* **125**, Elsevier, 1989, 1992.

[50] Post, E.L. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society* **52**, 1946.

[51] Rogers, H. *Theory of Recursive Functions and Effective Computability.* The MIT Press, 1987.

[52] Schnorr, C. P. A unified approach to the definition of random sequences. *Mathematical Systems Theory* **5**:246–258, 1971.

[53] Shafer, G. A counterexample to Richard von Mises' theory of collectives. Translation with introduction of an extract from Ville's *Étude Critique de la Notion de Collectif* [62], available from http://www.probabilityandfinance.com.

[54] Simpson, S.G. *Subsystems of Second-Order Arithmetic. Perspectives in Mathematical Logic*, Springer-Verlag, 1999.

[55] Smith, D., M. Eggen, and R. St. Andre. *A Transition to Advanced Mathematics*, third edition. Brooks/Cole Publishing Company, Wadsworth, Inc., 1990.

[56] Soare, R.I. *Recursively Enumerable Sets and Degrees. Perspectives in Mathematical Logic*, Springer-Verlag, 1987.

[57] Solovay, R. Draft of paper (or series of papers) on Chaitin's work. Unpublished notes, May 2004.

[58] Tennenbaum, S. Non-Archimedean models for arithmetic. *Notices of the American Mathematical Society* **6**:270, 1959.

[59] Turing, A.M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2, **42**:230–265 (1937). — A correction. *Proceedings of the London Mathematical Society*, Series 2, **43**:544–546 (1937).

[60] van Lambalgen, M. *Random Sequences.* Ph.D. thesis. University of Amsterdam, The Netherlands, 1987.

[61] van Lambalgen, M. The axiomatization of randomness. *Journal of Symbolic Logic* **55**(3):1143–1167, 1990.

[62] Ville, J. *Étude Critique de la Notion de Collectif.* Gauthier-Villars, Paris, 1939.

[63] Volchan, S.B. What is a random sequence. *American Mathematical Monthly* **109**(1):46–63, 2002.

[64] von Mises, R. *Probability, Statistics and Truth.* Translation of the third German edition, 1951; originally published 1928, Springer. George Allen and Unwin Ltd., London, 1957.

[65] Wald, A. Die Widerspruchsfreiheit des Kollektivbegriffs der Wahrscheinlichkeitsrechnung. *Ergebnisse eines mathematische Kollektives* **8**:38–72, 1936.

[66] Zambella, D. On sequences with simple initial segments. ILLC technical report ML-1990-05, University of Amsterdam, 1990.

[67] Zvonkin, A.K., and L.A. Levin. The complexity of finite objects and the development of concepts of information and randomness by the theory of algorithms. *Russian Mathematical Surveys* **25**(6):83–124, 1970.